Large-scale Collaborative Ranking in Near-Linear Time

Liwei Wu
Depts of Statistics and Computer Science
UC Davis

KDD'17, Halifax, Canada
August 13-17, 2017

Joint work with Cho-Jui Hsieh and James Sharpnack

Recommender Systems: Netflix Problem

Rating Matrix

Items

Users

	Movie 1	Movie 2								Movie 10	Movie 11
User 1	1			5			3		5		2
User 2		2		3			5		2	5	
User 3					3	?	5		3		
User 4	2		5			3		4		2	
User 5				5			5				1
User 6		5			1				5		
User 7	1			1				2			4

Matrix Factorization Approach $R \approx WH^T$

H^T

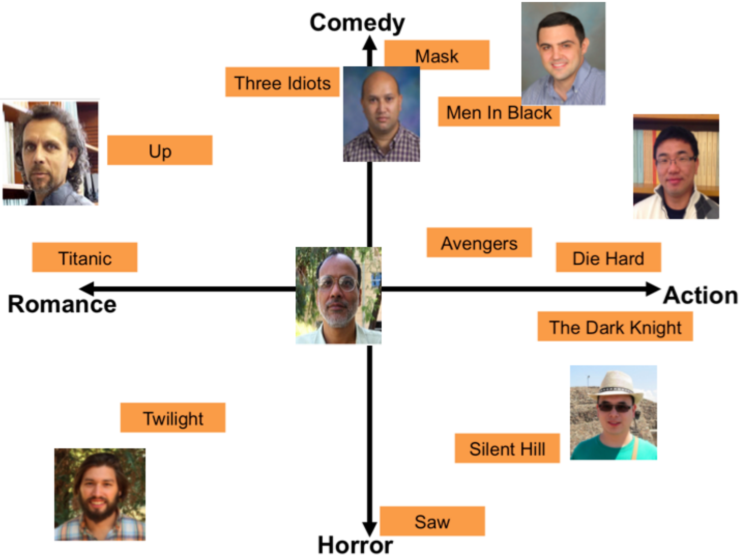
-0.07	-0.11	-0.53	-0.46	-0.06	-0.05	-0.53	-0.07	-0.35	-0.19	-0.14
0.13	-0.42	0.45	0.17	-0.25	-0.17	-0.18	0.27	-0.59	0.05	0.14
-0.21	-0.43	-0.23	0.16	0.08	0.17	0.57	-0.39	-0.37	-0.08	-0.15

W

-8.72	0.03	-1.03
-7.56	-0.79	0.62
-4.07	-3.95	2.55
-3.52	3.73	-3.32
-7.78	2.34	2.33
-2.44	-5.29	-3.92
-1.78	1.90	-1.68

1			5			3		5		2
	2		3			5		2	5	
				3	?	5		3		
2		5			3		4		2	
			5			5				1
	5			1				5		
1			1				2			4

Collaborative Filtering: Latent Factor Model



Collaborative Filtering: Matrix Factorization Approach

Latent Factor model fit the ratings directly in the objective function:

$$\min_{\substack{W \in \mathbb{R}^{d_1 \times r} \\ H \in \mathbb{R}^{d_2 \times r}}} \sum_{(i,j) \in \Omega} (R_{ij} - \boldsymbol{w}_i^T \boldsymbol{h}_j)^2 + \frac{\lambda}{2} \left(\|W\|_F^2 + \|H\|_F^2 \right),$$

- $\Omega = \{(i,j) \mid R_{ij} \text{ is observed}\}$
- Regularized terms to avoid over-fitting

Criticisms:

- Calibration drawback: users have different standards for their ratings
- Performance measured by quality of rating prediction, not the ranking of items for each user

Collaborative Filtering: Collaborative Ranking Approach

- Focus on ranking of items rather than ratings in the model
- Performance measured by ranking order of top k items for each user
  How?

Collaborative Filtering: Collaborative Ranking Approach

- Focus on ranking of items rather than ratings in the model
- Performance measured by ranking order of top k items for each user

  How?

- Given d_1 users, d_2 movies
- Each user has a subset of observed movie comparisons
- Goal: predict movie rankings for each user

Observations

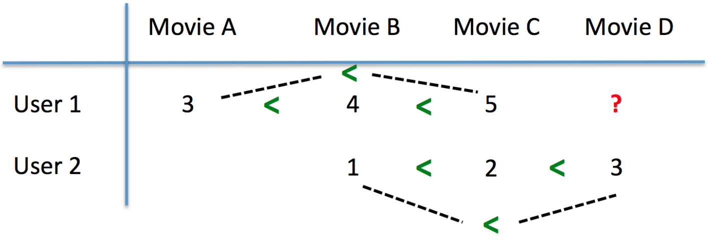
User 1:	B>A	C>B
User 2:	C>B	D>C
User 3:	B>C	C>D

Underlying Rankings

	A	B	C	D
User 1	1	2	3	4
User 2	1	2	3	4
User 3	-1	-2	-3	-4

Collaborative Ranking: Applications

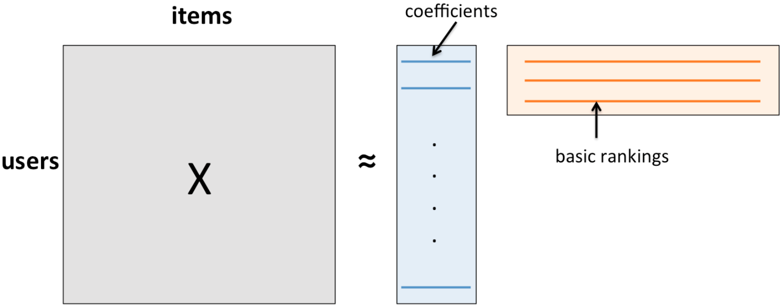
- Collaborative Ranking can be applied to classical recommender system
- For classical data (e.g., Netflix), we can transform original ratings to pairwise comparisons
- With the same data size, **ranking loss outperforms point-wise loss**



Collaborative Ranking: Assumption

- Assumption: the underlying scoring matrix is low-rank

$$X = UV^T$$



Collaborative Ranking: Model

- Collaborative Ranking:

$$\min_{U,V} \sum_{(i,j,k)\in\Omega} \ell\left(Y_{i,j,k} \cdot \left[(UV^T)_{ij} - (UV^T)_{ik} \right] \right) + \lambda(\|U\|_F^2 + \|V\|_F^2)$$

- The loss function ℓ we used is $\mathcal{L}_2$ hinge loss $\ell(a) = \max(0, 1-a)^2$
- The set $(i,j,k) \in \Omega$:
  - User i rates item $j >$ item $k \Leftrightarrow Y_{i,j,k} = 1$
  - If user i rates $\bar{d}_2$ movies, there will be $O(\bar{d}_2{}^2)$ pairs per user.

Collaborative Ranking: Model

- Collaborative Ranking:

$$\min_{U,V} \sum_{(i,j,k)\in\Omega} \ell\left(Y_{i,j,k} \cdot \left[(UV^T)_{ij} - (UV^T)_{ik} \right] \right) + \lambda(\|U\|_F^2 + \|V\|_F^2)$$

- The loss function ℓ we used is $\mathcal{L}_2$ hinge loss $\ell(a) = \max(0, 1-a)^2$
- The set $(i,j,k) \in \Omega$:
  - User i rates item $j >$ item $k \Leftrightarrow Y_{i,j,k} = 1$
  - If user i rates $\bar{d}_2$ movies, there will be $O(\bar{d}_2{}^2)$ pairs per user.
- Pros and Cons

  $+$ better prediction and recommendation

  \- slow, since it involves $O(|\Omega|) = O(d_1 \bar{d}_2{}^2)$ loss terms
  (original matrix completion only has $O(d_1 \bar{d}_2)$ terms)

Collaborative Ranking: Existing methods

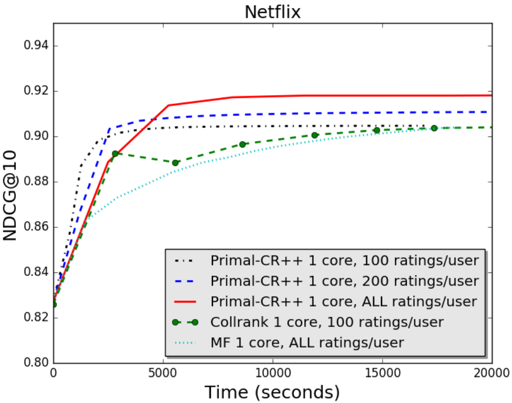
- State-of-the-art algorithms
  - CollRank (Park et al., ICML 2015): similar problem formulation
  - RobiRank (Yun et al., NIPS 2014): focus on binary observations
- All of them need

$$O(d_1 \bar{d_2}^2 r) \text{ time} \quad \text{and} \quad O(d_1 \bar{d_2}^2) \text{ memory}$$

  where d_1 is number of users, $\bar{d_2}$ is averaged ratings per user, r is the target rank.
- In full Netflix dataset: $d_1 \approx 2.5$ million, $\bar{d_2} \approx 200$, if $r = 100$
  Time per iteration: $d_1 \bar{d_2}^2 r \approx 10^{13}$ $\Rightarrow$ need days to train
  Memory: $d_1 \bar{d_2}^2 \approx 10^{11}$ $\Rightarrow$ 400GB memory
- Can't scale to full Netflix data

Our Algorithms: Primal-CR and Primal-CR++

- Classical matrix factorization: $O(d_1\bar{d}_2 r)$ time, $O(d_1\bar{d}_2)$ memory
- Previous collaborative ranking: $O(d_1\bar{d}_2{}^2 r)$ time, $O(d_1\bar{d}_2{}^2)$ memory
- Primal-CR: $O(d_1\bar{d}_2{}^2 + d_1\bar{d}_2 r)$ time, $O(d_1\bar{d}_2)$ memory
- Primal-CR++: $O(d_1\bar{d}_2(r + \log(\bar{d}_2)))$ time, $O(d_1\bar{d}_2)$ memory



General Framework: Alternating Minimization

- Consider the L2-hinge loss:

$$\min_{U,V} \left\{ \sum_{(i,j,k)\in\Omega} \max\left(0, 1 - Y_{i,j,k} \cdot (\boldsymbol{u}_i^T \boldsymbol{v}_j - \boldsymbol{u}_i^T \boldsymbol{v}_k)\right)^2 + \lambda \|U\|_F^2 + \lambda \|V\|_F^2 \right\}$$
$$:= f(U, V)$$

- Use alternating minimization.
- For $t = 1, 2, \ldots$
  - Fix V and update U by

$$U \leftarrow \arg\min_U f(U, V)$$

  - Fix U and update V by

$$V \leftarrow \arg\min_V f(U, V)$$

Update V

- Approach: Newton's method + conjugate gradient.
- Step 1: Compute gradient and Hessian:

$$\boldsymbol{g} = \nabla f(V) \in \mathbb{R}^{d_1 r} \qquad H = \nabla^2 f(V) \in \mathbb{R}^{d_1 r \times d_1 r}$$

- Update $V \leftarrow V - \eta H^{-1} \boldsymbol{g}$ (η is the step size)
- However, H is a huge matrix ($d_1 r$ can be millions)

  $\Rightarrow$ Cannot inverse H
- Use Conjugate Gradient (CG) to iteratively solve $H\boldsymbol{x} = \boldsymbol{g}$.
- Each iteration of CG only needs a matrix-vector product $H\boldsymbol{p}$
- Questions: (1) How to compute $\boldsymbol{g}$? (2) How to compute $H\boldsymbol{p}$

Compute Gradient

- How to compute the gradient?

$$\nabla_V f(V) = \sum_{i=1}^{d_1} \sum_{(j,k)\in\Omega_i} 2\max\left(0, 1-(\boldsymbol{u}_i^T\boldsymbol{v}_j - \boldsymbol{u}_i^T\boldsymbol{v}_k)\right)(\boldsymbol{u}_i\boldsymbol{e}_k^T - \boldsymbol{u}_i\boldsymbol{e}_j^T) + \lambda V$$

- Naive computation: $O(|\Omega| r) = O(d_1 \bar{d}_2{}^2 r)$

  But $d_1 \bar{d}_2{}^2 r \approx 10^{13}$ for Netflix

Compute Gradient

- How to compute the gradient?

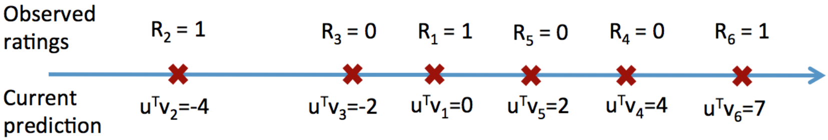
$$\nabla_V f(V) = \sum_{i=1}^{d_1} \sum_{(j,k)\in\Omega_i} 2\max\left(0, 1-(\boldsymbol{u}_i^T\boldsymbol{v}_j - \boldsymbol{u}_i^T\boldsymbol{v}_k)\right)(\boldsymbol{u}_i\boldsymbol{e}_k^T - \boldsymbol{u}_i\boldsymbol{e}_j^T) + \lambda V$$

- Naive computation: $O(|\Omega|r) = O(d_1\bar{d_2}^2 r)$

  But $d_1\bar{d_2}^2 r \approx 10^{13}$ for Netflix
- Idea (Primal-CR++): Fix k, do a linear scan of j after sorting
- For simplicity in illustation:
  - assume at k, observed rating is 1
  - ignore constant 1 in $\mathcal{L}_2$ hinge loss
  - ignore $\boldsymbol{u}_i\boldsymbol{e}_j^T$ in gradient equation

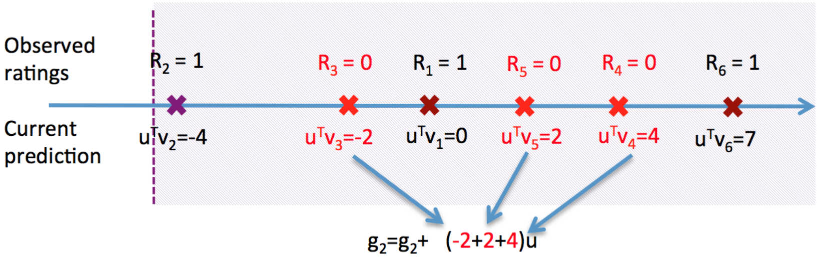
Even better way (Rethinking the main computation)

- Consider all the items for a user
- Sort items based on predictive values $\boldsymbol{u}^T \boldsymbol{v}_j$ ($O(\bar{d}_2 \log(\bar{d}_2))$ time)
- Now assume observed ratings can only be 0 or 1



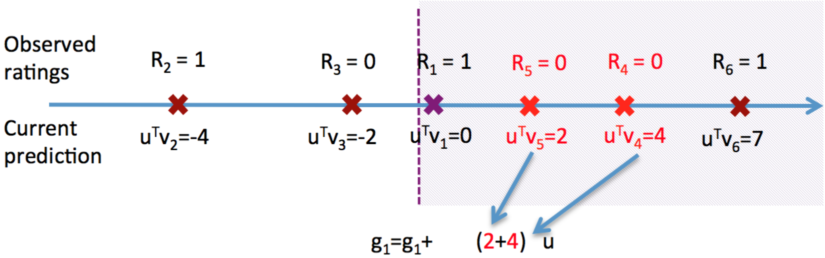
Even better way (Rethinking the main computation)

- Look at item 2 and sum over all items with larger predictive value but smaller observed ratings



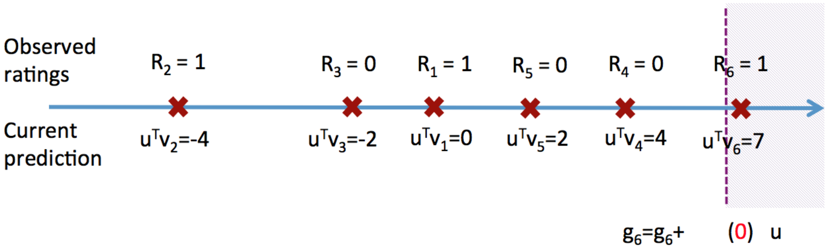
Even better way (Rethinking the main computation)

- Look at item 1 and sum over all items with larger predictive value but smaller observed ratings



Even better way (Rethinking the main computation)

- Look at item 6 and sum over all items with larger predictive value but smaller observed ratings



The linear-time algorithm for 0/1 ratings

- Let $s_k = \boldsymbol{u}^T \boldsymbol{v}_k$ (current prediction)
- Maintain prefix-sum at k:

$$P = \sum_{j: s_j > s_k \text{ and } R_j = 0} s_j$$

- Linea-scan algorithm:

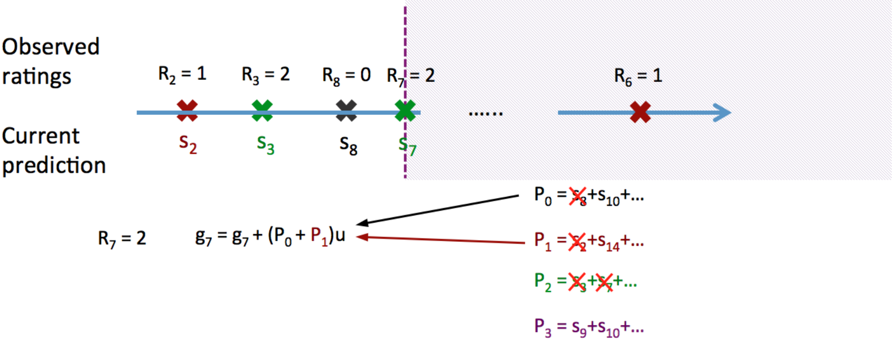
  Initially $P = \sum_{j:R_j=0} s_j$

  For $j = \pi(1), \pi(2), \cdots$

  If $R_j = 0$: Update $P \leftarrow P - s_j$

  If $R_j = 1$: Add $P \cdot u$ to g_j

General case: For T levels of ratings

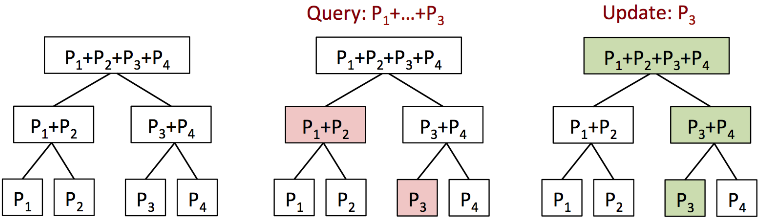


General case: For T levels of ratings

- Each step needs two operations:
  (1) Compute $P_1 + \cdots + P_\ell$ for some ℓ
  (2) Update one of P_j

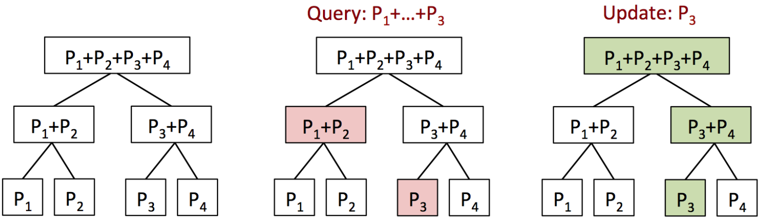
General case: For T levels of ratings

- Each step needs two operations:
  (1) Compute $P_1 + \cdots + P_\ell$ for some ℓ
  (2) Update one of P_j
- Can be done in $O(\log(T))$ time per query using
  Segment tree or Fenwick tree!



General case: For T levels of ratings

- Each step needs two operations:
  (1) Compute $P_1 + \cdots + P_\ell$ for some ℓ
  (2) Update one of P_j
- Can be done in $O(\log(T))$ time per query using
  Segment tree or Fenwick tree!



- Overall time complexity: $d_1 \bar{d_2}^2 r \approx 10^{13}$
  $\Rightarrow d_1 \bar{d}_2 r + d_1 \bar{d}_2 \log(\bar{d}_2) \approx 5.1 \times 10^{10}$ Order of 3 Speed-up!!
- Time complexity for standard matrix factorization:
  $\Rightarrow d_1 \bar{d}_2 r \approx 5 \times 10^{10}$

Experiments

- NDCG@10: measuring the quality of top-10 recommendations

$$\text{NDCG@10} = \frac{1}{d_1} \sum_{i=1}^{d_1} \frac{\text{DCG@10}(i, \pi_i)}{\text{DCG@10}(i, \pi_i^*)}, \tag{1}$$

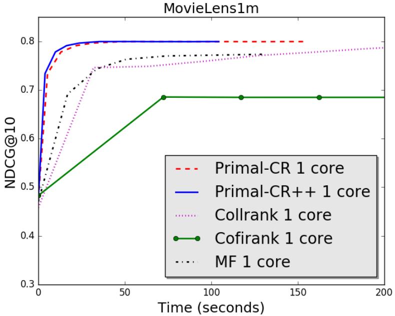
where i represents i-th user and

$$\text{DCG@10}(i, \pi_i) = \sum_{k=1}^{10} \frac{2^{M_i \pi_i(k)} - 1}{log_2(k+1)}. \tag{2}$$

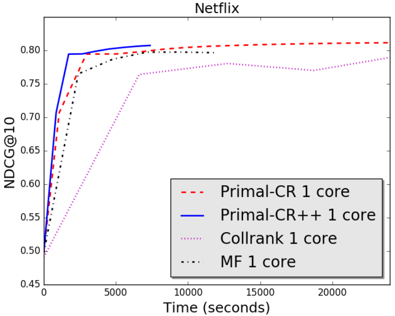
- Comparisons
  1. Single core subsampled data
  2. Parallelization
  3. Single core full data

Comparisons: Single Core Subsampled Data

- We subsampled each user to have exactly 200 ratings in training set and used the rest of ratings as test set, since previous approaches cannot scale up
- Users with fewer than $200 + 10$ ratings not included



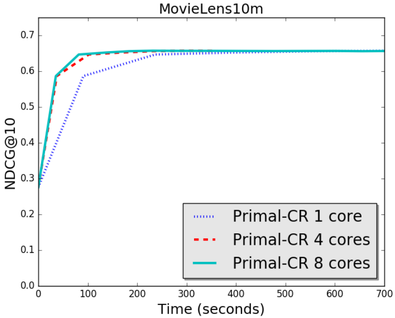
MoviLens1M $(6,040 \times 3,952)$



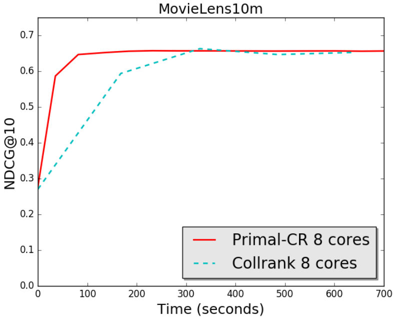
Netflix $(2,649,430 \times 17,771)$

Comparisons: Parallelization

- Our algorithm scales up well (see comparisons for 1 core, 4 cores and 8 cores) in the multi-core shared memory setting
- Primal-CR still much faster than Collrank when 8 cores are used



MoviLens10M $(71,567 \times 65,134)$



MoviLens10M $(71,567 \times 65,134)$

Comparisons: Single Core Full Data

- Due to the $O(|\Omega|r)$ complexity, existing algorithms always sub-sample a limited number of pairs per user
- Our algorithm is the first ranking-based algorithm that can scale to full Netflix data set using a single machine, and without sub-sampling
- A natural question:

  Does using more training data help us predict and recommend better?

Comparisons: Single Core Full Data

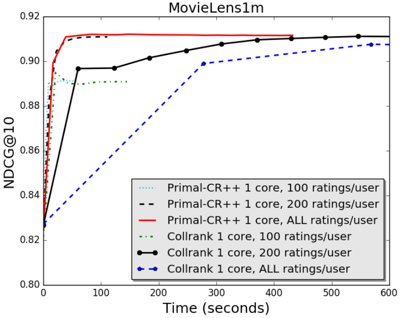
- Due to the $O(|\Omega|r)$ complexity, existing algorithms always sub-sample a limited number of pairs per user
- Our algorithm is the first ranking-based algorithm that can scale to full Netflix data set using a single machine, and without sub-sampling
- A natural question:

  Does using more training data help us predict and recommend better?

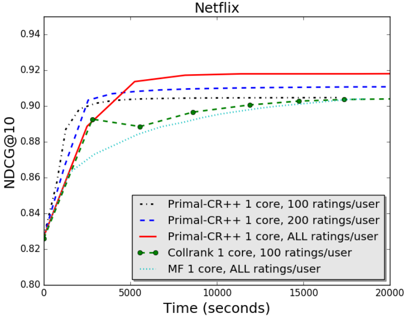
  The answer is yes!

Comparisons: Single Core Full Data

- Randomly choose 10 ratings as test data and out of the rest ratings randomly choose up to C ratings per user as training data
- Users with fewer than 20 ratings not included
- Comparing NDCG@10 when $C = 100, 200, d_2$



MoviLens1M $(6,040 \times 3,952)$



Netflix $(2,649,430 \times 17,771)$

Conclusions

- We show that CR can be used to replace matrix factorization in recommender systems
- We show that CR can be solved efficiently (almost same time complexity as matrix factorization)
- We show that it is always best to use all available pairwise comparisons if possible (subsampling gives suboptimal recommender results for top k items)
- Julia codes: https://github.com/wuliwei9278/ml-1m
- C++ codes (2x faster than Julia codes, with support for multithreading): https://github.com/wuliwei9278/primalCR

Thank You!