An Improved Training Algorithm for the Linear Ranking Support Vector Machine

Antti Airola, Tapio Pahikkala, and Tapio Salakoski

University of Turku and Turku Centre for Computer Science (TUCS)
Joukahaisenkatu 3-5 B, Turku, Finland
{antti.airola,tapio.pahikkala,tapio.salakoski}@utu.fi

Abstract. We introduce an $O(ms+m\log(m))$ time complexity method for training the linear ranking support vector machine, where m is the number of training examples, and s the average number of non-zero features per example. The method generalizes the fastest previously known approach, which achieves the same efficiency only in restricted special cases. The excellent scalability of the proposed method is demonstrated experimentally.

Keywords: binary search tree, cutting plane optimization, learning to rank, support vector machine

1 Introduction

The ranking support vector machine (RankSVM) [7, 8], is one of the most successful methods for learning to rank. The method is based on regularized risk minimization with a pairwise loss function, that provides a convex approximation of the number of pairwise mis-orderings in the ranking produced by the learned model. Related learning algorithms based on the pairwise criterion include methods such as RankBoost [5], and RankRLS [11], among others. RankSVM has been shown to achieve excellent performance on ranking tasks such as document ranking in web search [8, 3]. However, the scalability of the method leaves room for improvement. In this work we assume the so-called scoring setting, where each data instance is associated with a utility score reflecting its goodness with respect to the ranking criterion.

Previously, [9] has shown that linear RankSVM can be trained using cutting plane optimization very efficiently, when the number of distinct utility scores allowed is restricted. The introduced method has $O(ms+m\log(m)+rm)$ training complexity, where m is the number of training examples, s the average number of non-zero features per example, and r the number of distinct utility scores in the training set. A similar approach having same training complexity was also introduced by [3]. If r is assumed to be a small constant, the existing methods are computationally efficient. However, in the general case where unrestricted scores are allowed, if most of the training examples have different scores $r \approx m$ leading to $O(ms+m^2)$ complexity. This worst scale quadratic scaling limits the applicability of RankSVM in large scale learning.

In this work we generalize the work of [9] and present a training algorithm which has $O(ms + m\log(m))$ complexity even in the most general case, where arbitrary real-valued utility scores are allowed. The method is based on using binary search trees [2, 4] for speeding up the evaluations needed in the optimization process. Our experiments show the excellent scalability of the method in practice, allowing orders of magnitude faster training times than the fastest previously known methods in case of unrestricted utility scores. Due to space constraints more detailed description of the method and related proofs are left to an upcoming journal extension of the work [1].

2 Learning Setting

Let D be a probability distribution over a sample space $\mathcal{Z} = \mathbb{R}^n \times \mathbb{R}$. An example $z = (\mathbf{x}, y) \in \mathcal{Z}$ is a pair consisting of an n-dimensional column vector of real-valued features, and an associated real-valued utility score. Let the sequence $Z = ((\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_m, y_m)) \in \mathcal{Z}^m$ drawn according to D be a training set of m training examples. $X \in \mathbb{R}^{n\times m}$ denotes the $n \times m$ data matrix whose columns contain the feature representations of the training examples, and $\mathbf{y} \in \mathbb{R}^m$ is a column vector containing the utility scores in the training set.

Our task is to learn from the training data a ranking function $f : \mathbb{R}^n \to \mathbb{R}$. In the linear case such a function can be represented as $f(\mathbf{x}) = \mathbf{w}^{\mathrm{T}}\mathbf{x}$, where $\mathbf{w} \in \mathbb{R}^n$ is a vector of parameters. Where the ranking task differs in the scoring setting from that of simple regression is that the actual values taken by the ranking function are typically not of interest. Rather, what is of interest is how well the ordering acquired by sorting a set of new examples according to their predicted scores matches the true underlying ranking. This is a reasonable criterion for example in the web search engines and recommender systems, where the task is to choose a suitable order in which to present web pages or products to the end user. A popular way to model this criterion is by considering the pairwise preferences induced by a ranking (see e.g. [6]). We say that an example z_i is preferred over example z_j, if $y_i > y_j$. In this case one would require from the ranking function that $f(\mathbf{x}_i) > f(\mathbf{x}_j)$. The performance of a ranking function can be measured by the pairwise ranking error defined as

$$\frac{1}{N}\sum_{y_i<y_j} H(f(\mathbf{x}_i) - f(\mathbf{x}_j)) \ , \tag{1}$$

where H is the Heaviside step function defined as

$$H(a) = \begin{cases} 1, & \text{if } a > 0 \\ 1/2, & \text{if } a = 0 \\ 0, & \text{if } a < 0 \end{cases} ,$$

and N is the number of pairs for which $y_i < y_j$. The equation (1) counts the number of swapped pairs between the true ranking and the one produced by f.

In some learning to rank settings instead of having a total order over all examples, the sample space is divided into disjoint subsets, and pairwise preferences are induced only from pairwise comparisons between the scores of examples in the same subset. An example of an application settings where this approach is commonly adopted is document retrieval, where data consists of query-document pairs, and the scores represent the utility of the document with respect to the associated user query [8]. Preferences are induced only between query-document pairs from the same query, never between examples from different queries. In such settings we can calculate (1) separately for each subset, and take the average value as the final error.

Minimizing (1) directly is computationally intractable, successful approaches to learning to rank according to the pairwise criterion typically minimize convex relaxations instead. The relaxation considered in this work is the pairwise hinge loss, which together with a quadratic regularizer forms the objective function of RankSVM.

3 Algorithm Description

The RankSVM optimization problem can be formulated as the unconstrained regularized risk minimization problem

$$\arg\min_{\mathbf{w}\in\mathbb{R}^n} \frac{1}{N} \sum_{y_i<y_j} \max(0, 1 + \mathbf{w}^{\mathrm{T}}\mathbf{x}_i - \mathbf{w}^{\mathrm{T}}\mathbf{x}_j) + \lambda\|\mathbf{w}\|^2, \tag{2}$$

where $\mathbf{w}$ is the vector of parameters to be learned, N is the number of pairs for which $y_i < y_j$ holds true, and $\lambda \in \mathbb{R}^+$ is a parameter. The first term is the empirical risk measuring how well $\mathbf{w}$ fits the training data, and the second term is the quadratic regularizer measuring the complexity of the hypotheses.

[9] proposed minimizing the RankSVM risk using cutting plane optimization. A more general treatment of this optimization approach, together with improved convergence analysis can be found in [12], where the method is known as the bundle method for regularized risk minimization. The cutting plane method needs $O(\frac{1}{\lambda\epsilon})$ iterations to converge to ϵ-accurate solution for convex nonsmooth loss functions, independent of the training set size [12]. By ϵ-accurate we mean that the difference between the regularized risk for the found solution, and for the optimal solution is smaller than a user defined parameter ϵ.

Due to space constraints detailed description of the cutting plane method is not possible here, but the central insight necessary for implementing fast training algorithms is as follows. On each iteration, given the current solution, the cutting plane method needs the value of the empirical risk, as well as that of its subgradient. For large dataset sizes it is these computations that dominate the runtime, since none of the other computations needed in the optimization are dependent on the sample size or dimensionality. To develop fast training methods a necessary and sufficient condition is to have an efficient algorithm for computing the risk, and its subgradient.

At first glance, it would appear that computing the empirical risk requires $O(m^2)$ comparisons between the training examples. However, as noted by [9, 13], we can rewrite the empirical risk as

$$\frac{1}{N}\sum_{y_i<y_j}\max(0, 1+\mathbf{w}^{\mathrm{T}}\mathbf{x}_i-\mathbf{w}^{\mathrm{T}}\mathbf{x}_j)=\frac{1}{N}\sum_{i=1}^{m}(c_i-d_i)\mathbf{w}^{\mathrm{T}}\mathbf{x}_i+c_i \tag{3}$$

where c_i is the frequency how many times $y_i < y_j$ and $\mathbf{w}^{\mathrm{T}}\mathbf{x}_i > \mathbf{w}^{\mathrm{T}}\mathbf{x}_j - 1$, and d_i is the frequency how many times $y_i > y_j$ and $\mathbf{w}^{\mathrm{T}}\mathbf{x}_i < \mathbf{w}^{\mathrm{T}}\mathbf{x}_j + 1$. A subgradient with respect to $\mathbf{w}$ can be calculated as

$$\frac{1}{N}\sum_{i=1}^{m}(c_i-d_i)\mathbf{x}_i\ . \tag{4}$$

Inner product evaluations, scalar-vector multiplications and vector summations are needed to compute (3) and (4). These take each $O(s)$ time.

Assuming that we know the values of c_i, and d_i for all $1 \leq i \leq m$, both the empirical risk and the subgradient require $O(ms)$ time.

[9] proposes an algorithm for computing these frequencies, and subsequently the loss and the subgradient. However, the work assumes that the range of possible utility score values is restricted to r different values, with r assumed to be a small constant. The method has the computational complexity $O(ms + m\log(m) + rm)$. If the number of allowed scores is not restricted, at worst case $r = m$ and the method has $O(ms+m^2)$ complexity, meaning quadratic behavior in m. In this work we present a more general algorithm, for which the time complexity of evaluating the loss and the subgradient is $O(ms + m\log(m))$ also in the most general case, where arbitrary real valued utility scores are allowed.

To formulate the algorithm we need for bookkeeping purposes a data structure which stores floating point numbers as elements. What is required is that if h is the current number of stored elements, it supports the following operations in $O(\log(h))$ time: insertion of a new element, and query to find out the number of values in the data structure with a larger/smaller value than the given query value. Finally, the data structure must allow the storage of duplicate values.

For logarithmic time insertion and computation of the desired order statistics, a suitable choice is a self-balancing search tree. Our implementation is based on the order statistics tree [4], which is a red-black tree [2] modified so that each node stores the size of the subtree, whose root node it is. Further, we modify the basic data structure to allow the insertion of several duplicate values to the same node. The self-balancing property is crucial, as it guarantees logarithmic worst case performance.

Algorithm 1 illustrates the $O(ms+m\log(m))$ time calculation for calculating the loss and the subgradient. First, the algorithm calculates the predicted scores for the training examples using the current model $\mathbf{w}$. Next, an index list π is created, where the indices of the training examples are ordered in an increasing order, according to the magnitudes of their predicted scores. Then, the algorithm calculates the frequencies needed in evaluating (3) and (4). In lines $7 - 12$ we

Algorithm 1: Subgradient and loss computation

```
   Input: $X$, $\mathbf{y}$, $\mathbf{w}$, $N$
   Output: $\mathbf{a}$, loss
1  $\mathbf{p} \leftarrow X^{\mathrm{T}}\mathbf{w}$;
2  $\mathbf{c} \leftarrow m$ length column vector of zeros;
3  $\mathbf{d} \leftarrow m$ length column vector of zeros;
4  $\pi \leftarrow$ training set indices, sorted in ascending order according to $\mathbf{p}$;
5  $s \leftarrow$ new empty search tree;
6  $j \leftarrow 1$;
7  foreach $i \in \{1 \ldots m\}$ do
8      $k \leftarrow \pi[i]$;
9      while $(j \leq m)$ and $(\mathbf{p}[k] - \mathbf{p}[\pi[j]] > -1)$ do
10         s.insert($\mathbf{y}[\pi[j]]$);
11         $j \leftarrow j + 1$;
12     $\mathbf{c}[k] \leftarrow$s.count_larger($\mathbf{y}[k]$);
13 $s \leftarrow$ new empty search tree;
14 $j \leftarrow m$;
15 foreach $i \in \{m \ldots 1\}$ do
16     $k \leftarrow \pi[i]$;
17     while $(j \geq 1)$ and $(\mathbf{p}[k] - \mathbf{p}[\pi[j]] < 1)$ do
18         s.insert($\mathbf{y}[\pi[j]]$);
19         $j \leftarrow j - 1$;
20     $\mathbf{d}[k] \leftarrow$s.count_smaller($\mathbf{y}[k]$);
21 loss$\leftarrow \frac{1}{N}(\mathbf{p}^{\mathrm{T}}(\mathbf{c} - \mathbf{d}) + \mathbf{1}^{\mathrm{T}}\mathbf{c})$;
22 $\mathbf{a} \leftarrow \frac{1}{N}X(\mathbf{c} - \mathbf{d})$;
```

go through the examples in ascending order, as defined by the predicted scores. When considering a new example $\mathbf{x}_i$, the examples are scanned further, in lines $9-11$, to ensure that the true utility scores of such examples, for which, $\mathbf{w}^{\mathrm{T}}\mathbf{x}_i > \mathbf{w}^{\mathrm{T}}\mathbf{x}_j - 1$ holds true, are stored in the search tree. After this is ensured, the value of c_i is simply the number of scores in the search tree, for which $y_i < y_j$ holds. In lines $15 - 20$ we go through the examples in a reversed direction, and the values of d_i are calculated in an analogous manner. Once these values have been calculated, the loss and the subgradient can be evaluated as in (3) and (4). These operations are performed on lines 21 and 22 as vector-vector and matrix-vector operations, $\mathbf{1}$ represents a column vector of ones.

The computational complexity of the operation $X^{\mathrm{T}}\mathbf{w}$ needed to calculate the predicted scores is $O(ms)$. The cost of sorting the index list π according to these scores is $O(m\log(m))$. The $O(\log(m))$ time insertions on lines 10 and 18, as well as the $O(\log(m))$ time queries on lines 12 and 20 are each called exactly m times, leading to $O(m\log(m))$ cost.

The vector operations needed in calculating the loss have $O(m)$ complexity, and the matrix-vector multiplication necessary for computing the subgradient has $O(ms)$ complexity. Thus, the complexity of calculating the loss and the subgradient is $O(ms + m\log(m))$. The exact value of N can be computed in $O(m\log(m))$ by sorting the true utility scores of the training examples.

As discussed previously, in some ranking settings we do not have a global ranking over all examples. Instead, the training data may be divided into separate subsets, over each of which a ranking is defined. Let the training data set be divided into R subsets, each consisting on average of $\frac{m}{R}$ examples. Then we can

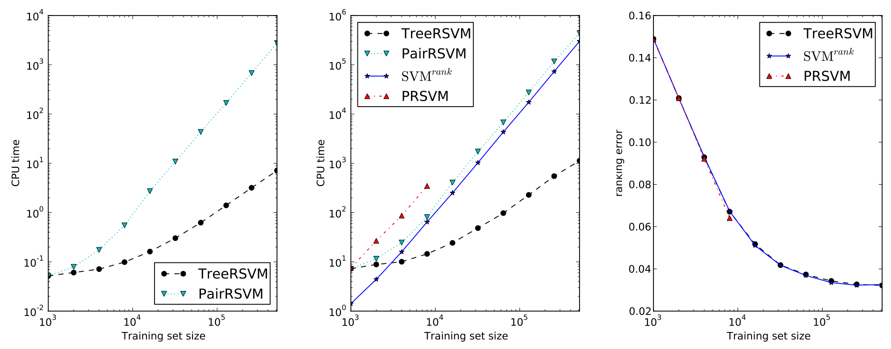

Fig. 1. Average iteration cost (left), runtimes (middle), test error plots (right).

calculate the loss and the subgradient as the average over the losses and subgradients for each subset. The computational complexity becomes $O(R * (\frac{m}{R}s + \frac{m}{R}\log(\frac{m}{R})) = O(ms + m\log(\frac{m}{R}))$.

4 Computational Experiments

In the computational experiments we compare the scalability of the proposed $O(ms + m\log(m))$ time training algorithm to the fastest previously known approach. In addition, we compare our implementation to the existing publicly available RankSVM solvers. The considered data set contains a global ranking, and the utility scores are real valued. This means that $r \approx m$, and the number of pairwise preferences in the training sets grows quadratically with m.

We implement the proposed method, denoted as TreeRSVM, as well as a baseline method PairRSVM, which iterates over all pairs to compute the loss and the subgradient. The methods are implemented mostly in Python using the NumPy, SciPy and CVXOPT libraries, the most computationally demanding parts of the subgradient and loss computations are for both methods implemented in C language due to efficiency reasons.

In addition, we compare our method to the fastest publicly available previous implementations of RankSVM. The SVM^{rank} software is a C-language implementation of the method described in [9]. In theory SVM^{rank} and PairRSVM implement the same method, though the use of different quadratic optimizers, and the inclusion of certain additional heuristics within SVM^{rank}, mean that there may be some differences in their behavior. PRSVM implements in MATLAB a truncated Newton optimization based method for training RankSVM [3]. PRSVM optimizes a slightly different objective function than the other implementations, since it minimizes a squared version of the pairwise hinge loss.

TreeRSVM has $O(ms + m\log(m))$ training time complexity, whereas all the other methods have $O(ms+m^2)$ training time complexity. Therefore, TreeRSVM should on large datasets scale substantially better than the other implementations. Further, all the methods other than PRSVM have $O(ms)$ memory complexity due to cost of storing the data matrix. PRSVM has $O(ms+m^2)$ memory complexity, since it also forms a sparse data matrix that contains two entries per each pairwise preference in the training set. [3] also describe an improved version of PRSVM that has similar scalability as SVM^{rank}, but there is no publicly available implementation of this method.

The experiments are run on a desktop computer with 2.4 GHz Intel Core 2 Duo E6600 processor, 8 GB of main memory, and 64-bit Ubuntu Linux 10.10 operating system. For TreeRSVM, PairRSVM and SVM^{rank} we use the termination criterion $\epsilon < 0.001$, which is the default setting of SVM^{rank}. For PRSVM we use the termination criterion Newton decrement $< 10^{-6}$, as according to [3] this is roughly equivalent to the termination criterion we use for the other methods. SVM^{rank} and PRSVM use a regularization parameter C that is multiplied to the empirical risk term rather than λ, and do not normalize the empirical risk by the number of pairwise preferences N. Therefore, we use the conversion $C = \frac{1}{\lambda N}$, when setting the parameters.

We run scalability experiments on a data set constructed from the Reuters RCV1 collection [10], which consists of approximately 800000 documents. Here, we use a high dimensional feature representation, with each example having approximately 50000 tf-idf values as features. The data set is sparse, meaning that most features are zero-valued. The utility scores are generated as follows. First, we remove one target example randomly from the data set. Next, we compute the dot products between each example and the target example, and use these as utility scores. In effect, the aim is now to learn to rank documents according to how similar they are to the target document. Similarly to the scalability experiments in [3], we compute the running times using a fixed value for the regularization parameter, and a sequence of exponentially growing training set sizes. The presented results are for $\lambda = 10^{-5}$, and the training set sizes are from the range [1000, 2000, . . . 512000].

In Figure 1 are the experimental results. First, we plot the average time needed for subgradient computation by the TreeRSVM and the PairRSVM. It can be seen that the results are consistent with the computational complexity analysis, the proposed method scales much better than the one based on iterating over the pairs of training examples in subgradient and loss evaluations. Second, we compare the scalability of the different RankSVM implementations. As expected, TreeRank achieves orders of magnitude faster training times than the other alternatives. PRSVM could not be trained beyond 8000 examples due to large memory consumption. With 512000 training examples training SVM^{rank} took 83 hours, and training PairRSVM took 122 hours, whereas training TreeRSVM took only 18 minutes in the same setting. Finally, we plot the pairwise ranking errors, as measured on an independent test set of 20000 examples. PairRSVM is left out of the comparison, since it always reaches exactly the same solution as TreeRSVM. The results show that TreeRSVM and

SVM^{rank} have similar performance as expected, as does PRSVM which optimizes a squared version of the pairwise hinge loss.

5 Conclusion

In this work we have introduced an improved training algorithm for the linear RankSVM, allowing efficient training also in case of unrestricted utility scores. The experiments demonstrate orders of magnitude improvements in training time on large enough data sets.

Acknowledgment This work has been supported by the Academy of Finland.

References

1. Airola, A., Pahikkala, T., Salakoski, T.: Training linear ranking SVMs in linearithmic time using red-black trees. Pattern Recognition Letters (2011), In Press
2. Bayer, R.: Symmetric binary B-trees: Data structure and maintenance algorithms. Acta Informatica 1, 290–306 (1972)
3. Chapelle, O., Keerthi, S.S.: Efficient algorithms for ranking with SVMs. Information Retrieval 13, 201–215 (2010)
4. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. MIT Press (2001)
5. Freund, Y., Iyer, R., Schapire, R.E., Singer, Y.: An efficient boosting algorithm for combining preferences. Journal of Machine Learning Research 4, 933–969 (2003)
6. Fürnkranz, J., Hüllermeier, E.: Preference learning. Künstliche Intelligenz 19(1), 60–61 (2005)
7. Herbrich, R., Graepel, T., Obermayer, K.: Support vector learning for ordinal regression. In: 9th International Conference on Articial Neural Networks. pp. 97–102. Institute of Electrical Engineers (1999)
8. Joachims, T.: Optimizing search engines using clickthrough data. In: Hand, D., Keim, D., Ng, R. (eds.) 8th ACM SIGKDD Conference on Knowledge Discovery and Data Mining. pp. 133–142. ACM Press (2002)
9. Joachims, T.: Training linear SVMs in linear time. In: Eliassi-Rad, T., Ungar, L., Craven, M., Gunopulos, D. (eds.) 12th ACM SIGKDD conference on Knowledge discovery and data mining. pp. 217–226. ACM Press (2006)
10. Lewis, D.D., Yang, Y., Rose, T.G., Li, F.: RCV1: A new benchmark collection for text categorization research. Journal of Machine Learning Research 5, 361–397 (2004)
11. Pahikkala, T., Tsivtsivadze, E., Airola, A., Boberg, J., Järvinen, J.: An efficient algorithm for learning to rank from preference graphs. Machine Learning 75(1), 129–165 (2009)
12. Smola, A.J., Vishwanathan, S.V.N., Le, Q.: Bundle methods for machine learning. In: McCallum, A. (ed.) Advances in Neural Information Processing Systems 20. MIT Press (2007)
13. Teo, C.H., Vishwanathan, S.V., Smola, A., Le, Q.V.: Bundle methods for regularized risk minimization. Journal of Machine Learning Research 11, 311–365 (2010)