Journal of Intelligent Information Systems (2021) 56:49–72
https://doi.org/10.1007/s10844-020-00601-0

Distributed matrix factorization based on fast optimization for implicit feedback recommendation

Lian Chen[1] · Wangdong Yang[1] · Kenli Li[1] · Keqin Li[1,2]

Received: 9 August 2019 / Revised: 29 March 2020 / Accepted: 29 March 2020 /
Published online: 10 June 2020

Abstract
In big data scenarios, matrix factorization (MF) is widely used in recommendation systems as it can offer high accuracy and scalability. However, when using MF to process large-scale implicit feedback data, the following two problems arise. One is that it is difficult to effectively obtain negative feedback information, which causes relatively poor recommendation accuracy. The other is that the limited resources of a single machine make the model training inefficient, and in particular, the acquisition of negative feedback information further increases the time complexity of model training. In order to solve the above two problems, we first propose a user-activity and item-popularity weighted matrix factorization (UIWMF) recommendation algorithm, which assigns every missing data different weight based on user activity and item popularity, gets negative feedback information more realistically, and leads to better recommendation accuracy. Meanwhile, in order to reduce the additional computational overhead caused by the weight strategy, we develop a fast optimization strategy to enhance the efficiency. In order to break the resource constraints of a single machine, we propose a distributed UIWMF (DUIWMF) algorithm based on Spark, which adopts an efficient parallel learning algorithm to train the model and utilizes cached in-block and out-block information to effectively reduce the communication overhead in a distributed environment. We conduct experiments on three public datasets, and the experimental results demonstrate that, comparing with the baseline MF methods, DUIWMF model has comparable performance in terms of recommendation accuracy and model training efficiency.

Keywords Personalized recommendation · Collaborative filtering · User and item recommendation · Fast optimization · Distributed matrix factorization · Spark

✉ Wangdong Yang
yangwangdong@163.com

Extended author information available on the last page of the article.

1 Introduction

1.1 Motivation

The recommendation system usually provides users with customized recommendations according to their interests. With the rapid development of information technology, the scale of recommendation data has increased dramatically. Additionally, the data is getting more and more sparse, which makes it harder to provide an accurate recommendation. Meanwhile, such large-scale recommendation data also poses a critical challenge to the construction of the model. Hence, based on such large-scale sparse data, effectively constructing models and making accurate recommendations becomes a very challenging issue (Karydi and Margaritis 2016).

Matrix factorization (MF) is a dimensionality reduction technique, which is widely used in recommendation systems and has been proven to be exceptionally accurate in processing large-scale and high-dimensional sparse data (Karydi and Margaritis 2016; Koren et al. 2009). However, Most of the previous studies on MF algorithms for recommendation focused on the explicit feedback (Koren et al. 2009; Rendle and Schmidt-Thieme 2008), where the ratings of users directly reflected their preference on each item. Collecting these ratings requires users to evaluate items subjectively, which is not always easy to realize. Thus, researchers paid attention to the implicit data (Kabbur et al. 2013), which is usually produced incessantly and much easier to obtain. For example, referring to purchase histories and web browsing histories, the implicit data of users can be easily acquired. This kind of implicit data is usually a binary decision on some items, while it may be biased since it inherently lacks negative feedbacks, which is known as the one-class problem (Pan et al. 2008). Specifically, when a user is browsing for some products on the Internet, commodities that attract the user, i.e., with positive feedbacks, are more likely to be browsed. In the meanwhile, the user may hardly click on the products with negative feedbacks. As a result, the corresponding browsing history contains much more products with positive feedbacks. Therefore, how to obtain the negative feedback from the missing data is essential to improve the accuracy of recommendation derived from implicit feedbacks. To make up for negative feedbacks, Hu et al. (2008) proposed to allocate uniform weights to all the missing data, while this weighting strategy is not consistent with the practical scenario. In the above purchasing example, the missing browsing data contains both the negative feedbacks that the user saw but not browsed, as well as the unlabeled data that the user did not see. The weighting strategy was improved in He et al. (2016), where the authors considered the item popularity. Particularly, since a popular item is more natural to be seen by the user, the missing feedbacks of a popular item deserves a higher weight. A similar weighting strategy based on the activity of the user was proposed in Li et al. (2018), where the authors believed that active users were more likely to interact with more items, then the missing data from active users should be designated higher weights. However, the above two methods assigned uniform weights to the missing data of each user or each item, respectively. This assumption only considers the differences among users or items, ignoring the unique characteristic of each missing data. Besides, the use of weight strategy significantly increases the time complexity of the implicit MF model, so it is necessary to develop more efficient learning algorithms. He et al. (2016) designed a new learning algorithm based on the element-wise Alternating Least Squares (eALS) technique. The main idea of eALS is to avoid the massive repeated computations by introducing a cache matrix. eALS algorithm provides a new idea for dealing with implicit MF, on the basis of which Li et al. (2018) developed an efficient learning algorithm to enhance the efficiency. Therefore, how to balance the accuracy

and time complexity of the algorithm also needs to be considered in the design of weight strategy.

Another critical issue is how to train models for large-scale recommendation efficiently. Due to the limited memory and CPU resources of a single machine, model training locally is very time-consuming. Hence, the design of parallel and distributed recommender algorithm becomes necessary (Chen et al. 2018; Yu et al. 2014). Thanks to the development of cloud computing, MapReduce-based distributed memory computing framework, e.g., Apache Spark (2019a) and Apache Flink (2019), could provide high computing power for large-scale data processing. To effectively leverage the computing power of existing frameworks, we need an efficient parallel learning algorithm based on MapReduce to train the implicit MF model. Stochastic gradient descent (SGD) and alternating least squares (ALS) are two major learning algorithms for training the MF model. However, both methods are not suitable for MapReduce-based large-scale data processing. Specifically, SGD is challenging to be parallelized efficiently in the MapReduce-based framework. This is due to the fact that the MapReduce platform usually executes synchronous distributed learning algorithms in a data-parallel mode, while SGD is inherently serial. More importantly, SGD usually suffers from slow convergence, and the convergence speed is sensitive to the parameters (Yu et al. 2014). Although ALS is straightforward to parallelize and suitable for the MapReduce programming model, its computation complexity is cubic, which is not scalable to large-scale datasets (Yu et al. 2014).

1.2 Our contributions

An efficient implicit MF recommendation system cannot ignore the influence of missing negative feedbacks. In addition to this, it has to possess the ability to process large-scale sparse data. This paper addresses the above two issues, and endeavors to design a distributed recommendation system for implicit feedback recommendation. Specifically, the main contributions of this paper are summarized as follows:

- *Comprehensive weighting strategy:* Different from the weighting strategies mentioned above, which did not address the negative feedbacks adequately, we give full consideration to both user activity and item activity, and firstly propose a centralized matrix factorization algorithm to improve the accuracy of recommendations. Compared with the current uniform weight strategy (Hu et al. 2008; He et al. 2016; Li et al. 2018), the proposed comprehensive weighting strategy, i.e., the user-activity and item-popularity weighted matrix factorization (UIWMF) recommendation algorithm, can distinguish negative feedbacks from missing data more accurately and make recommendations more precisely. Meanwhile, in order to avoid the increase of the time complexity of the algorithm caused by the weight strategy, we use eALS to learn the model, and skillfully design four cache matrices to effectively avoid the massive repeated computations.
- *Low-cost distributed implementation:* To handle large-scale implicit recommendation data, we further propose the distributed UIWMF (DUIWMF) algorithm such that the model training can be carried out in a parallel and distributed fashion. As a parallel extension of UIWMF, DUIWMF is based on eALS and can be implemented efficiently on Spark. To reduce the cost of distributed implementation, we adopt a block matrix partition strategy and utilize cached in-block and out-block information. In this way, the communication overhead generated along with the computation task offloading can be restricted.

– *Experiments on public datasets:* Experimental results testify both the recommendation accuracy and the model training efficiency of DUIWMF on three public datasets. Notably, in addition to the previously mentioned algorithms, we design practical training procedures and deploy them on a commercial cloud server. The results of real-world experiments demonstrate that the DUIWMF model has comparable performance in terms of recommendation accuracy and model training efficiency.

1.3 Paper organization

The rest of this paper is organized as follows. In Section 2, we review the related research on weighting strategies of missing data and distributed optimization methods for implicit MF recommendation. In Section 3, we give a detailed explanation to our implicit MF method, including a novel missing data weighting strategy and a fast optimization algorithm. In Section 4, we describe the parallel implementation of our method on Spark. In Section 5, we demonstrate the performance comparison results in our extensive experiments. In Section 6, we conclude the paper.

2 Related work

2.1 Weight strategy of missing data for implicit MF recommendation

Implicit feedbacks are inherently lacking in negative feedback information. Thus, how to obtain negative feedback information effectively is critical to improving the accuracy of recommendation. To this end, two different strategies were proposed, which were sample-based learning (Marlin et al. 2012; Rendle et al. 2009) and whole-based learning (Hu et al. 2008; He et al. 2016). The former strategy randomly sampled negative instances from the missing data, while the latter one treated all the missing values as negative instances. The sample-based method is more efficient than the other one since it does not need to consider all the missing data, while this inevitably leads to a risk of losing valuable information. As a limit of the sample-based method, the whole-based method utilizes all the data, which is potential with higher coverage. The corresponding efficiency, nevertheless, becomes a problem. To retain model's fidelity, we persist in the whole-data based learning, developing an efficient learning algorithm to solve the problem of inefficiency.

Most existing whole-data based methods (Hu et al. 2008; Pilászy et al. 2010; Volkovs and Yu 2015; Devooght et al. 2015; Steck 2010) assigned uniform weights to all the missing data, and assumed that the missing entries had the same probability of being negative feedbacks. This kind of strategy makes the algorithm more efficient (He et al. 2016), but overestimates the effect of the missing entries (Liang et al. 2016). In order to obtain negative feedbacks more effectively, some studies have proposed non-uniform weighting strategies for missing data. Pan et al. (2008) proposed two different non-uniform weighting strategies for missing data, i.e., user-oriented and item-oriented, respectively. The former one assumed that the user with more positive feedback instances could be more possibly to hide the negative feedbacks in the missing data. The latter one assumed that the item with fewer positive examples would be more likely negatively commented in the missing data. However, the time complexity of this method is cubic, which makes it not scalable to large-scale data (Pilászy et al. 2010). Intuitively, the missing feedbacks of a popular item are more natural to be seen by the user and thus should be assigned higher weights. In He et al. (2016), the authors adopted this idea, proposed an item-oriented missing data weighting strategy, and

further developed an efficient algorithm for model training. Liang et al. (2016) developed a probabilistic model called *Exposure MF* to evaluate whether the user saw an item, and also adopted a missing data weighting strategy based on the item popularity to separate negative feedback information from unlabeled information. Li et al. (2018) adopted a user-aware weighting strategy which assigned missing values with different confidence according to user popularity. However, all of these weighting strategies mentioned above only took into account the differences between either different users or different items, ignoring the unique characteristics of each missing data. Recently, a study is devoted to solving the above problems. He et al. (2019) perform truncated SVD on the weight matrix of missing data, using a more compact low-rank model to represent the weights of missing entries. As a result, how to initialize the weight matrix is the key to get the effective missing data weight. However, the initialization scheme based on user activity is still used in the paper. In addition, the scheme based on SVD weight strategy will also increase the time complexity of the algorithm.

2.2 Optimization strategy for distributed implicit MF

The *Stochastic Gradient Descent* (SGD) and *Coordinate Descent* (CD) are the two typical learning methods of MF. In order to improve the model training efficiency of large-scale implicit recommendation data, many researchers studied parallel implementation based on the above two algorithms in a distributed environment.

The SGD learning methods are implemented in a parallel and distributed fashion in works such as Gemulla et al. (2011), Yun et al. (2014), and Chen et al. (2018) to accelerate MF. Although the SGD algorithm has low time complexity and is easy to process large-scale data, it is unsuitable for whole-data based MF due to a large number of training instances (Hu et al. 2008). More importantly, it is challenging to efficiently implement the typically sequential SGD method in a distributed environment based on the MapReduce platform, which is suitable for synchronous distributed learning algorithm in a data-parallel mode. Another popular method is the *Alternating Least Squares* (ALS), which updates either the entire user feature vector or the entire item feature vector at a time, and can be regarded as a particular CD algorithm. Schelter et al. (2013) proposed a data-parallel low-rank matrix factorization with ALS on Hadoop, which used a series of broadcast-joins to avoid expensive shuffle operations. However, this method needs to store the user and item feature matrix separately on each node, which will cause additional storage overhead, and more importantly, the corresponding feature matrix needs to be broadcast during each iteration, which will lead to great communication overhead. Additionally, the Hadoop platform is not suitable for iterations because it needs to read data from disk repeatedly. Apache Spark Mllib (Apache Spark 2019b) contained a parallel implementation of ALS, which used a block matrix partition strategy and used block cache information to reduce communication overhead between different nodes effectively. However, limited by the high time complexity of ALS, the algorithm is not scalable to large-scale datasets (Yu et al. 2014). Another standard method of coordinate descent is Cyclic Coordinate Descent (CCD), which updates one feature element at a time to avoid expensive matrix transpose operations. Yu et al. (2014) proposed a scalable and efficient coordinate descent method named CCD++ and made a distributed implementation based on MPI. The main idea of CCD++ is that the updated sequence of feature elements will affect the final convergence result. However, in the process of parallel implementation of CCD++, each node needs to broadcast its updated feature elements to all other nodes, which will incur huge communication overhead, especially on the MapReduce-based platform.

3 Our implicit MF method

In this section, we first propose the definition of the implicit MF method. Next, in order to improve the recommendation accuracy of the implicit MF method, we propose a missing data weighting strategy that combines user activity and item popularity. Last, we develop an efficient optimization strategy to optimize our objective function.

3.1 Implicit MF method

MF separately maps rows and columns of the original user-item interaction matrix $\boldsymbol{R} \in \mathbb{R}^{M\times N}$ into a low-dimension (K-dimensional) user latent factor space $\boldsymbol{P} \in \mathbb{R}^{M\times K}$ and an item latent factor space $\boldsymbol{Q} \in \mathbb{R}^{N\times K}$ (Koren et al. 2009). The interaction R_{ui} is the (u, i)-th element in $\boldsymbol{R}$. Intuitively, we have $R_{ui} = 1$ if user u and item i exist interaction information, otherwise $R_{ui} = 0$. More specifically, the element R_{ui} is modeled as the inner product $< \boldsymbol{p}_u^T, \boldsymbol{q}_i >$ in that space, where $\boldsymbol{p}_u$ and $\boldsymbol{q}_i$ are latent factor vectors of user u and item i. The symbols and descriptions are shown in (Table 1). The objective function for model learning is usually formed as an error-based regression, which can be mathematically written as

$$L = \sum_{u=1}^{M}\sum_{i=1}^{N} W_{ui}(R_{ui} - \hat{R}_{ui})^2 + \lambda\left(\sum_{u=1}^{M}\|\boldsymbol{p}_u\|^2 + \sum_{i=1}^{N}\|\boldsymbol{q}_i\|^2\right), \tag{1}$$

where W_{ui} denotes the weight of R_{ui} and we use $\boldsymbol{W} = [W_{ui}]_{M\times N}$ to represent the weight matrix, $\hat{R}_{ui} = \boldsymbol{p}_u^T\boldsymbol{q}_i$ is the predicted value of R_{ui}, and λ is the regularization parameter to prevent over-fitting.

3.2 Our weighting strategy on missing data

In the user-item interaction matrix, the missing data contains both negative feedback data and unlabeled data. The negative feedback data should be given a higher weight in the objective function to distinguish from unlabeled data. How to filter out negative feedback information from the missing data is the key to improve recommendation accuracy. Some uniform weighting strategies of missing data have been proposed in previous works (Pan et al. 2008; He et al. 2016; Li et al. 2018). Their weight strategies only take into account the differences between different users or items, which is suboptimal for real application scenarios.

There are two situations in the real recommendation system. On the one hand, popular items have a higher probability to be exposed to users, which makes the missing of ratings more probable to come from deliberate choices. In order to capture this feature, He et al. (2016) proposed a popularity-aware weighting strategy which assigns each missing item a unified weight according to its frequency in the implicit feedback data. It can be expressed by the following formula

$$c_i = \frac{f_i^{\alpha}}{\sum_{j=1}^{N} f_j^{\alpha}}, \tag{2}$$

where $f_i = |\mathcal{R}_i^{\mathrm{U}}|/|\mathcal{R}|$ represents the proportion of interaction data of item i to the total interaction data, α controls the significance level of popular items over unpopular ones.

On the other hand, active users have a higher chance to see unrated items. Thus, the unobserved data for this user is negative with a higher probability. Li et al. (2018) adopted a user-aware weighting strategy which assigned missing values with different confidence

Table 1 Symbols and description

Symbols	Description
u	User id
i	Item id
M	Number of users
N	Number of items
K	Length of latent factor vector
$\boldsymbol{P}$	User latent factor matrix($\boldsymbol{P} \in \mathbb{R}^{M \times K}$)
$\boldsymbol{p}_u$	Latent factor vector of user u
$\boldsymbol{Q}$	Item latent factor matrix($\boldsymbol{Q} \in \mathbb{R}^{N \times K}$)
$\boldsymbol{q}_i$	Latent factor vector of item i
$\boldsymbol{R}$	User-item interaction matrix($\boldsymbol{R} \in \mathbb{R}^{M \times N}$)
$\boldsymbol{W}$	Weight matrix
$\hat{\boldsymbol{R}}$	Prediction rating matrix($\hat{\boldsymbol{R}} \in \mathbb{R}^{M \times N}$)
$\mathcal{R}_u{}^{\mathtt{I}}$	The set of items observed by user u
$\mathcal{R}_i^{\mathtt{U}}$	The set of users who have interaction information item i
$\mathcal{R}$	The set including observed interactive (u, i) pairs
$\lvert\mathcal{R}\rvert$	Number of observed interactive data
$\lvert\mathcal{R}_u^{\mathtt{I}}\rvert$	Number of items observed by user u
$\lvert R_i^{\mathtt{U}}\rvert$	Number of users who have interaction information with item i
t_0	Overall weight of the missing data
c_i	Popularity of item i
d_u	Activity of user u
λ	Regularization parameter

according to user frequency in the implicit feedback data. It can also be expressed by the following formula

$$d_u = \frac{h_u^{\beta}}{\sum_{j=1}^{M} h_j^{\beta}}, \tag{3}$$

where $h_u = |\mathcal{R}_u^{\mathtt{I}}|/|\mathcal{R}|$ represents the proportion of the interaction data of user u to the total interaction data, β controls the significance level of active users over inactive ones. However, both cases exist in implicit feedback recommendation. In order to further improve the accuracy of the recommendation, we combined the above two cases, and finally proposed a weighting strategy combining user activity and product popularity. We assign each missing data a different weight based on the cumulative results of user activity and item popularity. To account for this effect, we set T_{ui} as

$$T_{ui} = t_0(\eta c_i + (1 - \eta)d_u) \tag{4}$$

where t_0 represents the overall weight of the missing data, d_u represents the user activity, and c_i represents the item popularity. Parameter η controls the extent to which user activity and item popularity affect the weight of a single missing value. If η is equal to 0, the problem is transformed to finding a weighting strategy based on the user activity. Otherwise, if $\eta = 1$, the strategy is based on the item popularity. If both α and β are set to 0, $W_{ui} = t_0(\eta/N + (1 - \eta)/M)$ indicates the uniform weighting strategy.

Based on the above weighting strategy, we construct the following objective function:

$$L = \sum_{(u,i)\in\mathcal{R}} W_{ui}(R_{ui} - \hat{R}_{ui})^2 + \sum_{u=1}^{M}\sum_{i\notin R_u^{\mathrm{I}}} T_{ui}(\hat{R}_{ui})^2 + \lambda\left(\sum_{u=1}^{M}\|\boldsymbol{p}_u\|^2 + \sum_{i=1}^{N}\|\boldsymbol{q}_i\|^2\right). \tag{5}$$

3.3 Fast optimization using eALS

Since we adopt a full data-based missing data weighting strategy, the training efficiency of the model is a crucial issue. Thus we use an element-wise ALS (eALS) (He et al. 2016) to optimize the objective function, the basic idea of which is to update a single variable at a time while keep others fixed. This approach reduces the time complexity by avoiding the expensive computation of matrix inversion. In addition, we use an efficient optimization strategy to avoid massive unnecessary computations. In particular, our missing data weight strategy allocates nonuniform weights for each missing data, which requires a very high space complexity to store. We use an ingenious way to calculate the weight of missing data with less time complexity in the update process. For example, if only one variable P_{uf} is allowed to change while fixing the other variables, the original optimization problem turned into a univariate quadratic problem, the corresponding solution can be obtained as follow:

$$P_{uf} = \frac{\sum_{i\in\mathcal{R}_u^{\mathrm{I}}} W_{ui}(R_{ui} - \hat{R}_{ui}^f)Q_{if} - \sum_{i\notin\mathcal{R}_u^{\mathrm{I}}} T_{ui}\hat{R}_{ui}^f Q_{if}}{\sum_{i\in\mathcal{R}_u^{\mathrm{I}}} W_{ui}Q_{if}^2 + \sum_{i\notin\mathcal{R}_u^{\mathrm{I}}} T_{ui}Q_{if}^2 + \lambda}, \tag{6}$$

where $\hat{R}_{ui}^f = \boldsymbol{p}_u^T\boldsymbol{q}_i - P_{uf}Q_{if}$. Since the original interaction matrix is so sparse that the amount of missing data is much larger than the known data, the main calculations are concentrated on terms $\sum_{i\notin\mathcal{R}_u^{\mathrm{I}}} T_{ui}\hat{R}_{ui}^f Q_{if}$ and $\sum_{i\notin\mathcal{R}_u^{\mathrm{I}}} T_{ui}Q_{if}^2$. We perform the following conversion:

$$\sum_{i\notin\mathcal{R}_u^{\mathrm{I}}} T_{ui}\hat{R}_{ui}^f Q_{if} = \sum_{i=1}^{N} T_{ui}Q_{if}\sum_{k\neq f} P_{uk}Q_{ik} - \sum_{i\in\mathcal{R}_u{}^{\mathrm{I}}} T_{ui}\hat{R}_{ui}^f Q_{if}, \tag{7}$$

which can be further converted as

$$\sum_{i=1}^{N} T_{ui}Q_{if}\sum_{k\neq f} P_{uk}Q_{ik} = \sum_{k\neq f} p_{uk}\sum_{i=1}^{N} t_0\eta c_i Q_{if}Q_{ik} + d_u\sum_{k\neq f} p_{uk}\sum_{i=1}^{N} t_0(1-\eta)Q_{if}Q_{ik}. \tag{8}$$

It can be found that the computational overhead mainly depends on $\sum_{i=1}^{N} t_0\eta c_i Q_{if}Q_{ik}$ and $\sum_{i=1}^{N} t_0(1-\eta)Q_{if}Q_{ik}$, which need to iterate over all the items in $\boldsymbol{Q}$. It is worth noting that, both calculations are independent with u, thus we can accelerate the computation by

maintaining cache matrices $\boldsymbol{H}^{q1} = \sum_{i=1}^{N} t_0 \eta c_i \boldsymbol{q}_i \boldsymbol{q}_i^T$ and $\boldsymbol{H}^{q2} = \sum_{i=1}^{N} t_0 (1-\eta) \boldsymbol{q}_i \boldsymbol{q}_i^T$. Equation (7) is further transformed to

$$\sum_{i \notin \mathcal{R}_u^{\mathrm{I}}} T_{ui} \hat{R}_{ui}^f Q_{if} = \sum_{k \neq f} p_{uk} H_{kf}^{q1} + d_u \sum_{k \neq f} p_{uk} H_{kf}^{q2} - \sum_{i \in \mathcal{R}_u^{\mathrm{I}}} T_{ui} \hat{R}_{ui}^f Q_{if}, \tag{9}$$

where $\hat{R}_{ui}^f = \boldsymbol{p}_u^T \boldsymbol{q}_i - P_{uf} Q_{if}$, and H_{kf}^{q1} denotes the (k, f)-th element in $\boldsymbol{H}^{q1}$ cache. Similarly, the denominator in (6) can be evaluated as:

$$\sum_{i \notin \mathcal{R}_u^{\mathrm{I}}} T_{ui} Q_{if}^2 = H_{ff}^{q1} + d_u H_{ff}^{q2} - \sum_{i \in \mathcal{R}_u^{\mathrm{I}}} T_{ui} Q_{if}. \tag{10}$$

By bringing (9) and (10) into (6), we obtain

$$P_{uf} = \left(\sum_{i \in \mathcal{R}_u^{\mathrm{I}}} (W_{ui} R_{ui} - (W_{ui} - T_{ui}) \hat{R}_{ui}^f) Q_{if} - d_u \sum_{k \neq f} P_{uk} H_{kf}^{q2} - \sum_{k \neq f} P_{uk} H_{kf}^{q1} \right) / C, \tag{11}$$

where $C = \sum_{i \in \mathcal{R}_u^{\mathrm{I}}} (W_{ui} - T_{ui}) Q_{if}^2 + d_u H_{ff}^{q2} + H_{ff}^{q1} + \lambda$. Similarly, we can derive the update rule for Q_{if}:

$$Q_{if} = \left(\sum_{u \in \mathcal{R}_i^{\mathrm{U}}} (W_{ui} R_{ui} - (W_{ui} - T_{ui}) \hat{R}_{ui}^f) P_{uf} \right. \tag{12}$$

$$\left. - c_i \sum_{k \neq f} Q_{ik} H_{kf}^{p2} - \sum_{k \neq f} Q_{ik} H_{kf}^{p1} \right) / Z, \tag{13}$$

where $Z = \sum_{u \in \mathcal{R}_i^{\mathrm{U}}} (W_{ui} - T_{ui}) P_{uf}^2 + c_i H_{ff}^{p2} + H_{ff}^{p1} + \lambda$. The cache matrices are $\boldsymbol{H}^{p1} = \sum_{u=1}^{M} t_0 (1-\eta) d_u \boldsymbol{p}_u \boldsymbol{p}_u^T$ and $\boldsymbol{H}^{p2} = \sum_{u=1}^{M} t_0 \eta \boldsymbol{p}_u \boldsymbol{p}_u^T$. Algorithm (1) summarizes the process of UIWMF.

3.4 Discussions

3.4.1 Time complexity analysis

From (9), the time complexity of updating a user feature factor is $\mathcal{O}(|\mathcal{R}_u^{\mathrm{I}}| + K)$. Therefore, the time complexity of updating the user feature matrix $\boldsymbol{P}$ is $\mathcal{O}(|\mathcal{R}|K + MK^2)$. Similarly, from (13), the time complexity of updating an item feature factor is $\mathcal{O}(|\mathcal{R}_i^{\mathrm{U}}| + K)$. Thus, the time complexity of updating the item feature matrix $\boldsymbol{Q}$ is $\mathcal{O}(|\mathcal{R}|K + NK^2)$. Hence, the whole-time complexity of an iteration is $\mathcal{O}(|\mathcal{R}|K + (M+N)K^2)$, which has a significant improvement of the time complexity compared with the ALS method ($\mathcal{O}(|\mathcal{R}|K^2 + (M+N)K^3)$).

3.4.2 Space complexity analysis

For the implementation of UIWMF, first of all, we need to store the original sparse rating matrix $\boldsymbol{R}$, which needs the storage space of size $|\mathcal{R}|$. Additionally, we need to store the user feature matrix $\boldsymbol{P}$ ($\boldsymbol{P} \in \mathbb{R}^{M \times K}$) and the item feature matrix $\boldsymbol{Q}$ ($\boldsymbol{Q} \in \mathbb{R}^{N \times K}$). Besides, cache

matrices $\boldsymbol{H}^{p1}$, $\boldsymbol{H}^{p2}$, $\boldsymbol{H}^{q1}$ and $\boldsymbol{H}^{q2}$ need to be stored to accelerate the calculation, which require the storage space of size K^2. In the whole, the space complexity is $\mathcal{O}(|\mathcal{R}|)$.

3.4.3 Parallelism analysis

The training process for UIWMF is straightforward to parallelize. On the one hand, it can be found from the update process of the Algorithm (1), the update of each user feature vector $\boldsymbol{p}_u$ is independent of each other, so as the item feature vector $\boldsymbol{q}_i$. Therefore, the update of the user feature vector and the item feature vector can be updated in parallel. On the other hand, the cache calculations of $\boldsymbol{H}^{p1}$, $\boldsymbol{H}^{p2}$, $\boldsymbol{H}^{q1}$ and $\boldsymbol{H}^{q2}$ are standard matrix multiplication operations with efficient parallelism.

Algorithm 1 UIWMF.

Require: User-item interaction matrix R, weighting matrix W, item popularity vector c, user activity vector d, length of latent factor vector K, regularization parameter λ.
Ensure: MF parameters P and Q.

Randomly initialize P and Q;
for $(u, i) \in \mathcal{R}$ **do** $\hat{R}_{ui} \leftarrow \boldsymbol{p}_u^T \boldsymbol{q}_i$;
while Stopping criteria is not met **do**
  $H^{q1} \leftarrow \sum_{i=1}^{N} t_0 \eta c_i \boldsymbol{q}_i \boldsymbol{q}_i^T$;
  $H^{q2} \leftarrow \sum_{i=1}^{N} t_0 (1-\eta) \boldsymbol{q}_i \boldsymbol{q}_i^T$;
  for $u \leftarrow 1$ **to** M **do**
    for $f \leftarrow 1$ **to** K **do**
      for $i \in \mathcal{R}_u^{\mathrm{I}}$ **do** $\hat{R}_{ui}^f \leftarrow \hat{R}_{ui} - P_{uf} Q_{if}$;
      $P_{uf} \leftarrow (11)$;
      for $i \in \mathcal{R}_u^{\mathrm{I}}$ **do** $\hat{R}_{ui} \leftarrow \hat{f}_{ui}^f + P_{uf} Q_{if}$;
    end for
  end for
  $H^{p1} \leftarrow \sum_{u=1}^{M} t_0 (1-\eta) d_u \boldsymbol{p}_u \boldsymbol{p}_u^T$;
  $H^{p2} \leftarrow \sum_{u=1}^{M} t_0 \eta \boldsymbol{p}_u \boldsymbol{p}_u^T$;
  for $i \leftarrow 1$ **to** N **do**
    for $f \leftarrow 1$ **to** K **do**
      for $u \in \mathcal{R}_i^{\mathrm{U}}$ **do** $\hat{R}_{ui}^f \leftarrow \hat{R}_{ui} - P_{uf} Q_{if}$;
      $Q_{if} \leftarrow (12)$;
      for $u \in \mathcal{R}_i^{\mathrm{U}}$ **do** $\hat{R}_{ui} \leftarrow \hat{f}_{ui}^f + P_{uf} Q_{if}$;
    end for
  end for
end while
return P and Q.

4 Parallelization of UIWMF on spark

In order to process large-scale recommendation data with MF, especially when the matrices $\boldsymbol{R}$, $\boldsymbol{P}$, and $\boldsymbol{Q}$ exceed the memory capacity of a single machine, exploiting the advantages of cloud computing or fog computing and training models in a distributed fashion are necessary (Zhu et al. 2018).

In order to solve the above problems, we propose a distributed UIWMF (DUIWMF) algorithm on Spark, which utilizes the inherent model parallelism of the algorithm. We adapt a blocked parallel approach and construct a cached information routing table to reduce the communication overhead in distributed environments effectively.

4.1 Model partitioning strategy

To exploit the computational resources of multiple machines in a distributed environment, model partitioning is inevitable. Matrices $\boldsymbol{R}$, $\boldsymbol{P}$, and $\boldsymbol{Q}$ need to be stored in the training process of the algorithm. In the previous study (Schelter et al. 2013), $\boldsymbol{R}$ was simply divided into different machines, and a separate copy of $\boldsymbol{P}$, and $\boldsymbol{Q}$ was kept on each machine. However, in today's commercial environment, the scale of users and items is usually at the level of 10 million or even hundreds of millions, e.g., Suppose $M, N = 50$ million, $K = 128$, which requires 5.12 Gbytes of additional storage space for each machine, which requires a considerable amount of memory capacity. More importantly, in each iteration, we need to broadcast $\boldsymbol{P}$, and $\boldsymbol{Q}$ frequently to each machine, which will greatly increase the network communication overhead. Therefore, we also divide $\boldsymbol{P}$, and $\boldsymbol{Q}$ into different machines, and in the subsequent update process, we design a special data structure to minimize the cost of communication.

Assuming that the distributed system is composed of l machines, we consider to divide the rows of $\boldsymbol{P}$ and $\boldsymbol{Q}$ into l parts, where $\{\mathcal{P}_1, \mathcal{P}_2, \cdots, \mathcal{P}_l\}$ stores the index partition of the rows of $\boldsymbol{P}$, and $\{\mathcal{Q}_1, \mathcal{Q}_2, \cdots, \mathcal{Q}_l\}$ consists the index partition of the rows of $\boldsymbol{Q}$. Each machine is responsible for storing and calculating the assigned sub-matrices. Particularly, when there is a multi-core CPU or multiple CPUs on each machine, we divide the matrices $\boldsymbol{P}$ and $\boldsymbol{Q}$ according to the number of CPU cores. Moreover, the memory required to store the rating matrix $\boldsymbol{R}$ is usually much larger than that for $\boldsymbol{W}$ and $\boldsymbol{H}$ when a relatively small K value is set. Thus, we should avoid the transmission of $\boldsymbol{R}$ in the distributed implementation. From the update rule of UIWMF, only the elements $R_{ui}, \forall i \in \mathcal{R}_u^{\mathrm{I}}$ are required for the update of $\boldsymbol{p}_u$, which means $R_{ui}, \forall u \in \mathcal{P}_r$ should be stored in machine r. Similarly, $R_{ui}, \forall i \in \mathcal{Q}_r$ should also be stored in machine r. Thus, we put two copies of R_{ui} in the distributed system. One is divided by the index of the user ID, and the other is divided by the index of the item ID. From algorithm (1), we can find that maintaining a prediction rating matrix $\hat{\boldsymbol{R}}$ can accelerate the calculation process of UIWMF. Accordingly, additional space of size $|\mathcal{R}|$ is needed to store $\hat{\boldsymbol{R}}$. However, if we follow the same partitioning strategy as $\boldsymbol{R}$, one iteration of $\boldsymbol{P}$ needs to update all the elements in $\hat{\boldsymbol{R}}$ according to the update rules. In order to maintain the consistency of $\hat{\boldsymbol{R}}$, we need to broadcast the whole matrix to update the related $\hat{R}_{ui}$ held by the $\mathcal{Q}_r$. We need to broadcast $\hat{\boldsymbol{R}}$ twice in each iteration, which will cause huge communication overhead. To solve this problem, before the update of $\boldsymbol{q}_i$ and $\boldsymbol{p}_u$, we calculate $\hat{\boldsymbol{R}}$ in advance. The additional time complexity for each iteration is $\mathcal{O}(k|\mathcal{R}|)$, which is a relatively acceptable computational overhead compared to the total time complexity. The complete partitioning strategy is shown in Fig. 1.

4.2 Data communication optimization

Since we divide the feature matrices $\boldsymbol{P}$ and $\boldsymbol{Q}$ into different machines, the communication overhead of updating feature vector is inevitable. For example, during the update of the feature vector $\boldsymbol{p}_u$, we need to get $\boldsymbol{q}_i, i \in R_u$. A simple strategy used in Yu et al. (2014) and Schelter et al. (2013) is to broadcast the matrix $\boldsymbol{Q}$ to all other nodes in the cluster. Hence,

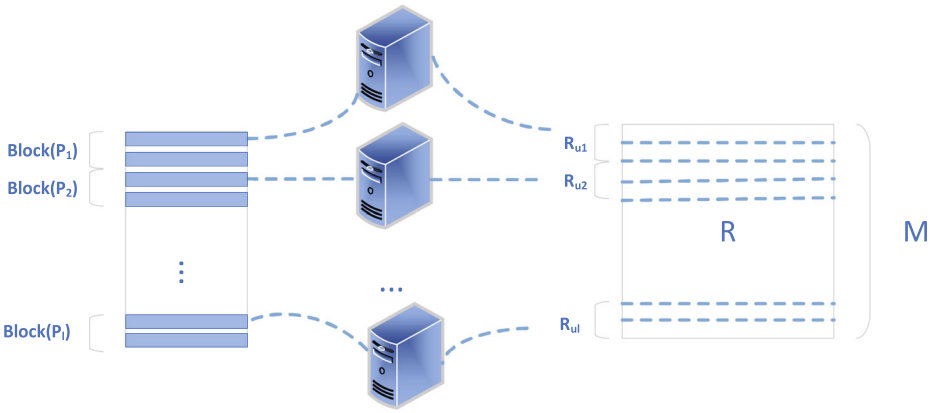

Fig. 1 Model and data partitioning strategies

$(l-1)(m+n)$ feature factor need to be sent over the network in each iteration. The shortcoming of this strategy is obvious, i.e., the network overhead will increase linearly as the number of machines increases, which leads to poor scalability of the distributed algorithm.

To solve the above problem, according to the previous work (Apache Spark 2019b), which cached the block information transmitted over the network to avoid repeated transmissions, we reduce the network overhead by building appropriate caches. Particularly, we

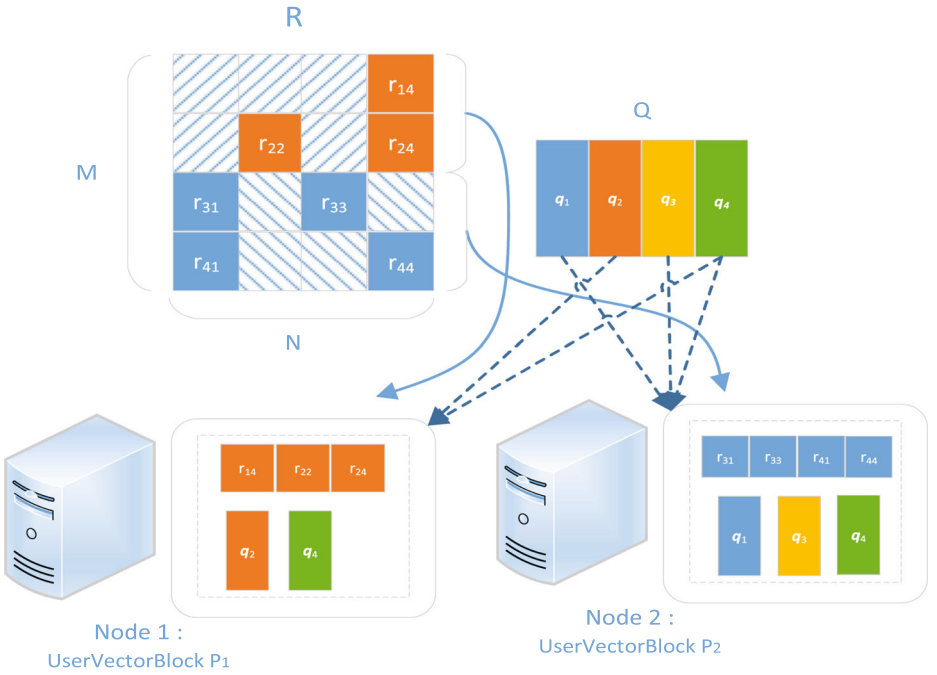

Fig. 2 Update process of user feature vectors

construct the cached in-block and out-block information to cache user and item feature vectors that each machine needs to fetch in each iteration. An example for updating the user feature vector is shown in Fig. 2. The number of machines l is 2. The user feature vectors $\boldsymbol{p}_1$, $\boldsymbol{p}_2$ and the item feature vectors $\boldsymbol{q}_1, \boldsymbol{q}_2$ are divided to machine 1. Accordingly, machine 2 holds the user feature vectors $\boldsymbol{p}_3$, $\boldsymbol{p}_4$ and the item feature vectors $\boldsymbol{q}_3, \boldsymbol{q}_4$. Updating the user feature vector only needs to obtain the item feature vectors $\boldsymbol{q}_2, \boldsymbol{q}_4$ for node 1, and $\boldsymbol{q}_1, \boldsymbol{q}_3, \boldsymbol{q}_4$ for node 2. There is no need to transmit the whole item feature vectors ($\boldsymbol{q}_3$ and $\boldsymbol{q}_4$) from node 2 to node 1, and the item vectors ($\boldsymbol{q}_4$) that need to be reused in node 1 only need to be sent once. Based on this principle, we construct an information routing table to record which item feature vector should be sent to node 1 and node 2. After using the cache strategy, the number of user feature factors needed to be sent per iteration becomes $\sum_{r=1}^{l} \sum_{i=1}^{N} \mathbb{1}\{i \in \mathcal{P}_r\}$. Recalling that the rating matrix is extremely sparse, $\sum_{i=1}^{N} \mathbb{1}\{i \in \mathcal{P}_r\}$ is much smaller than N, which greatly reduces the data traffic. Besides, as the number of machines increases, the values of $\sum_{i=1}^{N} \mathbb{1}\{i \in \mathcal{P}_r\}$ will be further reduced. Therefore, this strategy can effectively enhance the scalability of the distributed algorithm.

Algorithm 2 Creation process of user's in-block and out-block caches.

Require: RDD_0: an RDD object of the original rating data; $uBlock$: Number of user block; $iBlock$: Number of item block.

Ensure: RDD_{UO}: an RDD object of user out-block information; RDD_{UI}: an RDD object of the user in-block information.

1: **//Create user in-block information**
2: $RDD_{UB} \leftarrow dataBlockByUser(RDD_0, uBlock)$;
3: $RDD_{UBL} \leftarrow RDD_{UB}.\boldsymbol{map}$
4: $(userIds, itemLocalIndices, ratings) \leftarrow$
5: $convertItemIdtoLocalBlockIndex(RatingBlock)$
6: **end** ***map***
7: $RDD_{UI} \leftarrow RDD_{UBL}.\boldsymbol{groupByKey}(userBlockId)$
8: $itemLocalIndices \leftarrow Encode((itemBlockId, LocalIndices))$
9: $InblockBlock \leftarrow COOIntoCSC((userIds, itemLocalIndices, ratings))$
10: **end** ***groupByKey***
11: //Create user out-block information
12: $RDD_{UO} \leftarrow RDD_{UI}.\boldsymbol{map}$
13: for $(r : userIds)$ do
14: $outBlock(iBlockId) \leftarrow r$;
15: **endfor**
16: **end** ***map***
17: **return** RDD_{UO} and RDD_{UI}.

Algorithm 2 describes the creation process of in-block and out-block caches on Spark. To create an in-block cache for user feature vectors, first of all, we need to divide the rating data RDD_0 according to the user ID by a hash method and generate the corresponding user block ratings RDD_{UB}. Then by converting item id to local block index, RDD_{UBL} is obtained. Lastly, the in-block information RDD_{UI} is derived by applying the $groupByKey$ operation, where we use two strategies to effectively reduce the memory overhead. 1) we encode the item block id and local index to a single integer; 2) we convert the data from coordinate list (COO) format into compressed sparse column (CSC) format. After that, RDD_{UO} is generated by performing the map operation on RDD_{UI}. It contains for every item block

a two-dimensional array $outBlock$ indicating which item blocks should each user vector be sent to. The in-block and out-block caches of item feature vectors can be generated in the same way.

Algorithm 3 Parallel training process of the DUIWMF.

Require: K: length of latent factor vector; h: number of iterations; RDD_0: an RDD object of the original rating data; RDD_{UO}, RDD_{UI}, RDD_{IO} and RDD_{II}: in-block and out-block caches of user and item.

Ensure: RDD_{UF}: an RDD object of user feature matrix; RDD_{IF}: an RDD object of item feature matrix.

Compute item popularity RDD_{Ci} and user activity RDD_{Du} with (3)
$RDD_{UF} \leftarrow initial(RDD_{UO}, K)$;
$RDD_{IF} \leftarrow initial(RDD_{IO}, K)$;
for $i = 0$ to $(h - 1)$ **do**
    $RDD_{H^{q1}}$ and $RDD_{H^{q2}} \leftarrow RDD_{IF}.\boldsymbol{aggregate}$;
    $RDD_{UIF} \leftarrow RDD_{UI}.\mathbf{join}.(RDD_{UF})$;
    $RDD_{IOF} \leftarrow RDD_{IO}.\mathbf{join}(RDD_{IF}).\mathbf{groupByKey}(userBlockId)$;
    $RDD_{UF} \leftarrow RDD_{UIF}.\mathbf{join}(RDD_{IOF})$;
        Update the $userFactor$ with (11);
    end ***join***
    $RDD_{UF}.persist()$
    $RDD_{H^{p1}}$ and $RDD_{H^{p2}} \leftarrow RDD_{UF}.\boldsymbol{aggregate}$
    $RDD_{IIF} \leftarrow RDD_{II}.\mathbf{join}.(RDD_{IF})$;
    $RDD_{UOF} \leftarrow RDD_{UO}.\mathbf{join}(RDD_{UF}).\mathbf{groupByKey}(itemBlockId)$;
    $RDD_{IF} \leftarrow RDD_{IIF}.\mathbf{join}(RDD_{UOF})$;
        Update the $itemFactor$ with (12);
    end ***join***
    $RDD_{IF}.persist()$
end for
return RDD_{UF} and RDD_{IF}.

4.3 Parallel training process of DUIWMF

Algorithm 3 presents the parallel training process of DUIWMF on Spark. At the beginning of the algorithm, we need to initialize variables corresponding to the item popularity and the user activity. Then both the user feature matrix and the item feature matrix need to be initialized by the *map* operation. It is worth noting that using the out-block information to perform the initialization operation of the feature matrix can unify the partition of the feature matrix and the out-of-block information, such that the additional shuffle overhead during iterations can be avoided. Next, we update the user and item feature matrices iteratively. To update the user feature vector, we first calculate the caches H^{q1} and H^{q2}. Particularly, the corresponding matrix inner product calculation is performed by the $aggregate$ operation. After that, to aggregate the user feature vectors and the corresponding ratings, the user's in-block information RDD_{UI} needs to be combined with the user feature vector matrix RDD_{UF} by $join$ operator, and the corresponding result is stored in RDD_{UIF}. Next, joining the item feature vector RDD_{IF} with the item out-block information RDD_{IO}, we can get the transmission relationship between the user block and the item feature vectors, and the transmission

of item feature vector information takes place in operation $groupByKey$. Finally, the user feature matrix RDD_{UF} is updated by conducting the $join$ operation to group RDD_{UIF} and RDD_{IOF}. After that, we need to persist RDD_{UF} to reuse in later calculations. It is worth noting that we need to perform checkpoint operation periodically to truncate RDD lineage to prevent spark stack overflow exceptions. Similarly, we can update the item feature matrix in the same way. We illustrate the parallel training process of DUIWMF in Fig. 3.

5 Experiments

In this section, we compare the performance of the proposed DUIWMF with several baseline models, including the recommendation accuracy and the model training efficiency. All the experiments are performed on the Alibaba Cloud platform where we employ a cluster consisting of eight "ecs.r5.xlarge" (4 cores, 32GB RAM) instances. Each node executes in CentOS 7.4 and is configured with Hadoop 2.8.5 and Spark 2.4.3. The algorithm is implemented on Scala 2.11.12.

5.1 Setting

Datasets: We use three public recommendation datasets that are widely used in previous works (He et al. 2016, 2018): Amazon Movies, Yelp and Ciao. We transfer the dataset into the implicit data following the common practice in Rendle et al. (2009) and He et al. (2018). Specifically, each entry in the original rating data is marked as 0/1, indicating whether the user has interacted with the item. In order to make the recommendation results more

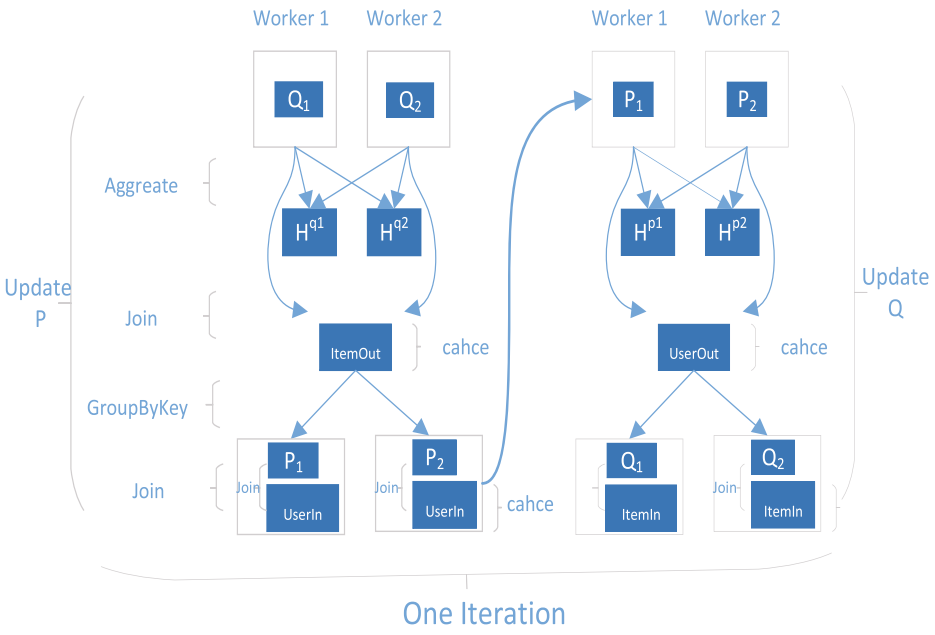

Fig. 3 Parallel training process of the DUIWMF model

meaningful and easier to be explained intuitively, we remove the users with less than 10 interactions. The statistics of the selected datasets are shown in Table 2.

Metrics: We adopt the leave-one-out (He et al. 2016, 2018; Rendle et al. 2009) protocol to evaluate the recommendation accuracies. Two evaluation metrics are used to evaluate the performance of our model, i.e., *hit ratio* (HR) and *normalized discounted cumulative gain* (NDCG). Both metrics are widely used to evaluate the recommendation performance with implicit feedbacks (He et al. 2016, 2018; Li et al. 2018). Since there is only one test item for each user, we set the threshold of the ranking list to 100 to ensure a relatively large HR and NDCG. Particularly, HR is computed by

$$\mathtt{HR} = \sum_{i \in u^{\mathtt{test}}} \frac{\mathbb{I}\{\mathtt{rank}(i) \leq 100\}}{|u^{\mathtt{test}}|}, \tag{14}$$

where $\mathbb{I}(\cdot)$ is the indicator function which returns 1 only when the test item are included in the recommendation list, and 0 otherwise. The metric NDCG emphasizes the importance of the top ranks by logarithmically discounting ranks, which is computed by

$$\mathtt{NDCG} = \frac{1}{|u^{\mathtt{test}}|} \sum_{i \in u^{\mathtt{test}}} \frac{\log(\hat{r}+1)}{\log(\mathtt{rank}(i)+1)}, \tag{15}$$

where $\hat{r}$ denotes the perfect ranking of the test item which is always set to 1 in the experiments.

Comparison methods: We compared our proposed DUIWMF models with the following methods:

- *ALS* (Hu et al. 2008): This is the conventional ALS method that optimizes the whole-data based MF model. The negative feedback information is obtained by assigning uniform weights to the missing data.
- *FastALS* (He et al. 2016): It is an implicit feedback recommendation algorithm based on MF, which assigns nonuniform weights to the missing data according to the item popularity, and uses the element-wise alternating least-squares (eALS) technique to update the model.
- (Li et al. 2018): WRMF-U is a whole-data based implicit MF model, which assigns nonuniform weights to the missing data based on the user activity. It also uses the eALS technique to update the model.
- *BPR* (Rendle et al. 2009):BPR is the first work that introduces the pairwise preference assumption into the task of recommendation with implicit feedback.
- *SLIM* (Ning and Karypis 2011): SLIM is a sparse linear method for top-K recommender systems which improves upon the traditional item-based nearest neighbor collaborative filtering approaches by learning item-item similarity matrix directly from the rating data.

Table 2 Statistics of the evaluation datasets

DataSet	#Review	#Item	#User	Sparsity
Amazon	5260450	171074	121725	97.47%
Yelp	731671	25815	25677	99.89%
Ciao	40189	13226	1141	99.73%

Parameter settings: The parameter settings in the experiment can be obtained from the experimental results or proposed by the literature work. Specifically, for all the matrix factorization-based approaches, we employ grid search in {2, 0.5, 0.1, 0.05, 0.01, 0.005, 0.001} to find out the optimal setting for regularization parameter. The number of latent factors is set to 64, 128, respectively to study the influence of different feature length on recommendation accuracy. The weight of observed interactions, we set it uniformly as 1, a default setting by previous works (He et al. 2016; Li et al. 2018). For all models learned through SGD, the learning rate is tuned on { 0.1, 0.01, 0.001, 0.0001}.

5.2 Effects of parameters

In this section, we analyze the influence of parameters on the recommendation accuracy. There are four major parameters determining the weighting scheme in our DUIWMF model, i.e. t_0, α, β, and η. Specifically, t_0 determines the overall weight of the missing data, α controls the significance level of popular items over unpopular ones, β controls the significance level of active users over inactive ones, η controls the impact of the user activity and the item popularity on the weight of a single missing value. Since NDCG and HR have similar variation tendencies, we do not distinguish both metrics in the following analysis.

First, we set a uniform weight distribution, i.e. $\alpha = 0, \beta = 0, \eta = 1$, and vary t_0 to study how the weight of missing data effect the performance. For the *Amazon* dataset, Fig. 4a1 shows that, the best performance locates $t_0 = 64$, corresponding to the uniform weight that each missing data is 0.000374 (t_0/N). Then, we fixed t_0 to 64 and study the impact of item popularity α on recommendation accuracy. As can be seen from Fig. 4a2, the recommendation accuracy reaches the optimum when $\alpha = 0.5$, which means that the weighting strategy based on the user activity is effective for improving the recommendation performance. Note that in order to keep the overall weight of the missing data unchanged, when η is changed,

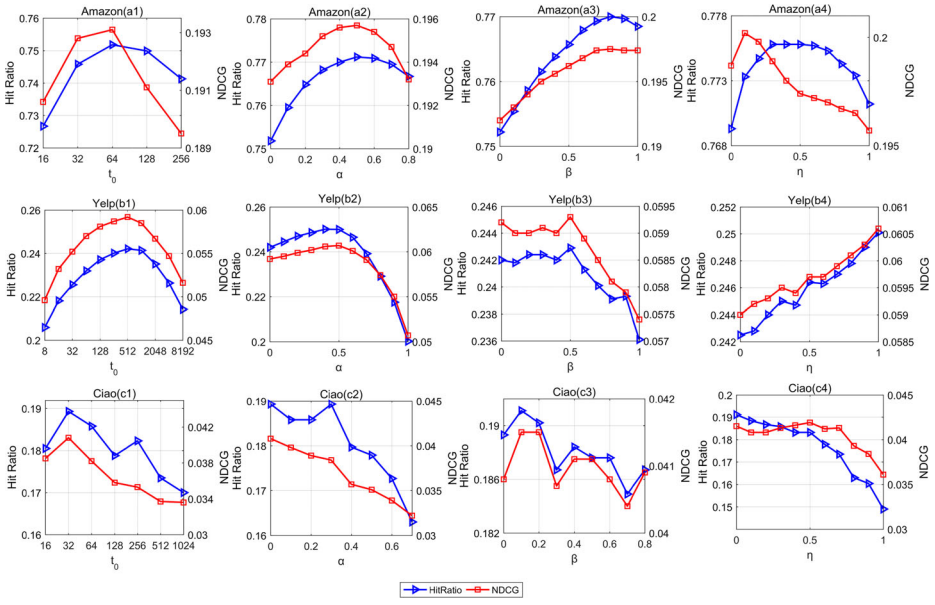

Fig. 4 Effect of parameters on different datasets

the corresponding t_0 should also be changed by $t_0 = t_{best}(\eta/N + (1-\eta)/M)$. Next, we set t_0 to the best value, i.e., $t_0 = 64$, and check the performance change through varying β. The result is revealed in Fig. 4a3, where the recommendation accuracy is enhancing as the increase of β and reaches the maximum when $\beta = 0.8$. This result also shows the weighting strategy based on user activity can effectively improve recommendation accuracy. Moreover, by fixing t_0, α, and β to the optimal value, i.e., $t_0 = 64$, $\alpha = 0.5$, and $\beta = 0.8$, and changing η, we study the influence of different weights of user popularity and item popularity strategies on the final recommendation accuracy. Figure 4a4 shows that the best value is obtained when $\eta = 0.3$, which reach the performance not only better than the weighting strategy of using the user activity alone ($\eta = 1$), but also better than the weighting strategy of using the item popularity alone ($\eta = 0$). It also illustrates that the weighting strategy based on user activity contributes more to the improvement of the overall recommendation accuracy than the weighting strategy of item popularity.

We conduct experiments on the *Yelp* dataset and the *ciao* dataset, while different results are obtained. Particularly, as shown in Fig. 4b4, the final recommendation accuracy of the $Yelp$ dataset increases with the increase of η and reaches the optimal value when $\eta = 1$, which means it is reduced to a weighting strategy only based on the item popularity, and assigning weights to the user activity cannot contribute to the improvement of the recommended accuracy. As for the *ciao* dataset, Fig. 4c2 reveals some differences. Different from the other two datasets, the increase of α will lead to a reduction of the recommendation performance in the *ciao* dataset, which means that the weighting strategy of item popularity will cause damage to the final recommendation accuracy.

5.3 Recommendation accuracy

To verify the performance of our model, we compare DUIWMF with three baselines models on *Amazon*, *Yelp*, and *Ciao* datasets. The experimental results are shown in Fig. 5, where the length of the feature vector K is set to 64 and 128, respectively. Since the similarity matrix of SLIM model is fixed, we put the result of SLIM model at the end. From Fig. 5, we have the following results:

- We find that different lengths of K have different effects on the recommendation accuracy in different datasets. For all datasets, when the length of the K changes from 64 to 128, the recommendation accuracies of all models are improved to a certain extent.
- Both FastALS and WRMF-U put forward different weight strategy on the basis of ALS to improve the accuracy of recommendation. However, Both FastALS and WRMF-U do not work for all datasets, which is consistent with the conclusions drawn in He et al. (2018). For the *Yelp* dataset, WRMF-U has little improvement compared to ALS in terms of the recommendation accuracy. However, FastALS has a significant improvement over ALS, which means only the weighting strategy based on the item popularity is helpful to improve recommendation accuracy. On the contrary, only the weighting strategy based on the user activity contributes to the improvement of recommendation accuracies on the *Ciao* dataset.
- For *Amazon* datasets, the NDCG value of the BPR model is better than all other models. We think BPR's strength in ranking top items is due to its optimization objective, which is a pairwise ranking function tailored for ranking correct item high. Similar results appear in He et al. (2016).
- DUIWMF achieves the best result over all the methods on each dataset. We believe that our proposed weighting strategy based on user activity and item popularity does a favor

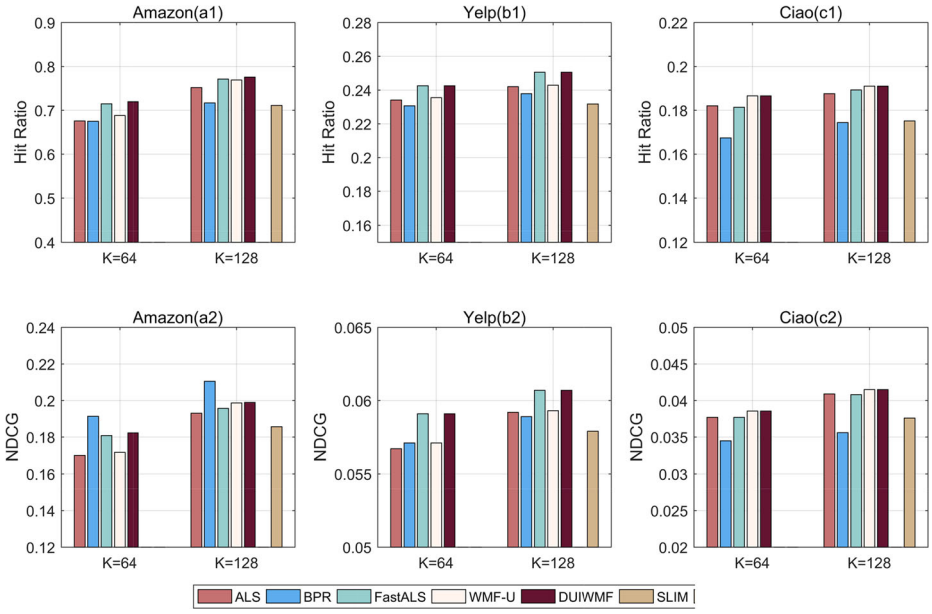

Fig. 5 Recommendation accuracy comparisons with different datasets ($K = 64$, $K = 128$)

to the improvement of recommendation accuracies. Specially, we find that DUIWMF and FastALS have the same recommendation accuracy on the *Yelp* dataset. This is due to the fact that DUIWMF can be adaptively transformed into one particular weighting strategy based on either the user activity or the item popularity if only one weighting strategy is more effective.

5.4 Recommendation efficiency

In this section, we first compare the offline model training efficiency of ALS, FastALS and DUIWMF on Spark. We use ALS learning algorithm implemented in spark machine learning library MLlib and name it Spark-ALS. In order to verify the correctness of our proposed communication strategy, we use the method in Yu et al. (2014) to implement the distributed FastALS on spark, which updates the model by broadcasting the feature matrix, and we name it Broadcast-FastALS. Since WRMF-U performs model training using a learning algorithm similar to FastALS and shares almost the same training speed, we just show the results of Broadcast-FastALS. Additionally, we analyze the effect of a different number of blocks on the execution efficiency. Besides, we analyze the scalability of DUIWMF on Spark.

5.4.1 Average execution time on different datasets

Experiments are performed to compare the average model training time of DUIWMF, Broadcast-FastALS and Spark-ALS on relatively large datasets, i.e., *Amazon* and *Yelp*. In these experiments, the length of K is set to 32, 64, and 128, respectively. The number of iterations is set to 50, and the number of Spark worker node is set to 4. The experimental results are presented in Fig. 6.

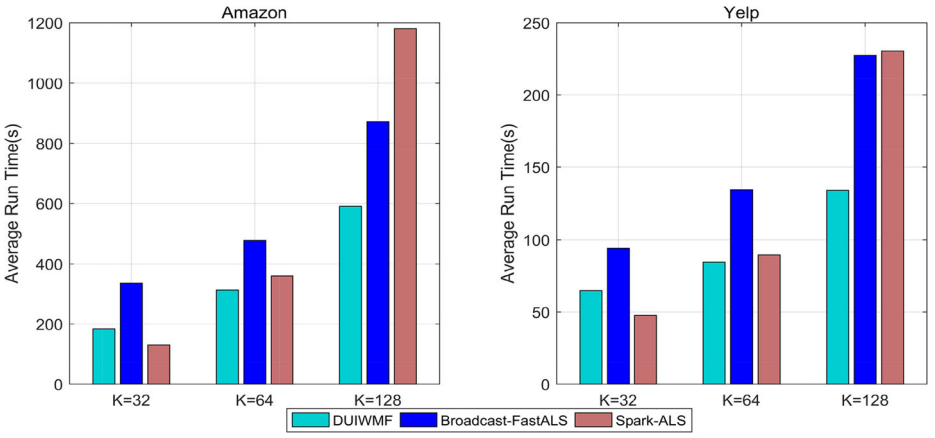

Fig. 6 Average execution time comparisons with different K

We can find that, as K increases, DUIWMF shows more and more performance advantages over Spark-ALS. For example, when K is 128, DUIWMF is 2.01 times faster than Spark-ALS on the *Amazon* dataset and is 1.72 times faster than Spark-ALS on the *Yelp* dataset. The reason is that DUIWMF has lower time complexity ($\mathcal{O}(|R|k + (M + N)k^2)$) compared to Spark-ALS ($\mathcal{O}(|R|k^2 + (M + N)k^3)$). However, it is worth noting that, when the length of the feature vector is $K = 32$, DUIWMF cannot benefit from low time complexity, i.e., DUIWMF has a longer execution time than Spark-ALS. The main reasons are as follows: 1) we need to calculate prediction rating matrix $\hat{\boldsymbol{R}} = \boldsymbol{p}_u^T \boldsymbol{q}_i$ in each iteration to avoid expensive shuffle overhead. However, this computational overhead cannot be ignored, especially when the number of blocks increases; 2) Our weighting strategy allocates different weights to each missing data and needs a space of size $M \times N$ to store the weight information, which is a pretty large space. In order to solve this problem, we calculate the weight information in the iterative process, which adds additional computational overheads to the algorithm. Another important finding is that the average execution time of Broadcast-FastALS is even longer than Spark-ALS for *Yelp* dataset which is a relatively small dataset. After analysis, we find that this is because user feature matrix and item feature matrix need to be broadcast in each iteration, which involves two major operations in spark: *collect* and *broadcast*, and these two operations involve a lot of communication overhead. For Amazon data, due to the low time complexity of Broadcast-FastALS, the execution time is less than that Spark-ALS. However, on the whole, the execution time of Broadcast-FastALS is much longer than DUIWMF, which is benefited from our optimization in communication to greatly improve the performance of the algorithm.

5.4.2 Average execution time for different number of blocks

To verify the impact of a different number of blocks used in DUIWMF, we configure 4 slave nodes (16 cores) and evaluate the case with $N \in \{16, 32, 48, 64\}$ blocks. The experimental results are presented in Fig. 7. It can be found when the number of blocks is larger than the number of cluster CPU cores, the training time for DUIWMF increases. Although our proposed in-block and out-block strategy can effectively reduce the feature vectors that are repeatedly sent in the same block, with the increase of the number of blocks, the advantage

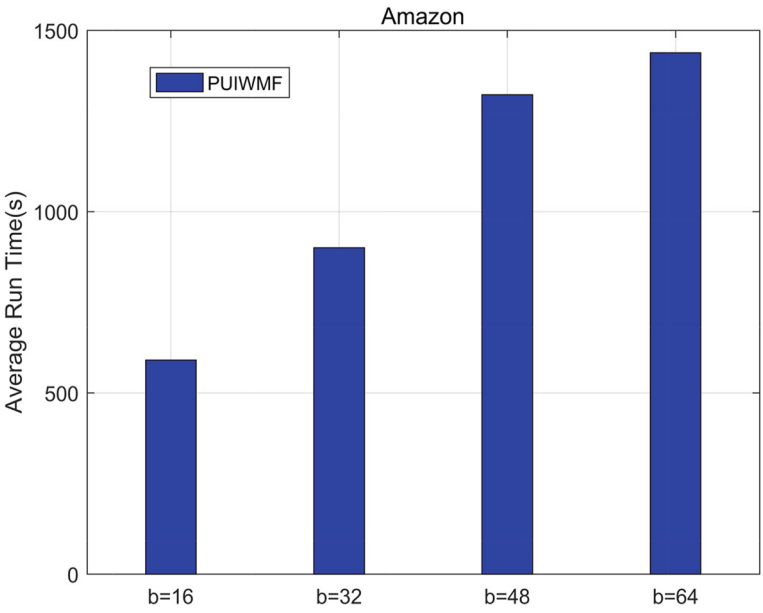

Fig. 7 Average execution time for different number of blocks

of this strategy is weakened. There are more and more repetitive feature vectors sent by each node, resulting in an increase of communication overhead, which further degrades the execution performance.

5.4.3 Average execution time for different cluster scales

To verify the scalability of DUIWMF, experiments are performed on the Spark cloud platform with a different number of nodes. Particularly, we evaluate the speedup of DUIWMF. Let $T(i, sa)$ be the execution time for carrying out DUIWMF in dataset i employing one work node. The speedup of using p machines is defined as $T_{i,p}/T_{i,sa}$, where $T_{i,p}$ is the time taken on p machines. The number of slave nodes is gradually increased from 1 to 8. The experiment results are presented in Fig. 8.

It can be found that the average execution time for the *Yelp* dataset does not always decrease as the number of nodes increases. Specifically, in the beginning, as the number of nodes increases, the execution time for DUIWMF decreases obviously. However, when the number of nodes is greater than 4, the training time begins to increase. This is because the execution time of the algorithm is determined by the number of users M, the number of items N, the number of ratings $|\mathcal{R}|$, and the number of feature vector blocks l. Recall that the best number of blocks is equal to the number of CPU cores in the cluster. When M, N, and $|\mathcal{R}|$ are fixed, with the increase of the number of nodes in the cluster, the number of matrix blocks needs to increase, which accordingly leads to a decrease in the number of calculation tasks allocated to each node and an increase in the communication overhead of the cluster. Particularly, when the number of nodes exceeds the equilibrium point (the point where the average execution time starts to increase), the communication overhead caused by the parallelism grows faster than the performance improvement. Thus, the average execution time will increase. In addition, larger M, N, and $|\mathcal{R}|$ will augment the threshold, which is

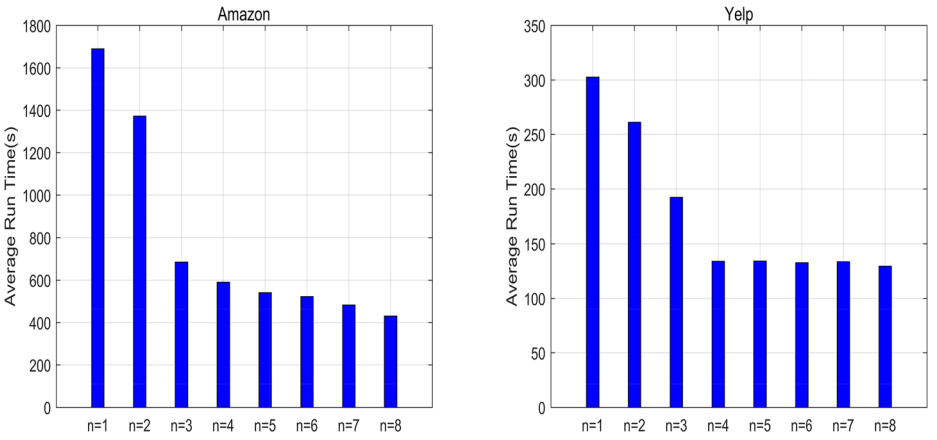

Fig. 8 Average execution time for different cluster scales

consistent with the results in the *Amazon* dataset. Besides, we can infer that the equilibrium point of *Amazon* will appear if we further increase the cluster scales.

The corresponding results of speedup ratios are shown in Fig. 9. It can be inferred that, for large-scale recommended datasets, the increase of parallelism will greatly improve the speedup.

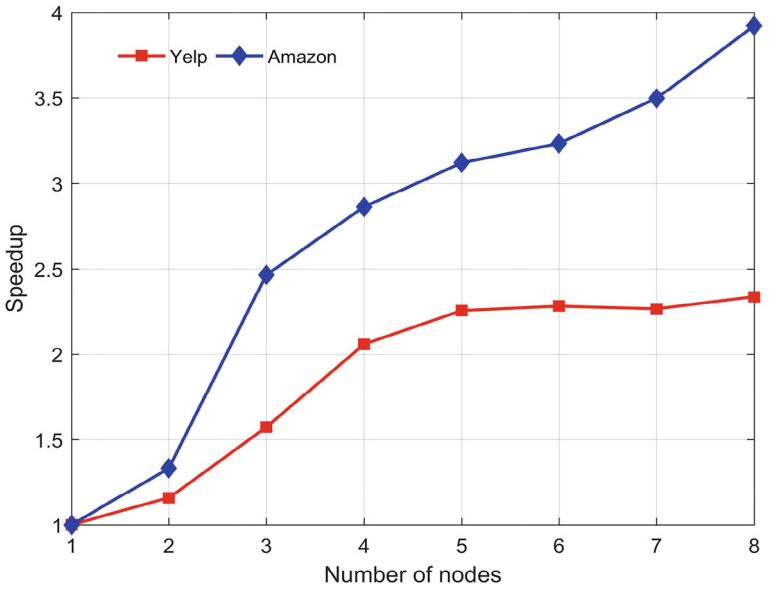

Fig. 9 Speedup for different cluster scales

6 Conclusion

In this paper, we first propose a weighted MF recommendation algorithm UIWMF, which assigns different weights to every missing data based on user activity and item popularity. The proposed approach can get negative feedback information more realistically and lead to a higher recommendation accuracy. Then we further modify UIWMF and propose a distributed UIWMF (DUIWMF) algorithm based on Spark, which adopts efficient parallel learning algorithm for model training and utilize cached in-block and out-block information to effectively reduce the communication overhead in a distributed environment. Experiments have been conducted on three public datasets, which demonstrate that compared with the baseline MF methods, the proposed DUIWMF model has comparable performance in terms of the recommendation accuracy and the model training efficiency. Our future work will focus on how to update the model in real-time scenarios where the implicit feedback data is generated continuously, and further combine the flow processing capability of Spark to deal with large-scale real-time data.

Acknowledgments The authors deeply appreciate the anonymous reviewers for their comments on the manuscript. The research was partially funded by the National Key R&D Program of China (Grant Nos. 2018YFB1003401), the National Natural Science Foundation of China (Grant Nos. 61872127, 61572175, 61751204, 61472124), the National Outstanding Youth Science Program of National Natural Science Foundation of China (Grant No. 61625202), the Key Program of National Natural Science Foundation of China (Grant No. 61432005), and the Natural Science Foundation of Hunan Province, China(Grant No. 2018JJ2022)

References

Apache Spark (2019a). website. http://spark-project.org.

Apache Spark (2019b). mllib-als. http://spark.apache.org/docs/latest/.

Apache Flink (2019). website. http://flink.apache.org.

Chen, C., Liu, Z., Zhao, P., Li, L., Zhou, J., Li, X. (2018). Distributed collaborative hashing and its applications in ant financial. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge discovery and data mining* (pp. 100–109): ACM.

Devooght, R., Kourtellis, N., Mantrach, A. (2015). Dynamic matrix factorization with priors on unknown values. In *Proceedings of the 21th ACM SIGKDD international conference on knowledge discovery and data mining* (pp. 189–198): ACM.

Gemulla, R., Nijkamp, E., Haas, P.J., Sismanis, Y. (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* (pp. 69–77): ACM.

He, X., Zhang, H., Kan, M.-Y., Chua, T.-S. (2016). Fast matrix factorization for online recommendation with implicit feedback. In *Proceedings of the 39th International ACM SIGIR conference on Research and Development in Information Retrieval* (pp. 549–558): ACM.

He, Y., Wang, C., Jiang, C. (2018). Correlated matrix factorization for recommendation with implicit feedback. *IEEE Transactions on Knowledge and Data Engineering*, *31*(3), 451–464.

He, X., Tang, J., Du, X., Hong, R., Ren, T., Chua, T.-S. (2019). Fast matrix factorization with nonuniform weights on missing data. *IEEE transactions on neural networks and learning systems*.

Hu, Y., Koren, Y., Volinsky, C. (2008). Collaborative filtering for implicit feedback datasets. In *ICDM*, (Vol. 8 pp. 263–272): Citeseer.

Kabbur, S., Ning, X., Karypis, G. (2013). Fism: factored item similarity models for top-n recommender systems. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining* (pp. 659–667): ACM.

Karydi, E., & Margaritis, K. (2016). Parallel and distributed collaborative filtering: a survey. *ACM Computing Surveys (CSUR)*, *49*(2), 37.

Koren, Y., Bell, R., Volinsky, C. (2009). Matrix factorization techniques for recommender systems. *Computer* (8), 30–37.

Li, H., Diao, X., Cao, J., Zheng, Q. (2018). Collaborative filtering recommendation based on all-weighted matrix factorization and fast optimization. *IEEE Access*, *6*, 25248–25260.
Liang, D., Charlin, L., McInerney, J., Blei, D.M. (2016). Modeling user exposure in recommendation. In *Proceedings of the 25th International Conference on World Wide Web, International World Wide Web Conferences Steering Committee* (pp. 951–961).
Marlin, B., Zemel, R.S., Roweis, S., Slaney, M. (2012). Collaborative filtering and the missing at random assumption, arXiv:1206.5267.
Ning, X., & Karypis, G. (2011). Slim: Sparse linear methods for top-n recommender systems. In *2011 IEEE 11th International Conference on Data Mining* (pp. 497–506): IEEE.
Pan, R., Zhou, Y., Cao, B., Liu, N.N., Lukose, R., Scholz, M., Yang, Q. (2008). One-class collaborative filtering. In *2008 Eighth IEEE International Conference on Data Mining* (pp. 502–511): IEEE.
Pilászy, I., Zibriczky, D., Tikk, D. (2010). Fast als-based matrix factorization for explicit and implicit feedback datasets. In *Proceedings of the fourth ACM conference on Recommender systems* (pp. 71–78): ACM.
Rendle, S., & Schmidt-Thieme, L. (2008). Online-updating regularized kernel matrix factorization models for large-scale recommender systems. In *Proceedings of the 2008 ACM conference on Recommender systems* (pp. 251–258): ACM.
Rendle, S., Freudenthaler, C., Gantner, Z., Schmidt-Thieme, L. (2009). Bpr: Bayesian personalized ranking from implicit feedback. In *Proceedings of the twenty-fifth conference on uncertainty in artificial intelligence* (pp. 452–461): AUAI Press.
Schelter, S., Boden, C., Schenck, M., Alexandrov, A., Markl, V. (2013). Distributed matrix factorization with mapreduce using a series of broadcast-joins. In *Proceedings of the 7th ACM Conference on Recommender Systems* (pp. 281–284): ACM.
Steck, H. (2010). Training and testing of recommender systems on data missing not at random. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining* (pp. 713–722): ACM.
Volkovs, M., & Yu, G.W. (2015). Effective latent models for binary feedback in recommender systems. In *Proceedings of the 38th international ACM SIGIR conference on research and development in information retrieval* (pp. 313–322): ACM.
Yu, H.-F., Hsieh, C.-J., Si, S., Dhillon, I.S. (2014). Parallel matrix factorization for recommender systems. *Knowledge and Information Systems*, *41*(3), 793–819.
Yun, H., Yu, H.-F., Hsieh, C.-J., Vishwanathan, S., Dhillon, I. (2014). Nomad: Non-locking, stochastic multi-machine algorithm for asynchronous and decentralized matrix completion. *Proceedings of the VLDB Endowment*, *7*(11), 975–986.
Zhu, Z., Liu, T., Jin, S., Luo, X. (2018). Learn and pick right nodes to offload. In *Proceedings of IEEE GLOBECOM* (pp. 1–6).

Affiliations

Lian Chen[1] · Wangdong Yang[1] · Kenli Li[1] · Keqin Li[1,2]

✉ Kenli Li
lkl@hnu.edu.cn

Lian Chen
chenlian@hnu.edu.cn

Keqin Li
lik@newpaltz.edu

[1] College of Information Science and Engineering, Hunan University, Changsha, Hunan 410008, China

[2] Department of Computer Science, State University of New York, New Paltz, NY 12561, USA