Maxios: Large Scale Nonnegative Matrix Factorization for Collaborative Filtering

Simon Shaolei Du **Boyi Chen** **Yilin Liu**
University of California Berkeley
simonshaoleidu@berkeley.edu

Lei Li*
Baidu Research
lilei22@baidu.com

Abstract

Nonnegative matrix factorization proved useful in many applications, including collaborative filtering – from existing ratings data one would like to predict new product ratings by users. However, factorizing a user-product score matrix is computation and memory intensive. We propose Maxios, a novel approach to fill missing values for large scale and highly sparse matrices efficiently and accurately. We formulate the matrix-completion problem as weighted nonnegative matrix factorization. In addition, we develop distributed update rules using alternating direction method of multipliers. We have implemented the Maxios system on top of Spark, a distributed in-memory computation framework. Experiments on commercial clusters show that Maxios is competitive in terms of scalability and accuracy against the existing solutions on a variety of datasets.

1 Introduction

Matrix factorization techniques have been successfully applied to missing value imputation [10, 12, 1, 5, 4, 3, 2]. In this paper, we exploit a matrix factorization tool, non-negative matrix factorization (NMF), to recover missing values for large and sparse matrices. We present a reformulation of the NMF objective. To solve the NMF efficiently and in a scalable way, we propose Maxios, an iterative algorithm that can distribute the computation over multiple machines. The proposed mathematical formalism and implemented system can be generalized to other matrix factorization techniques such as singular value decomposition (SVD).

We consider the following objective of non-negative matrix factorization with missing values,

$$\arg \min_{U \geq 0, V \geq 0} \frac{1}{2} \|W \odot (D - U \cdot V)\|_F^2 \tag{1}$$

Where D is a $n \times m$ matrix, and W is an indicating matrix for missing values (0=missing). Such formulation is different from the traditional NMF formulation without the weight matrix W, which is unsuitable for matrices with missing observations. U and V are latent factors to search for and should be nonnegative. The formulation is first introduced as weighted NMF implicitly by Zhang et al [17] and later formally by Kim and Choi [9].

Without missing observations, there exist two classes of solutions for NMF: algorithms using multiplicative update to maintain non-negativity, pioneered by Lee and Seung [11, 12], and (block-)coordinate descent algorithms such as alternating least square [6, 7] and active-set methods [8]. These algorithms have been ported to distributed computing platform with carefully engineered data-partition schemes [13]. In case of missing observations, the existing solutions can be categorized into two types: (a) using traditional NMF solutions in Expectation-Maximization framework [17];

*Corresponding author. The majority of the work was completed while the author was at Computer Science Division of University of California, Berkeley.

and (b) alternating least square (ALS) for weighted NMF formulation Equation (1) [17, 9]. For large scale problems, one could extend the EM-NMF approach to map-reduce framework using the same techniques proposed in [13]. Unfortunately, such approach requires in each iteration instantiating the full prediction matrix (millions by hundreds of thousands), which will be enormous even on today's cloud platform. ALS based solutions need to find optimal latent factors inside each iteration, which itself is expensive. Therefore neither approach scales well to the size of the data.

Our goal is to produce a scalable solution when the data matrix D is large and sparse. To accelerate computations of each iteration and improve the recovery accuracy, we are motivated by a few key properties of the data and the nature of such kind of computation.

Data sparsity: The data matrix has very large dimensions. However, there are very few observed entries. A full instantiation will take long to compute and take huge memory store. The problem gets worse in the distributed setting because transferring a full instantiation across machines can take very long time. Therefore an efficient algorithm need not compute the full instantiation during its intermediate steps. Alternating least square NMF(ALS-NMF) utilizes these properties to speedup the process [9] while expectation-maximization based approaches EM-NMF needs to evaluate the whole missing part [17].

Non-negativity constraints: Inside each iteration of the algorithm, the computation should be relatively fast. Directly solving for non-negativity constraints usually takes a nontrivial calculation step. Alternating direction methods of multiplier approaches (ADMM) provides a framework to separate the minimization objective from the constraints [1, 15].

Distributed in-memory computation: By partitioning the data and latent factors, an ideal approach should be strongly scalable with respect to the number of partitions. Hadoop-based implementation for NMF (but not for missing values) is an early success to distributed NMF [13], however it is limited by additional disk-writes. Instead of using Hadoop, we could further significantly reduce the overhead per iteration by using a distributed in-memory computation framework.

To construct a both faster and more accurate algorithm, we design Maxios specifically to address the following aspects:

- Maxios formulates the optimization problem using weights on the objective. The missing values are only calculated at the end of the algorithm once; therefore the computation time is independent of the number of missing entries.
- Maxios reformulates the non-negativity constrained optimization using ADMM, which leads to faster computation for each iteration. This is because ADMM separates the objective from the constraints.
- Maxios is implemented on top of Spark [16]. Spark enables distributed in-memory map-reduce computation, and it achieves 100x speedup on analytical tasks [16]. In addition, our algorithm only requires exchanging small partitions of U and V (note both U and V are much smaller than the data matrix), thus diminishing the communication overhead.

We evaluate our system on MovieLens(1 millon observed 22 million missing)[1], Netflix Movie Ratings(100 million observed, 8.5 billion missing), and Yahoo! Music Ratings dataset (120 million observed, 191 billion missing)[2]. We also do our best to implement base algorithms. Our experiments on commercial clusters show expected improvement in recovering accuracy within a fixed amount of computation time. The experimental settings and results are presented in Section 3.

2 Proposed Method

In this section, we will describe our proposed Maxios. We start with the mathematical formalism of the original problem, then describe how to use ADMM framework to derive the main algorithm. We also analyze the complexity and a property about the convergence to provide the theoretical grantee of Maxios. In addition, we will discuss several further extensions to the base case.

[1] http://movielens.umn.edu/

[2] http://webscope.sandbox.yahoo.com/catalog.php?datatype=r

2.1 Formulation and Update Rule

To enable embarrassingly parallel computation, we take the non-negative constraints out and consider the following equivalent formulation of the original problem in Equation (1):

$$\min \frac{1}{2} \|D - W \odot (UV)\|_F^2 \text{ s.t.} U = X, V = Y, X, Y \geq 0 \tag{2}$$

Where $U, X \in \mathbb{R}^{m \times q}$ and $V, Y \in \mathbb{R}^{q \times n}$. X and Y are axillary matrices. The augmented Lagrangian of Equation (2) is

$$\begin{aligned} L(U, V, X, Y, \Lambda_1, \Lambda_2) = & \frac{1}{2} \|D - W \odot (UV)\|_F^2 + \Lambda_1 \bullet (U - X) + \Lambda_2 \bullet (V - Y) \\ & + \frac{\alpha_1}{2} \|U - X\|_F^2 + \frac{\alpha_2}{2} \|V - Y\|_F^2 \end{aligned} \tag{3}$$
$$X, Y \geq 0$$

Where $\Lambda_1 \in \mathbb{R}^{m \times q}, \Lambda_2 \in \mathbb{R}^{q \times n}$ are Lagrangian multipliers, and α_1, α_2 are penalty parameters. Note that unlike conventional full Lagrangian accommodating all constraints, Equation (3) only includes the penalties for equality constraints, and leaves the non-negativity intact. Therefore we name it *augmented partial Lagrangian*. The reason to keep non-negativity constraints is to facilitate our calculation of auxiliary matrices X and Y. Such treatment leads to our ADMM solution.

The ADMM method for (3) can be derived by successively minimizing the Lagrangian with respect to U, V, X, Y one at time while fixing others, i.e.,

$$\begin{aligned} U^{k+1} &= \arg\min_U \ \frac{1}{2} \left\|D - W \odot (UV^k)\right\|_F^2 + \Lambda_1^k \cdot (U - X^k) + \frac{\alpha_1}{2} \left\|U - X^k\right\|_F^2 \\ V^{k+1} &= \arg\min_V \ \frac{1}{2} \left\|D - W \odot (U^{k+1}V)\right\|_F^2 + \Lambda_2^k \bullet (V - Y^k) + \frac{\alpha_2}{2} \left\|V - Y^k\right\|_F^2 \\ X^{k+1} &= \arg\min_{X \geq 0} \ \Lambda_1 \bullet (U^{k+1} - X) + \frac{\alpha_1}{2} \left\|U^{k+1} - X\right\|_F^2 \\ Y^{k+1} &= \arg\min_{Y \geq 0} \ \Lambda_2 \bullet (V^{k+1} - Y) + \frac{\alpha_2}{2} \left\|V^{k+1} - Y\right\|_F^2 \end{aligned}$$

All of the update rules above could be written in closed forms as opposed to methods like ALS-WNMF[9], which need iterative updates. For U's update, every row is independent of the others, so we can update U in parallel. Similarly, we can update V's columns in parallel. After updating U, V, X, and Y we update multipliers using standard ADMM formulas. The specific update rules for $U, V, X, Y, \Lambda_1, \Lambda_2$ are described in Algorithm 1

2.2 ℓ_2 regularization

There is often risk of over-fitting when the number non-zero entries are small comparing to the missing values. One commonly used solution is to penalize, either with ℓ_1 or ℓ_2 penalty. Both penalty will constrain the complexity of the model. Here we illustrate the case with ℓ_2. Our algorithm still works when adding ℓ_2 penalty to prevent over-fitting problem. With the ℓ_2 penalty, the new ADMM formulation is:

$$\min \frac{1}{2} \|D - W \odot (UV)\|_F^2 + \frac{\lambda}{2}(\|U\|_2 |^2 + \|V\|_2^2) \text{s.t.} U = X, V = Y, X, Y \geq 0$$

Where λ is the L_2 penalty Note that the penalty term is only relevant to U and V. Therefore we only need to change the update rule of U and V:

$$\begin{aligned} U_{i\bullet}^{k+1} =& ((V^k \odot W_{i\bullet}^q)^T D_{i\bullet} + \alpha_1 X_{i\bullet}^k - \Lambda_{1,i\bullet}^k) \cdot ((V^k \odot W_{i\bullet}^q)(V^k \odot W_{i\bullet}^q)^T + (\alpha_1 + \lambda) I_q)^{-1} \\ V_{\bullet j}^{k+1} =& ((U \odot W_{\bullet j}^q)^T (U \odot W_{\bullet j}^q) + (\alpha_2 + \lambda) I_q)^{-1} \cdot (U_{j\bullet} \odot W_{\bullet j} D_{\bullet j} + \alpha_2 Y_{\bullet j} - \Lambda_{2,\bullet,j}) \end{aligned}$$

Maxios can be easily generalized to a family of loss functions induced by Bregman Divergence[14].

Algorithm 1 Maxios

input : C with missing values as NaN, $maxIter > 0$, $U_0 \in \mathbb{R}^{m \times q} \geq 0$, $V_0 \in \mathbb{R}^{q \times n}, \epsilon^{pri}, \epsilon^{dual}, \alpha_1, \alpha_2 > 0$

output: U $\in R^{m \times q} \geq 0$, V $\in R^{q \times n} \geq 0$

Construct W using C such that $W_{ij} = 0$ if $C_{ij} =$ NaN, $W_{ij} = 1$ if $C_{ij} \neq$ NaN
Construct D using C such that $D_{ij} = 0$ if $C_{ij} =$ NaN, $D_{ij} = Cij$ if $C_{ij} \neq$ NaN
Set $X = U_0, Y = V_0$
Set Λ_1 a zero matrix $\in \mathbb{R}^{m \times q}$, Λ_2 a zero matrix $\in \mathbb{R}^{q \times n}$
for $k \leftarrow 0$ **to** $maxIter$ **do**

for i = 1,2,...m, update each row of U in parallel

$$U_{i\bullet}^{k+1} = ((V^k \odot W_{i\bullet}^q)^T D_{i\bullet} + \alpha_1 X_{i\bullet}^k - \Lambda_{1,i\bullet}^k) \cdot ((V^k \odot W_{i\bullet}^q)(V^k \odot W_{i\bullet}^q)^T + \alpha_1 I_q)^{-1} \tag{4}$$

for j = 1,2,...n, update each column of V in parallel

$$V_{\bullet j}^{k+1} = ((U^{k+1} \odot W_{\bullet j}^q)^T (U^{k+1} \odot W_{\bullet j}^q) + \alpha_2 I_q)^{-1} \cdot ((U^{k+1} \odot W_{\bullet j}^q)^T D_{\bullet j} + \alpha_2 Y_{\bullet j}^k - \Lambda_{2,\bullet j}^k) \tag{5}$$

$$X^{k+1} = P_+(U^{k+1} + \frac{\Lambda_1^k}{\alpha_1}) \tag{6}$$

$$Y^{k+1} = P_+(V^{k+1} + \frac{\Lambda_2^k}{\alpha_2}) \tag{7}$$

$$\Lambda_1^{k+1} = \Lambda_1^k + \alpha_1(U^{k+1} - X^{k+1}) \tag{8}$$

$$\Lambda_2^{k+1} = \Lambda_2^k + \alpha_2(V^{k+1} - Y^{k+1}) \tag{9}$$

$$r^{k+1} = \|D - W \odot (UV)\|_F^2 / \|W\|_F^2 \tag{10}$$

$$s^{k+1} = (\|U - X\|_F^2 + \|V - Y\|_F^2)/q(m+n) \tag{11}$$

if $s^{k+1} - s^k| < \epsilon^{pri}$ *and* $s < \epsilon^{dual}$ *and* $k \geq maxIter$ **then**
  exit and output(U^k,V^k)

2.3 Complexity

One main advantage of our algorithm is that Maxios does not require computing the full recovered data matrix explicitly during the iterations. Therefore, our algorithm works much faster than traditional Expectation-Maximization based algorithms [17]. In algorithm 1, we do not actually instantiate matrix W. Operations $V \odot W_{i\bullet}^q$ and $U \odot W_{\bullet j}^q$ are equivalent to select columns of V according to the nonzero entries of $W_{i\bullet}$ and select rows of U according to the nonzero entries of $W_{j\bullet}$. Therefore, (4) takes $O(q^2 m r_{max})$ where r_{max} is the biggest number of non-zeros of all rows of W. Similarly (5) takes $O(q^2 n c_{max})$ where c_{max} is the biggest number of non-zeros of all columns of W. For (6), (8), (7), and (9) we just need to iterate U, V, X, Y, Λ_1,and Λ_2, which take $O((m+n)q)$ total. Thus the algorithm takes $O(q^2(mr_{max} + nc_{max}))$ for each iteration. Notice that the dominating part (4),(5) can be implemented in parallel. The more detailed memory efficient implementation is described in next section.

2.4 Implementation

It is straightforward to implement the above algorithm on a single-threaded machine. However, technical challenges arise when we want to parallelize and distribute the computation over multiple machines. More specifically, there are a couple of optimizations we did in our implementation to speed up the system. We implement our method using Spark [16], a distributed in-memory map-reduce framework. Each row of U and X are stored and computed on worker nodes, and similarly each column of V and Y. The sparse data matrix is replicated on each of the worker nodes too. These latent factors are then broadcast to worker nodes in each iteration. Thanks to the high sparsity of the observation matrix, Maxios does not produce full prediction matrices and therefore reduce the communication overhead. We also implemented a special purpose scheduler to accommodate the straggler problem – worker nodes with smaller number of observations will wait for those with larger portion of data.

3 Experiments

We are concerned about four key properties of each algorithm:

Convergence Rate and Accuracy: The algorithm we want should converge fast with good accuracy. Thus we plot training and testing root mean squared error(RMSE) VS time to compare algorithms' convergence rate and prediction accuracy.

Scalability with Data Matrix: The desire algorithm should scale well with number of rows, number of columns, sparsity and scale of the ratings. Therefore we chose matrices of various sizes from real world to test how algorithms scale with different input data.

Scalability with Number of Latent Factor: Often we want try out different number of latent features(q) to see which one is the best fit for the data matrix. The desire algorithm should not be prohibited by the increase of number of latent factors. To see this property of the algorithms, we test each dataset using different q.

We compared Maxios, a distributed version of WNMF-ANLS[9] and a distributed version of WNMF-Mult[17] in terms of convergence speed, prediction accuracy and scalability. We only compared these three methods because they do not need to instantiate the user-item matrix and thus converge much faster than EM-based methods. In particular, it is infeasible to run the distributed NMF [13] for large dataset with missing values, because direct use of distributed NMF in an EM framework for missing observations require huge memory to instantiate the full prediction matrix.

We tested these algorithms on real commercial datasets, ranging from different number of columns and rows, sparsity and scales of ratings. All datasets we used is summarized in table 1. For all datasets, we uniformly sampled $4/5$ observed entries for training and $1/5$ entries for testing. For small dataset MoviLens Movie Ratings, which has 1000209 ratings on 3952 movies rated by 6040 users, we implemented all three algorithms in Matlab R2011 and ran on Macbook Pro with dual cores of 2.2G i7 CPU and 8GB memory. For large datasets, Yahoo! Music Ratings, which has contains 11557943 ratings of 98211 artists, and Netflix Movie Ratings, which consists of 17770 movies and 480189 users with more than 100 million ratings , we implemented algorithms in Scala,

Table 1: Dataset Summary.

Data Set	# of rows	# of columns	sparsity	score range
MovieLens	6040	3952	4.19%	[0, 5]
Netflix Movie Ratings	480189	17770	1.18 %	[0,5]
Yahoo! Music Ratings	1947155	98213	0.06%	[0,100]

and used Spark as cluster computing framework and EC2 r1 extra large instances with each instance having 4 threads.

Prediction Accuracy and Convergence Rate

First we compared the training and testing reconstruction error. For MovieLens dataset, we ran each algorithm with number of latent factor $q = 10, 30, 50$ and stopped running when the running time exceeded $1000s$. For Yahoo! Music Ratings, and Netflix Movie Ratings datasets, we also ran each algorithm with $q = 10, 30, 50$ and stopped running when the running time exceeded $3h$ or has run 1000 iterations. For MovieLens dataset, we set $\epsilon^{pri} = 5 \times 10^{-4}$, $\epsilon^{dual} = 5 \times 10^{-4}$, $\alpha_1 = 1, \alpha_2 = \alpha_1 m/n$. and L2 penalty $\lambda = 1, 9, 25$ for different number of latent factors $q = 10, 30, 50$.

For Netflix Movie Rating dataset, we set $\epsilon^{pri} = 5 \times 10^{-4}$, $\epsilon^{dual} = 5 \times 10^{-4}$, $\alpha_1 = 1, \alpha_2 = \alpha_1 m/n$. and L2 penalty $\lambda = 1, 9, 25$ for different number of latent factors $q = 10, 30, 50$.

For Yahoo! Music Rating dataset, we set $\epsilon^{pri} = 5 \times 10^{-4}$, $\epsilon^{dual} = 5 \times 10^{-4}$, $\alpha_1 = 10, \alpha_2 = \alpha_1 m/n$. and L2 penalty $\lambda = 100, 900, 2500$ for different number of latent factors $q = 10, 30, 50$.

Fig(1)-Fig(4) show the training and testing RMSE against time on different numbers of nodes with different numbers of latent factor. Maxios always converges faster than the other two methods no matter what number of latent factors and how many machines we used.

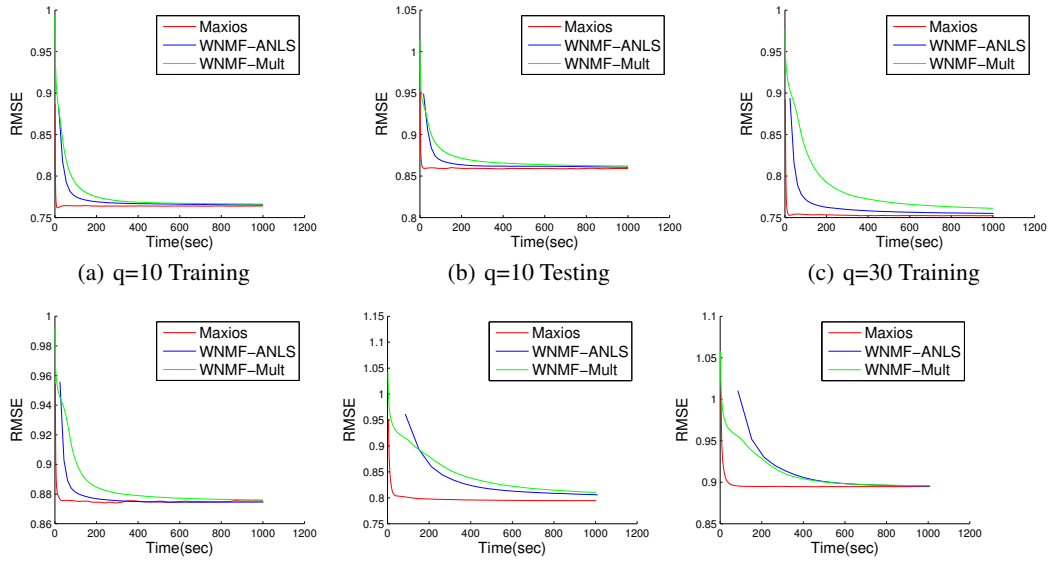
WNMF-ANLS always converges faster than WNMF-Mult when q is small however when q becomes larger, some time WNMF-ANLS is slower than WNMF-Mult. The reason is that each iteration WNMF-ANLS needs to optimize $(m + n)$ vectors with length q. We used projected-newton optimization method as stated in [9], which has complexity $O(q^3)$ and usually needs tens of iterations to converge.

Note that for Netflix dataset, WNMF-ANLS needs to optimize about 500K vectors and for Yahoo! dataset, WNMF-ANLS needs to optimize 2M vectors. This is why WNMF-ANLS behaves differently for these two datasets. On the other hand, Maxios and WNMF-Mult do not have inner optimization step, so they scale well with q and fully utilize the sparse property of the input matrix.

4 Conclusion

Large scale collaborative filtering is a challenge task. In this paper, we formulated the problem as weighted nonnegative matrix factorization. We proposed Maxios, a distributed algorithm that can complete user-product matrices effectively and efficiently. We implemented our algorithm on top of Spark, an in-memory cloud computing framework. Our experiments demonstrated our proposed Maxios not only generates more accurate predictions but also converges faster than existing solutions.

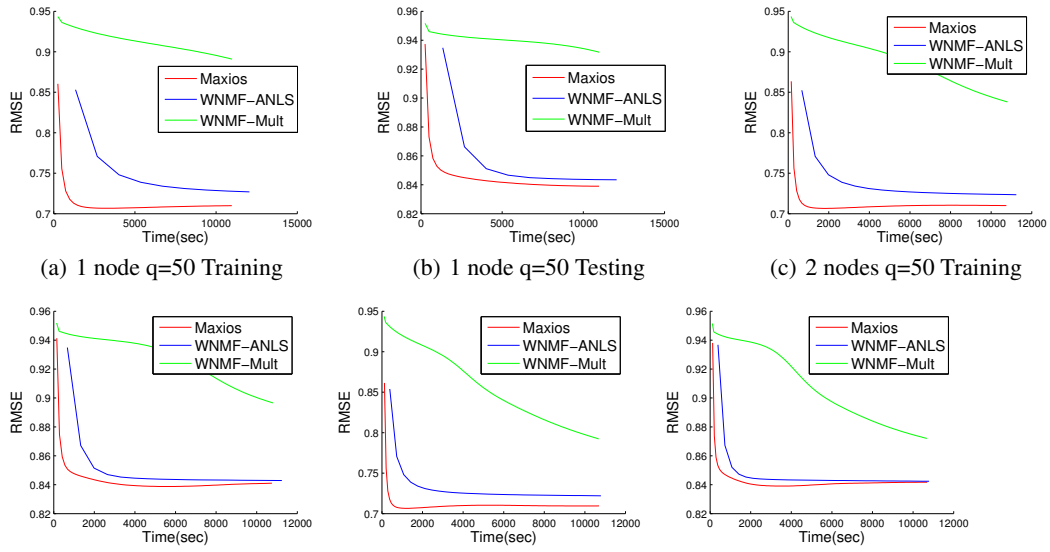
While we focus on the NMF with square loss, it is straightforward to generalize the system to handle other loss functions such as Bregman divergence, or losses induced from Poisson and Exponential prior [13].



(a) q=10 Training (b) q=10 Testing (c) q=30 Training

(d) q=30 Testing (e) q=50 Training (f) q=50 Testing

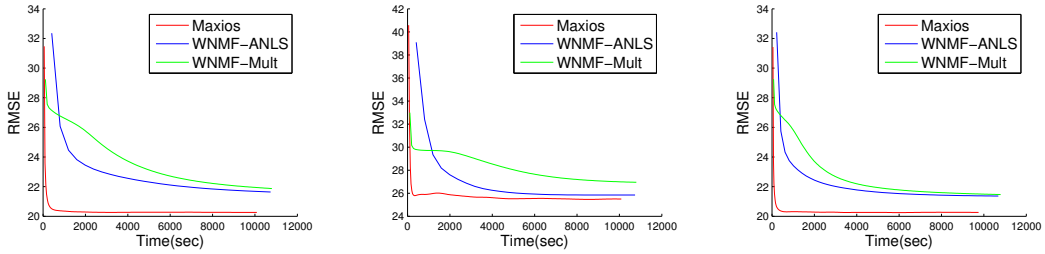
Figure 1: Convergence Rate and Accuracy of MovieLens Movie Rating Dataset. Maxios always converges faster than the other two methods and they have similar testing RMSE when they converge. WNMF-ANLS behaves better than WNMF-Mult when q is small, but it converges slower than WNMF-Mult when $q = 50$.



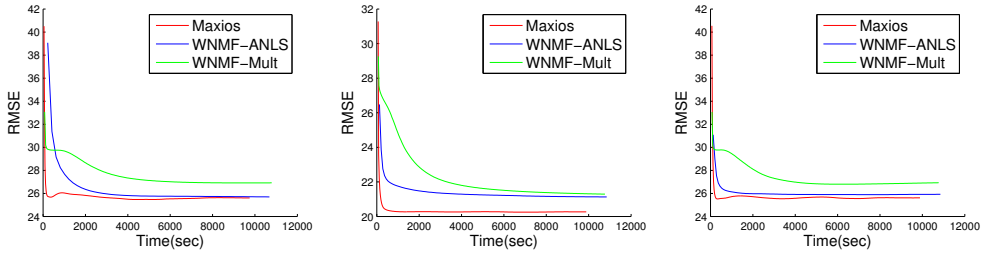
(a) 1 node q=50 Training (b) 1 node q=50 Testing (c) 2 nodes q=50 Training

(d) 2 nodes q=50 Testing (e) 4 nodes q=50 Training (f) 4 nodes q=50 Testing

Figure 2: Convergence Rate and Accuracy of Netflix Dataset with $q = 50$. For Netflix Movie Ratings dataset, even $q = 50$, Maxios and WNMF-ANLS have similar performance. The reason is that in this case WNMF-ANLS only needs to optimize about 500K vectors with length q ,which is not the dominating part of the update.

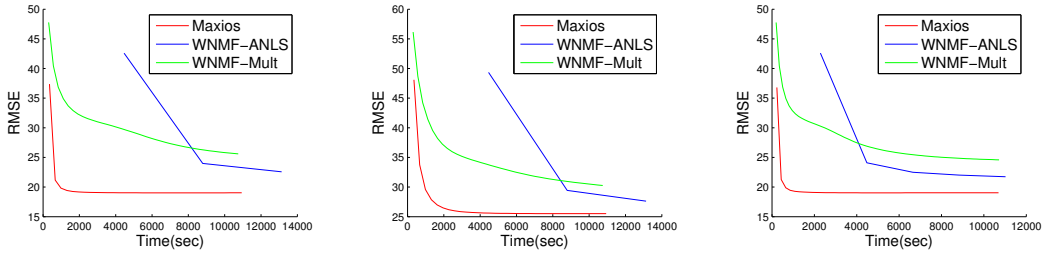


(a) 1 node q=10 Training (b) 1 node q=10 Testing (c) 2 nodes q=10 Training

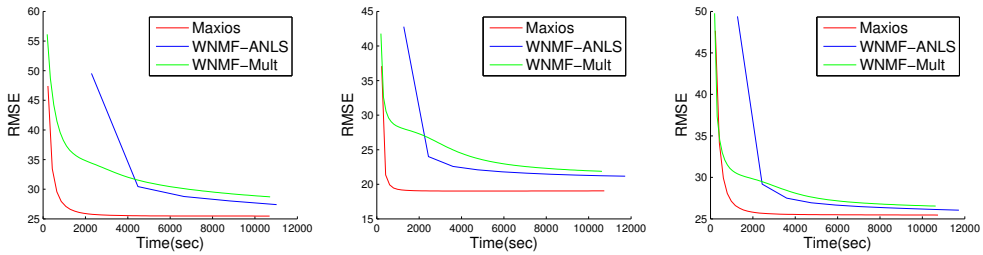


(d) 2 nodes q=10 Testing (e) 4 nodes q=10 Training (f) 4 nodes q=10 Testing

Figure 3: Convergence Rate and Accuracy of Yahoo! Music Rating with $q = 10$. Maxios converges faster than the other two and has lower prediction error. Also observe that WNMF-ANLS behaves better than WNMF-Mult. The reason is that q is very small in this case.



(a) 1 node q=50 Training (b) 1 node q=50 Testing (c) 2 nodes q=50 Training



(d) 2 nodes q=50 Testing (e) 4 nodes q=50 Training (f) 4 nodes q=50 Testing

Figure 4: Convergence Rate and Accuracy of Yahoo! Music Rating with $q = 50$. When $q = 50$, it takes very long time for each iteration of WNMF-ANLS. When running on 1 machine, WNMF-ANLS only completed 3 iterations and used 13000s. Compared with Netflix Movie Rating dataset, here WNMF-ANLS needs to optimize about 2M vectors and so the dominating part is the inner optimization step. On the other hand, Maxios and WNMF-Mult scale well with the increase of q.

References

[1] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein. Distributed optimization and statistical learning via the alternating direction method of multipliers. *Found. Trends Mach. Learn.*, 3(1):1–122, Jan. 2011.

[2] E. J. Candès and B. Recht. Exact matrix completion via convex optimization. *Foundations of Computational mathematics*, 9(6):717–772, 2009.

[3] Q. Gu, J. Zhou, and C. H. Ding. Collaborative filtering: Weighted nonnegative matrix factorization incorporating user and item graphs. In *SDM*, pages 199–210. SIAM, 2010.

[4] Y. Hu, D. Zhang, J. Ye, X. Li, and X. He. Fast and accurate matrix completion via truncated nuclear norm regularization. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 35(9):2117–2130, 2013.

[5] R. Khanna, L. Zhang, D. Agarwal, and B.-c. Chen. Parallel matrix factorization for binary response. In *Big Data, 2013 IEEE International Conference on*, pages 430–438. IEEE, 2013.

[6] H. Kim and H. Park. Nonnegative matrix factorization based on alternating nonnegativity constrained least squares and active set method. *SIAM Journal on Matrix Analysis and Applications*, 30(2):713–730, 2008.

[7] J. Kim, Y. He, and H. Park. Algorithms for nonnegative matrix and tensor factorizations: A unified view based on block coordinate descent framework. *Journal of Global Optimization*, 58(2):285–319, 2014.

[8] J. Kim and H. Park. Fast nonnegative matrix factorization: An active-set-like method and comparisons. *SIAM Journal on Scientific Computing*, 33(6):3261–3281, 2011.

[9] Y.-D. Kim and S. Choi. Weighted nonnegative matrix factorization. In *Acoustics, Speech and Signal Processing, 2009. ICASSP 2009. IEEE International Conference on*, pages 1541–1544. IEEE, 2009.

[10] Y. Koren, R. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 42(8):30–37, 2009.

[11] D. D. Lee and H. S. Seung. Learning the parts of objects by non-negative matrix factorization. *Nature*, 401(6755):788–791, Oct. 1999.

[12] D. D. Lee and H. S. Seung. Algorithms for non-negative matrix factorization. In *NIPS*, pages 556–562, 2000.

[13] C. Liu, H.-c. Yang, J. Fan, L.-W. He, and Y.-M. Wang. Distributed nonnegative matrix factorization for web-scale dyadic data analysis on mapreduce. In *Proceedings of the 19th international conference on World wide web*, pages 681–690. ACM, 2010.

[14] S. Sra and I. S. Dhillon. *Nonnegative matrix approximation: Algorithms and applications*. Computer Science Department, University of Texas at Austin, 2006.

[15] Y. Xu, W. Yin, Z. Wen, and Y. Zhang. An alternating direction algorithm for matrix completion with nonnegative factors. *Frontiers of Mathematics in China*, 7(2):365–384, 2012.

[16] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[17] S. Zhang, W. Wang, J. Ford, and F. Makedon. Learning from incomplete ratings using non-negative matrix factorization. In *SDM*. SIAM, 2006.