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

Abstract

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 the recent development of linear SVM for classification, linear rankSVM may give competitive performance for large and sparse data. Many existing works have studied linear rankSVM. Their focus is on the computational efficiency when the number of preference pairs is large. In this paper, we systematically study past works, discuss their advantages/disadvantages, and propose an efficient algorithm. Different implementation issues and extensions are discussed 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 in recent years, 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, for example, decision tree models and linear regression, directly learn the relevance score of each instance; *pairwise* methods like rankSVM (Herbrich et al., 2000) learn to classify preference pairs; *listwise* methods such as LambdaMART (Burges, 2010) try to optimize the measurement for evaluating the whole ranking list. Among them, rankSVM, as a pairwise approach, is one commonly used method. This method is extended from standard support vector machine (SVM) by Boser et al. (1992) and Cortes and Vapnik (1995). In 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 the more costly kernel SVM may give higher accuracy.[1] The same situation occurs for rankSVM. In this paper, we aim to 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] See, for example, Yuan et al. (2012) for more detailed discussion.

Notation	Explanation
$\boldsymbol{w}$	The weight vector obtained by solving (2) or (3)
$\boldsymbol{x}_i$	The feature vector of the i-th training instance
y_i	Label of the i-th training instance
q_i	Query of the i-th training instance
K	The set of relevance levels
Q	The set of queries
P	The set of preference pairs; see (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 non-zero 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

Table 1: Notation

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

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

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

$$\min_{\boldsymbol{w}} \quad \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} + C \sum_{(i,j)\in P} \max\big(0, 1-\boldsymbol{w}^T(\boldsymbol{x}_i-\boldsymbol{x}_j)\big)^2. \tag{3}$$

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 paper.

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

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

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

$$\binom{k}{2} \times O\left(\left(\frac{l}{k}\right)^2\right) = O(l^2). \tag{4}$$

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

$$\boldsymbol{w}^T\boldsymbol{x}_i \geq \boldsymbol{w}^T\boldsymbol{x}_j \text{ and } \boldsymbol{w}^T\boldsymbol{x}_j \geq \boldsymbol{w}^T\boldsymbol{x}_s \quad \Rightarrow \quad \boldsymbol{w}^T\boldsymbol{x}_i \geq \boldsymbol{w}^T\boldsymbol{x}_s, \tag{5}$$

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 works apply 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 (2) by a cutting plane method, in which an

$$O(l\bar{n} + l \log l + lk + n) \tag{6}$$

method is proposed to calculate the objective value and a sub-gradient of problem (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 non-zero features per training instance; $O(l \log l + lk)$ is for the sum of training losses in problem (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 et al. (2011) improve upon Joachims' work by reducing the complexity to

$$O(l\bar{n} + l \log l + l \log k + n). \tag{7}$$

The main breakthrough is that they employ order-statistic trees, so the $O(lk)$ term in Equation (6) is reduced to $O(l \log k)$. Another type of optimization methods considered is the truncated Newton methods for solving problem (3), in which the main computation is on Hessian-vector products. Chapelle and 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 (6) can be removed if $k = 2$. Therefore, similar to Joachims' 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 rankSVMs is an established method, it is known that gradient boosting decision trees (GBDT) by Friedman (2001) and its variant, LambdaMART (Burges, 2010), give better performance on web-search ranking data. In addition, random forest (Breiman, 2001) is also reported in Mohan et al. (2011) to perform well. Actually all the winning teams of *Yahoo Learning to Rank Challenge* (Chapelle and Chang, 2011) use decision tree based ensemble models. Note that GBDT and random forest 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 paper, we consider Newton methods for solving problem (3) and present the following results.

1. We give a clear overview and connection of past works 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 detailedly compare between linear rankSVM and linear/nonlinear pointwise methods including GBDT and random forest.

5. We release a public tool for linear rankSVM.

This paper is organized as follows. Section 2 introduces methods for the efficient calculation over all relevance pairs. Section 3 discusses former studies of linear rankSVM, and compares them with our method. Various types of experiments are shown in Section 4. In Section 5 we discuss some extensions and other possible algorithms for rankSVM. Section 6 concludes the paper. A supplementary file including additional analysis and experiments is available at `http://www.csie.ntu.edu.tw/~cjlin/papers/ranksvm/supplement.pdf`.

2 Efficient Calculation Over Relevance Pairs

As mentioned in Section 1, a difficulty to train 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. In this section, we consider a trust region Newton method by Lin and Moré (1999) as an example and investigate efficient methods for the calculation over pairs.

2.1 Newton, Truncated Newton and Trust Region Newton Methods

For minimizing a twice differentiable function $f(\boldsymbol{w})$, at the t-th iteration, Newton methods solve

$$\min_{\boldsymbol{s}} q_t(\boldsymbol{s}), \qquad \text{where} \qquad q_t(\boldsymbol{s}) \equiv \nabla f(\boldsymbol{w}^t)^T \boldsymbol{s} + \frac{1}{2}\boldsymbol{s}^T \nabla^2 f(\boldsymbol{w}^t)\boldsymbol{s},$$

and update $\boldsymbol{w}^t$ by

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

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

$$\nabla^2 f(\boldsymbol{w}^t)\boldsymbol{s} = -\nabla f(\boldsymbol{w}^t). \tag{8}$$

To ensure the convergence, usually a line search procedure is applied, so a truncated Newton step is used. For machine learning applications, $\nabla^2 f(\boldsymbol{w})$ is often too large to be stored, so conjugate gradient (CG) method is a common way to solve (8) by taking the special structure of $\nabla^2 f(\boldsymbol{w})$ into account; see more details in Section 2.2. Then the algorithm contains two levels of iterations. The outer one generates $\{\boldsymbol{w}^t\}$, while from $\boldsymbol{w}^t$ to $\boldsymbol{w}^{t+1}$ there are inner CG iterations.

Instead of using line search, in this paper, we consider another type of truncated Newton method called trust region Newton method (TRON). It finds the direction $\boldsymbol{s}$ by minimizing $q_t(\boldsymbol{s})$ within a region that we trust.

$$\min_{\boldsymbol{s}} \quad q_t(\boldsymbol{s}) \qquad \text{subject to} \qquad \|\boldsymbol{s}\| \le \Delta_t, \tag{9}$$

where Δ_t is the size of the trust region. TRON adjusts the trust region Δ_t according to the approximate function reduction $q_t(\boldsymbol{s})$ and the real function decrease. For details of trust region methods, a comprehensive book is by Conn et al. (2000). Our settings for updating Δ_t follows from those in Lin and Moré (1999).

Algorithm 1 Trust region Newton method

```
1. Given $\boldsymbol{w}^0$, $\Delta_0$.
2. For $t = 0, 1, 2, \ldots$
   2.1. If (10) is satisfied,
          return $\boldsymbol{w}^t$.
   2.2. Apply CG iterations until sub-problem (9) is solved or $\boldsymbol{s}$ reaches the trust-region
        boundary.
   2.3. Update $\boldsymbol{w}^t$ and $\Delta_t$ to $\boldsymbol{w}^{t+1}$ and $\Delta_{t+1}$.
```

The same difficulty of not being able to store $\nabla^2 f(\boldsymbol{w})$ for solving (8) also occurs for problem (9). For classification, Lin et al. (2008) apply the approach of Steihaug (1983) to run CG iterations until either a minimum of $q_t(\boldsymbol{s})$ is found or $\boldsymbol{s}$ touches the boundary of the trust region. At each CG iteration, we only need to calculate a Hessian-vector product

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

and it can be performed without storing the Hessian matrix. We will show that the same setting can be employed for rankSVM.

For the stopping condition, we follow that of TRON in the package LIBLINEAR (Fan et al., 2008) to check if the gradient is small enough compared with the initial gradient.

$$\|\nabla f(\boldsymbol{w}^k)\|_2 \leq \epsilon_s \|\nabla f(\boldsymbol{w}^0)\|_2, \tag{10}$$

where $\boldsymbol{w}^0$ is the initial iterate and ϵ_s is the stopping tolerance given by users. Algorithm 1 gives the framework of TRON.

2.2 Efficient Function/Gradient Evaluation and Matrix-vector Products

We discuss details of calculating function, gradient, and Hessian-vector products in Newton methods for solving L2-loss linear rankSVM (3). In particular, we focus on Hessian-vector products, which are the computational bottlenecks. In the rest of this paper, if not specified, $f(\boldsymbol{w})$ represents the objective function of (3).

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

$$A \equiv \begin{array}{c} \\ \vdots \\ (i,j) \\ \vdots \end{array} \begin{array}{c} \begin{array}{ccccc} \cdots & i & \cdots & j & \cdots \end{array} \\ \left[\begin{array}{ccccc} & & & & \\ 0\cdots0 & +1 & 0\cdots0 & -1 & 0\cdots0 \\ & & & & \end{array} \right] \end{array}.$$

That is, if $(i, j) \in P$ then a corresponding row in A has that the i-th entry is 1, the j-th entry is -1, and other entries are all zeros. By this definition, the objective function in (3) can be written as

$$f(\boldsymbol{w}) = \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} + C(\boldsymbol{e} - AX\boldsymbol{w})^T D_{\boldsymbol{w}}(\boldsymbol{e} - AX\boldsymbol{w}), \tag{11}$$

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

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

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

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

The gradient is

$$\begin{aligned} \nabla f(\boldsymbol{w}) &= \boldsymbol{w} - 2C \sum_{(i,j)\in P} (\boldsymbol{x}_i - \boldsymbol{x}_j) \max\left(0, 1 - (\boldsymbol{x}_i - \boldsymbol{x}_j)^T \boldsymbol{w}\right) \\ &= \boldsymbol{w} + 2CX^T(A^T D_{\boldsymbol{w}} AX\boldsymbol{w} - A^T D_{\boldsymbol{w}} \boldsymbol{e}). \end{aligned} \tag{12}$$

However, $\nabla^2 f(\boldsymbol{w})$ does not exist because (12) is not differentiable. Following Mangasarian (2002) and Lin et al. (2008), we define a generalized Hessian matrix

$$\nabla^2 f(\boldsymbol{w}) \equiv I + 2CX^T A^T D_{\boldsymbol{w}} AX, \tag{13}$$

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

$$\nabla^2 f(\boldsymbol{w})\boldsymbol{v} = \boldsymbol{v} + 2CX^T\left(A^T\left(D_{\boldsymbol{w}}\left(A(X\boldsymbol{v})\right)\right)\right). \tag{14}$$

Because A and $D_{\boldsymbol{w}}$ both have $O(p)$ non-zero elements, the complexity of calculating (14) is

$$O(l\bar{n} + p + n). \tag{15}$$

The right-to-left matrix-vector products in (14) are faster than $O(p\bar{n})$ if we obtain and store the matrix $X^T A^T D_{\boldsymbol{w}} AX$.

However, if $p = O(l^2)$, the cost of (14) is still high and the storage of A and $D_{\boldsymbol{w}}$ requires a huge amount of memory. To derive a faster method, we explore the structure of the generalized Hessian. We define

$$\text{SV}(\boldsymbol{w}) \equiv \{(i,j) \mid (i,j) \in P,\ 1 - \boldsymbol{w}^T(\boldsymbol{x}_i - \boldsymbol{x}_j) > 0\}, \quad \text{and} \quad p_{\boldsymbol{w}} \equiv |\text{SV}(\boldsymbol{w})|. \tag{16}$$

We will show in Appendix A that when (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 $\text{SV}(\boldsymbol{w})$ 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 $(i,j) \in \text{SV}(\boldsymbol{w})$. Thus (14) becomes

$$\nabla^2 f(\boldsymbol{w})\boldsymbol{v} = \boldsymbol{v} + 2CX^T A_{\boldsymbol{w}}^T A_{\boldsymbol{w}} X\boldsymbol{v}. \tag{17}$$

Observe that

$$(A_{\boldsymbol{w}}^T A_{\boldsymbol{w}})_{i,j} = \sum_s (A_{\boldsymbol{w}})_{s,i}(A_{\boldsymbol{w}})_{s,j}. \tag{18}$$

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

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

We define

$$\begin{aligned} \text{SV}_i^+(\boldsymbol{w}) &\equiv \{j \mid (j,i) \in \text{SV}(\boldsymbol{w})\}, \quad l_i^+(\boldsymbol{w}) \equiv |\text{SV}_i^+(\boldsymbol{w})|, \quad \alpha_i^+(\boldsymbol{w}, \boldsymbol{v}) \equiv \sum_{j \in \text{SV}_i^+(\boldsymbol{w})} \boldsymbol{x}_j^T \boldsymbol{v}, \\ \text{SV}_i^-(\boldsymbol{w}) &\equiv \{j \mid (i,j) \in \text{SV}(\boldsymbol{w})\}, \quad l_i^-(\boldsymbol{w}) \equiv |\text{SV}_i^-(\boldsymbol{w})|, \quad \alpha_i^-(\boldsymbol{w}, \boldsymbol{v}) \equiv \sum_{j \in \text{SV}_i^-(\boldsymbol{w})} \boldsymbol{x}_j^T \boldsymbol{v}. \end{aligned}$$

Then from (18),

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

Hence,

$$\begin{aligned} (A_{\boldsymbol{w}}^T A_{\boldsymbol{w}} X \boldsymbol{v})_i &= \sum_{j=1}^{l} (A_{\boldsymbol{w}}^T A_{\boldsymbol{w}})_{i,j} (X\boldsymbol{v})_j \\ &= \big(l_i^+(\boldsymbol{w}) + l_i^-(\boldsymbol{w})\big)\boldsymbol{x}_i^T \boldsymbol{v} - \sum_{j \in \text{SV}_i^+(\boldsymbol{w})} \boldsymbol{x}_j^T \boldsymbol{v} - \sum_{j \in \text{SV}_i^-(\boldsymbol{w})} \boldsymbol{x}_j^T \boldsymbol{v}. \end{aligned}$$

Therefore,

$$X^T A_{\boldsymbol{w}}^T A_{\boldsymbol{w}} X \boldsymbol{v} = X^T \begin{bmatrix} \Big(l_1^+(\boldsymbol{w}) + l_1^-(\boldsymbol{w})\Big)\boldsymbol{x}_1^T \boldsymbol{v} - \Big(\alpha_1^+(\boldsymbol{w}, \boldsymbol{v}) + \alpha_1^-(\boldsymbol{w}, \boldsymbol{v})\Big) \\ \vdots \\ \Big(l_l^+(\boldsymbol{w}) + l_l^-(\boldsymbol{w})\Big)\boldsymbol{x}_l^T \boldsymbol{v} - \Big(\alpha_l^+(\boldsymbol{w}, \boldsymbol{v}) + \alpha_l^-(\boldsymbol{w}, \boldsymbol{v})\Big) \end{bmatrix}. \tag{19}$$

If we already have the values of $l_i^+(\boldsymbol{w}), l_i^-(\boldsymbol{w}), \alpha_i^+(\boldsymbol{w}, \boldsymbol{v})$ and $\alpha_i^-(\boldsymbol{w}, \boldsymbol{v})$, the computation of the Hessian-vector product in (17) would just cost $O(l\bar{n} + n)$, where $O(l\bar{n})$ is for computing (19), and $O(n)$ is for the vector addition in (17).

Similarly, function and gradient evaluations can be more efficient by reformulating (11) and (12) to the following forms, respectively.

$$\begin{aligned} f(\boldsymbol{w}) &= \frac{1}{2}\boldsymbol{w}^T \boldsymbol{w} + C(A_{\boldsymbol{w}} X \boldsymbol{w} - \boldsymbol{e}_{\boldsymbol{w}})^T (A_{\boldsymbol{w}} X \boldsymbol{w} - \boldsymbol{e}_{\boldsymbol{w}}) \\ &= \frac{1}{2}\boldsymbol{w}^T \boldsymbol{w} + C(\boldsymbol{w}^T X^T (A_{\boldsymbol{w}}^T A_{\boldsymbol{w}} X \boldsymbol{w} - 2A_{\boldsymbol{w}}^T \boldsymbol{e}_{\boldsymbol{w}}) + p_{\boldsymbol{w}}), \end{aligned} \tag{20}$$

and

$$\nabla f(\boldsymbol{w}) = \boldsymbol{w} + 2CX^T(A_{\boldsymbol{w}}^T A_{\boldsymbol{w}} X\boldsymbol{w} - A_{\boldsymbol{w}}^T \boldsymbol{e}_{\boldsymbol{w}}), \tag{21}$$

where $\boldsymbol{e}_{\boldsymbol{w}} \in \mathbf{R}^{p_{\boldsymbol{w}} \times 1}$ is a vector of ones. In (20) and (21), $A_{\boldsymbol{w}}^T A_{\boldsymbol{w}} X\boldsymbol{w}$ can be calculated by (19). 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 (20) and (21) both cost $O(l\bar{n} + n)$ as well.

Note that for solving (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 sub-gradient evaluation; see more details in Section 3.2. Therefore, regardless of optimization methods used, an important common task is to efficiently calculate $l_i^+(\boldsymbol{w}), l_i^-(\boldsymbol{w}), \alpha_i^+(\boldsymbol{w}, \boldsymbol{v})$, and $\alpha_i^-(\boldsymbol{w}, \boldsymbol{v})$. Subsequently, we will discuss a direct method, followed by more efficient approaches.

2.3 A Direct Method to Calculate $l_i^+(\boldsymbol{w}), l_i^-(\boldsymbol{w}), \alpha_i^+(\boldsymbol{w}, \boldsymbol{v})$ and $\alpha_i^-(\boldsymbol{w}, \boldsymbol{v})$

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

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

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 easier description, we assume that

$$\boldsymbol{w}^T\boldsymbol{x}_1 \leq \cdots \leq \boldsymbol{w}^T\boldsymbol{x}_l. \tag{22}$$

We then notice that if

$$\text{count}^+(r) \equiv |\{j \mid y_j = r, \boldsymbol{w}^T\boldsymbol{x}_j < \boldsymbol{w}^T\boldsymbol{x}_i + 1\}|, \ \forall r \in K$$

are available, then

$$l_i^+(\boldsymbol{w}) = \sum_{r:r>y_i} \text{count}^+(r). \tag{23}$$

We can easily maintain $\text{count}^+(r)$ $\forall r$ when moving from i to $i+1$ by

$$\text{count}^+(y_j) \leftarrow \text{count}^+(y_j) + 1, \ \ \text{if } \boldsymbol{w}^T\boldsymbol{x}_i + 1 \leq \boldsymbol{w}^T\boldsymbol{x}_j < \boldsymbol{w}^T\boldsymbol{x}_{i+1} + 1.$$

We illustrate how this method obtains $l_i^+(\boldsymbol{w})$ by considering the following example.

i	1	2	3	4	5	6
$\boldsymbol{w}^T\boldsymbol{x}_i$	-0.9	-0.7	-0.1	0.15	0.2	1.6
y_i	2	1	2	3	3	2

For $i = 1$, we have

$$\begin{array}{cccccc} \downarrow & & \downarrow & & & \\ -0.9 & -0.7 & -0.1 & 0.15 & 0.2 & 1.6, \end{array}$$

where the first pointer indicates the current i, while the second one indicates the bound $\boldsymbol{w}^T\boldsymbol{x}_i + 1$. Thus, $\text{count}^+ = (1, 2, 0)$, and $l_1^+(\boldsymbol{w}) = 0$. For $i = 2$, we have

$$\begin{array}{cccccc} & \downarrow & & & \downarrow & \\ -0.9 & -0.7 & -0.1 & 0.15 & 0.2 & 1.6, \end{array}$$

and $\text{count}^+ = (1, 2, 2)$. Thus, $l_2^+(\boldsymbol{w}) = 2 + 2 = 4$.

The calculation for $l_i^-(\boldsymbol{w})$ is similar but goes through the whole data from l to 1 and maintains $|\{j \mid y_j = r, \boldsymbol{w}^T\boldsymbol{x}_j > \boldsymbol{w}^T\boldsymbol{x}_i - 1\}|$, $\forall r$.

Next, we discuss how to calculate $\alpha_i^+(\boldsymbol{w}, \boldsymbol{v})$ and $\alpha_i^-(\boldsymbol{w}, \boldsymbol{v})$. Notice that

$$l_i^+(\boldsymbol{w}) = \sum_{j \in \text{SV}_i^+(\boldsymbol{w})} 1, \quad \text{and} \quad \alpha_i^+(\boldsymbol{w}, \boldsymbol{v}) = \sum_{j \in \text{SV}_i^+(\boldsymbol{w})} \boldsymbol{x}_i^T \boldsymbol{v}. \tag{24}$$

Thus, $\alpha_i^+(\boldsymbol{w}, \boldsymbol{v})$ can be calculated by a similar method that maintains the following value.

$$\text{xv}^+(r) \equiv \sum_{j: y_j = r, \boldsymbol{w}^T\boldsymbol{x}_j < \boldsymbol{w}^T\boldsymbol{x}_i + 1} \boldsymbol{x}_j^T \boldsymbol{v}, \ \forall r \in S.$$

If $\boldsymbol{w}^T\boldsymbol{x}_i$ have been sorted before CG iterations, this approach needs $O(l + k)$ space and costs

$$O(l\bar{n} + lk + n) \tag{25}$$

time for one matrix-vector product. This type of approach has been used by Joachims (2005) and Chapelle and Keerthi (2010) for the situation of $k = 2$, although our discussion is more general for any k. A procedure with similar complexity for general k has been proposed by Joachims (2006). See more discussion in Section 3.2.

Because $O(lk) \leq O(p) = O(l^2)$, the current approach is better than the method by (14). However, the $O(lk)$ complexity is still high if k is large. Subsequently we will discuss methods to reduce this $O(lk)$ term to $O(l \log k)$.

2.4 Efficient Calculation by Storing Values in an Order-statistic Tree

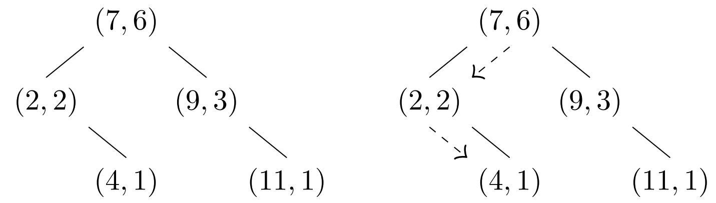
Airola et al. (2011) calculate $l_i^+(\boldsymbol{w})$ and $l_i^-(\boldsymbol{w})$ by an order-statistic tree, so the $O(lk)$ term in (25) is reduced to $O(l \log k)$. The optimization method used is a cutting plane method (Teo et al., 2010), which needs the sub-gradient. Our procedure here is extended from theirs, because we need to obtain not only the gradient but also Hessian-vector products in Newton methods.

Similar to the situation in Section 2.3, we assume that $\boldsymbol{w}^T\boldsymbol{x}_i$ are sorted as in (22). We observe that if elements in

$$\{j \mid \boldsymbol{w}^T\boldsymbol{x}_j < \boldsymbol{w}^T\boldsymbol{x}_i + 1\} \tag{26}$$

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 example.

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



(a) An example of arranging elements of the set (27) 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 $l_i^+(\boldsymbol{w})$

When $i = 1$, we assume

$$\{j \mid \boldsymbol{w}^T \boldsymbol{x}_j < \boldsymbol{w}^T \boldsymbol{x}_1 + 1\} = \{1, 2, 3, 4, 5, 6\}. \tag{27}$$

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

$$\begin{cases} \text{key} : y_j, \\ \text{size} : \text{number of instances in tree}(y_j), \end{cases} \tag{28}$$

where

$$\text{tree}(y) \equiv \text{tree with root } y,$$

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 to fulfill the property of a binary search tree. Clearly, for any node y,

$$\begin{aligned} &\text{size}(y) \\ =&\begin{cases} |\{j \mid y_j = y,\ j \in T\}| & \text{if } y \text{ is a leaf,} \\ \text{size}(y\text{'s left child}) + \text{size}(y\text{'s right child}) + |\{j \mid y_j = y,\ j \in T\}| & \text{otherwise.} \end{cases} \end{aligned} \tag{29}$$

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

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

Therefore, once a tree for the set (26) has been constructed, we can define the following function.

$$
\begin{aligned}
&\text{Larger}(y, y_i) \equiv |\{j \mid y_j > y_i, j \in \text{tree}(y)\}| \\
&= \begin{cases} 0 & \text{if } y \text{ is a leaf, and } y \le y_i, \\ \text{size}(y) & \text{if } y \text{ is a leaf, and } y > y_i, \\ \text{size}(y\text{'s right child}) & \text{if } y \text{ is not a leaf, and } y = y_i, \\ \text{Larger}(y\text{'s right child}, y_i) & \text{if } y \text{ is not a leaf, and } y < y_i, \\ \text{Larger}(y\text{'s left child}, y_i) & \\ \quad + \text{size}(y) - \text{size}(y\text{'s left child}) & \text{if } y \text{ is not a leaf, and } y > y_i, \end{cases}
\end{aligned}
\tag{30}
$$

and let

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

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

Once $l_i^+(\boldsymbol{w})$ has been calculated, we move on to 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 by the same way. The calculation for $l_i^-(\boldsymbol{w})$ is similar. Because

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

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 Smaller(y, y_i) similar to Larger(y, y_i) to obtain $l_i^-(\boldsymbol{w})$.

For the calculation of $\alpha_i^+(\boldsymbol{w}, \boldsymbol{v})$ and $\alpha_i^-(\boldsymbol{w}, \boldsymbol{v})$, by (24) and the fact that size(y) satisfies

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

at each node of the tree we can store a value xv(y) so that it follows a relation like (29).

$$
\begin{aligned}
&\text{xv}(y) \equiv \sum_{j: j \in \text{tree}(y)} \boldsymbol{x}_j^T\boldsymbol{v} \\
&= \begin{cases} \sum_{j: j \in T, y_j = y} \boldsymbol{x}_j^T\boldsymbol{v} & \text{if } y \text{ has no child,} \\ \text{xv}(y\text{'s left child}) + \text{xv}(y\text{'s right child}) + \sum_{j: j \in T, y_j = y} \boldsymbol{x}_j^T\boldsymbol{v} & \text{otherwise.} \end{cases}
\end{aligned}
\tag{31}
$$

The function Larger(y, y_i) defined in (30) can be directly extended to output both $l_i^+(\boldsymbol{w})$ and $\alpha_i^+(\boldsymbol{w}, \boldsymbol{v})$.

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

Algorithm 2 Obtaining $l_i^+(\boldsymbol{w}), l_i^-(\boldsymbol{w}), \alpha_i^+(\boldsymbol{w},\boldsymbol{v})$ and $\alpha_i^-(\boldsymbol{w},\boldsymbol{v})$ 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: $\boldsymbol{w}^T\boldsymbol{x}_{\pi(1)} \leq \cdots \leq \boldsymbol{w}^T\boldsymbol{x}_{\pi(l)}$.
3. $j \leftarrow 1$, $T \leftarrow$ an empty order-statistic tree.
4. For $i = 1, \ldots, l$
   4.1. While $j \leq l$ 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. $\big(l_{\pi(i)}^+(\boldsymbol{w}), \alpha_{\pi(i)}^+(\boldsymbol{w},\boldsymbol{v})\big) \leftarrow$ 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 $j \geq 1$ 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. $\big(l_{\pi(i)}^-(\boldsymbol{w}), \alpha_{\pi(i)}^-(\boldsymbol{w},\boldsymbol{v})\big) \leftarrow$ Smaller(root of $T$, $y_{\pi(i)}$).
```

discussion in Section 2.6. If $\boldsymbol{w}^T\boldsymbol{x}_i$ have been sorted before CG iterations, each matrix-vector product involves

$$O(l\bar{n} + l\log k + n) \tag{32}$$

operations, which are smaller than Equation (25) because the lk term is reduced to $l\log k$. Therefore, the cost of TRON using order-statistic trees is

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

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 (19). Thus an outer iteration of TRON 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 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 $\text{xv}(y)$. Further, we need $O(ln)$ space to store vectors.[2] Because the number of matrix-vector products is often not large, storing $\text{xv}(y)$ 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

$$y_{\pi(1)} \leq \cdots \leq y_{\pi(l)}, \tag{33}$$

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 only need to be sorted once in the whole training procedure. However, $\boldsymbol{w}^T\boldsymbol{x}_i$

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

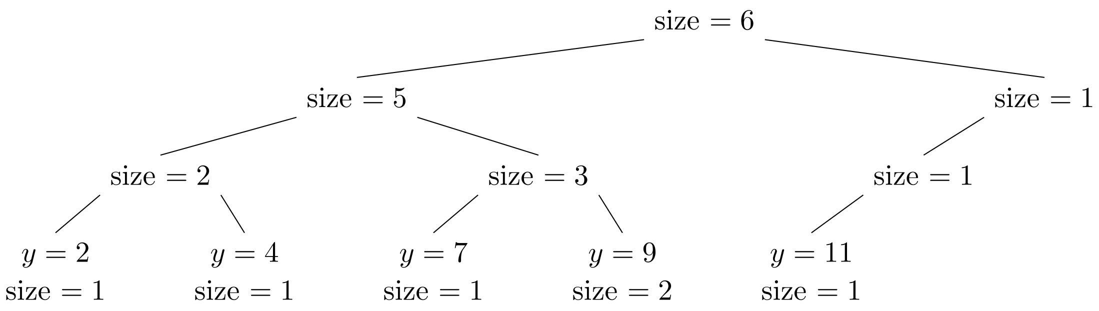


Figure 2: An example of storing y_j in leaf nodes.

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.

2.5 A Different Implementation by Storing Keys in Leaves of a Tree

Although the method in Section 2.4 successfully reduces the complexity, maintaining the trees makes the implementation more complicated. In this section, we consider a simpler method which 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 sub-tree, its size and xv are both the sum of that attribute of its children. For the same example considered in Section 2.4, the tree at $i = 1$ is shown in Figure 2. 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

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

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 $l_i^-(\boldsymbol{w})$, $\alpha_i^+(\boldsymbol{w}, \boldsymbol{v})$ and $\alpha_i^-(\boldsymbol{w}, \boldsymbol{v})$ 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 2.6. In Section 4.2, we will experimentally compare this approach with the method in Section 2.4.

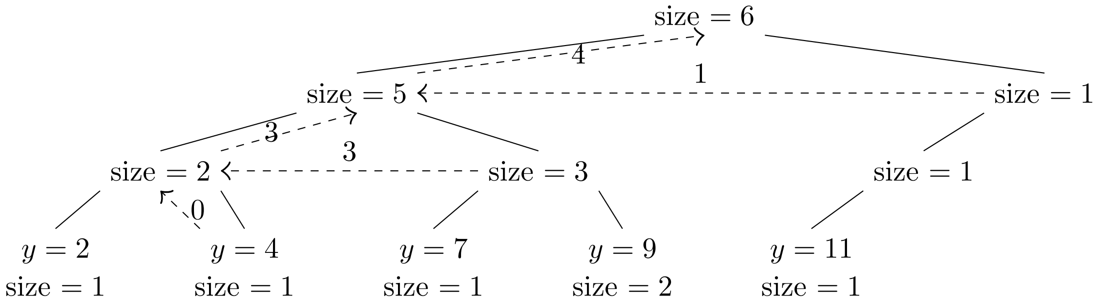


Figure 3: An example of finding $l_1^+(\boldsymbol{w})$ when storing y_j in leaf nodes. Note that we assume $y_1 = 4$.

2.6 A Discussion on Tree Implementations

For the method in Section 2.4, where each node has a key, we can consider balanced binary search trees such as AVL tree, red-black tree and AA tree. It is known that 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 4.2.

For the method in Section 2.5 to store keys in leaves, we have mentioned that 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 left 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 have been mentioned in Section 2 in compared with our proposed methods.

3.1 PRSVM and PRSVM+

We have discussed PRSVM by Chapelle and Keerthi (2010) in the beginning of Section 2.2. The complexity shown in (15) has a term $O(p)$, which becomes dominant for large p. To reduce the cost, Chapelle and Keerthi (2010) proposed PRSVM+ for solving (3) by a truncated Newton method. They first consider the case of $k = 2$ (i.e., two relevance levels). The algorithm for calculating $l_i^+(\boldsymbol{w}), l_i^-(\boldsymbol{w}), \alpha_i^+(\boldsymbol{w}, \boldsymbol{v})$ and $\alpha_i^-(\boldsymbol{w}, \boldsymbol{v})$ is related to Joachims

(2005) and is a special case of that in Section 2.3. 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. By replacing the $O(lk)$ term in (25) with O(size of each two-level set), the complexity of each matrix-vector product is

$$O(l\bar{n} + n) + \sum_{r\in K} O(|\{(i,j) \mid (i,j) \in P, y_i > r, y_j = r\}|). \tag{34}$$

If each relevance level takes about the same amount of $O(l/k)$ data, (34) becomes

$$O(l\bar{n} + n) + \sum_{m=2}^{k} O\left(\frac{lm}{k}\right) = O(l\bar{n} + lk + n), \tag{35}$$

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

3.2 TreeRankSVM

Joachims (2006) uses a cutting plane method to optimize (2). Airola et al. (2011) improve upon Joachims' 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 t-th 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 non-differentiable, then a sub-gradient 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 $L(\boldsymbol{w})$:

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

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

$$\boldsymbol{w}^{t+1} = \arg\min_{\boldsymbol{w}} \quad \frac{1}{2}\boldsymbol{w}^T\boldsymbol{w} + L_t^{\text{CP}}(\boldsymbol{w}). \tag{36}$$

For problem (2), a sub-gradient of its loss term is

$$\begin{aligned}\nabla^s\Big(C\sum_{(i,j)\in P, 1-\boldsymbol{w}^T(\boldsymbol{x}_i-\boldsymbol{x}_j)>0}\big(1-\boldsymbol{w}^T(\boldsymbol{x}_i-\boldsymbol{x}_j)\big)\Big) &= C\sum_{(i,j)\in P, 1-\boldsymbol{w}^T(\boldsymbol{x}_i-\boldsymbol{x}_j)>0}(\boldsymbol{x}_i-\boldsymbol{x}_j)\\ &= C\sum_{i=1}^{l}\big(l_i^+(\boldsymbol{w})-l_i^-(\boldsymbol{w})\big)\boldsymbol{x}_i. \qquad (37)\end{aligned}$$

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 similar to the method in Section 2.3, and the complexity at each iteration is shown in (6). However, the derivation is slightly more complicated. We leave the details in the supplementary materials. As mentioned in Section 2.4, the main improvement made by Airola et al. (2011) is to use order-statistic trees to reduce the $O(lk)$ term in calculating $l_i^+(\boldsymbol{w})$ and $l_i^-(\boldsymbol{w})$, $\forall i$ to $O(l\log k)$. In particular, red-black trees were adopted in their work. The overall cost is

$$\big(O(l\log l + l\bar{n} + l\log k + n) + \text{cost of (36)}\big) \times \#\text{iterations}.$$

3.3 sofia-ml

Sculley (2009) proposed sofia-ml to solve problem (2). It is a stochastic gradient descent method that randomly draws a preference pair from the training set at each iteration, and uses a sub-gradient 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.

4 Experiments

In this section, we first 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 an investigation of the performance difference between linear rankSVM and pointwise methods is conducted. Finally, an experiment on sparse data is shown. Programs used for experiments can be found at `http://www.csie.ntu.edu.tw/~cjlin/liblinear/exp.html`.

4.1 Experiment Setting

We consider three sources of web-search engine ranking: LETOR 4.0 (Qin et al., 2010), MSLR[3] and YAHOO LTRC (Chapelle and Chang, 2011). Both LETOR 4.0 and MSLR are

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

Data set	l	n	k	$\lvert Q\rvert$	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

Table 2: Statistics of training data sets. Note that 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.

from Microsoft Research, while YAHOO LTRC is from *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 pre-processing, 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 the experiments are conducted on a 64-bit machine with Intel Xeon 2.5GHz CPU (E5504), 12MB cache, and 16GB memory.

4.2 A Comparison Between Methods in Section 2: a Direct Counting Method and Different Order-statistic Trees

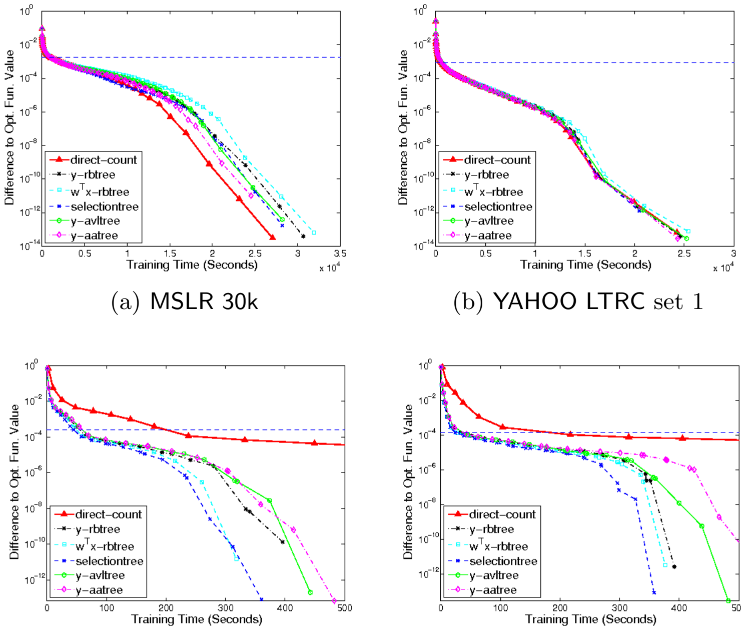
We solve (3) using TRON and compare the following methods for calculating $l_i^+(\boldsymbol{w})$, $l_i^-(\boldsymbol{w})$, $\alpha_i^+(\boldsymbol{w}, \boldsymbol{v})$ and $\alpha_i^-(\boldsymbol{w}, \boldsymbol{v})$.

- direct-count: the direct counting method discussed in Section 2.3.
- y-rbtree: the red-black tree using y_i as the key of nodes. See Section 2.4.
- $w^T x$-rbtree: the red-black tree using $\boldsymbol{w}^T \boldsymbol{x}_i$ as the key of nodes. See Section 2.4.
- selectiontree: the selection tree that stores keys in leaf nodes. See Section 2.5.
- y-avltree: the same as y-rbtree, except the order-statistic tree used is AVL tree. See Section 2.6.
- y-aatree: the same as y-rbtree, except the order-statistic tree used is AA tree. See Section 2.6.

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.

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

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



(a) MSLR 30k

(b) YAHOO LTRC set 1

(c) MQ2007-list

(d) MQ2008-list

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 the four data sets. The dotted horizontal line indicates the function value of TRON using default stopping tolerance $\epsilon_s = 0.001$ in (10).

where $\boldsymbol{w}^*$ is the optimum of (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 a default stopping condition in TRON of using $\epsilon_s = 10^{-3}$ in (10) has been satisfied. Solutions obtained below this line should have similar ranking performance to the optimum. From the figures, the method of direct counting is slow as expected when k (number of relevance levels) is large. In addition, although implementations of order-statistic trees have slightly different running time in the end, they are very similar otherwise. Therefore, we choose selection trees in subsequent experiments because of the simplicity.

4.3 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 and 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 the truncated Newton method.

- 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 (36) 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 and Kekäläinen, 2002). Several definitions of NDCG are available, so we follow the recommendation of each data source. Assume m is a pre-specified 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,

$$\text{NDCG@}m \equiv (N_m)^{-1} \sum_{i=1}^{m} (2^{y_{\pi'(i)}} - 1) d(i), \tag{38}$$

where

$$N_m = \sum_{i=1}^{m} (2^{y_{\pi(i)}} - 1) d(i) \quad \text{and} \quad d(i) = \begin{cases} \frac{1}{\log_2(\max(2,i))} & \text{for MSLR and LETOR 4.0,} \\ \frac{1}{\log_2(i+1)} & \text{for YAHOO LTRC.} \end{cases}$$

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 (38), 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.

We further consider pairwise accuracy as a measurement because it is directly related to the loss term of rankSVM.

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

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

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

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

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

	Problem (2) using L1 loss		Problem (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

Table 3: Best regularization parameter for each data set and each measurement. When problem (2) with L1 loss is used, TreeRankSVM failed to finish the parameter selection procedure on MSLR 30k and YAHOO LTRC set 1 after long running time. NDCG cannot be used for MQ2007-list and MQ2008-list because large k leads to the overflow of $2^{y_{\pi'(i)}}$ in (38).

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,[7] pairwise accuracy, and NDCG.

One could observe from the figures that the convergence speed of TreeRankSVM is slower than PRSVM+ and Tree-TRON. To rule out the implementation differences between Tree-TRON/PRSVM+ and TreeRankSVM, in the supplementary materials we check (CG) iterations versus objective values. 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. 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+, Figures 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 (32) and (35).

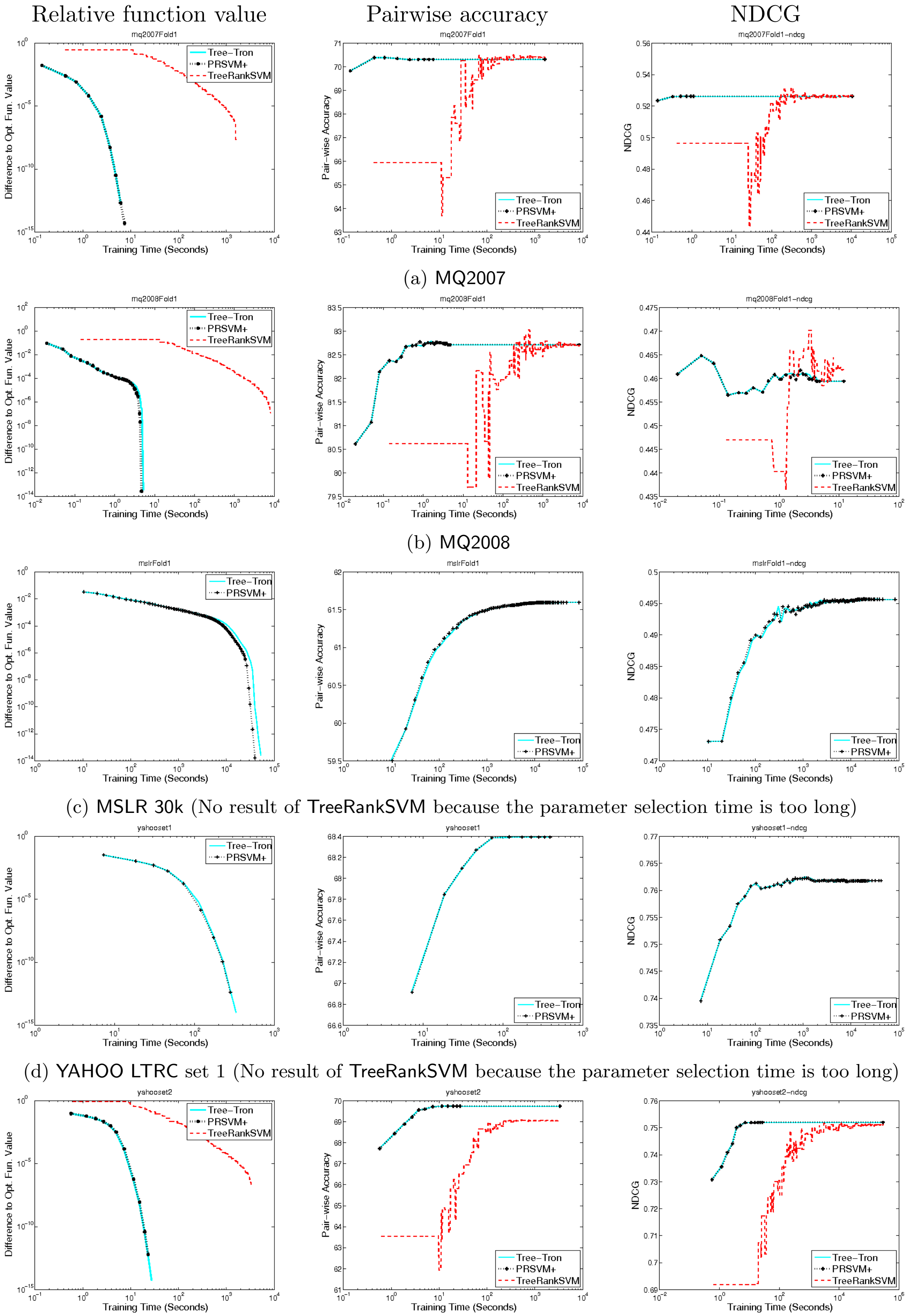
Experiments in this section also serve as a comparison between L1- and L2-loss linear rankSVM. Results show that their performances (NDCG and pairwise accuracy) are similar.

4.4 A Comparison Between Linear RankSVM, Linear Support Vector Regression, GDBT, and Random Forest

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 implementation are available in LIBLINEAR by solving primal and dual problems, respectively. We use the one that solves the primal problem by TRON.

[7] Parameters selected using (validation) pairwise accuracy are considered, but results of using NDCG are similar.



(a) MQ2007

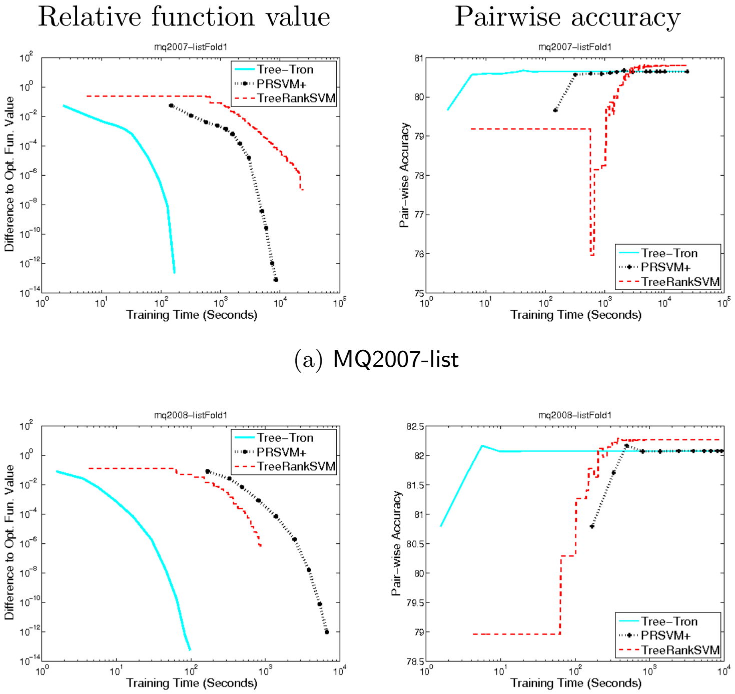
(b) MQ2008

(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.



(a) MQ2007-list

(b) MQ2008-list

Figure 6: A comparison between different linear rankSVM methods on function values and pairwise accuracy. The two sets MQ2007-listMQ2008-list have large k (number of relevance levels). NDCG cannot be used because of the overflow of $2^{y_{\pi'(i)}}$ in (38).

- GBDT (Friedman, 2001): this is a nonlinear pointwise model that is known to be powerful for web-search ranking problems. We use version 1.0 of the package Rt-Rank (Mohan et al., 2011) downloaded from `https://sites.google.com/site/rtranking/`.
- Random forest (Breiman, 2001): this is another nonlinear pointwise model that performs well on web-search data. We also use version 1.0 of Rt-Rank.

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 4.3 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 concern about.

	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 4: Best regularization parameter for each data set and each measurement of SVR

Next, we check GBDT and random forest. Their training time is very long, so we are not able to 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 = 20. For random forest, we use 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 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 $1,000$ trees.

From Tables 6-7, GBDT and random forest generally perform well, though they are not always better than linear rankSVM. For YAHOO LTRC set 2, random forest achieves 0.78 NDCG using $1,000$ trees, which is much better than 0.75 of linear rankSVM. This result is consistent with the fact that in *Yahoo Learning to Rank Challenge*, top performers all use decision tree based methods. However, the training cost of GBDT and random forest is much 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 are not always better than with few trees. This result seems to indicate that overfitting occurs and parameter selection is important. In contrast, random forest is more robust.

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 any problems.

4.5 A Comparison Between Linear and Nonlinear Models on Sparse Data

Recent research works have 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 forest rather than kernel rankSVM. One reason is that random forest is very robust in the previous experiment. We consider the following two CTR (click through rate) estimation problems, which can be either treated as regression or ranking problems.

- CTR: This is a data set used in Ho and Lin (2012).

	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*	0.4757*	0.5	0.5157
MQ2008	0.5	0.4571	3.4*	0.4153*	0.2	0.4450
MSLR 30k	1,601.6	0.4949	461.6	0.4742	202.4	0.4946
YAHOO LTRC set 1	334.8	0.7619	10.8	0.7586	172.7	0.7650
YAHOO LTRC set 2	11.2	0.7519	47.6	0.7470	20.8	0.7578

*: Reached maximum iteration of LIBLINEAR.

	L2-loss RankSVM		L1-loss SVR		L2-loss SVR	
Data set	Training time (s)	Pairwise accuracy	Training time (s)	Pairwise accuracy	Training time (s)	Pairwise accuracy
MQ2007	1.3	70.35%	23.9*	64.04%*	0.7	68.54%
MQ2008	0.5	82.70%	3.4*	77.72%*	0.3	82.17%
MSLR 30k	1,601.6	61.52%	65.4	60.10%	202.4	60.49%
YAHOO LTRC set 1	117.1	68.39%	2.4	67.77%	149.5	67.78%
YAHOO LTRC set 2	11.2	69.74%	3.3	68.37%	14.5	69.39%
MQ2007-list	38.7	80.67%	1.0	79.79%	5.0	79.70%
MQ2008-list	16.6	82.07%	1.1	81.61%	6.7	81.81%

*: Reached maximum iteration of LIBLINEAR.

Table 5: Comparison between rankSVM and SVR

- 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.

The two data sets both contain a single query and each comes with training/testing sets. To reduce the training time and the memory cost of random forest, we subsample from the two data sets and condense the features. The details are listed in Table 8. We use the same parameters of random forest as in Section 4.4. For a fair comparison, we fix $C = 1$ for rankSVM because the parameters of random forest are not well tuned. The results are shown in Table 9. We first notice the training time of random forest 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 forest is linear to the number of features, but that of rankSVM is linear to the average number of non-zero features. Regarding the performance, the difference is small for the two data sets, so linear rankSVM is very useful to quickly get competitive results. However, more experiments are needed to confirm these preliminary observations; we hope more public sparse ranking data will be available in the near future.

	Random forest			GBDT		
	Training	Pairwise	NDCG	Training	Pairwise	NDCG
Data set	time (s)	accuracy		time (s)	accuracy	
MQ2007	14.8	66.16%	0.4959	22.2	69.02%	0.5118
MQ2008	2.3	80.36%	0.4541	2.4	82.83%	0.4748
MSLR 30k	5,102.1	63.76%	0.5598	11,401.3	61.23%	0.5161
YAHOO LTRC set 1	1,672.2	70.69%	0.7797	9,680.5	66.61%	0.7546
YAHOO LTRC set 2	58.7	68.76%	0.7629	276.3	68.66%	0.7575
MQ2007-list	606.0	78.78%	NA	1,029.3	79.45%	NA
MQ2008-list	423.3	82.04%	NA	696.8	81.42%	NA

Table 6: Performance of GBDT and random forest with a small number of trees (40 for random forest and 20 for GBDT).

	Random forest			GBDT		
	Training	Pairwise	NDCG	Training	Pairwise	NDCG
Data set	time (s)	accuracy		time (s)	accuracy	
MQ2007	345.3	69.07%	0.5221	1,452.2	67.55%	0.4916
MQ2008	52.0	82.60%	0.4675	143.5	80.09%	0.4510
YAHOO LTRC set 2	1,406.9	71.91%	0.7801	16,481.9	71.68%	0.7718

Table 7: Performance of GBDT and random forest with 1,000 trees.

5 Discussion

In this section, we discuss some other methods for solving linear rankSVM problems and possible extensions.

5.1 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) that uses pairs with close relevance levels (i.e., y_i close to y_j). The concept is similar to Equation (5): if pairs with close relevance levels are ranked with the right order, then 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 (2) and (3) as classification problems with instances $\boldsymbol{x}_i - \boldsymbol{x}_j$. If Newton methods are considered for solving (3), by the approach in Equation (14), 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

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	$12,704,393$

Table 8: Statistics of sparse training data sets. To reduce the training time, only a small subset of each problem is used.

	Linear rankSVM			Random forest		
	Training	Pairwise	Mean	Training	Pairwise	Mean
Data set	time (s)	accuracy	NDCG	time (s)	accuracy	NDCG
CTR (0.1%)	4.3	60.83%	0.4822	$6,343.3$	60.45%	0.4732
KDD2012b (0.025%)	2.7	68.16%	0.5851	$5,223.1$	69.72%	0.5982

Table 9: Performance of linear rankSVM and random forest on sparse data. Random forest uses 40 trees.

	MQ2007-list			MQ2008-list		
		Training	Pairwise		Training	Pairwise
	C	time (s)	accuracy	C	time (s)	accuracy
Partial pairs	2^{-9}	19.9	79.07%	2^{-10}	10.5	81.78%
All pairs	2^{-12}	38.7	80.67%	2^{-14}	16.6	82.07%

Table 10: A comparison between using partial pairs and all pairs to train a model. LIBLINEAR using TRON for L2-loss SVM is used for the partial-pair setting, while Tree-TRON is used for all pairs.

using all pairs. This situation occurs because the sum of training losses in (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 sacrifice the performance for faster training speed.

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

- If k is small, we can apply the method in Section 2.3 that has an $O(lk)$ term for calculating $l_i^+(\boldsymbol{w}), l_i^-(\boldsymbol{w}), \alpha_i^+(\boldsymbol{w}, \boldsymbol{v})$ and $\alpha_i^-(\boldsymbol{w}, \boldsymbol{v})$.
- 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.

5.2 The Possibility of Solving the Dual Problem

From Appendix A, the dual problems of (2) and (3) both have p variables. It is difficult to solve such problems if $p = O(l^2)$. However, by removing some zero variables during the

Data sets	Percentage of zero dual variables
MQ2007	5.15%
MQ2008	22.44%
YAHOO LTRC set 2	11.14%
MQ2007-list	19.64%
MQ2008-list	23.86%

Table 11: Sparsity of the dual optimal solution of L2-loss linear rankSVM. $C = 2^{20}$ is used.

optimization procedure, we may efficiently solve a smaller problem. If L2 loss is used, from Appendix A, we have

$$\boldsymbol{w}^T(\boldsymbol{x}_i - \boldsymbol{x}_j) \geq 1 \quad \Leftrightarrow \quad \text{the corresponding dual variable is zero.}$$

If at the optimal solution, two pairs (i, j) and (j, s) in P are in the correct order with

$$\boldsymbol{w}^T\boldsymbol{x}_i - \boldsymbol{w}^T\boldsymbol{x}_j \geq 1 \quad \text{and} \quad \boldsymbol{w}^T\boldsymbol{x}_j - \boldsymbol{w}^T\boldsymbol{x}_s \geq 1,$$

then

$$\boldsymbol{w}^T\boldsymbol{x}_i - \boldsymbol{w}^T\boldsymbol{x}_s \geq 1 \tag{39}$$

and the corresponding dual variable is zero. Therefore, in the best situation, only $O(l)$ of the p variables are zero. Then the situation is similar to that of using partial pairs in Section 5.1. To check how good the sparsity is in practice, in Table 11 we present the percentage of zero elements of the dual optimal solution. We use a large penalty parameter $C = 2^{20}$ to get small training errors and therefore, better sparsity.[8] Unfortunately, the solution is very dense; more than 75% of the elements are non-zero. A further check shows that while most pairs are correctly classified, they satisfy

$$0 \leq \boldsymbol{w}^T(\boldsymbol{x}_i - \boldsymbol{x}_j) < 1$$

only, so the corresponding dual variable is still non-zero. Therefore, solving the dual problem may not be an efficient option.

6 Conclusions

In this paper, we systematically reviewed recent approaches for linear rankSVM. We show that, regardless of 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 at `http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/`.

[8] Because of the lengthy training time of using parameter $C = 2^{20}$, we report results of only five data sets.

References

Georgy Maximovich Adelson-Velsky and Evgenii Mikhailovich Landis. An algorithm for the organization of information. *Proceedings of the USSR Academy of Sciences*, 146:263–266, 1962.

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

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

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

Bernhard E. Boser, Isabelle Guyon, and Vladimir Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, pages 144–152. ACM Press, 1992.

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

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

Olivier Chapelle and Yi Chang. Yahoo! learning to rank challenge overview. In *JMLR Workshop and Conference Proceedings: Workshop on Yahoo! Learning to Rank Challenge*, volume 14, pages 1–24, 2011.

Olivier Chapelle and S. Sathiya Keerthi. Efficient algorithms for ranking with SVMs. *Information Retrieval*, 13(3):201–215, 2010.

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

Andrew R. Conn, Nicholas I. M. Gould, and Philippe L. Toint. *Trust-region Methods.* Society for Industrial and Applied Mathematics, Philadelphia, 2000.

Corina Cortes and Vladimir Vapnik. Support-vector network. *Machine Learning*, 20:273–297, 1995.

Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research*, 9: 1871–1874, 2008. URL `http://www.csie.ntu.edu.tw/~cjlin/papers/liblinear.pdf`.

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

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

Ralf Herbrich, Thore Graepel, and Klaus Obermayer. Large margin rank boundaries for ordinal regression. In Peter J. Bartlett, Bernhard Schölkopf, Dale Schuurmans, and Alexander J. Smola, editors, *Advances in Large Margin Classifiers*, pages 115–132. MIT Press, 2000.

Chia-Hua Ho and Chih-Jen Lin. Large-scale linear support vector regression. *Journal of Machine Learning Research*, 13:3323–3348, 2012. URL `http://www.csie.ntu.edu.tw/~cjlin/papers/linear-svr.pdf`.

Kalervo Järvelin and Jaana Kekäläinen. Cumulated gain-based evaluation of IR techniques. *ACM Transactions on Information Systems*, 20(4):422–446, 2002.

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

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

Donald E. Knuth. *The Art of Computer Programming*, volume 3. Addison-Wesley, Reading, MA, 1973.

Chih-Jen Lin and Jorge J. Moré. Newton's method for large-scale bound constrained problems. *SIAM Journal on Optimization*, 9:1100–1127, 1999.

Chih-Jen Lin, Ruby C. Weng, and S. Sathiya Keerthi. Trust region Newton method for large-scale logistic regression. *Journal of Machine Learning Research*, 9:627–650, 2008. URL `http://www.csie.ntu.edu.tw/~cjlin/papers/logistic.pdf`.

Ken-Yi Lin. Data selection techniques for large-scale rankSVM. Master's thesis, Department of Computer Science and Information Engineering, National Taiwan University, 2010.

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

Ananth Mohan, Zheng Chen, and Kilian Weinberger. Web-search ranking with initialized gradient boosted regression trees. In *JMLR Workshop and Conference Proceedings: Workshop on Yahoo! Learning to Rank Challenge*, volume 14, pages 77–89, 2011.

Yanzhi Niu, Yi Wang, Gordon Sun, Aden Yue, Brian Dalessandro, Claudia Perlich, and Ben Hamner. The Tencent dataset and KDD-Cup12. In *ACM SIGKDD KDD-Cup WorkShop*, 2012.

Tao Qin, Tie-Yan Liu, Jun Xu, and Hang Li. LETOR: A benchmark collection for research on learning to rank for information retrieval. *Information Retrieval*, 13(4):346–374, 2010.

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

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

Choon Hui Teo, S.V.N. Vishwanathan, Alex Smola, and Quoc V. Le. Bundle methods for regularized risk minimization. *Journal of Machine Learning Research*, 11:311–365, 2010.

Vladimir Vapnik. *The Nature of Statistical Learning Theory.* Springer-Verlag, New York, NY, 1995.

Kuan-Wei Wu, Chun-Sung Ferng, Chia-Hua Ho, An-Chun Liang, Chun-Heng Huang, Wei-Yuan Shen, Jyun-Yu Jiang, Ming-Hao Yang, Ting-Wei Lin, Ching-Pei Lee, Perng-Hwa Kung, Chin-En Wang, Ting-Wei Ku, Chun-Yen Ho, Yi-Shu Tai, I-Kuei Chen, Wei-Lun Huang, Che-Ping Chou, Tse-Ju Lin, Han-Jay Yang, Yen-Kai Wang, Cheng-Te Li, Shou-De Lin, and Hsuan-Tien Lin. A two-stage ensemble of diverse models for advertisement ranking in KDD Cup 2012. In *ACM SIGKDD KDD-Cup WorkShop*, 2012.

Guo-Xun Yuan, Chia-Hua Ho, and Chih-Jen Lin. Recent advances of large-scale linear classification. *Proceedings of the IEEE*, 100(9):2584–2603, 2012. URL `http://www.csie.ntu.edu.tw/~cjlin/papers/survey-linear.pdf`.

A The Dual Problem of (2) and (3)

Based on the original training data, we can construct a new set $\{\boldsymbol{x}_{i,j}, y_{i,j}\}$ $\forall (i,j) \in P$ 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, (2) and (3) can be viewed as L1-loss and L2-loss SVM problems with only one class of data, respectively. Then the dual problems of (2) and (3) are both in the following form.

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

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 $\text{SV}(\boldsymbol{w})$ defined in (16) corresponds to the set of support vectors.