Large-scale Collaborative Ranking in Near-Linear Time

Liwei Wu
Depts. of Statistics and Computer Science
University of California, Davis
liwu@ucdavis.edu

Cho-Jui Hsieh
Depts. of Statistics and Computer Science
University of California, Davis
chohsieh@ucdavis.edu

James Sharpnack
Dept. of Statistics
University of California, Davis
jsharpna@ucdavis.edu

ABSTRACT

In this paper, we consider the Collaborative Ranking (CR) problem for recommendation systems. Given a set of pairwise preferences between items for each user, collaborative ranking can be used to rank un-rated items for each user, and this ranking can be naturally used for recommendation. It is observed that collaborative ranking algorithms usually achieve better performance since they directly minimize the ranking loss; however, they are rarely used in practice due to the poor scalability. All the existing CR algorithms have time complexity at least $O(|\Omega|r)$ per iteration, where r is the target rank and $|\Omega|$ is number of pairs which grows quadratically with number of ratings per user. For example, the Netflix data contains totally 20 billion rating pairs, and at this scale all the current algorithms have to work with significant subsampling, resulting in poor prediction on testing data.

In this paper, we propose a new collaborative ranking algorithm called Primal-CR that reduces the time complexity to $O(|\Omega|+d_1\bar{d}_2 r)$, where d_1 is number of users and $\bar{d}_2$ is the averaged number of items rated by a user. Note that $d_1\bar{d}_2$ is strictly smaller and often much smaller than $|\Omega|$.

Furthermore, by exploiting the fact that most data is in the form of numerical ratings instead of pairwise comparisons, we propose Primal-CR++ with $O(d_1\bar{d}_2(r + \log \bar{d}_2))$ time complexity. Both algorithms have better theoretical time complexity than existing approaches and also outperform existing approaches in terms of NDCG and pairwise error on real data sets. To the best of our knowledge, this is the first collaborative ranking algorithm capable of working on the full Netflix dataset using all the 20 billion rating pairs, and this leads to a model with much better recommendation compared with previous models trained on subsamples. Finally, compared with classical matrix factorization algorithm which also requires $O(d_1\bar{d}_2 r)$ time, our algorithm has almost the same efficiency while making much better recommendations since we consider the ranking loss.

KEYWORDS

Collaborative Ranking, Recommendation Systems, Large-Scale.

KDD'17, August 13–17, 2017, Halifax, NS, Canada.

DOI: http://dx.doi.org/10.1145/3097983.3098071

1 INTRODUCTION

In online retail and online content delivery applications, it is commonplace to have embedded recommendation systems– algorithms that recommend items to users based on previous user behaviors and ratings. Online retail companies develop sophisticated recommendation systems based on purchase behavior, item context, and shifting trends. The Netflix prize [2], in which competitors utilize user ratings to recommend movies, accelerated research in recommendation systems. While the winning submissions agglomerated several existing methods, one essential methodology, latent factor models, emerged as a critical component. The latent factor model means that the approximated rating for user i and item j is given by $u_i^\top v_j$ where u_i, v_j are k-dimensional vectors. One interpretation is that there are k latent topics and the approximated rating can be reconstructed as a combination of factor weights. By minimizing the square error loss of this reconstruction we arrive at the incomplete SVD,

$$\min_{U,V} \sum_{i,j\in\Omega} \left(R_{i,j} - u_i^\top v_j\right)^2 + \frac{\lambda}{2}(\|U\|_F^2 + \|V\|_F^2), \tag{1}$$

where Ω contains sampled indices of the rating matrix, R.

Often the performance of recommendation systems is not measured by the quality of rating prediction, but rather the ranking of the items that the system returns for a given user. The task of finding a ranking based on ratings or relative rankings is called Collaborative Ranking. Recommendation systems can be trained with ratings, that may be passively or actively collected, or by relative rankings, in which a user is asked to rank a number of items. A simple way to unify the framework is to convert the ratings into rankings by making pairwise comparisons of ratings. Specifically, the algorithm takes as input the pairwise comparisons, $Y_{i,j,k}$ for each user i and item pairs j, k. This approach confers several advantages. Users may have different standards for their ratings, some users are more generous with their ratings than others. This is known as the calibration drawback, and to deal with this we must make a departure from standard matrix factorization methods. Because we focus on ranking and not predicting ratings, we can expect improved performance when recommending the top items. Our goal in this paper is to provide a collaborative ranking algorithm that can scale to the size of the full Netflix dataset, a heretofore open problem.

The existing collaborative ranking algorithms, (for a summary see section 2), are limited by the number of observed ratings per user in the training data and cannot scale to massive datasets, therefore, making the recommendation results less accurate and less useful in practice. This motivates our algorithm, which can make use of the entire Netflix dataset without sub-sampling. Our contribution can be summarized below:

- For input data in the form of pairwise preference comparisons, we propose a new algorithm Primal-CR that alternatively minimizes latent factors using Newton's method in the primal space. By carefully designing the computation of gradient and Hessian vector product, our algorithm reduces the sample complexity per iteration to $O(|\Omega|+d_1\bar{d}_2 r)$, while the state-of-the-art approach [17] have $O(|\Omega|r)$ complexity. Here $|\Omega|$ (total number of pairs), is much larger than $d_1\bar{d}_2$ (d_1 is number of users and $\bar{d}_2$ is averaged number of items rated by a user). For the Netflix problem, $|\Omega| = 2 \times 10^{10}$ while $d_1\bar{d}_2 = 10^8$.
- For input data in the form of ratings, we can further exploit the structure to speedup the gradient and Hessian computation. The resulting algorithm, Primal-CR++, can further reduce the time complexity to $O(d_1\bar{d}_2(r + \log \bar{d}_2))$ per iteration. In this setting, our algorithm has time complexity near-linear to the input size, and have comparable speed with classical matrix factorization model that takes $O(d_1\bar{d}_2 r)$ time, while we can achieve much better recommendation by minimizing the ranking loss.

We show that our algorithms outperform existing algorithms on real world datasets and can be easily parallelized.

2 RELATED WORK

Collaborative filtering methodologies are summarized in [20] (see [6] for an early work). Among them, matrix factorization [13] has been widely used due to the success in the Netflix Prize. Many algorithms have been developed based on matrix factorization [5, 11, 18, 19, 22], and many scalable algorithms have been developed [8, 13]. However, they are not suitable for ranking top items for a user due to the fact that their goal is to minimize the mean-square error (MSE) instead of ranking loss. In fact, MSE is not a good metric for recommendation when we want to recommend the top K items to a user. This has been pointed out in several papers [1] which argue normalized discounted cumulative gain (NDCG) should be used instead of MSE, and our experimental results also confirm this finding by showing that minimizing the ranking loss results in better precision and NDCG compared with the traditional matrix factorization approach that is targeting squared error.

Ranking is a well studied problem, and there has been a long line of research focuses on learning one ranking function, which is called Learning to Rank. For example, RankSVM [12] is a well-known pair-wise model, and an efficient solver has been proposed in [4] for solving rankSVM. [3] is a list-wise model implemented using neural networks. Another class of point-wise models fit the ratings explicitly but has the issue of calibration drawback (see [10]).

The collaborative ranking (CR) problem is essentially trying to learn multiple rankings together, and several models and algorithms have been proposed in literature. The Cofirank algorithm [25], which tailors maximum margin matrix factorization [23] for collaborative ranking, is a point-wise model for CR, and is regarded as the performance benchmark for this task. If the ratings are 1-bit, a weighting scheme is proposed to improve the usual point-wise Matrix Factorization approach [15]. List-wise models for Learning to Rank can also be extended to many rankings setting, [21]. However it is still quite similar to a point-wise approach since they only consider the top-1 probabilities.

For pairwise models in collaborative ranking, it is well known that they do not encounter the calibration drawback as do point-wise models, but they are computationally intensive and cannot scale well to large data sets [21]. The scalability problem for pairwise models is mainly due to the fact that their time complexity is at least proportional to $|\Omega|$, the number of pairwise preference comparisons, which grows quadratically with number of rated items for each user. Recently, [17] proposed a new Collrank algorithm, and they showed that Collrank has better precision and NDCG as well as being much faster compared with other CR methods on real world datasets, including Bayesian Personalized Ranking (BPR) [19]. Unfortunately their scalability is still constrained by number of pairs, so they can only run on subsamples for large datasets, such as Netflix. In this paper, our algorithm Primal-CR and Primal-CR++ also belong to the family of pairwise models, but due to cleverly re-arranging the computation, we are able to have much better time complexity than existing ones, and as a result our algorithm can scale to very large datasets.

There are many other algorithms proposed for many rankings setting but none of these mentioned below can scale up to the extent of using all the ratings in the full Netflix data. There are a few using Bayesian frameworks to model the problem [19], [16], [24], the last of which requires many specified parameters. Another one proposed retargeted matrix factorization to get ranking by monotonically transforming the ratings [14]. [9] proposes a similar model without making generative assumptions on ratings besides assuming low-rank and correctness of the ranking order.

3 PROBLEM FORMULATION

We first formally define the collaborative ranking problem using the example of item recommender system. Assume we have d_1 users and d_2 items, the input data is given in the form of "for user i, item j is preferred over item k" and thus can be represented by a set of tuples (i, j, k). We use Ω to denote the set of observed tuples, and the observed pairwise preferences are denoted as $\{Y_{ijk} \mid (i, j, k) \in \Omega\}$, where $Y_{ijk} = 1$ denotes that item j is preferred over item k for a particular user i and $Y_{ijk} = -1$ to denote that item k is preferred over item j for user i.

The goal of collaborative ranking is to rank all the unseen items for each user i based on these partial observations, which can be done by fitting a scoring matrix $X \in \mathbb{R}^{d_1 \times d_2}$. If the scoring matrix has $X_{ij} > X_{ik}$, it implies that item j is preferred over item k by the particular user i and therefore we should give higher rank for item j than item k. After we estimate the scoring matrix X by solving the optimization problem described below, we can then recommend top k items for any particular user.

The Collaborative Ranking Model referred to in this paper is the one proposed recently in [17]. It belongs to the family of pairwise models for collaborative ranking because it uses pairwise training losses [1]. The model is given as

$$\min_X \sum_{(i,j,k)\in\Omega} \mathcal{L}(Y_{ijk}(X_{ij} - X_{ik})) + \lambda||X||_*, \tag{2}$$

where $\mathcal{L}(.)$ is the loss function, $\|X\|_*$ is the nuclear norm regularization defined by the sum of all the singular value of the matrix X, and λ is a regularization parameter. The ranking loss defined in the first term of (2) penalizes the pairs when $Y_{ijk} = 1$ but $X_{ij} - X_{ik}$ is positive but small, and penalizes even more when the difference is negative. The second term in the loss function is based on the assumption that there are only a small number of latent factors contributing to the users' preferences which is analogous to the idea behind incomplete SVD for matrix factorization mentioned in the introduction. In general we can use any loss function, but since $\mathcal{L}_2$-hinge loss defined as

$$\mathcal{L}(a) = \max(0, 1 - a)^2 \tag{3}$$

gives the best performance in practice [17] and enjoys many nice properties, such as smoothness and differentiable, we will focus on L_2-hinge loss in this paper. In fact, our first algorithm Primal-CR can be applied to any loss function, while Primal-CR++ can only be applied to L_2-hinge loss.

Despite the advantage of the objective function in equation (2) being convex, it is still not feasible for large-scale problems since d_1 and d_2 can be very large so that the scoring matrix X cannot be stored in memory, not to mention how to solve it. Therefore, in practice people usually transform (2) to a non-convex form by replacing $X = UV^T$, and in that case since $\|X\|_* = min_{X=U^TV} \frac{1}{2}(\|U\|_F^2 + \|V\|_F^2)$ [23], problem (2) can be reformulated as

$$\min_{U,V} \sum_{(i,j,k)\in\Omega} \mathcal{L}(Y_{ijk} \cdot u_i^T(v_j - v_k)) + \frac{\lambda}{2}(\|U\|_F^2 + \|V\|_F^2), \tag{4}$$

We use u_i and v_j denote columns of U and V respectively. Note that [17] also solves the non-convex form (4) in their experiments, and in the rest of the paper we will propose a faster algorithm for solving (4).

4 PROPOSED ALGORITHMS

4.1 Motivation and Overview

Although collaborative ranking assumes that input data is given in the form of pairwise comparisons, in reality almost all the datasets (Netflix, Yahoo-Music, MovieLens, etc) contain user ratings to items in the form of $\{R_{ij} \mid (i,j) \in \bar{\Omega}\}$, where $\bar{\Omega}$ is the subset of observed user-item pairs. Therefore, in practice we have to transform the rating-based data into pair-wise comparisons by generating all the item pairs rated by the same user:

$$\Omega = \{(i,j,k) \mid j,k \in \bar{\Omega}_i\}, \tag{5}$$

where $\bar{\Omega}_i := \{j \mid (i,j) \in \bar{\Omega}\}$ is the set of items rated by user i. Assume there are averagely $\bar{d_2}$ items rated by a user (i.e., $\bar{d_2} = \text{mean}(|\bar{\Omega}_i|)$), then the collaborative ranking problem will have $O(d_1 \bar{d_2}^2)$ pairs and thus the size of Ω grows quadratically.

Unfortunately, all the existing algorithms have $O(|\Omega|r)$ complexity, so they cannot scale to large number of items. For example, the AltSVM (or referred to asCollrank) Algorithm in [17] will run out of memory when we subsample 500 rated items per user on Netflix dataset since its implementation[1] stores all the pairs in memory and therefore requires $O(|\Omega|)$ memory. So it cannot be used for the full Netflix dataset which has more than *20 billion pairs* and requires 300GB memory space. To the best of our knowledge, no collaborative ranking algorithms have been applied to the full Netflix data set. But in real life, we hope to make use of as much information as possible to make better recommendation. As shown in our experiments later, using full training data instead of sub-sampling (such as selecting a fixed number of rated items per user) achieves higher prediction and recommendation accuracy for the same test data.

To overcome this scalability issue, we propose two novel algorithms for solving problem (4), and both of them significantly reduce the time complexity over existing methods. If the input file is in the form of $|\Omega|$ pairwise comparisons, our proposed algorithm, Primal-CR, can reduce the time and space complexity from $O(|\Omega|r)$ to $O(|\Omega| + d_1\bar{d_2}r)$, where $\bar{d_2}$ is the average number of items compared by one user. If the input data is given as user-item ratings (e.g., Netflix, Yahoo-Music), the complexity is reduced from $O(d_1\bar{d_2}^2 r)$ to $O(d_1\bar{d_2}r + d_1\bar{d_2}^2)$.

If the input file is given in ratings, we can further reduce the time complexity to $O(d_1\bar{d_2}r + d_1\bar{d_2}\log\bar{d_2})$ using exactly the same optimization algorithm but smarter ways to compute gradient and Hessian vector product. This time complexity is much smaller than the number of comparisons $|\Omega| = O(d_1\bar{d_2^2})$, and we call this algorithm Primal-CR++.

We will first introduce Primal-CR in Section 4.2, and then present Primal-CR++ in Section 4.3.

4.2 Primal-CR: the proposed algorithm for pairwise input data

Algorithm 1 Primal-CR/ Primal-CR++: General Framework

Input: $\Omega, \{Y_{ijk} : (i,j,k) \in \Omega\}, \lambda \in \mathbb{R}^+$ ▷ for Primal-CR
Input: $M \in \mathbb{R}^{d_1 \times d_2}, \lambda \in \mathbb{R}^+$ ▷ for Primal-CR++
Output: $U \in \mathbb{R}^{r \times d_1}$ and $V \in \mathbb{R}^{r \times d_2}$

```
Randomly initialize U, V from Gaussian Distribution
while not converged do
    procedure Fix U and update V
        while not converged do
            Apply truncated Newton update (Algorithm 2)
        end while
    end procedure
    procedure Fix V and update U
        while not converged do
            Apply truncated Newton update (Algorithm 2)
        end while
    end procedure
end while
return U, V                                   ▷ recover score matrix X
```

In the first setting, we consider the case where the pairwise comparisons $\{Y_{ijk} \mid (i,j,k) \in \Omega\}$ are given as input. To solve problem (4), we alternatively minimize U and V in the primal space (see Algorithm 1). First, we fix U and update V, and the subproblem

[1] Collrank code is available on https://github.com/dhpark22/collranking.

Algorithm 2 Truncated Newton Update for V (same procedure can be used for updating U)

Input: Current solution U, V
Output: V

Compute $g = \text{vec}(\nabla f(V))$
Let $H = \nabla^2 f(V)$ (do not explicitly compute H)
procedure LINEAR CONJUGATE GRADIENT(g, F)
    Initialize $\delta_0 = 0$
    $r_0 = H\delta_0 - g, p_0 = -r_0$
    for $k = 0, 1, ..., maxiter$ **do**
        Compute the Hessian-vector product $q = Hp_k$
        $\alpha_k = -r_k^T p_k / p_k^T q$
        $\delta_{k+1} = \delta_k + \alpha_k p_k$
        $r_{k+1} = r_k + \alpha_k q$
        if $||r_{k+1}||_2 < ||r_0||_2 \cdot 10^{-2}$ **then**
            break
        end if
        $\beta_{k+1} = (r_{k+1}q)/p_k^T q$
        $p_{k+1} = -r_{k+1} + \beta_{k+1} p_k$
    end for
    return δ
end procedure
$V = V - s\delta$ (stepsize s found by line search)
return U or V

for V while U is fixed can be written as follows:

$$V = \underset{V \in \mathbb{R}^{r \times d_2}}{\text{argmin}} \left\{ \frac{\lambda}{2} ||V||_F^2 + \sum_{(i,j,k) \in \Omega} \mathcal{L}(Y_{ijk} \cdot u_i^T (v_j - v_k)) \right\} := f(V) \tag{6}$$

In [17], this subproblem is solved by stochastic dual coordinate descent, which requires $O(|\Omega|r)$ time and $O(|\Omega|)$ space complexity. Furthermore, the objective function decreases for the dual problem sometimes does not imply the decrease of primal objective function value, which often results in slow convergence. We therefore propose to solve this subproblem for V using the primal truncated Newton method (Algorithm 2).

Newton method is a classical second-order optimization algorithm. For minimizing a vector-valued function $f(x)$, Newton method iteratively updates the solution by $x \leftarrow x - (\nabla^2 f(x))^{-1} \nabla f(x)$. However, the matrix inversion is usually hard to compute, so a truncated Newton method computes the update direction by solving the linear system $\nabla^2 f(x) a = \nabla f(x)$ up to a certain accuracy, usually using a linear conjugate gradient method. If we vectorized the problem for updating V in eq (6), the gradient is a (rd_2)-sized vector and the Hessian is an (rd_2)-by-(rd_2) matrix, so explicitly forming the Hessian is impossible. Below we discuss how to apply the truncated Newton method to solve our problem, and discuss efficient computations for each part.

Derivation of Gradient. When applying the truncated Newton method, the gradient $\nabla f(V)$ is a $\mathbb{R}^{r \times d_2}$ matrix and can be computed explicitly:

$$\nabla f(V) = \sum_{i=1}^{d_1} \sum_{(j,k) \in \Omega_i} \mathcal{L}'(Y_{ijk} \cdot u_i^T (v_j - v_k))(u_i e_j^T - u_i e_k^T) Y_{ijk} + \lambda V, \tag{7}$$

where $\nabla f(V) \in \mathbb{R}^{r \times d_2}$, $\Omega_i := \{(j,k) \mid (i,j,k) \in \Omega\}$ is the subset of pairs that associates with user i, and e_j is the indicator vector used to add the u_i vector to the j-th column of the output matrix. The first derivative for L_2-hinge loss function (3) is

$$\mathcal{L}'(a) = 2 \min(a - 1, 0) \tag{8}$$

For convenience, we define $g := \text{vec}(\nabla f(V))$ to be the vectorized form of gradient. One can easily see that computing g naively by going through all the pairwise comparisons (j,k) and adding up arrays is time-consuming and has $O(|\Omega|r)$ time complexity, which is the same with Collrank [17].

Fast computation for gradient. Fortunately, we can reduce the time complexity to $O(|\Omega| + d_1 \bar{d}_2 r)$ by smartly rearranging the computations, so that the time is only linear to $|\Omega|$ and r, but not to $|\Omega|r$. The method is described below.

First, for each i, the first term of (7) can be represented by

$$\sum_{(j,k) \in \Omega_i} \mathcal{L}'(Y_{ijk} \cdot u_i^T (v_j - v_k))(u_i e_j^T - u_i e_k^T) Y_{ijk} = \sum_{j \in \bar{d}_2(i)} t_j u_i e_j^T, \tag{9}$$

where $\bar{d}_2(i) := \{j \mid \exists k \text{ s.t. } (i,j,k) \in \Omega\}$ and t_j is some coefficient computed by summing over all the pairs in Ω_i. If we have t_j, the overall gradient can be computed by $O(\bar{d}_2(i)r)$ time for each i. To compute t_j, we first compute $u_i^T v_j$ for all $j \in \bar{d}_2(i)$ in $O(\bar{d}_2(i)r)$ time, and then go through all the (j,k) pairs while keep adding the coefficient related to this pair to t_j and t_k. Since there is no vector operations when we go through all pairs, this step only takes $O(\Omega_i)$ time. After getting all t_j, we can then conduct $\sum_{j \in \bar{d}_2(i)} t_j u_i e_j^T$ in $O(\bar{d}_2(i)r)$ time. Therefore, the overall complexity can be reduced to $O(|\Omega| + d_1 \bar{d}_2 r)$. The pseudo code is presented in Algorithm 3.

Algorithm 3 Primal-CR: efficient way to compute $\nabla f(V)$

Input: Ω, $\{Y_{ijk} : (i,j,k) \in \Omega\}$, $\lambda \in \mathbb{R}^+$, current variables U, V
Output: g, m ▷ $g \in \mathbb{R}^{d_2 r}$ is the gradient for $f(V)$

Initialize $g = 0$ ▷ $g \in \mathbb{R}^{r \times d_2}$
for $i = 1, 2, \ldots, d_1$ **do**
    for all $j \in \bar{d}_2(i)$ **do**
        precompute $u_i^T v_j$ and store in a vector m_i
    end for
    Initialize a zero array t of size d_2
    for all $(j,k) \in \bar{d}_2(i)$ **do**
        if $Y_{ijk}(m_i[j] - m_i[k]) < 1$ **then**
            $s = 2(Y_{ijk}(m_i[j] - m_i[k]) - 1)$
            $t[j]$ += $Y_{ijk} s$
            $t[k]$ −= $Y_{ijk} s$ ▷ O(1) time per for loop iteration
        end if
    end for
    for all $j \in \bar{d}_2(i)$ **do**
        $g[:, j]$ += $t[j] \cdot u_i$
    end for
end for
$g = vec(g + \lambda V)$ ▷ vectorize matrix $g \in \mathbb{R}^{r \times d_2}$
Form a sparse matrix $m = [m_1 \ldots m_{d_1}]$ ▷ m can be reused later
return g, m

Derivation of Hessian-vector product. Now we derive the Hessian $\nabla^2 f(V)$ for $f(V)$. We define $\nabla_j f(V) := \frac{\partial}{\partial v_j} f(V) \in \mathbb{R}^r$ and $\nabla^2_{j,k} f(V) := \frac{\partial^2}{\partial v_j \partial v_k} f(V) \in \mathbb{R}^{r\times r}$ in the following derivations. From the gradient derivation, we have

$$\nabla_j f(V) = \sum_{i:j\in\bar{d}_2(i)} \sum_{\substack{k\in\bar{d}_2(i)\\ k\neq j}} \mathcal{L}'(Y_{ijk}\cdot u_i^T(v_j - v_k))u_i Y_{ijk} + \lambda v_j.$$

Taking derivative again we can obtain

$$\nabla^2_{j,k} f(V) =$$

$$\begin{cases} \sum_{i:(j,k)\in\bar{d}_2(i)} \mathcal{L}''(Y_{ijk}\cdot u_i^T(v_j - v_k))(-u_i u_i^T) & \text{if } j \neq k \\ \sum_{i:j\in\bar{d}_2(i)} \sum_{k\in\bar{d}_2(i),k\neq j} \mathcal{L}''(Y_{ijk}\cdot u_i^T(v_j - v_k))u_i u_i^T + \lambda I_{r\times r} & \text{if } j = k \end{cases}$$

and the second derivative for $\mathcal{L}_2$ hinge loss function is given by:

$$\mathcal{L}''(a) = \begin{cases} 2 & \text{if } a \leq 1 \\ 0 & \text{if } a > 1. \end{cases} \tag{10}$$

Note that if we write the full Hessian H as a $(d_2 r)$ by $(d_2 r)$ matrix, then $\nabla^2_{j,k} f(V)$ is an $r \times r$ block in H, where there are totally d_2^2 of these blocks. In the CG update for solving $H^{-1}g$, we only need to compute $H \cdot a$ for some $a \in \mathbb{R}^{d_2 r}$. For convenience, we also partition this a into d_2 blocks, each subvector a_j has size r, so $a = [a_1; \cdots; a_j]$. Similarly we can use subscript to denote the subarray $(H \cdot a)_j$ of the array $H \cdot a$, which becomes

$$\begin{aligned}(H\cdot a)_j &= \sum_{k\neq j} \nabla^2_{j,k} f(V)\cdot a_k + \nabla^2_{j,j} f(V)\cdot a_j \\ &= \lambda a_j + \sum_{i:j\in\bar{d}_2(i)} u_i \sum_{\substack{k\in\bar{d}_2(i)\\ k\neq j}} \mathcal{L}''(Y_{ijk}\cdot u_i^T(v_j - v_k))(u_i^T a_j - u_i^T a_k).\end{aligned}$$

Therefore, we have

$$\begin{aligned}H\cdot a &= \sum_j E_j (H\cdot a)_j \\ &= \lambda a + \sum_i \sum_{j\in\bar{d}_2(i)} E_j u_i \sum_{\substack{k\in\bar{d}_2(i)\\ k\neq j}} \mathcal{L}''(Y_{ijk}\cdot u_i^T(v_j - v_k))(u_i^T a_j - u_i^T a_k)\end{aligned} \tag{11}$$

where E_j is the projection matrix to the j-th block, indicating that we are only adding $(H \cdot a)_j$ to the j-th block of matrix, and setting 0 elsewhere.

Fast computation for Hessian-vector product . Similar to the case of gradient computation, using a naive way to compute $H \cdot a$ requires $O(|\Omega| r)$ time since we need to go through all the (i, j, k) tuples, and each of them requires $O(r)$ time. However, we can apply the similar trick in gradient computation to reduce the time complexity to $O(|\Omega| + d_1 \bar{d}_2 r)$ by pre-computing $u_i^T a_j$ and caching the coefficient using the array t. The detailed algorithm is given in Algorithm 4.

Note that in Algorithm 4, we can reuse the m (sparse array storing the current prediction) which has been pre-computed in the gradient computation (Algorithm 3), and that will cost only $O(d_1 \bar{d}_2)$ memory. Even without storing the m matrix, we can compute m in the loop of line 4 in Algorithm 4, which will not increase the overall computational complexity.

Algorithm 4 Primal-CR: efficient way to compute Hessian vector product

Input: $\Omega, \{Y_{ijk} : (i, j, k) \in \Omega\}, \lambda \in \mathbb{R}^+, a \in \mathbb{R}^{d_2 r}, m, U, V$
Output: Ha ▷ $Ha \in \mathbb{R}^{d_2 r}$ is needed in Linear CG

1: $Ha = 0$ ▷ $Ha \in \mathbb{R}^{d_2 r}$
2: **for** $i = 1, 2, \ldots, d_1$ **do**
3: **for all** $j \in \bar{d}_2(i)$ **do**
4: precompute $u_i^T a_j$ and store it in array b
5: **end for**
6: Initialize a zero array t of size d_2
7: **for all** $(j, k) \in \bar{d}_2(i)$ **do**
8: **if** $Y_{ijk}(m_i[j] - m_i[k]) < 1.0$ **then**
9: $s_{jk} = 2.0 \cdot (b[j] - b[k])$
10: $t[j]$ += s_{jk}
11: $t[k]$ −= s_{jk} ▷ O(1) time per for loop iteration
12: **end if**
13: **end for**
14: **for all** $j \in \bar{d}_2(i)$ **do**
15: $(Ha)[(p-1)\cdot r + 1 : p \cdot r]$ += $t[j] \cdot u_i$
16: **end for**
17: **end for**
18: **return** Ha

Fix V and Update U. After updating V by truncated Newton, we need to fix V and update U. The subproblem for U can be written as:

$$U = \underset{U\in\mathbb{R}^{r\times d_2}}{\operatorname{argmin}} \{\frac{\lambda}{2}||U||_F^2 + \sum_{i=1}^{d_1} \sum_{(j,k)\in\bar{d}_2(i)} \mathcal{L}(Y_{ijk}\cdot u_i^T(v_j - v_k))\} \tag{12}$$

Since u_i, the i-th column of U, is independent from the rest of columns, equation 12 can be decomposed into d_1 independent problems for u_i:

$$u_i = \underset{u\in\mathbb{R}^r}{\operatorname{argmin}} \frac{\lambda}{2}||u||_2^2 + \sum_{(j,k)\in\bar{d}_2(i)} \mathcal{L}(Y_{ijk}\cdot u^T(v_j - v_k)) := h(u) \tag{13}$$

Eq (13) is equivalent to an r-dimensional rankSVM problem. Since r is usually small, the problems are easy to solve. In fact, we can directly apply an efficient rankSVM algorithm proposed in [4] to solve each r-dimensional rankSVM problem. This algorithm requires $O(|\Omega_i| + r|\bar{d}_2(i)|)$ time for solving each subproblem with respect to u_i, so the overall complexity is $O(|\Omega| + r d_1 \bar{d}_2)$ time per iteration.

Summary of time and space complexity. When updating V, we first compute gradient by Algorithm 3, which takes $O(|\Omega| + d_1 \bar{d}_2 r)$ time, and each Hessian-vector product in 4 also takes the same time. The updates for U takes the same time complexity with updating V, so the overall time complexity is $O(|\Omega| + d_1 \bar{d}_2 r)$ per iteration. The whole algorithm only needs to store size $d_1 \times r$ and $d_2 \times r$ matrices for gradient and conjugate gradient method. The m matrix in Algorithm 3 is not needed, but in practice we find it can speedup the code by around 25%, and it only takes $d_1 \bar{d}_2 \leq |\Omega|$ memory space (less than the input size). Therefore, our algorithm is very memory-efficient.

Before going to Primal-CR++, we discuss the time complexity of Primal-CR when the input data is the user-item rating matrix. Assume $\bar{d}_2$ is the averaged number of rated items per user, then there will be $|\Omega| = O(d_1\bar{d}_2^2)$ pairs, leading to $O(d_1\bar{d}_2^2 + d_1\bar{d}_2 r)$ time complexity for Primal-CR. This is much better than the $O(d_1\bar{d}_2^2 r)$ complexity for all the existing algorithms.

Algorithm 5 Primal-CR++: compute gradient part for $f(V)$

Input: $M \in \mathbb{R}^{d_1 \times d_2}$, $\lambda \in \mathbb{R}^+$, current U, V
Output: g ▷ $g \in \mathbb{R}^{d_2 r}$ is the gradient for $f(V)$

Initialize $g = 0$ ▷ $g \in \mathbb{R}^{r \times d_2}$
for $i = 1, 2, ..., d_1$ **do**
  Let $\bar{d}_2 = |\bar{d}_2(i)|$ and $r_j = R_{i,j}$ for all j.
  Compute $m[j] = u_i^T v_j$ for all $j \in \bar{d}_2(i)$
  Sort $\bar{d}_2(i)$ according to the ascending order of m_i, so $m[\pi(1)] \le \cdots \le m[\pi(\bar{d}_2)]$
  Initialize $s[1], \ldots, s[L]$ and $c[1], \ldots, c[L]$ with 0
  (Store in segment tree or Fenwick tree.)
  $p \leftarrow 1$
  for all $j = 1, \ldots, \bar{d}_2$ **do**
    while $m[\pi(p)] \le m_j + 1$ **do**
      $s[r_{\pi(p)}] += m[\pi(p)]$, $c[r_{\pi(p)}] += 1$
      $p += 1$
    end while
    $S = \sum_{\ell \ge r_{\pi(p)}} s[\ell]$, $C = \sum_{\ell \ge r_{\pi(p)}} c[\ell]$
    $t^+[\pi(j)] = 2(C \cdot (m[\pi(j)] + 1) - S)$
  end for
  Do another scan j from $\bar{d}_2$ to 1 to compute $t^-[\pi(j)]$ for all j
  $g[:, j] += (t^+[j] + t^-[j]) \cdot u_i$ for all j
end for
$g = vec(g + \lambda V)$ ▷ vectorize matrix $g \in \mathbb{R}^{r \times d_2}$
return g

4.3 Primal-CR++: the proposed algorithm for rating data

Now we discuss a more realistic scenario, where the input data is a rating matrix $\{R_{ij} \mid (i,j) \in \bar{\Omega}\}$ and $\bar{\Omega}$ is the observed set of user-item ratings. We assume there are only L levels of ratings, so $R_{ij} \in \{1, 2, \ldots, L\}$. Also, we use $\bar{d}_2(i) := \{j \mid (i,j) \in \bar{\Omega}\}$ to denote the rated items for user i.

Given this data, the goal is to solve the collaborative ranking problem (4) with all the pairwise comparisons in the rating dataset as defined in (5). There are totally $O(d_1\bar{d}_2^2)$ pairs, and the question is: Can we have an algorithm with near-linear time with respect to number of observed ratings $|\bar{\Omega}| = d_1\bar{d}_2$? We answer this question in the affirmative by proposing Primal-CR++, a near-linear time algorithm for solving problem (4) with L2-hinge loss.

The algorithm of Primal-CR++ is exactly the same with Primal-CR, but we use a smarter algorithm to compute gradient and Hessian vector product in near-linear time, by exploiting the structure of the input data.

We first discuss how to speed up the gradient computation of (7), where the main computation is to compute (9) for each i. When the loss function is L2-hinge loss, we can explicitly write down the coefficients t_j in (9) by

$$t_j = \sum_{k \in \bar{d}_2(i)} 2(m_j - m_k - Y_{ijk}) I[Y_{ijk}(m_j - m_k) \le 1], \tag{14}$$

where $m_j := u_i^T v_j$ and $I[\cdot]$ is an indicator function such that $I[a \le b] = 1$ if $a \le b$, and $I[a \le b] = 0$ otherwise. By splitting the cases of $Y_{ijk} = 1$ and $Y_{ijk} = -1$, we get

$$\begin{aligned} t_j &= t_j^+ + t_j^- \\ &= \sum_{\substack{k \in \bar{d}_2(i) \\ m_k \le m_j + 1,\ Y_{ijk} = -1}} 2(m_j - m_k + 1) + \sum_{\substack{k \in \bar{d}_2(i) \\ m_k \ge m_j - 1,\ Y_{ijk} = 1}} 2(m_j - m_k - 1). \end{aligned} \tag{15}$$

Assume the indexes in $\bar{d}_2(i)$ are sorted by the the ascending order of m_j. Then we can scan from left to right, and maintain the current accumulated sum $s_1, ..., s_L$ and the current index counts $c_1, ..., c_L$ for each rating level. If the current pointer is p, then

$$s_\ell[p] = \sum_{j : m_j \le p, R_{ij} = \ell} m_j \ \text{ and } \ c_\ell[p] = |\{j : m_j \le p, R_{ij} = \ell\}|.$$

Since we scan from left to right, these numbers can be maintained in constant time at each step. Now assume we scan over the numbers $m_1 + 1, m_2 + 1, \ldots$, then at each point we can compute

$$t_j^+ = \sum_{\ell = R_{i,j}+1}^{L} 2\{(m_j + 1)c_\ell[m_j + 1] - s_\ell[m_j + 1]\},$$

which can be computed in $O(L)$ time.

Although we observe that $O(L)$ time is already small in practice (since L usually smaller than 10), in the following we show there is a way to remove the dependency on L by using a simple Fenwick tree [7], F+tree [26] or segment tree. If we store the set $\{s_1, \ldots, s_L\}$ in Fenwick tree, then each query of $\sum_{\ell \ge r} s_i$ can be done in in $O(\log L)$ time, and since each step we only need to change one element into the set, the updating time is also $O(\log L)$. Note that t_j^- can be computed in the same way by scanning from largest m_j to the smallest one.

To sum up, the algorithm first computes all m_j in $O(\bar{d}_2 r)$ time, then sort these numbers using $O(\bar{d}_2 \log \bar{d}_2)$ time, and then compute t_j for all j using two linear scans in $O(\bar{d}_2 \log L)$ time. Here $\log L$ is dominated by $\log \bar{d}_2$ since L can be the number of unique rating levels in the current set $\bar{d}_2(i)$. Therefore, after computing this for all users $i = 1, \ldots, d_1$, the time complexity for computing gradient is

$$O(d_1\bar{d}_2 \log \bar{d}_2 + d_1\bar{d}_2 r) = O(|\bar{\Omega}|(\log \bar{d}_2 + r)).$$

A similar procedure can also be used for computing the Hessian-vector product, and the computation of updating U with fixed V is simplier since the problem becomes decomposable to d_1 independent problems, see eq (13). Due to the page limit we omit the details here; interesting readers can check our code on github.

Compared with the classical matrix factorization, where both ALS and SGD requires $O(|\bar{\Omega}|r)$ time per iteration [13], our algorithm has almost the same complexity, since $\log \bar{d}_2$ is usually smaller than r (typically $r = 100$). Also, since all the temporary memory when computing user i can be released immediately, the

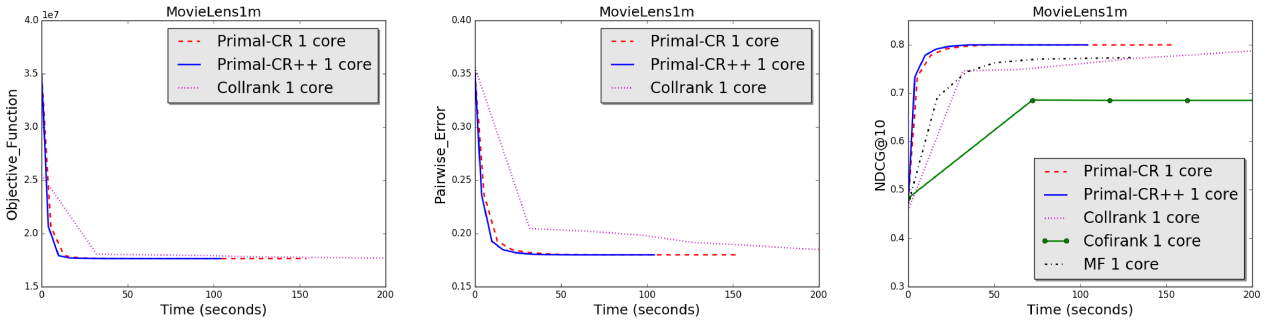

Figure 1: Comparing Primal-CR, Primal-CR++ and Collrank, MovieLens1m data, 200 ratings/user, rank 100, lambda = 5000

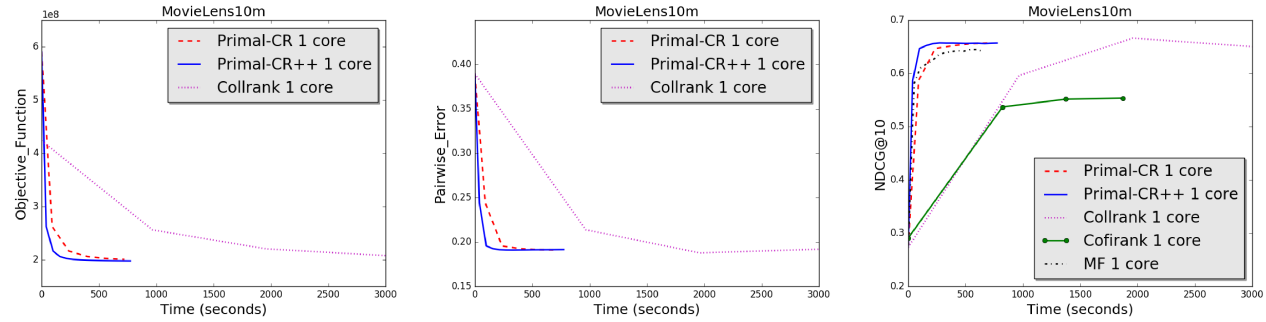

Figure 2: Comparing Primal-CR, Primal-CR++ and Collrank, MovieLens10m data, 500 ratings/user, rank 100, lambda = 7000

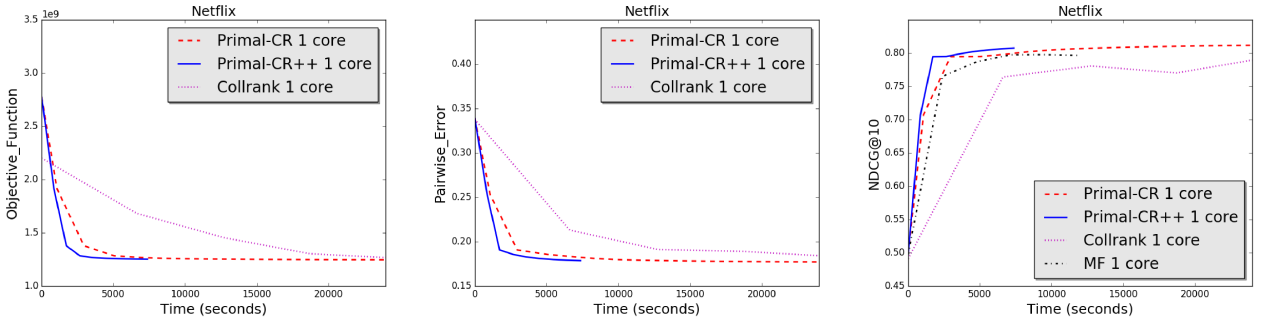

Figure 3: Comparing Primal-CR, Primal-CR++ and Collrank, Netflix data, 200 ratings/user, rank 100, lambda = 10000

only memory cost is still the same with Primal-CR++, which is $O(d_1 r + d_2 r)$.

4.4 Parallelization

Updating U while fixing V can be parallelized easily because each column of U is independent and we can actually solve d_1 independent subproblems at the same time. For the other side, updating V while fixing U can also be parallelized by parallelizing "computing g" part and "computing Ha" part respectively. We implemented the algorithm using parallel computing techniques in Julia by computing g and Ha distributedly and summing them up in the end. We show in section 5.2 that our parallel version of the proposed new algorithm works better than the paralleled version of Collrank algorithm [17].

5 EXPERIMENTS

In this section, we test the performance of our proposed algorithms Primal-CR and Primal-CR++ on real world datasets, and compare with existing methods. All experiments are conducted on the UC Davis Illidan server with an Intel Xeon E5-2640 2.40GHz CPU and 64G RAM. We compare the following methods:

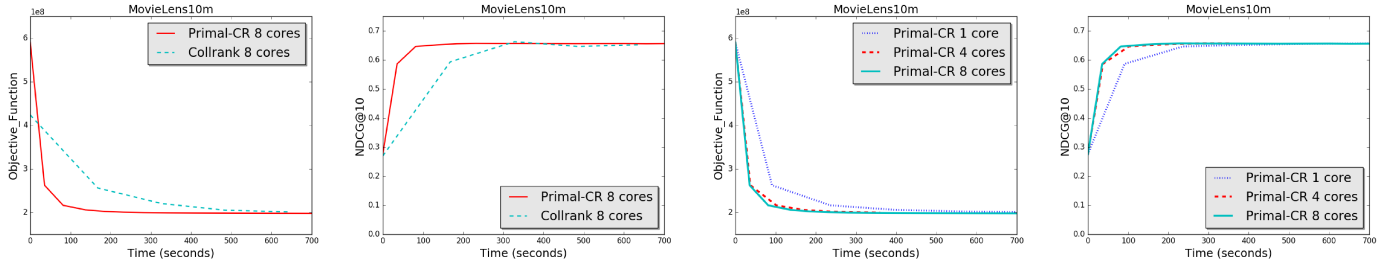

Figure 4: Comparing parallel version of Primal-CR and Collrank and Speedup of Primal-CR, MovieLens10m data, 500 ratings/user, rank 100, lambda = 7000

- Primal-CR and Primal-CR++: our proposed methods implemented in Julia. [2]
- Collrank: the collaborative ranking algorithm proposed in [17]. We use the C++ code released by the authors, and they parallelized their algorithm using OpenMP.
- Cofirank: the classical collaborative ranking algorithm proposed in [25]. We use the C++ code released by the authors.
- MF: the classical matrix factorization model in (1) solved by SGD [13].

We used three data sets (MovieLens1m, Movielens10m, Netflix data) to compare these algorithms. The dataset statistics are summarized in Table 1. The regularization parameter λ used for each datasets are chosen by a random sampled validation set. For the pair-wise based algorithms, we covert the ratings into pair-wise comparisons, by saying that item j is preferred over item k by user i if user i gives a higher rating to item j over item k, and there will be no pair between two items if they have the same rating.

We compare the algorithms in the following three different ways:

- Objective function: since Collrank, Primal-CR, Primal-CR++ have the same objective function, we can compare the convergence speed in terms of the objective function (4) with squared hinge loss.
- Predicted pairwise error: the proportion of pairwise preference comparisons that we predicted correctly out of all the pairwise comparisons in the testing data:

$$\text{pairwise error} = \frac{1}{|\mathcal{T}|} \sum_{\substack{(i,j,k)\in\mathcal{T} \\ Y_{ijk}=1}} 1(X_{ij} > X_{ik}), \tag{16}$$

  where $\mathcal{T}$ represents the test data set and $|\mathcal{T}|$ denotes the size of test data set.
- NDCG@k: a standard performance measure of ranking, defined as:

$$\text{NDCG@}k = \frac{1}{d_1} \sum_{i=1}^{d_1} \frac{\text{DCG@}k(i, \pi_i)}{\text{DCG@}k(i, \pi_i^*)}, \tag{17}$$

  where i represents i-th user and

$$\text{DCG@}k(i, \pi_i) = \sum_{l=1}^{k} \frac{2^{M_i \pi_i(l)} - 1}{log_2(l+1)}. \tag{18}$$

In the DCG definition, $\pi_i(l)$ represents the index of the l-th ranked item for user i in test data based on the score matrix $X = U^T V$ generated, M is the rating matrix and M_{ij} is the rating given to item j by user i. π_i^* is the ordering provided by the underlying ground truth of the rating.

5.1 Compare single thread versions using the same subsamples

Since Collrank cannot scale to the full dataset of Movielens10m and Netflix, we sub-sample data using the same approach in their paper [17] and compare all the methods using the smaller training sets. More specifically, for each data set, we subsampled N ratings for training data and used the rest of ratings as test data. For this subsampled data, we discard users with less than $N + 10$ ratings, since we need at least 10 ratings for test data to compute the NDCG@10.

As shown in Figure 1, 2, 3, both Primal-CR and Primal-CR++ perform considerably better than the existing Collrank algorithm. As data size increases, the performance gap becomes larger. As one can see, for Netflix data where $N = 200$, the speedup is more than 10 times compared to Collrank.

For Cofirank, we observe that it is even slower than Collrank, which confirms the experiments conducted in [17]. Furthermore, Cofirank cannot scale to larger datasets, so we omit the results in Figure 2 and 3.

We also include the classical matrix factorization algorithm in the NDCG comparisons. As shown in our complexity analysis, our proposed algorithms are competitive with MF in terms of speed, and MF is much faster than other collaborative ranking algorithms. Also, we observe that MF converges to a slightly worse solution in MovieLens10m and Netflix datasets, and converges to a much worse solution in MovieLens1m. The reason is that MF minimizes a simple mean square error, while our algorithms are minimizing ranking loss. Based on the experimental results, our algorithm Primal-CR++ should be able to replace MF in many real world recommender systems.

5.2 Compare parallel versions

Since Collrank can be implemented in a parallel fashion, we also implemented the parallel version of our algorithm in Julia. We want to show our algorithm scales up well and is still much faster than Collrank in the multi-core shared memory setting. As shown

[2] Our code is available on https://github.com/wuliwei9278/ml-1m.

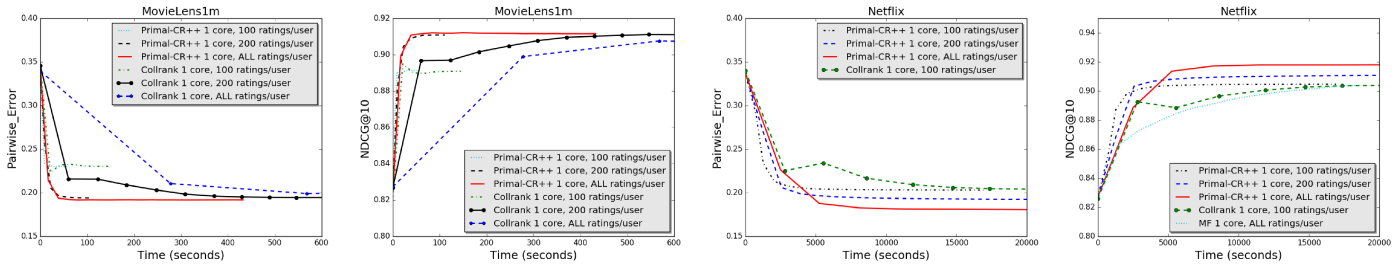

Figure 5: Varies number of ratings per user in training data, MovieLens1m and Netflix data, rank 100

	MovieLens1m	Movielens10m	Netflix
Users	6,040	71,567	2,649,430
Items	3,952	65,134	17,771
Ratings	1,000,209	10,000,054	99,072,112
λ	5,000	7,000	10,000

Table 1: Datasets used for experiments

# cores	1	4	8
Speedup for Primal-CR	1x	2.46x	3.12x
Speedup for Collrank	1x	2.95x	3.47x

Table 2: Scability of Primal-CR and Collrank on Movielens10m

in Figure 4, Primal-CR is still much faster than Collrank when 8 cores are used. Comparing our Primal-CR algorithm in 1 core, 4 cores and 8 cores on the same machine in Figure 4, the speedup is desirable. The speedup of Primal-CR and Collrank is summarized in the Table 2. One can see from the table that the speedup of our Primal-CR algorithm is comparable to Collrank.

5.3 Performance using Full Training Data

Due to the $O(|\Omega|k)$ complexity, existing algorithms cannot deal with large number of pairs, so they always sub-sample a limited number of pairs per user when solving MovieLens10m or Netflix data. For example, for Collrank, the authors fixed number of ratings per user in training as N and only reported N up to 100 for Netflix data. When we tried to apply their code for $N = 200$, the algorithm gets very slow and reports memory error for $N = 500$.

Using our algorithm, we have the ability to solve the full Netflix problem, so a natural question to ask is: Does using more training data help us predict and recommend better? The answer is yes! We conduct the following experiments to verify this: For all the users with more than 20 ratings, we randomly choose 10 ratings as test data and out of the rest ratings we randomly choose up to C ratings per user as training data. One can see in Figure 5, for the same test data, more training data leads to better prediction performance in terms of pairwise error and NDCG. Using all available ratings ($C = d_2$) gives lowest pairwise error and highest NDCG@10, using up to 200 ratings per user ($C = 200$) gives second lowest pairwise error and second highest NDCG@10, and using up to 100 ratings per user ($C = 100$) has the highest pairwise error and lowest NDCG@10. Similar phenomenon is observed for Netflix data in Figure 5. Collrank code does not work for $C = 200$ and $C = d_2$ and even for $C = 100$, it takes more than $20,000$ secs to converge while our Primal-CR++ takes less than $5,000$ secs for the full Netflix data. The speedup of our algorithm will be even more for a larger C or larger data size d_1 and d_2. We tried to create input file without subsampling for Collrank, we created 344GB input data file and Collrank reported memory error message "Segmentation Fault". We also tried $C = 200$, still got the same error message. It is possible to implement Collrank algorithm by directly working on the rating data, but the time complexity remains the same, so it is clear that our proposed Primal-CR and Primal-CR++ algorithms are much faster.

To the best of our knowledge, our algorithm is the first ranking-based algorithm that can scale to full Netflix data set using a single core, and without sub-sampling. Our proposed algorithm makes the Collaborative Ranking Model in (4) a clear better choice for large-scale recommendation system over standard Matrix Factorization techniques, since we have the same scalability but achieve much better accuracy. Also, our experiments suggest that in practice, when we are given a set of training data, we should try to use all the training data instead of doing sub-sampling as existing algorithms do, and only Primal-CR and Primal-CR++ can scale up to all the ratings.

6 CONCLUSIONS

We considered the collaborative ranking problem setting in which a low-rank matrix is fitted to the data in the form of pairwise comparisons or numerical ratings. We proposed our new optimization algorithms Primal-CR and Primal-CR++ where the time complexity is much better than all the existing approaches. We showed that our algorithms are much faster than state-of-the-art collaborative ranking algorithms on real data sets (MovieLens1m, Movielens10m and Netflix) using same subsampling scheme, and moreover our algorithm is the only one that can scale to the full Movielens10m and Netflix data. We observed that our algorithm has the same efficiency with matrix factorization, while achieving better NDCG since we minimize ranking loss. As a result, we expect our algorithm to be able to replace matrix factorization in many real applications.

REFERENCES

[1] Suhrid Balakrishnan and Sumit Chopra. 2012. Collaborative ranking. In *Proceedings of the fifth ACM international conference on Web search and data mining*. ACM, 143–152.

[2] James Bennett, Stan Lanning, and Netflix Netflix. 2007. The Netflix Prize. In *In KDD Cup and Workshop in conjunction with KDD*.

[3] Zhe Cao, Tao Qin, Tie-Yan Liu, Ming-Feng Tsai, and Hang Li. 2007. Learning to rank: from pairwise approach to listwise approach. In *Proceedings of the 24th international conference on Machine learning*. ACM, 129–136.

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

[5] Kai-Yang Chiang, Cho-Jui Hsieh, and Inderjit S Dhillon. 2015. Matrix completion with noisy side information. In *Advances in Neural Information Processing Systems*. 3447–3455.

[6] Mukund Deshpande and George Karypis. 2004. Item-based top-n recommendation algorithms. *ACM Transactions on Information Systems (TOIS)* 22, 1 (2004), 143–177.

[7] P. M. Fenwick. 1994. A new data structure for cumulative frequency tables. *Software: Practice and Experience* (1994).

[8] Rainer Gemulla, Erik Nijkamp, Peter J Haas, and Yannis Sismanis. 2011. Large-scale matrix factorization with distributed stochastic gradient descent. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 69–77.

[9] Suriya Gunasekar, Oluwasanmi O Koyejo, and Joydeep Ghosh. 2016. Preference Completion from Partial Rankings. In *Advances in Neural Information Processing Systems*. 1370–1378.

[10] Severin Hacker and Luis Von Ahn. 2009. Matchin: eliciting user preferences with an online game. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 1207–1216.

[11] Cho-Jui Hsieh, Nagarajan Natarajan, and Inderjit S Dhillon. 2015. PU Learning for Matrix Completion. In *ICML*.

[12] T. Joachims. 2002. Optimizing search engines using clickthrough data. In *ACM SIGKDD*.

[13] Yehuda Koren, Robert Bell, and Chris Volinsky. 2009. Matrix factorization techniques for recommender systems. *Computer* 42, 8 (2009).

[14] Oluwasanmi Koyejo, Sreangsu Acharyya, and Joydeep Ghosh. 2013. Retargeted matrix factorization for collaborative filtering. In *Proceedings of the 7th ACM conference on Recommender systems*. ACM, 49–56.

[15] Rong Pan, Yunhong Zhou, Bin Cao, Nathan N Liu, Rajan Lukose, Martin Scholz, and Qiang Yang. 2008. One-class collaborative filtering. In *Data Mining, 2008. ICDM'08. Eighth IEEE International Conference on*. IEEE, 502–511.

[16] Weike Pan and Li Chen. 2013. GBPR: Group Preference Based Bayesian Personalized Ranking for One-Class Collaborative Filtering.. In *IJCAI*, Vol. 13. 2691–2697.

[17] Dohyung Park, Joe Neeman, Jin Zhang, Sujay Sanghavi, and Inderjit S Dhillon. 2015. Preference Completion: Large-scale Collaborative Ranking from Pairwise Comparisons. *stat* 1050 (2015), 16.

[18] Steffen Rendle. 2010. Factorization machines. In *Data Mining (ICDM), 2010 IEEE 10th International Conference on*. IEEE, 995–1000.

[19] Steffen Rendle, Christoph Freudenthaler, Zeno Gantner, and Lars Schmidt-Thieme. 2009. BPR: Bayesian personalized ranking from implicit feedback. In *Proceedings of the twenty-fifth conference on uncertainty in artificial intelligence*. AUAI Press, 452–461.

[20] J Ben Schafer, Dan Frankowski, Jon Herlocker, and Shilad Sen. 2007. Collaborative filtering recommender systems. In *The adaptive web*. Springer, 291–324.

[21] Yue Shi, Martha Larson, and Alan Hanjalic. 2010. List-wise learning to rank with matrix factorization for collaborative filtering. In *Proceedings of the fourth ACM conference on Recommender systems*. ACM, 269–272.

[22] Si Si, Kai-Yang Chiang, Cho-Jui Hsieh, Nikhil Rao, and Inderjit S Dhillon. 2016. Goal-directed inductive matrix completion. In *KDD*.

[23] Nathan Srebro, Jason DM Rennie, and Tommi S Jaakkola. 2004. Maximum-Margin Matrix Factorization.. In *NIPS*, Vol. 17. 1329–1336.

[24] Maksims Volkovs and Richard S Zemel. 2012. Collaborative ranking with 17 parameters. In *Advances in Neural Information Processing Systems*. 2294–2302.

[25] Markus Weimer, Alexandros Karatzoglou, Quoc Viet Le, and Alex Smola. 2007. Maximum margin matrix factorization for collaborative ranking. *Advances in neural information processing systems* (2007), 1–8.

[26] Hsiang-Fu Yu, Cho-Jui Hsieh, Hyokun Yun, SVN Vishwanathan, and Inderjit S Dhillon. 2015. A scalable asynchronous distributed algorithm for topic modeling. In *Proceedings of the 24th International Conference on World Wide Web*. ACM, 1340–1350.