A Lossless Irregular Tensor Factorization for Cross-domain Recommender System with Genetic Algorithm on Spark

Guodong XUE† Seiki MIYAMOTO†† Takumi ZAMAMI††† and Hayato YAMANA††††

† Graduate School of Fundamental Science and Engineering, Waseda University 3-4-1 Okubo, Shinjuku-ku, Tokyo, 169-8555 Japan

†† Faculty of Science and Engineering, Waseda University 3-4-1 Okubo, Shinjuku-ku, Tokyo, 169-8555 Japan

††† Ceo's office of DMM.com 3-2-1 Roppongi, Minato-ku, Tokyo, Japan

†††† Faculty of Science and Engineering, Waseda University 3-4-1 Okubo, Shinjuku-ku, Tokyo, 169-8555 Japan

E-mail: † xueguodong@fuji.waseda.jp, †† smiyamoto14@yama.info.waseda.ac.jp, †††zamami-takumi@dmm.com †††† yamana@waseda.jp

Abstract Users may have sufficient experience in their focused domains but lack experience in unfamiliar domains, leading to a drop in the accuracies of recommender systems owing to the unavailability of data with which to train such domains. Therefore, the cross-domain recommender system, which analyzes user information from both the user's familiar and unfamiliar domains, has become an important emerging subject of research. Weighted irregular tensor factorization (WITF) has been proposed as an algorithm to leverage cross-domain feedback across all users and considers the dataset as an irregular tensor containing different items for each domain. According to tensor theory, an irregular tensor must be transferred into a regular tensor (in which each domain contains the same items), before executing tensor factorization to obtain the user's latent factors. Therefore, WITF must assign a weight on each domain to ensure minimal data loss during the transformation. Searching for the optimal domain weight configuration is a complex and time-consuming procedure. Thus, WITF adopts an empirical domain weight configuration. In this study, we propose a model that combines Apache Spark® and a genetic algorithm to search efficiently for the optimal domain weight configurations for WITF, and then uses the optimal configuration obtained to generate more precise users' latent factors to make recommendations. Experimental evaluation using two datasets of different sizes show 4.2% and 6.3% higher accuracies in comparison with the WITF alone

1. Intorduction

With the development of the Internet, the number and variety of content on the World Wide Web continue to increase, and user demand is higher for searches and recommendations. Recommender systems, which make accurate and personalized recommendations using users' preferences, interests, and experiences, have become more and more important and are widely used in many fields.

Collaborative filtering (CF) [1] is the algorithm most utilized in recommender systems, and it has been used in many applications, including Amazon and Netflix. However, the data sparsity problem (many users do not have data in the CF recommender systems) always affects the performance of CF recommender systems. In general, users always have sufficient experience in their focused domains. In some researches, the recommender system uses data from other domains to supply more data, and thus generates more accurate recommendations. Therefore, the cross-domain recommender systems, which leverage cross-domain feedback data across all users to learn the user's latent factors, have become an important emerging research topic.

Weighted irregular tensor factorization (WITF) [2] is a model that considers cross-domain information by considering the multiple domains dataset as an irregular tensor. In this model, a weight must be assigned to each domain to ensure minimal data loss after transferring into a regular tensor, followed by processing the tensor factorization to make recommendations. However, the WITF model adopts an empirical domain weight configuration that decides the weight based on the number of items in each domain. Therefore, a method for selecting the optimal weight configuration for the WITF model is necessary to improve its performance.

Genetic algorithms [3] are commonly used to generate high-quality solutions for optimization and search problems. Furthermore, the WITF model and genetic

algorithm are suitable for parallel processing. For this, Apache Spark [4] is a powerful distributed computing framework that has become an important part of the big data analytics ecosystem.

This study focuses on combining the WITF model with a genetic algorithm on Spark to search for the optimal weight configurations for making more accurate recommendations and reducing the execution time.

2. Background

2.1 Notations

Table 1 describes notations referred to in this study.

Table 1. Notations

Notation	Description
$\mathcal{X}$	A tensor
$\boldsymbol{X}$	A matrix (user-item rating matrix)
$\boldsymbol{X_k}$	The matrix of domain
$\boldsymbol{X_{k,i,:}}$	A row (vector) of domain matrix $\boldsymbol{X_k}$
$\boldsymbol{X_{k,:,j}}$	A column (vector) of domain matrix $\boldsymbol{X_k}$
$\boldsymbol{X_{i,:}}$	A row (vector) of matrix $\boldsymbol{X}$
$\boldsymbol{X_{:,j}}$	A column (vector) of matrix $\boldsymbol{X}$
$\boldsymbol{U}$	User latent factor matrix
$\boldsymbol{V}$	Item latent factor matrix
$\boldsymbol{C}$	Domain latent factor matrix
$\boldsymbol{I}$	Identity matrix
$\{\omega_k\}$	A set of domain weights, $1 \le k \le \boldsymbol{K}$
$\|\boldsymbol{X}\|_F$	Frobenius norm of matrix $\boldsymbol{X}$
$\circledast$	Hadamard product

2.1 Cross-domain Recommender System

In the real-world environment, users have sufficient experience in their focused domains but lack experience in other domains. When information from users' familiar domains is used as auxiliary data, recommender systems perform better at recommending potentially desirable items in domains unfamiliar to users. Thus, cross-domain recommender systems have become an important emerging subject of research. Cross-domain collaborative filtering (CDCF) [5] is an important subject of research that focuses on product domains. Product domains, time domains, spatial domains, and other domains are also adopted in research related to CDCF. Because matrix factorization (MF) is a widely used model of CF recommender systems, cross-domain matrix factorization (CDMF) [6] has been proposed as an improvement technique for the CDCF method.

2.2 Irregular Tensor Factorization

In CF recommender systems, a dyadic user–item relationship is considered a core relationship. To use additional data, such as tags and time, some studies adopt tensor factorization (TF) [7] to process the triadic relationships, e.g., user–item–tag, user–item–time. Tensor factorization has two models: Candecomp/Parafac (CP) [7] decomposition and Tucker [7] decomposition. As shown in Figure 1, CP decomposition decomposes a tensor into a sum of component vectors, e.g., $\boldsymbol{U_{i,:}}$, $\boldsymbol{C_{i,:}}$, and $\boldsymbol{V_{i,:}}$, of the three latent factor matrices $\boldsymbol{U}$, $\boldsymbol{V}$, and $\boldsymbol{C}$. Compared to CP decomposition, Tucker decomposition decomposes a tensor into one core tensor $\boldsymbol{g}$ and three latent factor matrices, e.g., $\boldsymbol{U}$, $\boldsymbol{V}$, and $\boldsymbol{C}$.

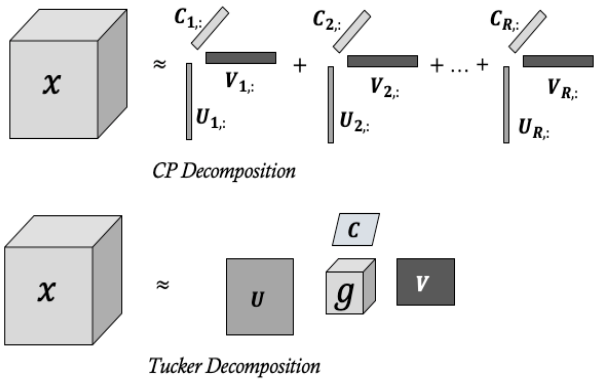

Figure 1. Two models of tensor factorization

To adopt the two models, the tensor must be a regular tensor with each of its domains containing the same items. Unfortunately, almost all real-world datasets cannot be considered as regular tensors with the same items in each domain. These datasets can be considered as irregular tensors, each of whose domains contain specific items. As shown in Figure 2, it is necessary to transfer the irregular tensor into the regular tensor that has the same set of virtual items. However, during the transfer, data loss is inevitable [2].

Therefore, it must be ensured that the data loss is minimal because the origin data from the irregular tensor is to be utilized maximally, and the most accurate latent factors are then obtained by processing the factorization for the regular tensor that has been transferred from the irregular tensor.

We define this kind of irregular tensor factorization as *lossless irregular tensor factorization*, where *lossless* represents the minimal data loss while transferring an irregular tensor to a regular tensor.

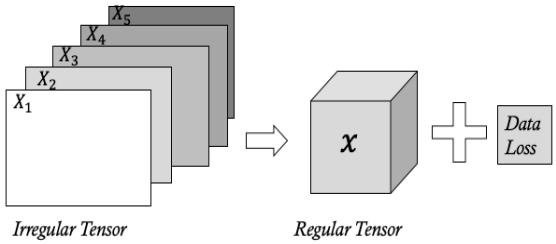

Figure 2. Irregular Tensor Factorization

2.3 Genetic Algorithm

Genetic algorithms are commonly used to generate high-quality solutions for optimization and search problems by relying on bio-inspired genetic operations such as mutation, crossover, and selection. These bio-inspired operators can be processed in parallel to speed up the operation. To improve the algorithm, researchers have refined each bio-inspired operator. For example, roulette wheel selection [3] and tournament selection [8] have been proposed to improve the selection operator. Binary mutation and Gaussian mutation [9] have been proposed to improve the mutation operator. Decades after the genetic algorithm was proposed, it is still widely used.

2.4 Apache Spark

Although MapReduce has been successfully implemented in large-scale data-intensive applications on commodity clusters, some other applications are not suitable for MapReduce. These applications contain many iterative machine learning algorithms that reuse a working set of data across multiple parallel operations. Thus, the new Spark framework distributed by Apache, to support these applications. Spark contains two main new features, which retain the scalability and fault tolerance of MapReduce. They are resilient distributed datasets (RDDs) [4] that save data on memory and various parallel operations.

The architecture of a Spark cluster contains three components: Spark driver program, cluster manager, and Spark workers. The Spark driver program converts the user's applications into tasks and schedules, then sends these tasks to Spark workers. The cluster manager manages the resources of whole clusters as a Spark plugin, and Spark can run different external cluster managers. A Spark worker contains several executors, which have multiple cores (≥ 1), to execute tasks and return the results to the Spark driver program.

3. Related Work

3.1 Weighted Irregular Tensor Factorization (WITF)

Hu et al. proposed a weighted irregular tensor factorization (WITF) model [2] to leverage cross-domain feedback data across all users to learn users' latent factors that are more accurate than users' latent factors only in a particular domain. Epinions dataset [10], which is used to evaluate WITF, contains multiple domains, with each domain containing different items. WITF considers the dataset as an irregular tensor in which each domain contains different items and then executes tensor factorization to obtain the user's latent factors. According to tensor theory, the irregular tensor must be transferred into a regular tensor, in which each domain contains the same items, to execute the tensor factorization.

Due to the inevitable data loss in irregular tensor transformation, the WITF model utilizes a set of domain weights to reduce the data loss as shown in Figure 3.

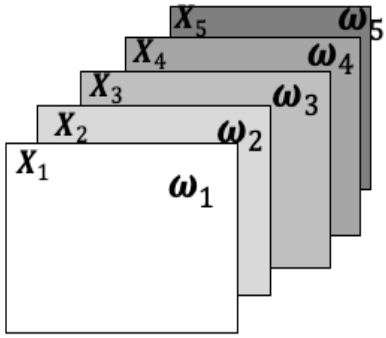

Figure 3. Domain Weight Configuration

The WITF model provides an objective function, based on CP decomposition, to describe the data loss. It is presented as formula 1 [2], where ω_k is the domain weight for each domain $\boldsymbol{X_k}$, $\mathbf{W_k}$ is the weight for each entry in domain $\boldsymbol{X_k}$, $\boldsymbol{\Sigma_k}$ is the diagonal matrix constructed based on the latent factor vector $\boldsymbol{C_k}$ that belongs to domain matrix $\boldsymbol{C}$, and $\boldsymbol{P_k}$ is the additional constraint [11] for obtaining a unique value of the Frobenius norm ($\|\ldots\|_F$) in the objective function.

$$J = \underset{U,V,C}{argmin} \frac{1}{2} \sum_{k=1}^{K} \omega_k \left\| \boldsymbol{W}_k \circledast (\boldsymbol{X}_k - \boldsymbol{U}\boldsymbol{\Sigma}_k(\boldsymbol{P}_k\boldsymbol{V})^T) \right\|_F^2 \qquad s.t. P_k^T P_k = \boldsymbol{I} \qquad (1)$$

The WITF model adopts a specific set of domain weights $\{\omega_k\}$, then iterates while updating $\boldsymbol{U}$, $\boldsymbol{V}$, $\boldsymbol{C}$, and $\boldsymbol{P_k}$ in each iteration until the objective function converges. The convergence value of the objective function can be considered as the minimal data loss of the irregular tensor factorization using a specific set of domain weights $\{\omega_k\}$. Furthermore, the WITF model also proves that the optimal domain weight configuration can minimize the data loss.

Searching for the optimal domain weight configuration is a complex and time-consuming procedure. Therefore, the WITF model adopts an empirical domain weight configuration that assigns the domain weight based on the ratios of the ratings in each domain. Since the performance of WITF can be optimized by searching for the optimal weight configuration on each domain, a method to search for the optimal weight configuration for the WITF model is necessary.

4. Proposed Method

4.1 Overview

We propose a model that combines Apache Spark and a genetic algorithm to search efficiently for the optimal domain weight configurations for the WITF model; then, we use the optimal configuration obtained to generate more precise users' latent factors with which to make recommendations. In this respect, Spark is an efficient and powerful large-scale data processing engine. A genetic algorithm is easily parallelized and widely used to search for solutions to complex problems. It is suitable for complex computing problems such as searching for the optimal domain weight configuration for the WITF model. Furthermore, the WITF model is also suitable for parallelization. Therefore, parallelization of the genetic algorithm and the WITF can speed up computation when using a genetic algorithm to search for the optimal domain weight configurations for the WITF model.

4.2 Architecture

Although the genetic algorithm is easy to parallelize and widely used to search for the solutions to a complex problem, a genetic algorithm is not suitable for problems with a complex fitness evaluation. However, the proposed method utilizes the WITF model as a fitness evaluation. Therefore, we propose a two-phase Spark parallelization scheme to process the WITF model and genetic algorithm efficiently.

The Spark RDDs are one of the core components of Spark. The RDDs store the data in memory to reduce the time for data input/output (I/O) and have various efficient methods available to users. Therefore, Spark users always separate large data into partitions and store them in RDDs, then process these data on Spark in parallel.

For our proposed method, WITF and genetic algorithm operators, such as crossover and mutation, are the time-consuming procedures. To speed up these operations, the WITF model is processed as the first phase parallelization. In the first phase, the data of the WITF, e.g., $\boldsymbol{X_k}$, $\boldsymbol{U}$, $\boldsymbol{V}$, and $\boldsymbol{C}$, are stored in RDDs and computed in parallel inside the WITF model. This phase is presented as Algorithm 1.

Algorithm 1: Parallelization of **WITF Model [2]**

$[\boldsymbol{U},\boldsymbol{V},\boldsymbol{C},\{\boldsymbol{P_k}\}]$ = **(function) Parallel_WITF**$(\{\boldsymbol{X_k}\},\{\omega_k\})$

Input: $\{\omega_k\}$ is set of domain weights

$\boldsymbol{X_k}$ is the data matrix for each domain

Output: $\boldsymbol{U}$ is the latent factor matrix for users

$\boldsymbol{C}$ is the latent factor matrix for domains

$\boldsymbol{V}$, $\boldsymbol{P_k}$ are the latent factor matrices for items

Begin

***Initialization*:**

1: Randomly Initialize $\boldsymbol{U,C}$

2: $\boldsymbol{V = I}$

3: $\boldsymbol{P_k = AB^T}$, with the SVD: $\boldsymbol{X_k^T U \Sigma_k V^T \approx A \Sigma B^T}$

***Iteration*:**

4: Generate tensor with each $\boldsymbol{X_k}$

5: Update $\boldsymbol{U_i}$ of each user $\boldsymbol{i}$ in parallel by WITF theory

6: Update $\boldsymbol{C_k}$ of each domain $\boldsymbol{k}$ in parallel by WITF theory

7: Update $\boldsymbol{V}$ as a whole by WITF theory

8: Update $\boldsymbol{P_k}$ of each domain $\boldsymbol{k}$ in parallel by WITF theory

Repeat 4-8 until the objective function convergence

9: Return $\boldsymbol{U,V,C}$, $\{\boldsymbol{P_k}\}$

End

The second phase parallelization is for genetic algorithm operators. We use the root mean square error (RMSE) [12] metric, which is calculated using the returned $\boldsymbol{U,V,C}$, $\{\boldsymbol{P_k}\}$ from the WITF model as a fitness value for each individual ($\{\omega_k\}$). The RMSE is defined as:

$$RMSE = \sqrt{\frac{\sum_{i=1}^{N}(Y_i - \widehat{Y}_i)^2}{N}} \qquad (2)$$

where Y_i is the user's real rating for item $\boldsymbol{i}$, and $\widehat{Y}_i$ is the predicted rating of item $\boldsymbol{i}$ for that user. We store the individuals in RDDs and process these operators in parallel on Spark. This phase is presented as Algorithm 2.

Algorithm 2: Parallel Genetic Algorithm with WITF

Input: $\{\omega_k\}$ is the initial set of domain weights

$\boldsymbol{X_k}$ is the data matrix for each domain

$\boldsymbol{G_{max}}$ is the maximum number of generations

Output: $\{\boldsymbol{\psi_k}\}$is the optimal set of domain weights

Begin

***Initial Population*:**

1: Make initial population by $\{\omega_k\}$

***Generation Iteration of Genetic Algorithm*:**

2: **for each** individual $\{\omega_k\}$ in population **do**

3: $[\boldsymbol{U,C,V}, \boldsymbol{P_k}]$ = **Parallel_WITF**$(\{\boldsymbol{X_k}\},\{\omega_k\})$

4: Calculate RMSE by $[\boldsymbol{U,C,V}, \boldsymbol{P_k}]$

5: Save the RMSE as the fitness value of $\{\omega_k\}$

6: **end for**

7: Selection of the population in parallel

8: Crossover of the population in parallel

9: Mutation of the population in parallel

***Repeat 2-9 for G_{max} times*:**

10: $\{\boldsymbol{\psi_k}\}$ is the $\{\omega_k\}$ that obtains minimal RMSE

11: Return $\{\boldsymbol{\psi_k}\}$

End

5. Experimental Evaluation

5.1 Datasets and Environment

The Epinions dataset [10] is extracted from the Epinions website, which is an e-commerce website with multiple domains of products. The users can provide reviews and ratings (which are integers from 1 to 5) to products of different domains, and products on each domain are unique to that domain. Five domains from the Epinions dataset [10] that have the most ratings were selected, and the data were pre-processed to discard users with less than five and ten ratings on the five selected domains. In the *small* dataset, first, the users who have at least ten ratings in each selected domain are retained, and then the items that have been rated by at least ten of the retained users, are retained. Similarly, in the *large* dataset, users who have at least five ratings in each selected domain are retained, and then the items that have been rated by at least five of the retained users are retained. The two datasets are presented in Table 2.

Table 2. Dataset Statistic

Size	*Small*			*Large*		
	#Users	*#Items*	*#Ratings*	*#Users*	*#Items*	*#Ratings*
Domain 1	2,403	1,104	14,960	6,682	1,771	27,786
Domain 2	2,403	320	7,350	6,682	702	13,802
Domain 3	2,403	1,362	30,490	6,682	2,318	47,972
Domain 4	2,403	1,050	10,223	6,682	2,459	22,448
Domain 5	2,403	761	8,733	6,682	1,786	18,925

The experiment was processed on four servers, and the environment of each server is presented in Table 3. In the experimental Spark cluster, one server was configured as the Spark master server to process the Spark driver program and Spark cluster manager. The other three servers were configured as Spark workers (also called Spark slave servers) to process the Spark tasks.

Table 3. Experiment Environment

OS	CPU	#CPU(Core)	Memory
CentOS 7.5	Intel Xeon CPUE5-2620 v4 @ 2.10GHz	2 Physical 16 Logical	128GB

5.2 Genetic Algorithm Encoding and Operators

The individuals (chromosomes) of the initial population in a genetic algorithm are a set of domain weights $\{\omega_k\}$. Each weight ω_k is considered as a gene inside an individual (chromosome). This initial encoding method is called a float-encoding genetic algorithm [13].

A genetic algorithm has three operators: selection, crossover, and mutation. For the selection operator, tournament selection [8] was adopted. It executes several *tournaments* among a few individuals chosen at random from the population, and the individual with the best fitness value becomes the winner to be selected for crossover. A pair of individuals will cross their genes when processing the crossover operator for which whole arithmetic recombination [14] was adopted. It is the most commonly used crossover operator for the float-encoding genetic algorithm and works by taking the weighted sum of the two parental alleles for each gene in the children. After processing the crossover operator, Gaussian mutation [9] was adopted. It adds a random value from a Gaussian distribution to a randomly selected gene of an individual to create a new offspring.

5.3 Efficiency Evaluation

5.3.1 Parallel WITF model on Spark

A Spark cluster contains many computing resources that are considered as workers. To manage these workers in the cluster, Spark provides a driver program to dispatch tasks, which contain data and instructions, and to receive the task results when workers finish their assigned tasks. An efficient program on Spark should execute its codes on the Spark workers for as long as possible.

Among the procedures of the proposed method, the most time-consuming procedures are the fitness evaluation, which computes the RMSE of the WITF model using the domain weight configuration of each individual. As discussed in section 3.1, the WITF model executes iterations to update its parameters, e.g., $\boldsymbol{U}$, $\boldsymbol{C}$, $\boldsymbol{V}$, and $\boldsymbol{P_k}$, until reaching convergence. Therefore, the efficiency of each iteration of the WITF model is important for the overall efficiency of the proposed method. By implementing the WITF model on Spark using the Spark Python API and web user interface to monitor the program, the timeline of an implemented iteration of the WITF model can be analyzed. The timeline of a WITF iteration using the *small* dataset is shown in Figure 4.

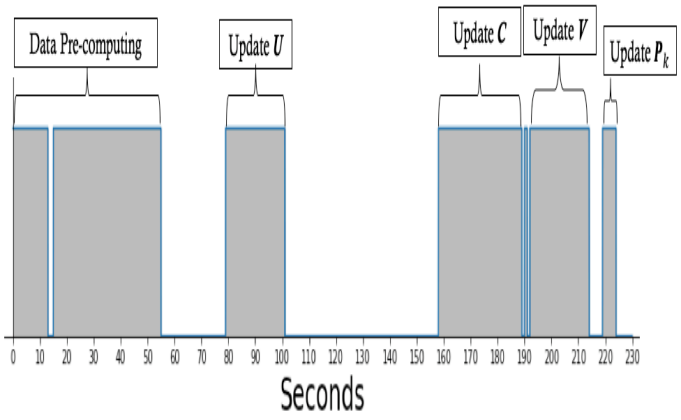

Figure 4. Timeline of a WITF iteration on Spark

The gray bars in Figure 4 represent the period of Spark workers executing the tasks. The white spaces in the figure represent the period during which the Spark driver program is executing, and the workers are idle. The time ratio between the workers executing time and a WITF iteration executing time is 67.9%.

5.3.2 Efficiency of the WITF model on a Spark cluster

A series of experiments were processed using two datasets of different sizes to evaluate the implementation of the WITF model on a Spark cluster, which contains one master server and three slave servers (16 cores in each server, 48 cores in total). Experiments with different numbers of Spark executors with one core were processed first. Then experiments with different numbers of Spark executors with multiple cores were processed, and all 48 cores were used.

First, three group experiments for the two datasets of different sizes were processed by setting 16, 32, and 48 executors with one core. The results of the execution time (in minutes) of the WITF iteration is shown in Figure 5. According to the results, the execution time of the WITF iteration shows a slight increase with an increase in the number of Spark executors used while processing either the *small* or *large* dataset. The reason the execution time increased when more executors (cores) were utilized is that these executors contain only one core and the computing power of each executor was limited. Furthermore, communication between the Spark driver program and the Spark executors, e.g., data broadcast and task results return, was increased by utilizing more executors. Thus, the execution time of the WITF iteration increased, although more computing resources were utilized.

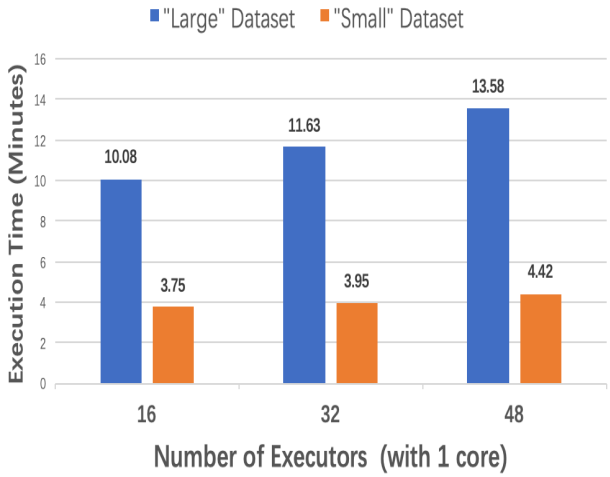

Figure 5. Execution Time of a WITF Iteration

Because the experiment results from setting the Spark executors to only one core show that the executor's computing power is limited, another series of experiments on the two datasets were processed with the Spark executors set to multiple cores (48 cores in total). The execution time results of a WITF iteration for the two datasets are presented in Figure 6. The execution time of the WITF iteration decreased with the increase in the number of cores in each executor, and the decrease in the number of executors. The execution time tends to be stable when each executor has more than eight cores (number of executors is less than six, with 48 cores in total). When using the *large* dataset, the best execution time, which was obtained with three executors (16 cores in each), is 59.96% faster than the worst execution time obtained with 48 executors (one core in each). When using the *small* dataset, the best execution time, which was obtained with three executors (16 cores in each), is 40.04% faster than the worst execution time obtained with 48 executors (one core in each).

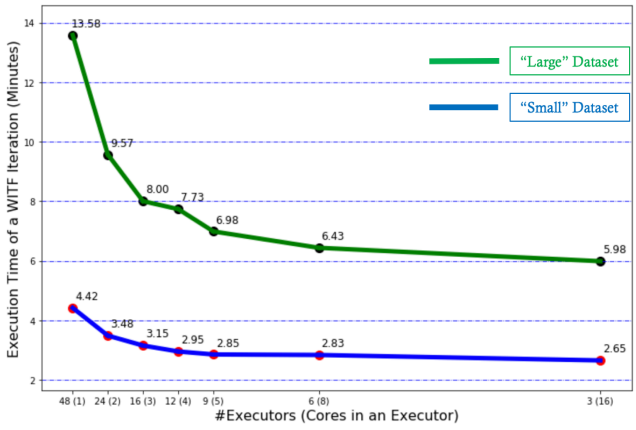

Figure 6. Execution Time of a WITF Iteration with different multiple-core Spark executors

Regarding the ratio between the executing time per executor and the entire execution time of a WITF iteration, the results for the two datasets are shown in Figure 7. Like the results in Figure 6, the execution time ratios also decreased with an increase in the number of cores in each executor and a decrease in the number of the executors. It also tends to be stable when the number of cores in each executor is more than eight (and the number of executors is less than six, with 48 cores in total).

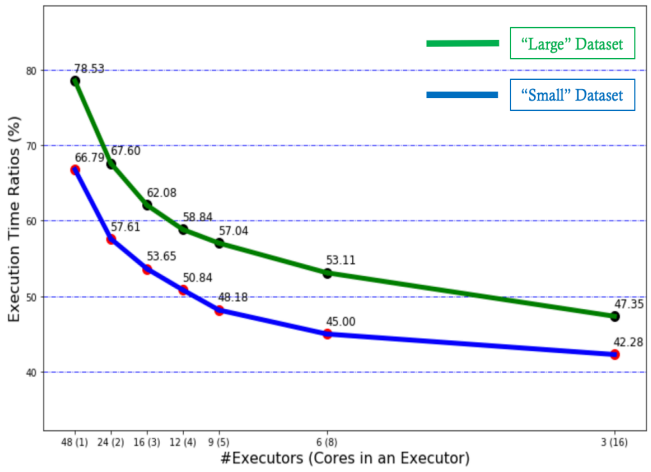

Figure 7. Execution Time Ratios of a WITF Iteration with different multiple-core Spark executors

The results in Figure 6 and 7 show that more cores in each Spark executor increases the computing power of each Spark executor and reduces the execution time for the tasks processed on each Spark executor. Furthermore, reducing the number of executors decreases communication between the Spark driver program and the Spark executors, consequently decreasing the execution time of the Spark driver program. The execution time ratio, which is between the time of the Spark executors and the total execution time, also decreases. This means that the decrease in the Spark driver program execution time is much less than the decrease in the Spark executors execution time while setting more cores in each executor and deploying fewer executors, as in our experiments. In other words, the implementation of the WITF model would have a more efficient performance in a Spark cluster with executors that have high computing power.

5.4 Efficiency of the Proposed Method on Spark

The population of the genetic algorithm was initialized with 16 individuals and three executors set to 16 cores in each and 48 in total. An implementation of the proposed method was then executed for 50 generations using the two datasets. Based on the execution time, the execution time statistics shown in Table 4 were obtained.

Table 4. Genetic Algorithm Execution Time

(a). *Small* Dataset

Procedure	Average Execution time (min.)
1 Generation	172.62
1 Individual	10.63

(b). *Large* Dataset

Procedure	Average Execution time (min.)
1 Generation	382.93
1 Individual	23.93

5.5 Genetic Algorithm Generation Evaluation

The mean fitness (RMSE) value of all 16 individuals in each generation was calculated, and the minimum fitness (RMSE) value was recorded as the best RMSE of all 16 individuals in each generation. Furthermore, the RMSE value was calculated using the WITF model's empirical domain weight configuration, which determines the domain weights by adopting the number of ratings in each domain as the baseline.

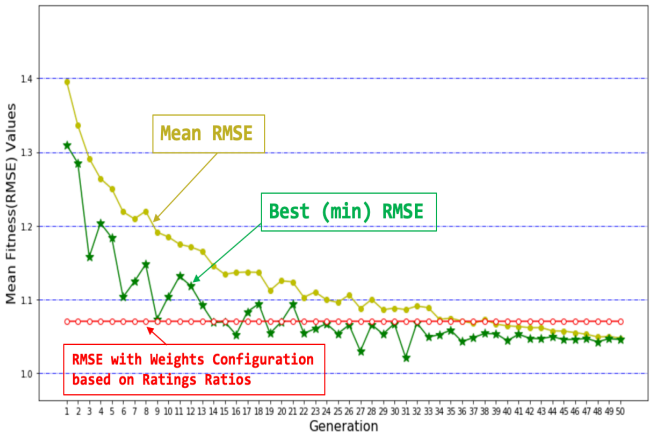

Figure 8. Genetic Algorithm Generations Evaluation using the *small* dataset

For the *small* dataset, the comparison of those three values in each generation is presented in Figure 8. The red line represents the baseline. The golden polyline represents the mean fitness (RMSE) value of each generation, and the green polyline represents the best fitness (RMSE) value of each generation. Figure 8 shows that the mean fitness and best fitness value of each generation tend to be stable and better than the initial individual's mean and best fitness values as the generation grows. After the 22nd generation, the best fitness values are better than the baseline and tend to be stable after the 41st generation. Before the 34th generation, the mean fitness values are worse than the baseline. Then after the 39th generation, the mean fitness values are better than the baseline and approach to the best fitness RMSE. This demonstrates that the search result of the genetic algorithm reaches convergence in the proposed method.

Furthermore, the best RMSE using the *small* dataset among all generations is improved by 4.2% compared with the baseline.

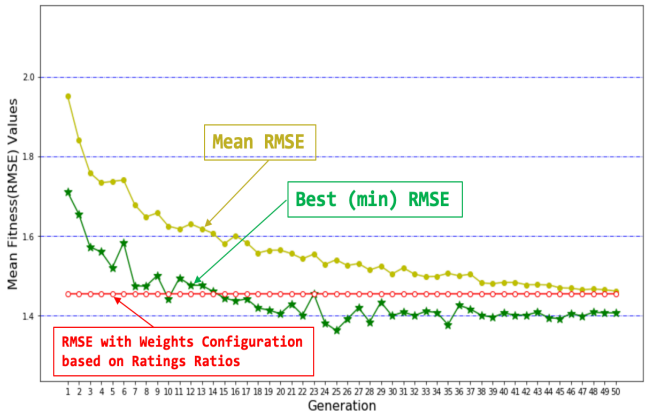

Figure 9. Genetic Algorithm Generations Evaluation using the *large* dataset

For the *large* dataset, the result, which is shown in Figure 9, is similar to the results of the *small* dataset. Because the *large* dataset is sparser than the *small* dataset, the baseline, mean fitness RMSE, and best fitness RMSE are worse than the fitness value for the *small* dataset, and the mean fitness RMSE is worse than the baseline among all the 50 generations. However, like the result of the *small* dataset, the mean fitness RMSE would approach to the best fitness RMSE after enough generations. After the 14^{th} generation, the best fitness values are better than the baseline, and then tend to be stable after the 38^{th} generation. Furthermore, the best RMSE using the *large* dataset among all generations is improved by 6.3% compared with the baseline.

The results show that the proposed method can search and obtain the optimal domain weight configuration for WITF to make recommendations that have the best RMSE.

6. Conclusion

In this study, we proposed a method that combines the WITF model and a genetic algorithm to search for the optimal domain weight configuration of a WITF model for more accurate recommendations. To make the computation efficiently, we adopted parallelization in both the WITF model and the genetic algorithm, then implemented and evaluated the proposed method on a Spark platform.

We evaluated two different sizes of datasets on a Spark cluster containing 48 cores and analyzed the experimental results for each dataset. The results of the evaluation show that the proposed method and its implementation can search for the optimal domain weight configuration for the WITF model. For the *small* dataset, the proposed method searched for a domain weight configuration with a 4.2% RMSE improvement. For the *large* dataset, the proposed method searched for a domain weight configuration with a 6.3% RMSE improvement. The efficiency evaluation results show that parallelization implementation of the WITF model has a more efficient performance in a Spark cluster with executors that have high computing power.

References

[1] P. Resnick, N. Iacovou, M.Suchak, P.Bergstrom, and J.Riedl, GroupLens: An Open Architecture for Collaborative Filtering of Netnews, Proceedings of ACM Conference on Computer Supported Cooperative Work (CSCW1994), pp.175-186, 1994.

[2] L. Hu, L. Cao, J. Cao, Z. Gu, G. Xu, and D. Yang. Learning informative priors from heterogeneous domains to improve recommendation in cold-start user domains, ACM Trans. Inf. Syst. Vol.35, No.2, Article 13, 2016.

[3] D.E. Goldberg, Genetic Algorithms in Search, Optimization, and Machine Learning, Addison-Wesley. 1989

[4] Zaharia M, Chowdhury M, Das T, Dave A, Ma j, McCauley M, Franklin M, Shenker S, Stoica I, Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing, Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, Berkeley, 2012.

[5] B. Li. Cross-domain collaborative filtering: A brief survey, Proceedings of the 2011 IEEE 23rd nternational Conference on Tools with Artificial Intelligence, 2011.

[6] W. Pan, E. W. Xiang, N. N. Liu, and Q. Yang, Transfer learning in collaborative filtering for sparsity reduction. Proceedings of the 24th AAAI Conference on Artificial Intelligence, 2010.

[7] T. G. Kolda and B. W. Bader, Tensor decompositions and applications, SIAM Rev., Vol.51, No.3, pp.455–500, 2009.

[8] D. E. Goldberg and K. Deb, A comparative analysis of selection schemes used in genetic algorithms, Foundations of Genetic Algorithms, Vol.1, pp.69-93, 1991.

[9] T. Back and H.-P. Schwefel, An Overview of Evolutionary Algorithms for Parameter Optimization, Evolutionary Computation,Vol.1, No.1, pp.1-23, 1993.

[10] S. Meyffret , E. Guillot , L. Médini, and F. Laforest, RED: a Rich Epinions Dataset for Recommender Systems, Dataset for Recommender Systems, 2014.

[11] H. A. L. Kiers, J. M. F. Ten Berge, and R. Bro. Parafac2—part I. A direct fitting algorithm for the parafac2 model, Journal of Chemometrics, Vol.13, pp.275-294, 1999.

[12] CJ. Willmott, K. Matsuura, Advantages of the mean absolute error (MAE) over the root mean square error (RMSE) in assessing average model performance, Climate Research, Vol.30, No.1, pp.79-82, 2005.

[13] C. T. Su and W. T. Tyen, A Genetic Algorithm Approach Employing Systems, Proceedings of International Congress on Modeling and Simulation, pp. 1444-1449, 1997.

[14] Z. Michalewicz. Genetic Algorithms + Data Structures = Evolution Programs. Springer, 3rd edition, 1996.