LETTER Communicated by Chun-Nam Yu

Large-Scale Linear RankSVM

Ching-Pei Lee
r00922098@csie.ntu.edu.tw
Chih-Jen Lin
cjlin@csie.ntu.edu.tw
Department of Computer Science, National Taiwan University, Taipei 10617, Taiwan

Linear rankSVM is one of the widely used methods for learning to rank. Although its performance may be inferior to nonlinear methods such as kernel rankSVM and gradient boosting decision trees, linear rankSVM is useful to quickly produce a baseline model. Furthermore, following its recent development for classification, linear rankSVM may give competitive performance for large and sparse data. A great deal of works have studied linear rankSVM. The focus is on the computational efficiency when the number of preference pairs is large. In this letter, we systematically study existing works, discuss their advantages and disadvantages, and propose an efficient algorithm. We discuss different implementation issues and extensions with detailed experiments. Finally, we develop a robust linear rankSVM tool for public use.

1 Introduction

Learning to rank is an important supervised learning technique because of its application to search engines and online advertisement. According to Chapelle and Chang (2011) and others, state-of-the-art learning to rank models can be categorized into three types. Pointwise methods (e.g., decision tree models and linear regression) directly learn the relevance score of each instance; pairwise methods like rankSVM (Herbrich, Graepel, & Obermayer, 2000) learn to classify preference pairs; and listwise methods such as LambdaMART (Burges, 2010) try to optimize the measurement for evaluating the whole ranking list. Some methods lie between two categories; for example, GBRank (Zheng, Chen, Sun, & Zha, 2007) combines pointwise decision tree models and pairwise loss. Among them, rankSVM, as a pairwise approach, is one commonly used method. This method is extended from standard support vector machine (SVM) by Boser, Guyon, and Vapnik (1992) and Cortes and Vapnik (1995). In the SVM literature, it is well known that linear (i.e., data are not mapped to a different space) and kernel SVMs are suitable for different scenarios, where linear SVM is more efficient, but

Neural Computation **26, 781–817** (2014)
doi:10.1162/NECO_a_00571

the more costly kernel SVM may give higher accuracy.[1] The same situation may occur for rankSVM. In this letter, we study large-scale linear rankSVM.

Assume we are given a set of training label-query-instance tuples $(y_i, q_i, \boldsymbol{x}_i), y_i \in K \subset \mathbf{R}, q_i \in Q \subset \mathbf{Z}, \boldsymbol{x}_i \in \mathbf{R}^n,\ i = 1, \ldots, l$, where K is the set of possible relevance levels with $|K| = k$ and Q is the set of queries. By defining the set of preference pairs as

$$P \equiv \{(i, j) \mid q_i = q_j, y_i > y_j\} \text{ with } p \equiv |P|, \tag{1.1}$$

L1-loss linear rankSVM minimizes the sum of training losses and a regularization term,

$$\min_{\boldsymbol{w}} \quad \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} + C\sum_{(i,j)\in P} \max(0, 1 - \boldsymbol{w}^T(\boldsymbol{x}_i - \boldsymbol{x}_j)), \tag{1.2}$$

where $C > 0$ is a regularization parameter. If L2 loss is used, the optimization problem becomes

$$\min_{\boldsymbol{w}} \quad \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} + C\sum_{(i,j)\in P} \max(0, 1 - \boldsymbol{w}^T(\boldsymbol{x}_i - \boldsymbol{x}_j))^2. \tag{1.3}$$

In prediction, for any test instance $\boldsymbol{x}$, a larger $\boldsymbol{w}^T\boldsymbol{x}$ implies that $\boldsymbol{x}$ should be ranked higher. In Table 1, we list the notation used in this letter.

The sum of training losses can be written in the following separable form:

$$\sum_{q\in Q} \sum_{\substack{(i,j):q_i=q_j=q \\ y_i>y_j}} \max(0, 1 - \boldsymbol{w}^T(\boldsymbol{x}_i - \boldsymbol{x}_j)).$$

Because each query involves an independent training subset, the outer summation over all $q \in Q$ can be easily handled. Therefore, in our discussion, we assume a single query in the training set. Hence, if on average l/k instances are with the same relevance level, the number of pairs in P is

$$\binom{k}{2} \times O\left(\left(\frac{l}{k}\right)^2\right) = O(l^2). \tag{1.4}$$

The large number of pairs becomes the main difficulty in training rankSVM. Many existing studies have attempted to address this difficulty. By taking the property that

$$y_i > y_j \quad \text{and} \quad y_j > y_s \ \Rightarrow\ y_j > y_s, \tag{1.5}$$

[1] See, for example, Yuan, Ho, and Lin (2012) for more detailed discussion.

Table 1: Notation.

Notation	Explanation
$\boldsymbol{w}$	The weight vector obtained by solving problem 1.2 or 1.3
$\boldsymbol{x}_i$	The feature vector of the ith training instance
y_i	Label of the ith training instance
q_i	Query of the ith training instance
K	The set of relevance levels
Q	The set of queries
P	The set of preference pairs; see equation 1.1
l	Number of training instances
k	Number of relevance levels
p	Number of preference pairs
n	Number of features
$\bar{n}$	Average number of nonzero features per instance
l_q	Number of training instances in a given query
k_q	Number of relevance levels in a given query
T	An order-statistic tree

it is possible to avoid the $O(l^2)$ complexity of going through all pairs in calculating the objective function, gradient, or other information needed in the optimization procedure. Interestingly, although existing work applies different optimization methods, their approaches to avoid considering the $O(l^2)$ pairs are very related. Next, we briefly review some recent results. Joachims (2006) solves problem 1.2 by a cutting plane method in which an

$$O(l\bar{n} + l\log l + lk + n) \tag{1.6}$$

method is proposed to calculate the objective value and a subgradient of problem 1.2. The $O(l\bar{n})$ cost is for calculating $\boldsymbol{w}^T\boldsymbol{x}_i$, $\forall i$, where $\bar{n}$ is the average number of nonzero features per training instance; $O(l\log l + lk)$ is for the sum of training losses in problem 1.2; $O(n)$ is for the regularization term $\boldsymbol{w}^T\boldsymbol{w}/2$. This method is efficient if k (number of relevance levels) is small but becomes inefficient when $k = O(l)$. Airola, Pahikkala, and Salakoski (2011) improve on Joachims's work by reducing the complexity to

$$O(l\bar{n} + l\log l + l\log k + n).$$

The main breakthrough is that they employ order-statistic trees, which are extended from balance binary search trees, so the $O(lk)$ term in equation 1.6 is reduced to $O(l\log k)$. Another type of optimization method considered is the truncated Newton method for solving problem 1.3 in which the main computation is on Hessian vector products. Chapelle & Keerthi (2010) showed that if $k = 2$ (i.e., only two relevance levels), the cost of each

function, gradient, or Hessian vector product evaluation is $O(l\bar{n} + l \log l + n)$. Their method is related to that by Joachims (2006) because we can see that the $O(lk)$ term in equation 1.6 can be removed if $k = 2$. For $k > 2$, they decompose the problem into $k - 1$ subproblems, each with only two relevance levels. Therefore, similar to Joachims's approach, Chapelle and Keerthi's approach may not be efficient for larger k. Regarding optimization methods, an advantage of Newton methods is the faster convergence. However, they require the differentiability of the objective function, so L2 loss must be used. In contrast, cutting plane methods are applicable to both L1 and L2 losses.

Although linear rankSVM is an established method, it is known that gradient boosting decision trees (GBDT) by Friedman (2001) and its variant, LambdaMART (Burges, 2010), give competitive performance on web search ranking data. In addition, random forests (Breiman, 2001) are also reported in Mohan, Chen, and Weinberger (2011) to perform well. Actually all the winning teams of the Yahoo Learning to Rank Challenge (Chapelle & Chang, 2011) use decision-tree-based ensemble models. Note that GBDT and random forests are nonlinear pointwise methods, and LambdaMART is a nonlinear listwise method. Their drawback is the longer training time. We will conduct experiments to compare the performance and training time between linear and nonlinear ranking methods.

In this letter, we consider Newton methods for solving problem 1.3 and present the following results:

1. We give a clear overview and connection of past work on the efficient calculation over all relevance pairs.
2. We investigate several order-statistic tree implementations and show their advantages and disadvantages.
3. We finish an efficient implementation that is faster than existing works for linear rankSVM.
4. We compare linear rankSVM and linear and nonlinear pointwise methods, including GBDT and random forests, in detail.
5. We release a public tool for linear rankSVM.

This letter is organized as follows. Section 2 introduces methods for the efficient calculation of all relevance pairs. Section 3 discusses past studies of linear rankSVM and compares them with our method. Various types of experiments are shown in section 4. In section 5 we discuss another possible algorithm for rankSVMs when k is large. Section 6 concludes. A supplementary file including additional analysis and experiments is available online at http://www.mitpressjournals.org/doi/suppl/10.1162/NECO_a_00571.

2 Efficient Calculation over Relevance Pairs

A difficulty in training rankSVM is that the number of pairs in the loss term can be as large as $O(l^2)$. This difficulty occurs in any optimization

method that needs to calculate the objective value. Further more, other values used in optimization procedures, such as subgradient, gradient, or Hessian-vector products, face the same difficulty. In this section, we consider truncated Newton methods as an example and investigate efficient methods for the calculation over pairs.

2.1 Information Needed in Optimization Procedures and an Example Using Truncated Newton Methods. Many optimization methods employ gradient or even higher-order information at each iteration of an iterative procedure. From problems 1.2 and 1.3, it is clear that the summation over the p pairs remains in the gradient and the Hessian. Therefore, the difficulty of handling $O(l^2)$ pairs occurs beyond the calculations of the objective value. Here we consider the truncated Newton method as an example to see what kind of information it requires.

A Newton method obtains a direction at the tth iteration by solving

$$\min_{s} \quad g_t(s),$$

where

$$g_t(s) \equiv \nabla f(w^t)^T s + \frac{1}{2} s^T \nabla^2 f(w^t) s, \tag{2.1}$$

and updates w^t by

$$w^{t+1} = w^t + s.$$

Note that $g_t(s)$ is the second-order Taylor approximation of $f(w^t + s) - f(w^t)$. If $\nabla^2 f(w^t)$ is invertible, step s is obtained by solving the following linear system;

$$\nabla^2 f(w^t) s = -\nabla f(w^t). \tag{2.2}$$

To ensure convergence, usually a line search or a trust region technique is applied to obtain a truncated Newton step. For machine learning applications, $\nabla^2 f(w)$ is often too large to be stored, so the conjugate gradient (CG) method is a common way to solve equation 2.2 by taking the special structure of $\nabla^2 f(w)$ into account (Keerthi & DeCoste, 2005; Lin, Weng, & Keerthi, 2008). At each CG iteration, we only need to calculate a Hessian-vector product,

$$\nabla^2 f(w^t) v, \text{ for some vector } v,$$

and it can be performed without storing the Hessian matrix. Then the algorithm contains two levels of iterations. The outer one generates $\{w^t\}$,

Algorithm 1: Truncated Newton Methods.

1. Given $\boldsymbol{w}^0$.
2. For $t = 0, 1, 2, \ldots$
   - 2.1. If the stopping condition is satisfied, return $\boldsymbol{w}^t$.
   - 2.2. Apply CG iterations to solve equation 2.2, together with some additional constraints for ensuring convergence. A truncated Newton step is obtained.
   - 2.3. $\boldsymbol{w}^{t+1} \leftarrow \boldsymbol{w}^t + \boldsymbol{s}$.

while from $\boldsymbol{w}^t$ to $\boldsymbol{w}^{t+1}$ there are inner CG iterations. Algorithm 1 gives the framework of truncated Newton methods.

The discussion shows that in a truncated Newton method, the calculations of objective value, gradient, and Hessian-vector product all face the difficulty of handling p pairs. In the rest of this section, we discuss the efficient calculation of these values.

2.2 Efficient Function and Gradient Evaluation and Matrix-Vector Products. We consider L2-loss linear rankSVM (see problem 1.3) and discuss details of calculating the function, gradient, and Hessian-vector products. These are the computational bottlenecks in Newton methods. In the rest of this letter, if not specified, $f(\boldsymbol{w})$ represents the objective function of problem 1.3.

To indicate the preference pairs, we define a p by l matrix:

$$A \equiv \begin{array}{c} \\ \vdots \\ (i,\, j) \\ \vdots \end{array} \begin{array}{c} \begin{array}{ccccccc} & \cdots & i & \cdots & & j & \cdots \end{array} \\ \begin{bmatrix} 0\cdots0 & +1 & 0\cdots0 & -1 & 0\cdots0 \end{bmatrix} \end{array}.$$

That is, if $(i, j) \in P$, then a corresponding row in A has that the ith entry is 1, the jth entry is -1, and other entries are all zeros. By this definition, the objective function in problem 1.3 can be written as

$$f(\boldsymbol{w}) = \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} + C(\boldsymbol{e} - AX\boldsymbol{w})^T D_{\boldsymbol{w}}(\boldsymbol{e} - AX\boldsymbol{w}), \tag{2.3}$$

where $\boldsymbol{e} \in \mathbf{R}^{p\times 1}$ is a vector of ones,

$$X \equiv \left[\boldsymbol{x}_1, \ldots, \boldsymbol{x}_l\right]^T,$$

and $D_{\boldsymbol{w}}$ is a p by p diagonal matrix with for all $(i, j) \in P$,

$$(D_{\boldsymbol{w}})_{(i,j),(i,j)} \equiv \begin{cases} 1 & \text{if } 1 - \boldsymbol{w}^{\mathrm{T}}(\boldsymbol{x}_{\mathrm{i}} - \boldsymbol{x}_{\mathrm{j}}) > 0, \\ 0 & \text{otherwise.} \end{cases}$$

The gradient is

$$\begin{aligned} \nabla f(\boldsymbol{w}) &= \boldsymbol{w} - 2C \sum_{(i,j)\in P} (\boldsymbol{x}_i - \boldsymbol{x}_j) \max\left(0, 1 - (\boldsymbol{x}_i - \boldsymbol{x}_j)^T \boldsymbol{w}\right) \\ &= \boldsymbol{w} + 2CX^T(A^T D_{\boldsymbol{w}} AX\boldsymbol{w} - A^T D_{\boldsymbol{w}} \boldsymbol{e}). \end{aligned} \tag{2.4}$$

However, $\nabla^2 f(\boldsymbol{w})$ does not exist because equation 2.4 is not differentiable. Following Mangasarian (2002) and Lin et al. (2008), we define a generalized Hessian matrix,

$$\nabla^2 f(\boldsymbol{w}) \equiv I + 2CX^T A^T D_{\boldsymbol{w}} AX, \tag{2.5}$$

where I is the identity matrix.

The main computation at each CG iteration is a Hessian-vector product. For any vector $\boldsymbol{v} \in \mathbf{R}^n$, the truncated Newton method PRSVM by Chapelle and Keerthi (2010) calculates

$$\nabla^2 f(\boldsymbol{w})\boldsymbol{v} = \boldsymbol{v} + 2CX^T\left(A^T\left(D_{\boldsymbol{w}}\left(A(X\boldsymbol{v})\right)\right)\right). \tag{2.6}$$

Because both A and $D_{\boldsymbol{w}}$ have $O(p)$ nonzero elements, the complexity of calculating equation 2.6 is

$$O(l\bar{n} + p + n). \tag{2.7}$$

The right-to-left matrix-vector products in equation 2.6 are faster than $O(p\bar{n})$ if we obtain and store the matrix $X^T A^T D_{\boldsymbol{w}} AX$.

However, if $p = O(l^2)$, not only is the cost of equation 2.6 still high, but also the storage of A and $D_{\boldsymbol{w}}$ requires a huge amount of memory. To derive a faster method, it is essential to explore the structure of the generalized Hessian. We define

$$\begin{aligned} &\mathrm{SV}(\boldsymbol{w}) \equiv \{(i, j) \mid (i, j) \in P,\ 1 - \boldsymbol{w}^T(\boldsymbol{x}_i - \boldsymbol{x}_j) > 0\}, \quad \text{and} \\ &\quad p_{\boldsymbol{w}} \equiv |\mathrm{SV}(\boldsymbol{w})|. \end{aligned} \tag{2.8}$$

We show in appendix A that when problem 1.3 is treated as an SVM classification problem with feature vectors $\boldsymbol{x}_i - \boldsymbol{x}_j$ and labels being 1 for all $(i, j) \in P$, the set $\mathrm{SV}(\boldsymbol{w})$ corresponds to the support vectors.

We then remove the matrix $D_{\boldsymbol{w}}$ by defining a new matrix $A_{\boldsymbol{w}} \in \mathbf{R}^{p_{\boldsymbol{w}} \times l}$ such that

$$A_{\boldsymbol{w}}^T A_{\boldsymbol{w}} = A^T D_{\boldsymbol{w}} A,$$

where $A_{\boldsymbol{w}}$ includes rows of A such that $(i, j) \in \mathrm{SV}(\boldsymbol{w})$. Thus, equation 2.6 becomes

$$\nabla^2 f(\boldsymbol{w})\boldsymbol{v} = \boldsymbol{v} + 2CX^T A_{\boldsymbol{w}}^T A_{\boldsymbol{w}} X\boldsymbol{v}. \tag{2.9}$$

Observe that

$$(A_{\boldsymbol{w}}^T A_{\boldsymbol{w}})_{i,j} = \sum_s (A_{\boldsymbol{w}})_{s,i}(A_{\boldsymbol{w}})_{s,j}. \tag{2.10}$$

Because each row of $A_{\boldsymbol{w}}$ contains two nonzero elements, $(A_{\boldsymbol{w}})_{s,i}(A_{\boldsymbol{w}})_{s,j} \neq 0$ occurs only under the following situations:

$$(A_{\boldsymbol{w}})_{s,i}, (A_{\boldsymbol{w}})_{s,j}$$
$$= \begin{cases} 1, 1 & \text{if } i = j \text{ and } s \text{ corresponds to } (i,t) \in \mathrm{SV}(\boldsymbol{w}) \\ & \quad \text{for some } 1 \le t \le l, \\ -1, -1 & \text{if } i = j \text{ and } s \text{ corresponds to } (t,i) \in \mathrm{SV}(\boldsymbol{w}) \\ & \quad \text{for some } 1 \le t \le l, \\ 1, -1 & \text{if } i \neq j \text{ and } s \text{ corresponds to } (i,j) \in \mathrm{SV}(\boldsymbol{w}), \\ -1, 1 & \text{if } i \neq j \text{ and } s \text{ corresponds to } (j,i) \in \mathrm{SV}(\boldsymbol{w}). \end{cases}$$

We define

$$\mathrm{SV}_i^+(\boldsymbol{w}) \equiv \{j \mid (j,i) \in \mathrm{SV}(\boldsymbol{w})\}, \quad l_i^+(\boldsymbol{w}) \equiv |\mathrm{SV}_i^+(\boldsymbol{w})|,$$
$$\gamma_i^+(\boldsymbol{w}, \boldsymbol{v}) \equiv \sum_{j \in \mathrm{SV}_i^+(\boldsymbol{w})} \boldsymbol{x}_j^T \boldsymbol{v},$$
$$\mathrm{SV}_i^-(\boldsymbol{w}) \equiv \{j \mid (i,j) \in \mathrm{SV}(\boldsymbol{w})\}, \quad l_i^-(\boldsymbol{w}) \equiv |\mathrm{SV}_i^-(\boldsymbol{w})|,$$
$$\gamma_i^-(\boldsymbol{w}, \boldsymbol{v}) \equiv \sum_{j \in \mathrm{SV}_i^-(\boldsymbol{w})} \boldsymbol{x}_j^T \boldsymbol{v}.$$

Then from equation 2.10,

$$(A_{\boldsymbol{w}}^T A_{\boldsymbol{w}})_{i,j} = \begin{cases} l_i^+(\boldsymbol{w}) + l_i^-(\boldsymbol{w}) & \text{if } i = j, \\ -1 & \text{if } i \neq j, \text{ and } (i,j) \text{ or } (j,i) \in \mathrm{SV}(\boldsymbol{w}), \\ 0 & \text{otherwise.} \end{cases}$$

Hence,

$$(A_{\boldsymbol{w}}^T A_{\boldsymbol{w}} X\boldsymbol{v})_i = \sum_{j=1}^{l} (A_{\boldsymbol{w}}^T A_{\boldsymbol{w}})_{i,j} (X\boldsymbol{v})_j$$
$$= \left(l_i^+(\boldsymbol{w}) + l_i^-(\boldsymbol{w})\right)\boldsymbol{x}_i^T\boldsymbol{v} - \sum_{j \in \mathrm{SV}_i^+(\boldsymbol{w})} \boldsymbol{x}_j^T\boldsymbol{v} - \sum_{j \in \mathrm{SV}_i^-(\boldsymbol{w})} \boldsymbol{x}_j^T\boldsymbol{v}.$$

Therefore,

$$X^T A_{\boldsymbol{w}}^T A_{\boldsymbol{w}} X\boldsymbol{v} = X^T \begin{bmatrix} \left(l_1^+(\boldsymbol{w}) + l_1^-(\boldsymbol{w})\right)\boldsymbol{x}_1^T\boldsymbol{v} - \left(\gamma_1^+(\boldsymbol{w},\boldsymbol{v}) + \gamma_1^-(\boldsymbol{w},\boldsymbol{v})\right) \\ \vdots \\ \left(l_l^+(\boldsymbol{w}) + l_l^-(\boldsymbol{w})\right)\boldsymbol{x}_l^T\boldsymbol{v} - \left(\gamma_l^+(\boldsymbol{w},\boldsymbol{v}) + \gamma_l^-(\boldsymbol{w},\boldsymbol{v})\right) \end{bmatrix}. \tag{2.11}$$

If we already have the values of $l_i^+(\boldsymbol{w})$, $l_i^-(\boldsymbol{w})$, $\gamma_i^+(\boldsymbol{w},\boldsymbol{v})$, and $\gamma_i^-(\boldsymbol{w},\boldsymbol{v})$, the computation of the Hessian-vector product in equation 2.9 would just cost $O(l\bar{n}+n)$, where $O(l\bar{n})$ is for computing equation 2.11 and $O(n)$ is for the vector addition in equation 2.9.

Similarly, function and gradient evaluations can be more efficient by reformulating equations 2.3 and 2.4 to the following forms, respectively:

$$f(\boldsymbol{w}) = \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} + C(A_{\boldsymbol{w}}X\boldsymbol{w} - \boldsymbol{e}_{\boldsymbol{w}})^T(A_{\boldsymbol{w}}X\boldsymbol{w} - \boldsymbol{e}_{\boldsymbol{w}})$$
$$= \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} + C(\boldsymbol{w}^T X^T (A_{\boldsymbol{w}}^T A_{\boldsymbol{w}} X\boldsymbol{w} - 2A_{\boldsymbol{w}}^T\boldsymbol{e}_{\boldsymbol{w}}) + p_{\boldsymbol{w}}) \tag{2.12}$$

and

$$\nabla f(\boldsymbol{w}) = \boldsymbol{w} + 2CX^T(A_{\boldsymbol{w}}^T A_{\boldsymbol{w}} X\boldsymbol{w} - A_{\boldsymbol{w}}^T\boldsymbol{e}_{\boldsymbol{w}}), \tag{2.13}$$

where $\boldsymbol{e}_{\boldsymbol{w}} \in \mathbf{R}^{p_{\boldsymbol{w}} \times 1}$ is a vector of ones. In equations 2.12 and 2.13, $A_{\boldsymbol{w}}^T A_{\boldsymbol{w}} X\boldsymbol{w}$ can be calculated by equation 2.11. We also have

$$A_{\boldsymbol{w}}^T\boldsymbol{e}_{\boldsymbol{w}} = \begin{bmatrix} l_1^-(\boldsymbol{w}) - l_1^+(\boldsymbol{w}) \\ \vdots \\ l_l^-(\boldsymbol{w}) - l_l^+(\boldsymbol{w}) \end{bmatrix}, \quad \text{and} \quad p_{\boldsymbol{w}} = \sum_{i=1}^{l} l_i^+(\boldsymbol{w}) = \sum_{i=1}^{l} l_i^-(\boldsymbol{w}).$$

Thus, the computation of both equations 2.12 and 2.13 costs $O(l\bar{n}+n)$ as well.

Note that for solving problem 1.2 by cutting plane methods, Joachims (2006) and Airola et al. (2011) have identified that $l_i^+(\boldsymbol{w})$ and $l_i^-(\boldsymbol{w})$ are needed for efficient function and subgradient evaluation (see more details in section 3.2). Therefore, regardless of the optimization methods used, an important common task is to efficiently calculate $l_i^+(\boldsymbol{w})$, $l_i^-(\boldsymbol{w})$, $\gamma_i^+(\boldsymbol{w}, \boldsymbol{v})$, and $\gamma_i^-(\boldsymbol{w}, \boldsymbol{v})$. In the supplementary materials, we discuss in detail a direct method that costs $O(l + k)$ space excluding the training data and

$$O(l\bar{n} + lk + n) \tag{2.14}$$

time for one matrix-vector product. Although the cost is lower than that by equation 2.6, the $O(lk)$ complexity is still high if k is large. Subsequently, we discuss methods to reduce the $O(lk)$ term to $O(l \log k)$.

2.3 Efficient Calculation by Storing Values in an Order-Statistic Tree. Airola et al. (2011) calculate $l_i^+(\boldsymbol{w})$ and $l_i^-(\boldsymbol{w})$ by an order-statistic tree, so the $O(lk)$ term in equation 2.14 is reduced to $O(l \log k)$. The optimization method used is a cutting plane method (Teo, Vishwanathan, Smola, & Le, 2010), which calculates function and subgradient values. Our procedure here is an extension because in Newton methods, we further need Hessian-vector products. Notice that Airola et al. (2011) considered problem 1.2; we solve problem 1.3 and require the computation of $\gamma_i^+(\boldsymbol{w}, \boldsymbol{v})$ and $\gamma_i^-(\boldsymbol{w}, \boldsymbol{v})$ in addition.

To find $l_i^+(\boldsymbol{w})$, we must count the cardinality of the following set:

$$\mathrm{SV}_i^+(\boldsymbol{w}) = \{j \mid y_j > y_i, \boldsymbol{w}^T\boldsymbol{x}_j < \boldsymbol{w}^T\boldsymbol{x}_i + 1\}.$$

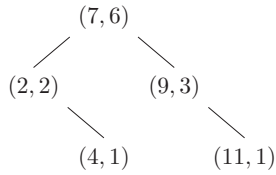
The main difficulty is that both the order of y_i and the order of $\boldsymbol{w}^T\boldsymbol{x}_i$ are involved. We can first sort $\boldsymbol{w}^T\boldsymbol{x}_i$ in ascending order. For an easy description, we assume that

$$\boldsymbol{w}^T\boldsymbol{x}_1 \le \cdots \le \boldsymbol{w}^T\boldsymbol{x}_l. \tag{2.15}$$

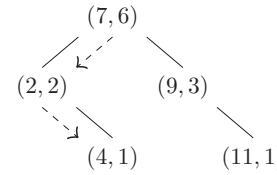
We observe that if elements in

$$\{j \mid \boldsymbol{w}^T\boldsymbol{x}_j < \boldsymbol{w}^T\boldsymbol{x}_i + 1\} \tag{2.16}$$

have been properly arranged in an order-statistic tree T by the value of y_j, then $l_i^+(\boldsymbol{w})$ can be obtained in $O(\log k)$ time. Consider the following



(a) An example of arranging elements of the set (2.17) in a tree. Each node contains $(y_j, \text{size}(y_j))$.



(b) If $y_1 = 4$, we find $l_1^+(\boldsymbol{w})$ by $\text{Larger}(7,4) = \text{Larger}(2,4) + 6 - 2 = \text{Larger}(4,4) + 4 = 4$.

Figure 1: An illustration of using an order-statistic tree to calculate $l_i^+(\boldsymbol{w})$.

example:

i	1	2	3	4	5	6	7	8	$\cdots$
y_i	4	7	9	9	2	11	5	7	$\cdots$

When $i = 1$, we assume

$$\{j \mid \boldsymbol{w}^T \boldsymbol{x}_j < \boldsymbol{w}^T \boldsymbol{x}_1 + 1\} = \{1, 2, 3, 4, 5, 6\}. \tag{2.17}$$

We construct a tree in Figure 1a so that each node includes

$$\begin{cases} \text{key} : y_j, \\ \text{size} : \text{number of instances in tree}(y_j), \end{cases} \tag{2.18}$$

where tree$(y) \equiv$ tree with root y, and nodes are arranged according to the keys (i.e., y_j values). For each node, we ensure that its right child has a larger key than its left child and the node itself. Clearly, for any node y,

$$\text{size}(y) = \begin{cases} |\{j \mid y_j = y,\ j \in T\}| & \text{if } y \text{ is a leaf,} \\ \text{size}(y\text{'s left child}) + \text{size}(y\text{'s right child}) & \\ \quad + |\{j \mid y_j = y,\ j \in T\}| & \text{otherwise.} \end{cases} \tag{2.19}$$

By $j \in T$, we mean the instance j has been inserted to the tree T.

To find $l_i^+(\boldsymbol{w})$, we traverse from the root of T to the node y_i by observing that

$$
\begin{aligned}
&|\{j \mid y_j > y_i \text{ and } j \in \text{tree}(y)\}| \\
&= \begin{cases}
|\{j \mid y_j > y_i \text{ and } j \in \text{tree}(y\text{'s right child})\}| & \text{if } y \le y_i, \\
|\{j \mid y_j > y_i \text{ and } j \in \text{tree}(y\text{'s left child})\}| & \\
\quad + \text{size}(y) - \text{size}(y\text{'s left child}) & \text{if } y > y_i.
\end{cases}
\end{aligned}
$$

Therefore, once a tree for the set 2.16 has been constructed, we can define the following function:

$$
\begin{aligned}
&\text{Larger}(y, y_i) \equiv |\{j \mid y_j > y_i,\ j \in \text{tree}(y)\}| \\
&= \begin{cases}
0 & \text{if } y \text{ is a leaf, and } y \le y_i, \\
\text{size}(y) & \text{if } y \text{ is a leaf, and } y > y_i, \\
\text{size}(y\text{'s right child}) & \text{if } y = y_i, \\
\text{Larger}(y\text{'s left child}, y_i) & \\
\quad + \text{size}(y) - \text{size}(y\text{'s left child}) & \text{if } y > y_i, \\
\text{Larger}(y\text{'s right child}, y_i) & \text{if } y < y_i.
\end{cases}
\end{aligned} \tag{2.20}
$$

We explain that equation 2.20 uses the idea of equation 1.5 on y. In the last case of equation 2.20, for any t in the left of y,

$$y_t < y$$

by the way we construct the tree. Then with

$$y < y_i,$$

we have

$$y_t < y_i,$$

so nodes on the left are not considered. Using equation 2.20, we have

$$l_i^+(\boldsymbol{w}) = \text{Larger}\big(\text{root of } T, y_i\big).$$

An example to traverse the tree for finding $l_1^+(\boldsymbol{w})$ is in Figure 1b.

Once $l_i^+(\boldsymbol{w})$ has been calculated, we insert the following instances into the tree:

$$\{j \mid \boldsymbol{w}^T\boldsymbol{x}_i + 1 \le \boldsymbol{w}^T\boldsymbol{x}_j < \boldsymbol{w}^T\boldsymbol{x}_{i+1} + 1\}.$$

Then, $l_{i+1}^+(\boldsymbol{w})$ can be calculated in the same way. The calculation for $l_i^-(\boldsymbol{w})$ is similar. Because

$$\mathrm{SV}_i^-(\boldsymbol{w}) = \{j \mid y_j < y_i, \boldsymbol{w}^T\boldsymbol{x}_j > \boldsymbol{w}^T\boldsymbol{x}_i - 1\},$$

we start from l and maintain a tree of the following set:

$$\{j \mid \boldsymbol{w}^T\boldsymbol{x}_j > \boldsymbol{w}^T\boldsymbol{x}_i - 1\}.$$

We then define a function $\mathrm{Smaller}(y, y_i)$ similar to $\mathrm{Larger}(y, y_i)$ to obtain $l_i^-(\boldsymbol{w})$.

For the calculation of $\gamma_i^+(\boldsymbol{w}, \boldsymbol{v})$ and $\gamma_i^-(\boldsymbol{w}, \boldsymbol{v})$, notice that

$$l_i^+(\boldsymbol{w}) = \sum_{j \in \mathrm{SV}_i^+(\boldsymbol{w})} 1, \quad \text{and} \quad \gamma_i^+(\boldsymbol{w}, \boldsymbol{v}) = \sum_{j \in \mathrm{SV}_i^+(\boldsymbol{w})} \boldsymbol{x}_i^T\boldsymbol{v} \tag{2.21}$$

and

$$\mathrm{size}(y) = \sum_{j:\, j \in \mathrm{tree}(y)} 1.$$

At each node of the tree, we can store a value $\mathrm{xv}(y)$ so that it follows a relation like equation 2.19:

$$\begin{aligned}\mathrm{xv}(y) &\equiv \sum_{j:\, j \in \mathrm{tree}(y)} \boldsymbol{x}_j^T\boldsymbol{v} \\ &= \begin{cases} \displaystyle\sum_{j:\, j \in T, y_j = y} \boldsymbol{x}_j^T\boldsymbol{v} & \text{if } y \text{ has no child,} \\ \mathrm{xv}(y\text{'s left child}) + \mathrm{xv}(y\text{'s right child}) & \text{otherwise.} \\ \quad + \displaystyle\sum_{j:\, j \in T, y_j = y} \boldsymbol{x}_j^T\boldsymbol{v} & \end{cases}\end{aligned} \tag{2.22}$$

The function $\mathrm{Larger}(y, y_i)$, defined in equation 2.20, can be directly extended to output both $l_i^+(\boldsymbol{w})$ and $\gamma_i^+(\boldsymbol{w}, \boldsymbol{v})$.

Details of the overall procedure are presented in algorithm 2. For binary search trees, we can consider, for example, the AVL tree (Adelson-Velsky

Algorithm 2: Obtaining $l_i^+(\boldsymbol{w}), l_i^-(\boldsymbol{w}), \gamma_i^+(\boldsymbol{w}, \boldsymbol{v})$ and $\gamma_i^-(\boldsymbol{w}, \boldsymbol{v})$ Using an Order-Statistic Tree.

1. Given X, $\boldsymbol{w}$ and $\boldsymbol{v}$, compute $X\boldsymbol{w}$ and $X\boldsymbol{v}$.
2. Sort $\boldsymbol{w}^T\boldsymbol{x}_i$ in ascending order: $\boldsymbol{w}^T\boldsymbol{x}_{\pi(1)} \le \cdots \le \boldsymbol{w}^T\boldsymbol{x}_{\pi(l)}$.
3. $j \leftarrow 1$, $T \leftarrow$ an empty order-statistic tree.
4. For $i = 1, \ldots, l$
   - 4.1. While $j \le l$ and $1 - \boldsymbol{w}^T\boldsymbol{x}_{\pi(j)} + \boldsymbol{w}^T\boldsymbol{x}_{\pi(i)} > 0$
     - 4.1.1. Insert $(y_{\pi(j)}, \boldsymbol{x}_{\pi(j)}^T\boldsymbol{v})$ into T.
     - 4.1.2. $j \leftarrow j + 1$.
   - 4.2. $\left(l_{\pi(i)}^+(\boldsymbol{w}), \gamma_{\pi(i)}^+(\boldsymbol{w}, \boldsymbol{v})\right) \leftarrow$ Larger(root of T, $y_{\pi(i)}$).
5. $j \leftarrow l$, $T \leftarrow$ an empty order-statistic tree.
6. For $i = l, \ldots, 1$
   - 6.1. While $j \ge 1$ and $1 - \boldsymbol{w}^T\boldsymbol{x}_{\pi(i)} + \boldsymbol{w}^T\boldsymbol{x}_{\pi(j)} > 0$
     - 6.1.1. Insert $(y_{\pi(j)}, \boldsymbol{x}_{\pi(j)}^T\boldsymbol{v})$ into T.
     - 6.1.2. $j \leftarrow j - 1$.
   - 6.2. $\left(l_{\pi(i)}^-(\boldsymbol{w}), \gamma_{\pi(i)}^-(\boldsymbol{w}, \boldsymbol{v})\right) \leftarrow$ Smaller(root of T, $y_{\pi(i)}$).

& Landis, 1962), red-black tree (Bayer, 1972), and AA tree (Andersson, 1993). These trees are reasonably balanced so that each insertion and computation of Larger/Smaller functions all cost $O(\log k)$ (see more discussion in section 2.5).If $\boldsymbol{w}^T\boldsymbol{x}_i$ have been sorted before CG iterations, each matrix-vector product involves

$$O(l\bar{n} + l\log k + n) \tag{2.23}$$

operations, which are smaller than equation 2.14 because the lk term is reduced to $l\log k$. Therefore, the cost of truncated Newton methods using order-statistic trees is

$$\left(O(l\log l) + O(l\bar{n} + l\log k + n) \times \text{ average \#CG iterations }\right) \times \#\text{outer iterations},$$

where the $O(l\log l)$ term is the cost of sorting.

Our algorithm constructs a tree for each matrix-vector product (or each CG iteration) because of the change of the vector $\boldsymbol{v}$ in equation 2.11. Thus, an outer iteration of truncated Newton method requires constructing several trees. If we store $\sum_{j:j\in\text{tree}(y)} \boldsymbol{x}_j$ instead of $\text{xv}(y)$ at each node, only one tree independent of $\boldsymbol{v}$ is needed at an outer iteration. However, because a

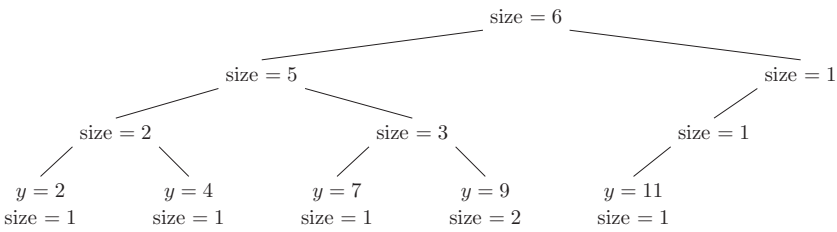


Figure 2: An example of storing y_j in leaf nodes.

vector is stored at a node, each update requires $O(\bar{n})$ cost. The total cost of maintaining the tree is $O(\bar{n}l \log k)$ because each insertion requires $O(\log k)$ updates. This is bigger than $O(l \log k + l\bar{n})$ for a tree of storing $\mathrm{xv}(y)$. Further, we need $O(ln)$ space to store vectors.[2] Because the number of matrix-vector products is often not large, storing $\mathrm{xv}(y)$ is more suitable.

Besides, instead of sorting $\boldsymbol{w}^T \boldsymbol{x}_i$ and using y_i as the keys, we may alternatively sort y_i such that

$$y_{\pi(1)} \leq \cdots \leq y_{\pi(l)} \tag{2.24}$$

and for $y_{\pi(i)}$ maintain a tree T of the following set:

$$\{j \mid y_{\pi(j)} > y_{\pi(i)}\}.$$

Then we can apply the same approach as above. An advantage of this approach is that y_i are fixed and need to be sorted only once in the training procedure. However, $\boldsymbol{w}^T \boldsymbol{x}_i$ become keys of nodes and are in general different, so the tree will eventually contain $O(l)$ rather than $O(k)$ nodes. Therefore, this approach is less preferred because maintaining a smaller tree is essential.

2.4 A Different Implementation by Storing Keys in Leaves of a Tree. Although the method in section 2.3 successfully reduces the complexity, maintaining the trees makes the implementation more complicated. In this section, we consider a simpler method that doubles the size of the tree and stores all instances in leaf nodes. This setting is similar to a selection tree (Knuth, 1973). We ensure that the k leaf nodes from left to right correspond to the ascending order of relevance levels. At a leaf node, we record the size and xv of a relevance level. For each internal node, which is the root of a subtree, its size and xv are both the sum of that attribute of its children. For the same example considered in section 2.3, the tree at $i = 1$ is shown in Figure 2.

[2]Note that $\sum_{j:\, j \in \mathrm{tree}(y)} \boldsymbol{x}_j$ is likely to be dense even if each $\boldsymbol{x}_j$ is sparse.

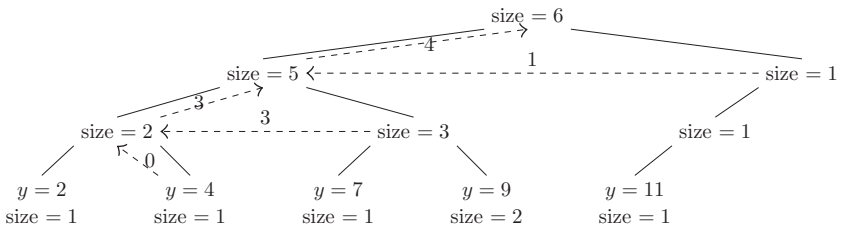


Figure 3: An example of finding $l_1^+(\boldsymbol{w})$ when storing y_j in leaf nodes. Note that we assume $y_1 = 4$.

To compute $l_i^+(\boldsymbol{w})$, we know that

$$|\{j \mid y_j > y_i\}| = \text{sum of the size attribute of leaf nodes on the right side of } y_i.$$

Therefore, for any node s in the tree, we can define:

$$\text{Larger}(s) \equiv \text{sum of the size attribute of leaf nodes on the right of } s:$$

$$= \begin{cases} \text{Larger(parent of } s\text{)} + \text{size(sibling of } s\text{)} & \text{if } s \text{ is the left child,} \\ \text{Larger(parent of } s\text{)} & \text{if } s \text{ is the right child,} \\ 0 & \text{if } s \text{ is the root,} \end{cases}$$

and let

$$l_i^+(\boldsymbol{w}) = \text{Larger(the leaf node with key } = y_i).$$

An illustration of finding $l_1^+(\boldsymbol{w})$ is in Figure 3. The procedures for obtaining $l_i^-(\boldsymbol{w})$, $\gamma_i^+(\boldsymbol{w}, \boldsymbol{v})$, and $\gamma_i^-(\boldsymbol{w}, \boldsymbol{v})$ are all similar.

An advantage of this approach is that because all y_j are stored in leaves, it is easier to maintain the trees. (See more discussion in section 2.5.) In section 4.3, we experimentally compare this approach with the method in section 2.3.

2.5 A Discussion on Tree Implementations. For the method in section 2.3, where each node has a key, we can consider balanced binary search trees such as an AVL tree, red-black tree, and AA tree. AVL trees use more complicated insertion operations to ensure being balanced. Consequently, the insertion is slower, but the order-statistic computation is usually faster compared to other order-statistic trees. In the comparison by

Heger (2004), an AA tree tends to be more balanced and faster than a red-black tree. However, previous studies also consider node deletions, which are not needed here, so we conduct an experiment in section 4.3.

For the method in section 2.4 to store keys in leaves, the selection tree is a suitable data structure. Note that selection trees were mainly used for sorting, but using it as a balanced binary search tree is a straightforward adaptation. An implementation method introduced in Knuth (1973) is to transfer k possible y_i values to $2^{\lceil \log_2 k \rceil}, 2^{\lceil \log_2 k \rceil} + 1 \ldots, 2^{\lceil \log_2 k \rceil} + k - 1$, and let the indices of the internal nodes be $1, 2, \ldots, 2^{\lceil \log_2 k \rceil} - 1$. Then for any node m, its parent is the node $\lfloor \frac{m}{2} \rfloor$. Moreover, if m is an odd number then it is a right child, and vice versa. By this method, we do not need to use pointers for constructing the tree, and thus the implementation is very simple. Another advantage is that this tree is fully balanced, so each leaf is of the same depth.

3 Comparison with Existing Methods

In this section, we introduce recent studies of linear rankSVM that are considered state of the art. Some of them were mentioned in section 2 in comparison with our proposed methods.

3.1 PRSVM and PRSVM+. We have discussed PRSVM by Chapelle and Keerthi (2010) at the beginning of section 2.2. The complexity shown in equation 2.7 has a term $O(p)$, which becomes dominant for large p. To reduce the cost, Chapelle and Keerthi (2010) proposed PRSVM+ for solving problem 1.3 by a truncated Newton method. They first consider the case of $k = 2$ (i.e., two relevance levels). The algorithm for calculating $l_i^+(\boldsymbol{w}), l_i^-(\boldsymbol{w}), \gamma_i^+(\boldsymbol{w}, \boldsymbol{v})$, and $\gamma_i^-(\boldsymbol{w}, \boldsymbol{v})$ is related to Joachims (2005) and is a special case of a direct counting method discussed in the supplementary material. For the general situation, they observe that

$$\sum_{(i,j)\in P} \max\left(0, 1 - \boldsymbol{w}^T(\boldsymbol{x}_i - \boldsymbol{x}_j)\right)^2 = \sum_{r\in K} \sum_{\substack{(i,j)\in P \\ y_i > r, y_j = r}} \max\left(0, 1 - \boldsymbol{w}^T(\boldsymbol{x}_i - \boldsymbol{x}_j)\right)^2.$$

The inner sum is for a subset of data in two relevance levels (r and $> r$). Then the algorithm for two-level data can be applied. When we replace the $O(lk)$ term in equation 2.14 with $O(\text{size of each two-level set})$, the complexity of each matrix-vector product is

$$O(l\bar{n} + n) + \sum_{r\in K} O(|\{(i, j) \mid (i, j) \in P, y_i > r, y_j = r\}|). \tag{3.1}$$

If each relevance level takes about the same amount of $O(l/k)$ data, equation 3.1 becomes

$$O(l\bar{n}+n)+\sum_{m=2}^{k} O\left(\frac{lm}{k}\right)=O(l\bar{n}+lk+n), \tag{3.2}$$

which is larger than the approach of using order-statistic trees.

3.2 TreeRankSVM. Joachims (2006) uses a cutting plane method to optimize problem 1.2. Airola et al. (2011) improve on Joachims's work and release a package TreeRankSVM.

Here we follow Teo et al. (2010, section 2) to describe the cutting plane method for minimizing a function

$$\frac{1}{2}\boldsymbol{w}^T\boldsymbol{w}+L(\boldsymbol{w}),$$

where $L(\boldsymbol{w})$ is the loss term. Let $\boldsymbol{w}^t$ be the solution obtained at the tth iteration. The first-order Taylor approximation of $L(\boldsymbol{w})$ is used to build a cutting plane $\boldsymbol{a}_t^T\boldsymbol{w}+b_t$ at $\boldsymbol{w}=\boldsymbol{w}^t$,

$$\begin{aligned} L(\boldsymbol{w}) &\geq \nabla L(\boldsymbol{w}^t)^T(\boldsymbol{w}-\boldsymbol{w}^t)+L(\boldsymbol{w}^t)\\ &= \boldsymbol{a}_t^T\boldsymbol{w}+b_t,\ \forall \boldsymbol{w}, \end{aligned}$$

where

$$\boldsymbol{a}_t \equiv \nabla L(\boldsymbol{w}^t) \quad \text{and} \quad b_t \equiv L(\boldsymbol{w}^t)-\boldsymbol{a}_t^T\boldsymbol{w}^t.$$

If $L(\boldsymbol{w})$ is nondifferentiable, then a subgradient is used for $\boldsymbol{a}_t$. The cutting plane method maintains $\boldsymbol{a}_m, b_m,\ m=1,\ldots,t$ to form a lower-bound function for $L(\boldsymbol{w})$,

$$L_t^{CP}(\boldsymbol{w}) \equiv \max_{1\leq m\leq t}(\boldsymbol{a}_m^T\boldsymbol{w}+b_m),$$

and obtains $\boldsymbol{w}^{t+1}$ by solving

$$\boldsymbol{w}^{t+1}=\arg\min_{\boldsymbol{w}} \quad \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w}+L_t^{CP}(\boldsymbol{w}). \tag{3.3}$$

For rankSVM, if problem 1.2 is considered, a subgradient of its loss term is

$$\nabla^s\Big(C\sum_{(i,j)\in P,1-\boldsymbol{w}^T(\boldsymbol{x}_i-\boldsymbol{x}_j)>0}\big(1-\boldsymbol{w}^T(\boldsymbol{x}_i-\boldsymbol{x}_j)\big)\Big)=C\sum_{(i,j)\in P,1-\boldsymbol{w}^T(\boldsymbol{x}_i-\boldsymbol{x}_j)>0}(\boldsymbol{x}_i-\boldsymbol{x}_j)$$

$$=C\sum_{i=1}^{l}\big(l_i^+(\boldsymbol{w})-l_i^-(\boldsymbol{w})\big)\boldsymbol{x}_i. \quad (3.4)$$

The function value also needs to be evaluated during the optimization procedure.

$$\frac{1}{2}\boldsymbol{w}^T\boldsymbol{w}+C\sum_{(i,j)\in P,1-\boldsymbol{w}^T(\boldsymbol{x}_i-\boldsymbol{x}_j)>0}\big(1-\boldsymbol{w}^T(\boldsymbol{x}_i-\boldsymbol{x}_j)\big)$$

$$=\frac{1}{2}\boldsymbol{w}^T\boldsymbol{w}+C\sum_{i=1}^{l}\big(l_i^+(\boldsymbol{w})-l_i^-(\boldsymbol{w})\big)\boldsymbol{w}^T\boldsymbol{x}_i.$$

For obtaining $l_i^+(\boldsymbol{w})$ and $l_i^-(\boldsymbol{w})$, Joachims (2006) uses a direct counting method; the complexity at each iteration is shown in equation 1.6. We leave the details to the supplementary materials. As mentioned in section 2.3, the main improvement Airola et al. (2011) made is to use order-statistic trees so that the $O(lk)$ term in calculating $l_i^+(\boldsymbol{w})$ and $l_i^-(\boldsymbol{w})$, $\forall i$ is reduced to $O(l\log k)$. In particular, red-black trees were adopted in their work. The overall cost is

$$\big(O(l\log l+l\bar{n}+l\log k+n)+\text{cost of equation 3.3})\times\#\text{iterations}.$$

3.3 sofia-ml. Sculley (2009) proposed sofia-ml to solve problem 1.2. It is a stochastic gradient descent (SGD) method that randomly draws a preference pair from the training set at each iteration and uses a subgradient on this pair to update $\boldsymbol{w}$. This method does not consider the special structure of the loss term. For going through the whole training data, the cost is $O(p\bar{n})$, which is worse than other methods discussed. Therefore, we do not include this method in our experiments.

In contrast, SGD is one of the state-of-the-art methods for linear SVM(e.g., Shalev-Shwartz, Singer, & Srebro, 2007.) The main reason is that special methods to consider the structure of the loss term have not been available.

4 Experiments

In this section, we begin by describing the details of a truncated Newton implementation of our approach. The first experiment is to evaluate

methods discussed in section 2. In particular, the speed of different implementations of order-statistic trees is examined. Next, we compare state-of-the-art methods for linear rankSVM with the proposed approach. Then we conduct an investigation of the performance difference between linear rankSVM and pointwise methods. Finally, an experiment on sparse data is shown. (Programs used for experiments can be found online at http://www.csie.ntu.edu.tw/∼cjlin/liblinear/exp.html.)

4.1 Implementation Using a Trust Region Newton Method. In our implementation of the proposed approach, we consider a trust region Newton method (TRON), that is, a truncated Newton method discussed in section 2.1. For details of trust region methods, a comprehensive book is by Conn, Gould, and Toint (2000). Here we mainly follow the setting in Lin and Moré (1999) and Lin et al. (2008).

TRON adopts the trust region technique to obtain the truncated Newton step. It finds the direction $\boldsymbol{s}$ by minimizing $g_t(\boldsymbol{s})$ in equation 2.1 over a region that we trust:

$$\min_{\boldsymbol{s}} \quad g_t(\boldsymbol{s}) \qquad \text{subject to} \quad \|\boldsymbol{s}\| \le \Delta_t, \tag{4.1}$$

where Δ_t is the size of the trust region. After solving problem 4.1, it decides whether to apply the obtained direction $\boldsymbol{s}^t$ according to the approximate function reduction $g_t(\boldsymbol{s})$ and the real function decrease, that is,

$$\boldsymbol{w}^{t+1} \to \begin{cases} \boldsymbol{w}^t, & \text{if } \rho_{\mathrm{k}} < \eta_0, \\ \boldsymbol{w}^t + \boldsymbol{s} & \text{if } \rho_{\mathrm{k}} \ge \eta_0, \end{cases} \tag{4.2}$$

where $\eta_0 \ge 0$ is a prespecified parameter and

$$\rho_k \equiv \frac{f(\boldsymbol{w}^t + \boldsymbol{s}^t) - f(\boldsymbol{w}^t)}{g_t(\boldsymbol{s}^t)}.$$

TRON adjusts the trust region Δ_t according to ρ_k. When it is too small, Δ_t is decreased. Otherwise, while ρ_k is large enough, TRON increases the value of Δ_t. More specifically, the following rule is considered:

$$\begin{aligned} &\Delta_{t+1} \in [\sigma_1 \min(\|\boldsymbol{s}^t\|, \Delta_t), \sigma_2 \Delta_t] \text{ if } \rho_k \le \eta_1, \\ &\Delta_{t+1} \in [\sigma_1 \Delta_t, \sigma_3 \Delta_t] \text{ if } \rho_k \in (\eta_1, \eta_2), \\ &\Delta_{t+1} \in [\Delta_t, \sigma_3 \Delta_t] \text{ if } \rho_k \ge \eta_2, \end{aligned} \tag{4.3}$$

where

$$\eta_0 = 10^{-4}, \eta_1 = 0.25, \eta_2 = 0.75,$$
$$\sigma_1 = 0.25, \sigma_2 = 0.5, \sigma_3 = 4.0.$$

For linear classification, Lin et al. (2008) apply the approach of Steihaug (1983) to run CG iterations until either a minimum of $g_t(\boldsymbol{s})$ is found or $\boldsymbol{s}$ touches the boundary of the trust region. We consider the same setting in our implementation.

Regarding the stopping condition, we follow that of TRON in the package LIBLINEAR (Fan, Chang, Hsieh, Wang, & Lin, 2008) to check if the gradient is relatively smaller than the initial gradient:

$$\|\nabla f(\boldsymbol{w}^k)\|_2 \leq \epsilon_s \|\nabla f(\boldsymbol{w}^0)\|_2, \tag{4.4}$$

where $\boldsymbol{w}^0$ is the initial iterate and ϵ_s is the stopping tolerance given by users. By default, we set $\epsilon_s = 10^{-3}$ and let $\boldsymbol{w}^0$ be the zero vector.

4.2 Experiment Setting. We consider three sources of web search engine ranking: LETOR 4.0 (Qin, Liu, Xu, & Li, 2010), MSLR,[3] and YAHOO LTRC (Chapelle & Chang, 2011). Both LETOR 4.0 and MSLR are from Microsoft Research, while YAHOO LTRC is from the Yahoo Learning to Rank Challenge. From LETOR 4.0, we take four sets: MQ2007, MQ2008, MQ2007-list, and MQ2008-list. For MSLR, we take the set with name 30k, which indicates the number of queries within it.[4] Each set from LETOR 4.0 or MSLR consists of five segmentations, and we take the first fold. YAHOO LTRC contains two sets, and both are considered. The details of these data sets are listed in Table 2. Each set comes with training, validation, and testing sets; we use the validation set only for selecting the parameters of each model. For preprocessing, we linearly scale each feature of YAHOO LTRC and MSLR data sets to the range $[0, 1]$, while the features of LETOR 4.0 data sets are already in this range.

All experiments are conducted on a 64-bit machine with Intel Xeon 2.5 GHz CPU (E5504), 12 MB cache, and 16 GB memory.

4.3 A Comparison Between Methods in Section 2: A Direct Counting Method and Different Order-Statistic Trees. We solve problem 1.3 using TRON and compare the following methods for calculating $l_i^+(\boldsymbol{w})$, $l_i^-(\boldsymbol{w})$, $\gamma_i^+(\boldsymbol{w}, \boldsymbol{v})$ and $\gamma_i^-(\boldsymbol{w}, \boldsymbol{v})$:

[3] http://research.microsoft.com/en-us/projects/mslr/.

[4] The number of queries shown in Table 2 is fewer because we report only the training set statistics.

Table 2: Statistics of Training Data Sets.

Data Set	l	n	k	$\|Q\|$	p	Average k_q/l_q over Queries
MQ2007 fold 1	42,158	46	3	1,017	246,015	0.0546
MQ2008 fold 1	9,630	46	3	471	52,325	0.1697
MSLR 30k fold 1	2,270,296	136	5	18,919	101,312,036	0.0492
YAHOO LTRC set 1	473,134	519	5	19,944	5,178,545	0.2228
YAHOO LTRC set 2	34,815	596	5	1,266	292,951	0.1560
MQ2007-list fold 1	743,790	46	1,268	1,017	285,943,893	1
MQ2008-list fold 1	540,679	46	1,831	471	323,151,792	1

Notes: All data sets are dense (i.e., $\bar{n} = n$). In the last column, l_q and k_q are the number of instances and the number of relevance levels in query q, respectively. See Table 1 for the meaning of other columns.

- direct-count: the direct counting method mentioned at the end of section 2.2 (see details in the supplementary material)
- y-rbtree: the red-black tree using y_i as the key of nodes (see section 2.3)
- $w^T x$-rbtree: the red-black tree using $\boldsymbol{w}^T \boldsymbol{x}_i$ as the key of nodes (see section 2.3)
- selectiontree: the selection tree that stores keys in leaf nodes (see section 2.4)
- y-avltree: the same as y-rbtree, except the order-statistic tree used is the AVL tree (see section 2.5)
- y-aatree: the same as y-rbtree, except the order-statistic tree used is the AA tree (see section 2.5)

The trust region Newton method, written in C/C++, is extended from the implementation in LIBLINEAR. To compare the convergence speed, we investigate the relative difference to the optimal function value,

$$\left| \frac{f(\boldsymbol{w}) - f(\boldsymbol{w}^*)}{f(\boldsymbol{w}^*)} \right|,$$

where $\boldsymbol{w}^*$ is the optimum of problem 1.3. We run the optimization algorithm long enough to obtain an approximation of $f(\boldsymbol{w}^*)$.

We take four data sets and set $C = 1$. The results of training time versus function values are shown in Figure 4. We also draw a horizontal line in the figure to indicate that the default stopping condition in TRON of using $\epsilon_s = 10^{-3}$ in equation 4.4 has been satisfied. Experiments in section 4.4 will show that solutions obtained below this line have similar ranking performance to the optimum. From the figures, the method of direct counting is slow when k (number of relevance levels) is large. This result is expected

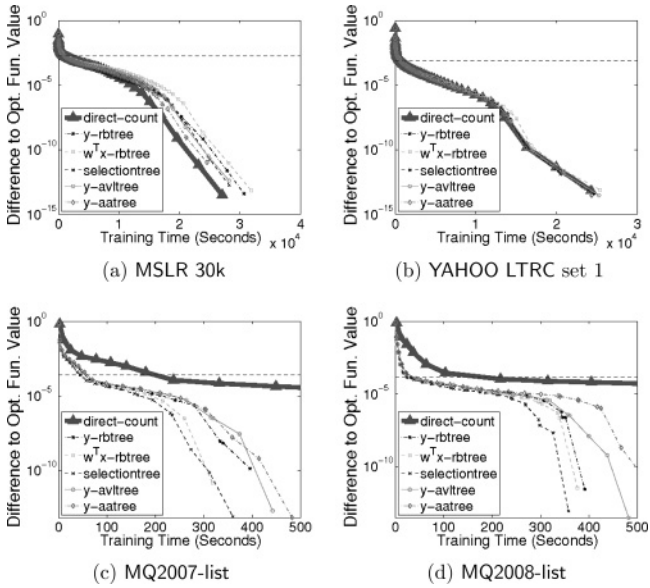


Figure 4: Comparison between different order-statistic tree implementations and the direct counting method. We present training time and relative difference to the optimal function value. $C = 1$ is set for all four data sets. The dotted horizontal line indicates the function value of TRON using the default stopping tolerance $\epsilon_s = 0.001$ in equation 4.4.

following the complexity analysis in equation 2.14. In addition, although implementations of order-statistic trees have slightly different running times in the end, they are very similar otherwise. Therefore, we choose selection trees in subsequent experiments because of the simplicity.

4.4 A Comparison Between Different Methods for Linear RankSVM. We compare the following methods for linear rankSVM:

- Tree-TRON. Our approach of using TRON with selection trees.
- PRSVM+ (Chapelle & Keerthi, 2010). This method was discussed in section 3.1. The authors did not release their code, so we make an implementation using the same framework of Tree-TRON. Therefore we apply trust region rather than line search in their truncated Newton procedure.
- TreeRankSVM (Airola et al., 2011). This method was discussed in section 3.2. We download version 0.1 from http://staff.cs.utu.fi/~aatapa/software/RankSVM/. Although this package is mainly implemented in Python, computationally intensive procedures such as

red-black trees and the minimization of equation 3.3 are written in C/C++ or Fortran.

Note that TreeRankSVM solves L1-loss rankSVM, but the other two consider L2 loss. Therefore they have different optimal function values. We separately obtain their own optima and compute the corresponding relative difference to the optimal function value. For prediction performance, we first check normalized discounted cumulative gain (NDCG), which is widely used for comparing ranked lists of information retrieval tasks (Järvelin & Kekäläinen, 2002). Several definitions of NDCG are available, so we follow the recommendation of each data source. Assume m is a prespecified positive integer, π is an ideal ordering with

$$y_{\pi(1)} \geq y_{\pi(2)} \geq \cdots \geq y_{\pi(l_q)}, \ \forall q \in Q,$$

and π' is the ordering being evaluated, where l_q is the number of instances in query q. Then

$$\text{NDCG@}m \equiv (N_m)^{-1} \sum_{i=1}^{m} (2^{y_{\pi'(i)}} - 1) d(i), \tag{4.5}$$

where

$$N_m = \sum_{i=1}^{m} (2^{y_{\pi(i)}} - 1) d(i) \quad \text{and}$$

$$d(i) = \begin{cases} \dfrac{1}{\log_2(\max(2, i))} & \text{for MSLR and LETOR 4.0,} \\ \dfrac{1}{\log_2(i+1)} & \text{for YAHOO LTRC.} \end{cases}$$

N_m is the score of an ideal ordering, where top-ranked instances are considered more important because of larger $(2^{y_{\pi(i)}} - 1) d(i)$. From equation 4.5, NDCG computes the relative score of the evaluated ordering to the ideal ordering. Regarding m, YAHOO LTRC considers

$$m = \min(10, l_q).$$

For MSLR and LETOR 4.0, we follow their recommendation to use mean NDCG:

$$\text{Mean NDCG} \equiv \frac{\sum_{i=1}^{l_q} \text{NDCG@}i}{l_q}.$$

We then report the average over all queries.

Table 3: Best Regularization Parameter for Each Data Set and Each Measurement.

	Problem 1.2 Using L1 Loss		Problem 1.3 Using L2 Loss	
Data Sets	Pairwise Accuracy	NDCG	Pairwise Accuracy	NDCG
MQ2007	2^{-1}	2^{8}	2^{-5}	2^{-15}
MQ2008	2^{8}	2^{-6}	2^{7}	2^{7}
MSLR 30k	NA	NA	2^{3}	2^{3}
YAHOO LTRC set 1	NA	NA	2^{-14}	2^{1}
YAHOO LTRC set 2	2^{-7}	2^{-4}	2^{-10}	2^{-10}
MQ2007-list	2^{5}	NA	2^{-12}	NA
MQ2008-list	2^{-14}	NA	2^{-14}	NA

Notes: When problem 1.2 with L1 loss is used, TreeRankSVM failed to finish the parameter selection procedure on MSLR 30k and YAHOO LTRC set 1 after a long running time. NDCG cannot be used for MQ2007-list and MQ2008-list because the large k leads to the overflow of $2^{y_{\pi'(i)}}$ in equation 4.5.

We further consider pairwise accuracy as a measurement because it is directly related to the loss term of rankSVM. Pairwise accuracy is widely adopted in areas such as statistics and medical data analysis, but with the name concordance index or Kendall's τ (Kendall, 1938):

$$\text{Pairwise accuracy} \equiv \frac{|\{(i, j) \mid (i, j) \in P,\ \boldsymbol{w}^T \boldsymbol{x}_i > \boldsymbol{w}^T \boldsymbol{x}_j\}|}{p}.$$

We adopt the algorithm of Christensen (2005) that uses an AVL tree to compute pairwise accuracy in $O(l \log l)$ time, but our implementation uses a selection tree.[5]

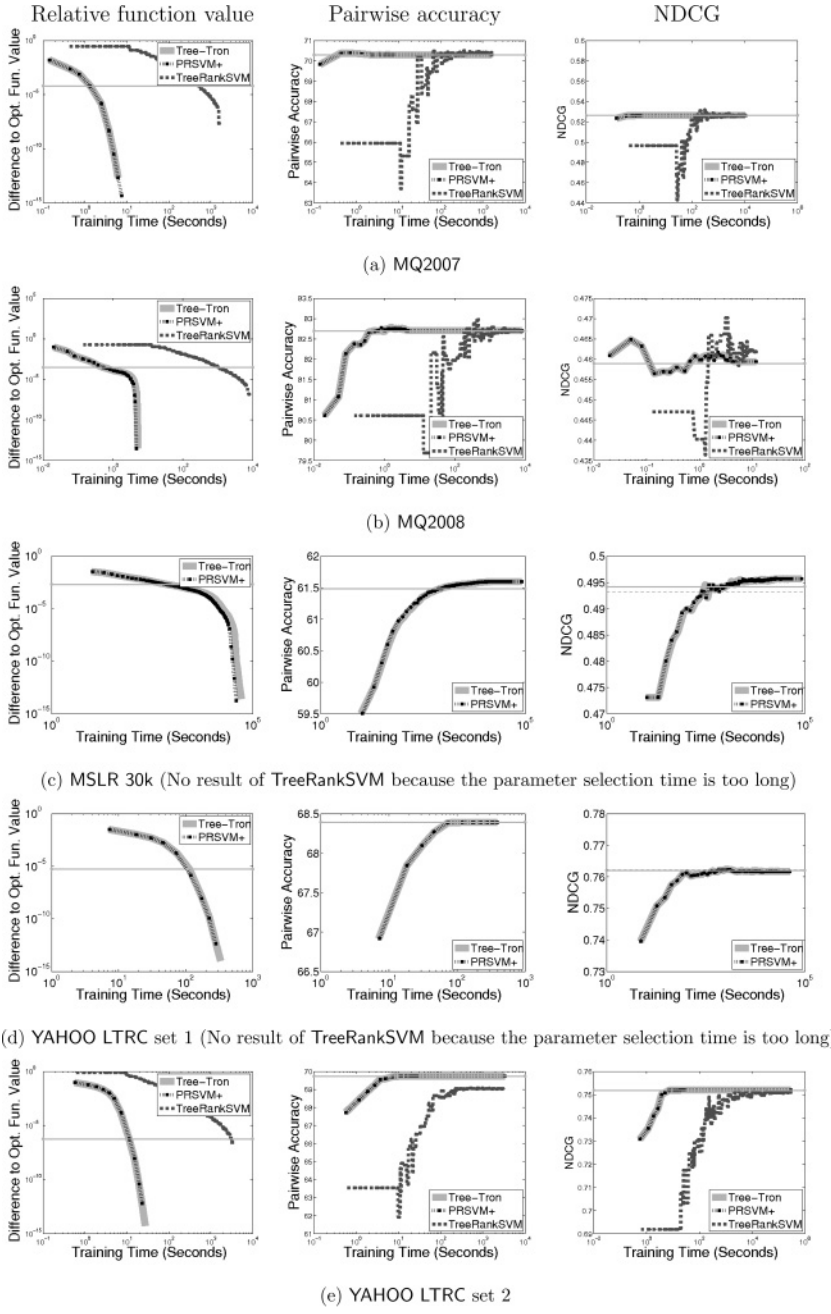
For each evaluation criterion, we find the best regularization parameter by checking the validation set result of $C \in \{2^{-15}, 2^{-14}, \ldots, 2^{10}\}$.[6] The selected regularization parameter C for each data set and each measurement is listed in Table 3. The results of comparing different approaches can be found in Figures 5 and 6. We present the relative difference to the optimal function value, pairwise accuracy, and (mean) NDCG.[7] We also draw the horizontal lines of the default stopping condition of Tree-TRON in the figures.

One could observe from the figures that the convergence speed of TreeRankSVM is slower than PRSVM+ and Tree-TRON. To rule out the

[5]Here l represents the number of testing data.

[6]In the implementation of TreeRankSVM, the formulation is scaled so the regularization parameter is $\lambda = 1/(Cp)$.

[7]For the function value, parameters selected using (validation) pairwise accuracy are considered, but results of using NDCG are similar.



(a) MQ2007

(b) MQ2008

(c) MSLR 30k (No result of TreeRankSVM because the parameter selection time is too long)

(d) YAHOO LTRC set 1 (No result of TreeRankSVM because the parameter selection time is too long)

(e) YAHOO LTRC set 2

Figure 5: A comparison between different linear rankSVM methods on function values, pairwise accuracy, and NDCG. The x-axis is in log scale.

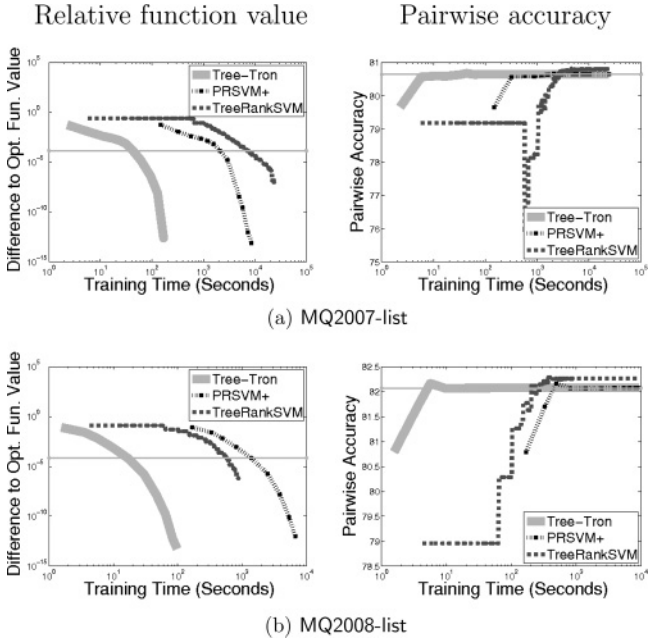


Figure 6: A comparison between different linear rankSVM methods on function values and pairwise accuracy. The two sets MQ2007-list MQ2008-list have large k (number of relevance levels). NDCG cannot be used because of the overflow of $2^{y_{\pi'(i)}}$ in equation 4.5. The x-axis is in log scale.

implementation differences between Tree-TRON/PRSVM+ and TreeRankSVM, in the supplementary materials, we check iterations versus relative function value and test performance.[8] Results still show that TreeRankSVM is slower, so for linear rankSVM, methods using second-order information seem to be superior. Regarding Tree-TRON and PRSVM+, Figure 5 shows that they are similar when the average k_q/l_q is small. However, from Figure 6, PRSVM+ is much slower if the number of preference levels is large (i.e., large k_q/l_q). This result is expected following the complexity analysis in equations 2.23 and 3.2. Another observation is that the performances of PRSVM+ and Tree-TRON are stable after the default stopping condition is satisfied. Thus, tuning ϵ_s in equation 4.4 is generally not necessary.

This experiment also serves as a comparison between L1- and L2-loss linear rankSVM. Results show that their performances (NDCG and pairwise accuracy) are similar.

[8]For Tree-TRON/PRSVM+, we use CG iterations rather than outer Newton iterations because each CG has a similar complexity to that of a cutting plane iteration.

Instead of using the best C after parameter selection, we investigate the training time under a fixed C for all methods. We check the situations that C is large, medium, and small by using $C = 100$, $C = 1$, and $C = 10^{-4}$, respectively. The results are shown in Figure 7. We can observe that Tree-TRON is always one of the fastest methods.

4.5 A Comparison Between Linear RankSVM, Linear Support Vector Regression, GBDT, and Random Forests. We compare rankSVM using Tree-TRON with the following pointwise methods:

- Linear support vector regression (SVR) by Vapnik (1995). We check both L1-loss and L2-loss linear SVR provided in the package LIBLINEAR (version 1.92). Their implementation details can be found in Ho and Lin (2012). For L2-loss linear SVR, two implementations are available in LIBLINEAR by solving primal and dual problems, respectively. We use the one that solves the primal problem by TRON.
- GBDT (Friedman, 2001). This is a nonlinear pointwise model that is known to be powerful for web search ranking problems. We use version 0.9 of the package pGBRT (Tyree, Weinberger, Agrawal, & Paykin, 2011) downloaded from http://machinelearning.wustl.edu/pmwiki.php/Main/Pgbrt.
- Random forests (Breiman, 2001). This is another nonlinear pointwise model that performs well on web search data. We use version 1.0 of Rt-Rank downloaded from https://sites.google.com/site/rtranking.

For linear SVR, we set the ϵ-insensitive parameter $\epsilon = 0$ because Ho and Lin (2012) showed that this setting often works well. We then conduct the same parameter selection procedure as in section 4.4 to find the best regularization parameters C and list them in Table 4. The training time, test NDCG, and test pairwise accuracy are shown in Table 5. We first observe that the performance of L1-loss SVR is worse than L2-loss SVR. The reason might be that L1 loss imposes a smaller training loss when the prediction error is larger than 1. Regarding L2-loss SVR and L2-loss rankSVM, their NDCG results are close, but rankSVM gives better pairwise accuracy. This result seems to be reasonable because rankSVM considers pairwise training losses. For training time, although the selected regularization parameters are different and hence results are not fully comparable, in general L2-loss SVR is faster. In summary, L2-loss SVR is competitive in terms of NDCG and training time, but rankSVM may still be useful if pairwise accuracy is what we are concerned about.

Next, we check GBDT and random forests. Their training time is long, so we do not conduct parameter selection. We consider a small number of trees and fix the parameters as follows. For GBDT, we use learning rate $= 0.1$, tree depth $= 4$, and number of trees $= 100$. For random forests, we use the number of sampled features for splitting in each node $= \lfloor \sqrt{n} \rfloor$ and number of trees $= 40$. We further use eight cores to reduce the training time. The

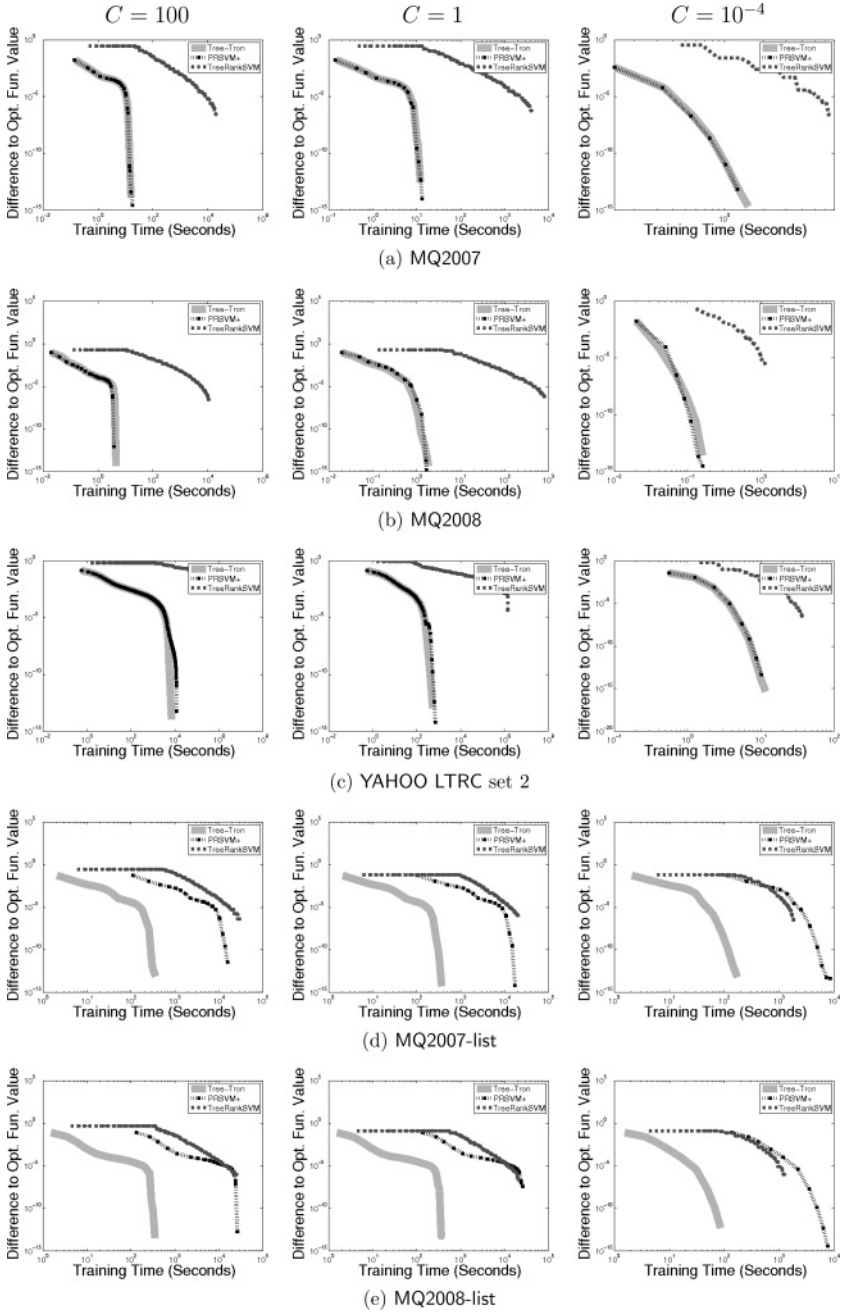


Figure 7: A comparison between different linear rankSVM methods on relative function value with the same C. We set C to be 100, 1 and 10^{-4}.

Table 4: Best Regularization Parameter for Each Data Set and Each Measurement of SVR.

	L1-Loss Linear SVR		L2-Loss Linear SVR	
Data Set	Pairwise Accuracy	NDCG	Pairwise Accuracy	NDCG
MQ2007	2^{8}	2^{8}	2^{-7}	2^{-11}
MQ2008	2^{8}	2^{8}	2^{-1}	2^{-4}
MSLR 30k	2^{-1}	2^{2}	2^{-2}	2^{-2}
YAHOO LTRC set 1	2^{-10}	2^{-5}	2^{-5}	2^{-2}
YAHOO LTRC set 2	2^{-3}	2^{1}	2^{-5}	2^{4}
MQ2007-list	2^{-9}	NA	2^{-15}	NA
MQ2008-list	2^{4}	NA	2^{-7}	NA

Table 5: Comparison Between rankSVM and SVR.

	L2-Loss RankSVM		L1-Loss SVR		L2-Loss SVR	
Data Set	Training Time (s)	NDCG	Training Time (s)	NDCG	Training Time (s)	NDCG
MQ2007	0.5	0.5211	23.9[a]	0.4756[a]	0.5	0.5157
MQ2008	0.5	0.4571	3.4[a]	0.4153[a]	0.2	0.4450
MSLR 30k	1601.6	0.4945	461.6	0.4742	202.4	0.4946
YAHOO LTRC set 1	334.8	0.7616	10.8	0.7579	172.7	0.7642
YAHOO LTRC set 2	11.2	0.7519	47.6	0.7470	20.8	0.7578
	Training Time (s)	Pairwise Accuracy	Training Time (s)	Pairwise Accuracy	Training Time (s)	Pairwise Accuracy
MQ2007	1.3	70.36%	23.9[a]	64.06%[a]	0.7	68.56%
MQ2008	0.5	82.70%	3.4[a]	77.72%[a]	0.3	82.17%
MSLR 30k	1601.6	61.52%	65.4	60.11%	202.4	60.49%
YAHOO LTRC set 1	117.1	68.45%	2.4	67.82%	149.5	67.83%
YAHOO LTRC set 2	11.2	69.74%	3.3	68.37%	14.5	69.39%
MQ2007-list	38.7	80.71%	1.0	79.82%	5.0	79.70%
MQ2008-list	16.6	82.11%	1.1	81.65%	6.7	81.85%

[a]Reached maximum iteration of LIBLINEAR.

results are shown in Table 6. For the smaller data sets MQ2007, MQ2008 and YAHOO LTRC set 2, we are able to train more trees in a reasonable time, so we present in Table 7 the result of using 1000 trees.

From Tables 6 and 7, GBDT and random forests generally perform well, though they are not always better than linear rankSVM. For YAHOO LTRC set 2, random forests achieves 0.78 NDCG using 1000 trees, which is much better than 0.75 of linear rankSVM. This result is consistent with the fact that in the Yahoo Learning to Rank Challenge, all top performers use

Table 6: Performance of GBDT and Random Forests with a Small Number of Trees.

	Random Forests			GBDT		
Data Set	Training Time (s)	Pairwise Accuracy	NDCG	Training Time (s)	Pairwise Accuracy	NDCG
MQ2007	14.8	66.16%	0.4959	1.4	69.78%	0.5182
MQ2008	2.3	80.36%	0.4541	0.4	82.83%	0.4706
MSLR 30k	5102.1	63.76%	0.5598	1339.3	62.77%	0.5375
YAHOO LTRC set 1	1672.2	70.69%	0.7797	557.7	69.22%	0.7707
YAHOO LTRC set 2	58.7	68.76%	0.7629	11.3	71.21%	0.7711
MQ2007-list	606.0	78.78%	NA	106.8	79.85%	NA
MQ2008-list	423.3	82.04%	NA	59.3	82.43%	NA

Note: Random forests: 40 trees; GBDT: 100 trees.

Table 7: Performance of GBDT and Random Forests with 1000 Trees.

	Random Forests			GBDT		
Data Set	Training Time (s)	Pairwise Accuracy	NDCG	Training Time (s)	Pairwise Accuracy	NDCG
MQ2007	345.3	69.07%	0.5221	13.7	67.58%	0.4892
MQ2008	52.0	82.60%	0.4675	3.6	79.78%	0.4491
YAHOO LTRC set 2	1406.9	71.91%	0.7801	108.7	71.70%	0.7720

decision-tree-based methods. However, the training cost of GBDT and random forests is in general higher than linear rankSVM. Therefore, linear rankSVM is useful to quickly provide a baseline result. We also note that the performance of GBDT with more trees is not always better than with few trees. This result seems to indicate that overfitting occurs and parameter selection is important. In contrast, random forests is more robust. The training time of GBDT is faster than random forests because in random forests, the tree depth is unlimited while we restrict the depth of trees in GBDT to be four.

Although pointwise methods perform well in this experiment, a potential problem is that they do not consider different queries. It is unclear if this situation may cause problems.

4.6 A Comparison Between Linear and Nonlinear Models on Sparse Data. Recent research has shown that linear SVM is competitive with nonlinear SVM on classifying large and sparse data (Yuan et al., 2012). We conduct an experiment to check if this property also holds for learning to rank. We consider rankSVM as the linear model for comparison, but for the nonlinear model, we use random forests rather than kernel rankSVM. One

Table 8: Statistics of Sparse Training Data Sets.

Data Set	l	n	$\bar{n}$	k	p
CTR	11,382,195	22,510,600	22.6	93,899	46,191,724,381,879
KDD2012b	68,019,906	79,901,700	35.3	6,896	198,474,800,029,148
CTR (0.1%)	11,382	73,581	22.6	1,087	46,020,848
KDD2012b (0.025%)	17,005	74,026	35.3	26	12,704,393

Note: To reduce the training time, only a small subset of each problem is used.

Table 9: Performance of Linear RankSvm and Random Forests on Sparse Data.

	Linear RankSvm			Random Forests		
Data Set	Training Time (s)	Pairwise Accuracy	Mean NDCG	Training Time (s)	Pairwise Accuracy	Mean NDCG
CTR (0.1%)	4.3	60.83%	0.4822	6343.3	60.45%	0.4732
KDD2012b (0.025%)	2.7	68.16%	0.5851	5223.1	69.72%	0.5982

Note: Random forests uses 40 trees.

reason is that random forests is very robust in the previous experiment. We consider the following two CTR (click through rate) estimation problems, which can be treated as regression or ranking problems:

- CTR. This is a data set used in Ho and Lin (2012).
- KDD2012b. This is the processed data generated by the winning team (Wu et al., 2012) of KDD Cup 2012 track 2 (Niu et al., 2012). It contains about one-third of the original data. The task of this competition is online advertisement ranking evaluated by AUC, while the labels are number of clicks and number of views. Note that pairwise accuracy is reduced to AUC when $k = 2$. We transform the labels into CTR (i.e., number of clicks over number of views).

The two data sets both contain a single query, and each comes with training and testing sets. To reduce the training time and the memory cost of random forests, we sample from the two data sets and condense the features. The details are listed in Table 8. We use the same parameters of random forests as in section 4.5. For a fair comparison, we fix $C = 1$ for rankSVM because the parameters of random forests are not well tuned. The results are shown in Table 9. We first notice the training time of random forests is several thousand times more than linear rankSVM on sparse data. The difference is larger than the case of dense data because the training cost of random forests is linear to the number of features, but that of rankSVM is linear to the average number of nonzero features. Regarding the performance, the difference is small for the two data sets, so linear rankSVM is

Table 10: A Comparison Between Using Partial Pairs and All Pairs to Train a Model.

	MQ2007-list			MQ2008-list		
	C	Training Time (s)	Pairwise Accuracy	C	Training Time (s)	Pairwise Accuracy
Partial pairs	2^{-9}	19.9	79.10%	2^{-10}	10.5	81.81%
All pairs	2^{-12}	38.7	80.71%	2^{-14}	16.6	82.11%

Note: LIBLINEAR using TRON for L2-loss SVM is used for the partial-pair setting, while Tree-TRON is used for all pairs.

very useful to get competitive results quickly. However, more experiments are needed to confirm these preliminary observations. We hope more public sparse ranking data will be available in the near future.

5 Using Partial Pairs to Train Models

To avoid considering the $O(l^2)$ pairs, a common practice in ranking is to use only a subset of pairs. An example is Lin (2010), which uses pairs with close relevance levels (i.e., y_i close to y_j). The concept is similar to equation 1.5: if pairs with close relevance levels are ranked with the right order, those pairs with larger distances should also be ranked correctly. When $k = O(l)$, this approach can reduce the number of pairs from $O(l^2)$ to be as small as $O(k) = O(l)$. However, if k is small, each pair is already formed by instances in two close relevance levels, so we cannot significantly reduce the number of pairs.

We take MQ2007-list and MQ2008-list to conduct experiments because these two data sets possess the property $k_q = l_q,\ \forall q \in Q$. Because in each q, the values of y_i are $1, \ldots, l_q$, we use the pairs $(i, j) \in P$ with $y_i = y_j + 1$. This setting of using two adjacent relevance levels leads to $O(l)$ pairs. Then we can directly consider problems 1.2 and 1.3 as classification problems with instances $\boldsymbol{x}_i - \boldsymbol{x}_j$. If Newton methods are considered for solving problem 1.3, by the approach in equation 2.6, each Hessian-vector product costs only $O(l\bar{n} + l + n)$. Therefore, we directly use the TRON implementation to solve L2-loss SVM in LIBLINEAR without applying any special method in section 2. After selecting the parameter C, we present pairwise accuracy in Table 10. It is observed that the selected C of using partial pairs is larger than that of using all pairs. This situation occurs because the sum of training losses in problem 1.3 on a smaller number of pairs must be penalized by a larger C. For training time and pairwise accuracy, it is as expected that the approach of using partial pairs slightly sacrifices the performance for faster training speed.

Because of the only slightly lower pairwise accuracy, we may say that this approach, together with past work, is already enough to train large-scale linear rankSVM:

- If k is small, we can apply the direct method mentioned in section 2.2 that has an $O(lk)$ term for calculating $l_i^+(\boldsymbol{w})$, $l_i^-(\boldsymbol{w})$, $\gamma_i^+(\boldsymbol{w}, \boldsymbol{v})$, and $\gamma_i^-(\boldsymbol{w}, \boldsymbol{v})$.
- If k is large, we can use only $O(l)$ pairs. Then any efficient methods to train linear SVM can be applied.

However, a caveat is that two different implementations must be used. In contrast, methods of using order-statistic trees can simultaneously handle situations of small and large k.

6 Conclusion

In this letter, we systematically reviewed recent approaches for linear rankSVM. We show that regardless of the optimization methods used, the computational bottleneck is on calculating some values over all preference pairs. Following Airola et al. (2011), we comprehensively investigate tree-based techniques for the calculation. Experiments show that our method is faster than existing implementations for linear rankSVM.

Based on this study, we release an extension of the popular linear classification/regression package LIBLINEAR for ranking. It is available online at http://www.csie.ntu.edu.tw/∼cjlin/libsvmtools/.

Appendix: The Dual Problem of Problems 1.1 and 1.2

Based on the original training data, we can construct a new set $\{y_{i,j}, \boldsymbol{x}_{i,j}\}\ \forall(i, j) \in P$ with

$$\boldsymbol{x}_{i,j} = \boldsymbol{x}_i - \boldsymbol{x}_j, \quad \text{and} \quad y_{i,j} = 1,\ \forall(i, j) \in P.$$

Using this training data set, problems 1.2 and 1.3 can be viewed as L1-loss and L2-loss SVM problems with only one class of data, respectively. Then the dual problems of problems 1.2 and 1.3 are both in the following form,

$$\begin{aligned} \min_{\boldsymbol{\alpha}} \quad & \tfrac{1}{2}\boldsymbol{\alpha}^T\bar{Q}\boldsymbol{\alpha} - \boldsymbol{e}^T\boldsymbol{\alpha} \\ \text{subject to} \quad & 0 \le \alpha_{i,j} \le U, \forall(i, j) \in P, \end{aligned}$$

where $\bar{Q} = Q + D$, D is a diagonal matrix and $Q = AX(AX)^T$. For L1 loss, $U = C$ and D is the zero matrix. For L2 loss, $U = \infty$ and $D = I/(2C)$, where I is the p by p identity matrix.

Notice that the KKT condition for L2-loss SVM gives

$$\max(0, 1 - \boldsymbol{w}^T \boldsymbol{x}_{i,j}) = \frac{\alpha_{i,j}}{2C}, \forall (i, j) \in P,$$

which implies

$$1 - \boldsymbol{w}^T (\boldsymbol{x}_i - \boldsymbol{x}_j) > 0 \quad \Leftrightarrow \quad \alpha_{i,j} > 0.$$

Thus the set $\mathrm{SV}(\boldsymbol{w})$ defined in equation 2.8 corresponds to the set of support vectors.

Acknowledgments

This work was supported in part by the National Science Council of Taiwan (grant 101-2221-E-002-199-MY3). We thank Chia-Hua Ho for discussion and inspiring the selection tree data structure. We also thank Husan-Tien Lin, Yuh-Jye Lee, and the anonymous reviewers for valuable comments.

References

Adelson-Velsky, G. M., & Landis, E. M. (1962). An algorithm for the organization of information. *Proceedings of the USSR Academy of Sciences*, *146*, 263–266.

Airola, A., Pahikkala, T., & Salakoski, T. (2011). Training linear ranking SVMs in linearithmic time using red–black trees. *Pattern Recognition Letters*, *32*(9), 1328–1336.

Andersson, A. (1993). Balanced search trees made simple. In *Proceedings of the Third Workshop on Algorithms and Data Structures* (pp. 60–71).

Bayer, R. (1972). Symmetric binary B-trees: Data structure and maintenance algorithms. *Acta Informatica*, *1*, 290–306.

Boser, B. E., Guyon, I., & Vapnik, V. (1992). A training algorithm for optimal margin classifiers. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory* (pp. 144–152). New York: ACM Press.

Breiman, L. (2001). Random forests. *Machine Learning*, *45*(1), 5–32.

Burges, C. J. C. (2010). *From RankNet to LambdaRank to LambdaMART: An overview* (Technical Report MSR-TR-2010-82). Microsoft Research.

Chapelle, O., & Chang, Y. (2011). Yahoo! learning to rank challenge overview. In *JMLR Workshop and Conference Proceedings: Workshop on Yahoo! Learning to Rank Challenge* (Vol. 14, pp. 1–24).

Chapelle, O., & Keerthi, S. S. (2010). Efficient algorithms for ranking with SVMs. *Information Retrieval*, *13*(3), 201–215.

Christensen, D. (2005). Fast algorithms for the calculation of Kendall's τ. *Computational Statistics*, *20*, 51–62.

Conn, A. R., Gould, N. I. M., & Toint, P. L. (2000). *Trust-region methods*. Philadelphia: Society for Industrial and Applied Mathematics.

Cortes, C., & Vapnik, V. (1995). Support-vector network. *Machine Learning*, *20*, 273–297.

Fan, R.-E., Chang, K.-W., Hsieh, C.-J., Wang, X.-R., & Lin, C.-J. (2008). LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research*, *9*, 1871–1874.

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

Heger, D. A. (2004). A disquisition on the performance behavior of binary search tree data structures. *European Journal for the Informatics Professional*, *5*(5), 67–75.

Herbrich, R., Graepel, T., & Obermayer, K. (2000). Large margin rank boundaries for ordinal regression. In P. J. Bartlett, B. Schölkopf, D. Schuurmans, & A. J. Smola (Eds.), *Advances in large margin classifiers* (pp. 115–132). Cambridge, MA: MIT Press.

Ho, C.-H., & Lin, C.-J. (2012). Large-scale linear support vector regression. *Journal of Machine Learning Research*, *13*, 3323–3348.

Järvelin, K., & Kekäläinen, J. (2002). Cumulated gain-based evaluation of IR techniques. *ACM Transactions on Information Systems*, *20*(4), 422–446.

Joachims, T. (2005). A support vector method for multivariate performance measures. In *Proceedings of the Twenty Second International Conference on Machine Learning (ICML)*.

Joachims, T. (2006). Training linear SVMs in linear time. In *Proceedings of the Twelfth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*.

Keerthi, S. S., & DeCoste, D. (2005). A modified finite Newton method for fast solution of large scale linear SVMs. *Journal of Machine Learning Research*, *6*, 341–361.

Kendall, M. G. (1938). A new measure of rank correlation. *Biometrika*, *30*(1/2), 81–93.

Knuth, D. E. (1973). *The art of computer programming*. Reading, MA: Addison-Wesley.

Lin, C.-J., & Moré, J. J. (1999). Newton's method for large-scale bound constrained problems. *SIAM Journal on Optimization*, *9*, 1100–1127.

Lin, C.-J., Weng, R. C., & Keerthi, S. S. (2008). Trust region Newton method for large-scale logistic regression. *Journal of Machine Learning Research*, *9*, 627–650.

Lin, K.-Y. (2010). *Data selection techniques for large-scale rankSVM*. Master's thesis, National Taiwan University.

Mangasarian, O. L. (2002). A finite Newton method for classification. *Optimization Methods and Software*, *17*(5), 913–929.

Mohan, A., Chen, Z., & Weinberger, K. (2011). Web-search ranking with initialized gradient boosted regression trees. In *JMLR Workshop and Conference Proceedings: Workshop on Yahoo! Learning to Rank Challenge* (Vol. 14, pp. 77–89).

Niu, Y., Wang, Y., Sun, G., Yue, A., Dalessandro, B., Perlich, C., & Hamner, B. (2012). The Tencent dataset and KDD-Cup12. In *ACM SIGKDD KDD-Cup WorkShop*.

Qin, T., Liu, T.-Y., Xu, J., & Li, H. (2010). LETOR: A benchmark collection for research on learning to rank for information retrieval. *Information Retrieval*, *13*(4), 346–374.

Sculley, D. (2009). Large scale learning to rank. In *NIPS 2009 Workshop on Advances in Ranking*.

Shalev-Shwartz, S., Singer, Y., & Srebro, N. (2007). Pegasos: Primal estimated subgradient solver for SVM. In *Proceedings of the Twenty Fourth International Conference on Machine Learning (ICML)*.

Steihaug, T. (1983). The conjugate gradient method and trust regions in large scale optimization. *SIAM Journal on Numerical Analysis*, *20*, 626–637.

Teo, C. H., Vishwanathan, S., Smola, A., & Le, Q. V. (2010). Bundle methods for regularized risk minimization. *Journal of Machine Learning Research*, *11*, 311–365.

Tyree, S., Weinberger, K. Q., Agrawal, K., & Paykin, J. (2011). Parallel boosted regression trees for web search ranking. In *Proceedings of the 20th International Conference on World Wide Web* (pp. 387–396).

Vapnik, V. (1995). *The nature of statistical learning theory*. New York: Springer-Verlag.

Wu, K.-W., Ferng, C.-S., Ho, C.-H., Liang, A.-C., Huang, C.-H., Shen, W.-Y., . . . Lin, H.-T. (2012). A two-stage ensemble of diverse models for advertisement ranking in KDD Cup 2012. In *ACM SIGKDD KDD-Cup WorkShop*. New York.

Yuan, G.-X., Ho, C.-H., & Lin, C.-J. (2012). Recent advances of large-scale linear classification. *Proceedings of the IEEE*, *100*(9), 2584–2603.

Zheng, Z., Chen, K., Sun, G., & Zha, H. (2007). A regression framework for learning ranking functions using relative relevance judgments. In *Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (pp. 287–294). New York: ACM.

Received August 5, 2013; accepted November 6, 2013.