**Title:** Speed up Differential Evolution for ranking of items in recommendation systems

**Author:** Urszula Boryczka, Michał Bałchanowski

**Citation style:** Boryczka Urszula, Bałchanowski Michał. (2021). Speed up Differential Evolution for ranking of items in recommendation systems. „Procedia Computer Science" (Vol. 192, 2021, s. 2229-2238), DOI:10.1016/j.procs.2021.08.236

Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 192 (2021) 2229–2238

Procedia Computer Science

www.elsevier.com/locate/procedia

25th International Conference on Knowledge-Based and Intelligent Information & Engineering Systems

# Speed up Differential Evolution for ranking of items in recommendation systems

Urszula Boryczka, Michał Bałchanowski*

*Institute of Computer Science, University of Silesia in Katowice, ul. Bedzińska 39, Sosnowiec 41-200, Poland*

**Abstract**

Recommendation systems can suggest users list of items they have not yet seen but might be interested in. To improve the quality of the generated recommendations, different techniques are often used which try to personalize recommendations. Usually user preferences are stored in the form of a vector in which individual values describe to what extent a given feature is desired by the user. To find this vector, metaheuristic algorithms can be used, however their main drawback is their computational complexity. Therefore, in this paper, a modification of the Differential Evolution algorithm is proposed to enable faster computation of the ranking score for each item in the system, which is used to create a recommendation list. Experiments have been performed on the current MovieLens 25m database and they show that our modification can significantly speed up the process of finding a preference vector, without losing their quality for the top-N recommendation task. We will also address the vulnerability of recommendation systems to profile injection attacks, as a result of which an attacker can influence the generated recommendations.



*Keywords:* recommendation systems; ranking function; learning to rank; differential evolution; metaheuristic; profile injection attack

## 1. Introduction

Recommendation systems have become an integral part of our lives. We can use them while shopping, watching movies, listening to music, or browsing social networks. These systems try to suggest new content and products by analyzing the current behavior of users. A well implemented recommendation system, in addition to financial benefits for a given company, can also have a positive influence on the satisfaction of users. Before generating recommendations, items and users must have a certain set of features that describe them. Unfortunately, it is often the case that we do not have such features or they are poor quality. If we do not have good quality features, we can create them by

* Corresponding author.
*E-mail address:* urszula.boryczka@us.edu.pl, michal.balchanowski@us.edu.pl

1877-0509 
10.1016/j.procs.2021.08.236

factorizing the user-item matrix. Such features are called "latent features", due to the fact that they cannot be observed directly. Their main disadvantage is that it is hard to explain what they mean.

In our previous article [4], we described how the Differential Evolution algorithm can be used to generate personalized recommendations by optimizing the average precision (AP) measure. Unfortunately, the main problem during our research was the long time needed to complete the computation, which prevented us from using larger datasets. Therefore, we decided to improve our algorithm and proposed a modification of the Differential Evolution algorithm allowing for faster calculation of the dot product for individual items. This value is later used to create a ranking of items. In our research, we used the publicly available, popular MovieLens 25m dataset (released in 2019) [11].

Additionally, in this article, we will discuss the problem of pre-filtering the data on the basis of which the recommendations are generated. This is important because recommendation systems analyze users behaviour in order to recommend products to other users on the basis of their choices. With this in mind, it should be noted that there is some potential danger here. It may turn out that some user profiles are fabricated in a special way. The goal of such an attack would be to influence the recommendation system in a specific way, which, for example, should recommend specific products. The problem is particularly difficult to solve in large datasets, where there is a large number of user profiles which cannot be reviewed manually.

This article is divided into 6 chapters. Chapter 2 includes literature overview which provides information about the current literature related to the subject of this article. Chapter 3 will present the definition of recommendation system along with an explanation of such concepts as "matrix factorization", "profile injection attack" and the "Differential Evolution" algorithm. Chapter 4 describes the proposed modification of the DE algorithm along with a description of the fitness function. In chapter 5 we will present the performed experiments and their results. The last chapter 6 will be dedicated to conclusions and suggestions for future work.

## 2. Literature overview

Recommender systems have become popular in recent years, mainly due to the "Netfix Prize" competition [1], where researchers have tried to improve the quality of the generated recommendations by reducing the RMSE (root mean square error). However, since recommendations are often presented to the user in the form of a Top-N list of suggested items, an appropriate approach is to check the quality of the system in this aspect as well [13] and many papers have been written on this topic [8] [9]. In this paper, learning to rank is applied to find the items that could be recommended to the active user. Learning to rank is often described in the context of information retrieval systems and this problem has been described in detail in the work [16]. This technique has also been applied to recommendation systems [21][17].

Often metaheuristic algorithms are used to directly optimize measures used in the learning to rank problem such as MAP, NDCG, which due to their specificity cannot be optimized using other techniques. An example of such work is the use of Differential Evolution (DE) [3] and Particle Swarm Optimization (PSO) [10] algorithm to optimize the mean average precision (MAP) measure in information retrieval systems. The application of Evolutionary Algorithms in recommender systems is well described in the paper [12] where an overview of current research and its results is presented.

This paper also addresses the problem of shilling attacks that aim to influence the generated recommendations. This problem has been described in several papers, where researchers try to use statistical metrics [7] or algorithms used for outlier analysis [6] to find such profiles in the system that may have been created by an attacker. Good survey of attack detection approaches in recommender systems can be found in [19].

## 3. Background of the research

### 3.1. Recommendation system

The purpose of recommendation systems is to suggest some content or services to the user that he might be interested in. The problem is not trivial and has been studied by researchers for many years. Let us consider some set of users $U = u_1, \ldots, u_{|U|}$ and some set of items $I = i_1, \ldots, i_{|I|}$. Each user $u \in U$, has rated some items $i \in I$, which we will refer to by $r_{ui}$. Therefore, the data can be represented as a three $(u, i, r_{ui})$, which in the case of our research will

mean that a certain user $u$ rated a certain movie $i$, giving it a rating of $r_{ui}$. All ratings given by all users for all items are often presented as a matrix of size $m \times n$. There are three main types of recommendation systems:

- content-based - suggesting new products is based on the similarity of features describing the items with the user profile. Their advantage is that they do not need other users to generate recommendations, but the main disadvantage is that they require good quality features.
- collaborative filtering – collaborative filtering is a technique that recommends new products to a user based on the activity of other, similar users in the system. Different similarity measures, such as Euclidean distance, are used to assess the similarity between users.
- hybrid – it is a system that combines different techniques. Today, most recommendation systems are based on this approach.

Frequently when working with recommendation systems, we need a certain set of features upon which we can, for example, calculate the similarity between particular users or items. However, a common problem in recommendation systems is that we do not have such features or they are poor quality. Therefore, it has been proposed to use algorithms that are designed to create such features from a user-item matrix. The most popular technique for obtaining these features is the matrix factorization technique. However, it should be noted that this matrix is very sparse (about 99%) and therefore standard techniques (e.g. Singular Value Decomposition (SVD)) cannot be used here. Therefore we factorize this matrix using algorithms that look for the best approximation of the original matrix. We can describe this technique as follows:

$$\hat{R} := UI^T \tag{1}$$

where $\hat{R}$ is a matrix of size $|U| \times |I|$ formed by the product of two smaller matrices $|U| : |U| \times k$ and $I : |I| \times k$, $k$ denotes the rank of approximated matrices. Each row $U_u$ in matrix $U$ represents features describing user $u$, and each row $I_i$ in matrix $I$ represents features describing item $i$. More information about this technique can be found in the paper [14].

### 3.2. Profile injection attack

Some people or bots may try to influence a recommendation system by creating fake user profiles which then promote (push attack) or discredit (nuke attack) selected items in the system. Attacks of this type are called “profile injection attack” or “shilling attack” and they can affect the quality of the generated recommendations. Of course, manual detection of such attacks is very time-consuming due to the amount of data that needs to be processed, so the only reasonable solution is to automate this process.

Various techniques are used to detect these types of attacks. From simple statistical measures examining a single user behavior to sophisticated algorithms combining various anomaly (outlier) detection techniques. Their aim is to filter users in the system and detect profiles that are highly likely to have been crafted by the attacker. In the paper [2], the authors observed that attacker profiles are often correlated with one another. Therefore, unsupervised learning can be used here, for example, to detect clusters of such users. In our work, we used the rating deviation from mean agreement (RDMA) measure, proposed in this paper [7] to detect such profiles:

$$RDMA_j = \frac{\sum_{n=0}^{N_j} \frac{r_{i,j} - Avg_i}{NR_i}}{N_j} \tag{2}$$

where $N_j$ is the number of items rated by user $j$, $r_{i,\,j}$ is the rating given by user $j$ for item $i$, $NR_i$ is the number of ratings given in the system for item $i$.

### 3.3. Differential Evolution

Differential evolution is an evolutionary technique that was developed by K. Price and R. Storn [20]. In this algorithm there is some population in which every individual is a solution to some optimization problem. This technique is often use in continuous optimization, in which case the individuals are d-dimensional vectors of real numbers, so

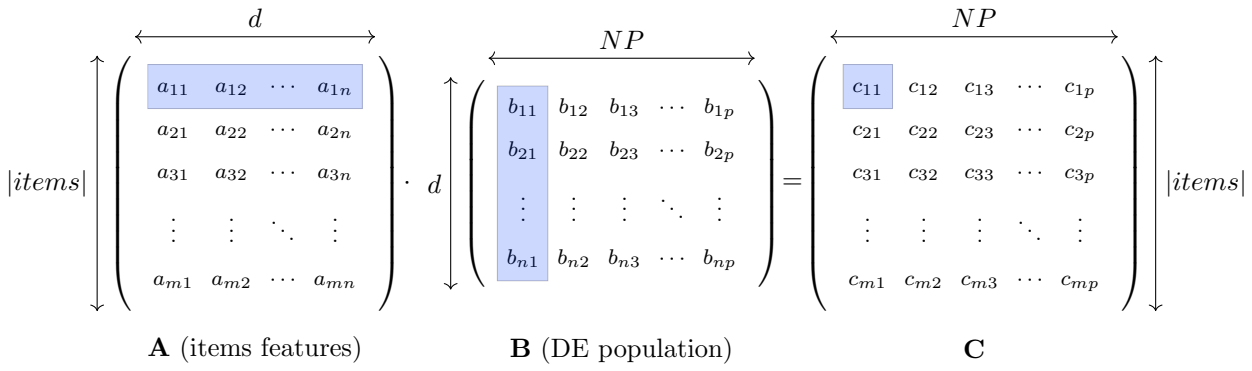


Fig. 1. Graphic representation of multiplication of items feature matrix by DE population matrix.

each individual is a solution to the optimization problem. The algorithm with each iteration tries to improve the population of individuals until a stopping criterion is reached. To evaluate whether one individual is better than another, a function, called the "fitness function", is used here. Its task is to evaluate the quality of each individual in the population, by which the algorithm can choose which individual should go to the next generation. There are two operators: mutation and crossover [5]. Mutation in the standard version of the algorithm creates a new individual by combining three randomly selected individuals from the population $P$ and is expressed by the following formula:

$$\vec{v_i} = \vec{x_{r_1}} + F(\vec{x_{r_2}} - \vec{x_{r_3}}), \tag{3}$$

where $r_1, r_2, r_3$ are three random individuals from population $P$, with $r_1 \neq r_2 \neq r_3$. The parameter $F$ is the amplification factor and takes a value between $[0, 1]$. Following the usage of mutation operator the crossover operator is used, which creates a new individual $\vec{u_i}$ according to formula 4 . The $CR$ parameter determines the crossover probability, and the $rand(j)$ function generates a random number between $[0, 1]$. $i_{rand}$ is a random integer from $[1, 2, ..., d]$.

$$u_{i,j} = \begin{cases} v_{i,j} & \text{if } (rand(j) \leq CR \text{ or } i = i_{rand}) \\ x_{i,j} & otherwise. \end{cases} \tag{4}$$

## 4. Proposed method FastRankDE

Our proposed FastRankDE method use matrix multiplication operation to compute the ranking score based on which the ranking of items will be created. To do this, we need to store the population of individuals and all the items present in the system as a matrix. Such an operation will allow us to multiply the matrix of individuals and the matrix of items. This is an operation that can be easily parallelized using programing libraries. Then, by sorting the values (dot products) found in each column, we can create a ranking of the items. A graphical representation of such an operation is presented in figure 1, and a diagram showing the system architecture is shown in figure 2.

### *4.1. Fitness function*

Fitness function makes it possible to evaluate the quality of a new individual created as a result of the usage of crossover and mutation operators. The fitness of the individuals is compared and the individual with the better fitness value is passed to the next generation. For the learning to rank problem, we propose to use a fitness function that uses matrix multiplication to calculate the values on the basis of which the items will be sorted. This way, we will obtain ranking scores (dot products) for all items and for the entire population in the DE algorithm. Pseudocode showing the algorithm for such a fitness function is presented in listing 1 and 2.

$$C = AB \tag{5}$$

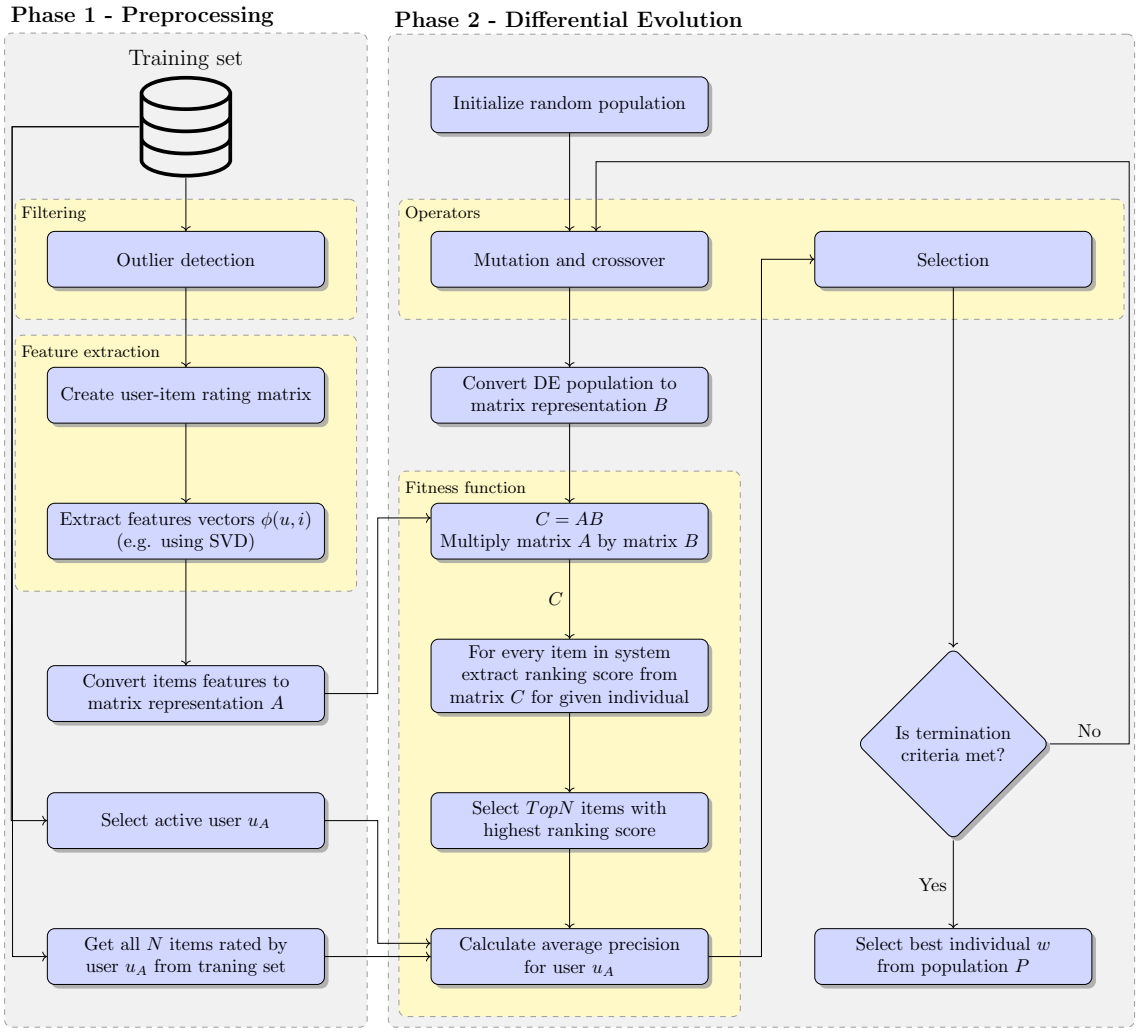


Fig. 2. System architecture.

By multiplying the items feature matrix $A$ by the population matrix $B$ according to the formula 5, we obtain the matrix product $C$. Then, for a given individual (column) we sort the items in a descending order according to the values in the corresponding row of this matrix, and then select the top-N items $S$ that would be recommended to user $u_A$. The fitness function is calculated for the active user in the system as follows:

$$Fitness = AP@k(R,S) \tag{6}$$

where $R$ is the set of items that the user $u_A$ rated in his training set. Formula 6 represents the average precision that would be computed between the sets $R$ and $S$, that is, between the set of items rated by the user and the set of items recommended by the system. In our system $k$ in average precision is equal to the number of rated items by user $u_A$ in his traning set. According to our experiments this is the best value since we are using all items from user $u_A$ training set.

**Algorithm 1** Pseudocode for DE

Generate random initial population $P^0$ of size $NP$ (d-dimensional vectors)
**repeat**
  **for each** individual $i$ in the population $P^t$ **do**
    three random integers $r_1, r_2, r_3 \in (1, NP)$, with $r_1 \neq r_2 \neq r_3 \neq i$
    generate a random integer $i_{rand} \in [1, 2, ..., d]$
    **for each** $j$-th gene in individual $i$ **do**
      $v_{i,j} = x_{r_1,j} + F(x_{r_2,j} - x_{r_3,j})$
      $u_{i,j} = \begin{cases} v_{i,j} & \text{if } (rand(j) \leq CR \text{ or } i = i_{rand}) \\ x_{i,j} & otherwise \end{cases}$
    **end for**
    add new individual $\vec{u_i}$ to population $P^{t+1}$
    matrixFitnessFunction($P^{t+1}$)
    replace $\vec{x_i}$ from population $P^t$ with the trial individual $\vec{u_i}$ from population $P^{t+1}$, if $\vec{u_i}$ is better
  **end for**
  t = t + 1
**until**

**Algorithm 2** Pseudocode for matrix fitness function

**procedure** MatrixFitnessFunction(population $P$)
  convert items features to matrix representation $A$
  convert population $P$ to matrix representation $B$
  $C = AB$
  **for** individual $i$ in population $P$ **do**
    **for** item $j$ in items $I$ **do**
      extract ranking score from matrix $C$ for given individual:
      $rankingScore = C[j, i]$
    **end for**
    sort items by ranking score
    select $TopN$ items with highest ranking score
    calculate average precision ($AP$) for active user $u_A$
    use $AP$ as fitness value for given individual $i$
  **end for**
**end procedure**

## 5. Experiments

Recommendations are usually displayed to the user in form of list. Therefore, to evaluate the quality of the generated recommendations, we used the average precision (AP) measure, which compares the top-N recommended items suggested by our system with the items that an active user has in his test set. This measure examines where the recommended items are positioned. The higher the more relevant items are on the list (closer to the first position), the larger the AP measure is. The problem is not trivial, due to the number of items from which we need to select these which are relevant. The experiments performed in this paper were done using the popular MovieLens 25m database [11]. This is currently the most up-to-date database provided by GroupLens group and is recommended for new research. The database contains 25 million ratings given on a scale of 1-5 by 162541 users for 62423 movies. Each user has rated at least 20 movies and is represented only by an id number (no other information is provided). Features for user and items in system were generated using the randomizedSVD algorithm, which implementation can be found in the

popular scikit-learn[1] library. The FastRankDE algorithm was implemented in Python and C#. Python was used to create the environment needed to conduct a reliable research and at the same time the Differential Evolution algorithm, due to the time-consuming computation, was implemented in C#. The research was conducted on a computer with Intel Core i5-7600 processor clocked at 3.50GHz with 16GB of RAM.

### 5.1. Description of the research

In order to check the quality of the generated recommendations, our system had to select the top-N items out of all the items that appeared in the training set, and which the user had not yet rated. Therefore, first the ratings for each user were sorted by the time, and then divided into two sets: training (80%) and test (20%). This approach is justified due to the fact that our algorithm tries to predict the user's future preferences based on his past activity. Due to the time-consuming computation of metaheuristic algorithms, 30 users were randomly selected for the study and recommendations were generated for them, subsequently the results (AP) were averaged which can be shown in result as Mean Average Precision (MAP). To demonstrate that our algorithm produces good results, we compared it with the Bayesian Personal Ranking (BPR) algorithm [18], implementation of which is available in the LigtFM library [15]. In addition, an standard SVD algorithm labeled "PureSVD" in the results was implemented. The parameters of the Differential Evolution algorithm are presented below, but some of them depend on the performed experiment. In addition, a push attack was simulated to test whether the system can detect injected profiles by an attacker. Due to the large number of profiles in the database, testing was performed on a subset of profiles. First, 5000 user profiles were selected and 50 (1%) fake profiles were injected. The injected profiles were then used to rate a few items that had low average ratings and were rated by a small number of users. The RDMA value was calculated for each profile and it was checked if injected profiles were correctly identified.

Table 1. Differential Evolution parameters. The population and number of iterations varies depending on the experiment.

| Parameter name | Value |
|---|---|
| Population | [10, . . . , 100] |
| Number of Iterations | [100, . . . , 1000] |
| Crossover's Probability | 0,9 |
| Amplification Factor F | 0,8 |
| Dimensions (number of latent features) | 15 |

### 5.2. Results

By analyzing the results presented in Table 2 and figure 3, it can be seen that as the number of iterations increases, the computation time increases as well. However, for the FastRankDE algorithm, the calculations performed on average 36% faster than for the algorithm without a modification. In addition, it should be noted that the quality of recommendations for both algorithms is practically identical. Further, Table 3 and Figure 3 present how the average computation time changed according to the number of individuals in the DE population. Here, the computation time was reduced on average by 56%, also without any loss in the quality of the generated recommendations. In addition, a study was performed to compare the quality of the generated recommendations with other algorithms such as "PureSVD" or "BPR" and the results are presented in Table 4, where it can be seen that the quality of the generated recommendations is higher than these techniques.

Analyzing the results presented in figure 4, it can be seen that our system correctly identified injected profiles as outliers. They were characterized by a significant value of RDMA compared to the other selected profiles. It can also be seen that some of the non-injected profiles also had a high RDMA value. To explain this, it is important to note that the MovieLens 25m database can consist of profiles created by attackers, or this profiles can belong to users with very specific tastes.

[1] https://scikit-learn.org/

Table 2. Average time (in seconds) and quality of generated recommendations (MAP) depending on the number of iterations (for 20 individuals).

| Iterations | DE run time | DE MAP | FastRankDE run time | FastRankDE MAP | Run time improved by |
|---|---|---|---|---|---|
| 100 | 3,32 s | 0,25 | 1,91 s | 0,26 | **42** % |
| 200 | 6,00 s | 0,26 | 3,72 s | 0,23 | **38** % |
| 300 | 8,85 s | 0,26 | 5,70 s | 0,30 | **36** % |
| 400 | 11,83 s | 0,27 | 7,23 s | 0,25 | **39** % |
| 500 | 14,77 s | 0,26 | 9,80 s | 0,27 | **34** % |
| 600 | 18,20 s | 0,25 | 11,30 s | 0,23 | **38** % |
| 700 | 20,22 s | 0,25 | 13,28 s | 0,24 | **34** % |
| 800 | 24,05 s | 0,26 | 14,94 s | 0,28 | **38** % |
| 900 | 26,42 s | 0,26 | 17,70 s | 0,25 | **33** % |
| 1000 | 29,13 s | 0,25 | 19,00 s | 0,25 | **35** % |

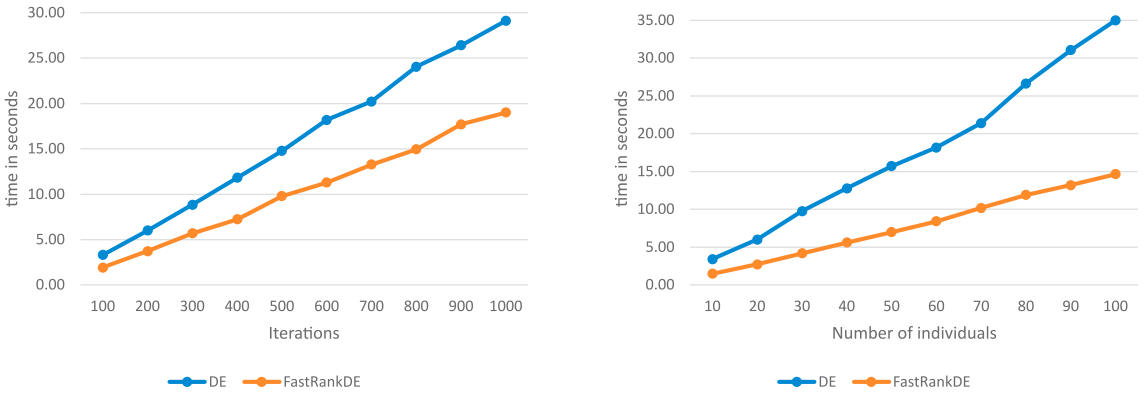


Fig. 3. Graph representing average time (in seconds): (left) depending on the number of iterations; (right) depending on the number of individuals.

Table 3. Average time (in seconds) and quality of generated recommendations (MAP) depending on the number of individuals (for 250 iterations).

| Individuals | DE run time | DE MAP | FastRankDE run time | FastRankDE MAP | Run time improved by |
|---|---|---|---|---|---|
| 10 | 3,42 s | 0,18 | 1,48 s | 0,23 | **57** % |
| 20 | 6,02 s | 0,26 | 2,74 s | 0,26 | **55** % |
| 30 | 9,77 s | 0,25 | 4,18 s | 0,26 | **57** % |
| 40 | 12,79 s | 0,25 | 5,63 s | 0,26 | **56** % |
| 50 | 15,71 s | 0,24 | 7,00 s | 0,23 | **55** % |
| 60 | 18,18 s | 0,25 | 8,42 s | 0,25 | **54** % |
| 70 | 21,40 s | 0,27 | 10,19 s | 0,25 | **52** % |
| 80 | 26,64 s | 0,25 | 11,90 s | 0,25 | **55** % |
| 90 | 31,07 s | 0,25 | 13,19 s | 0,27 | **58** % |
| 100 | 35,00 s | 0,25 | 14,65 s | 0,27 | **58** % |

Table 4. Quality of generated recommendations (MAP) for 20 individuals and 250 iterations. The rest of the parameters according to table 1

| experimentId | mean FastRankDE run time per user | Random | Most Popular | SVD MAP | BPR MAP | DE MAP |
|---|---|---|---|---|---|---|
| 1 | 3,0 s | 0,001 | 0,022 | 0,19 | 0,20 | **0,24** |
| 2 | 3,1 s | 0,003 | 0,018 | 0,19 | 0,20 | **0,21** |
| 3 | 2,9 s | 0,004 | 0,014 | 0,19 | 0,20 | **0,23** |
| 4 | 3,2 s | 0,002 | 0,025 | 0,19 | 0,20 | 0,19 |
| 5 | 3,0 s | 0,006 | 0,024 | 0,19 | 0,20 | **0,22** |

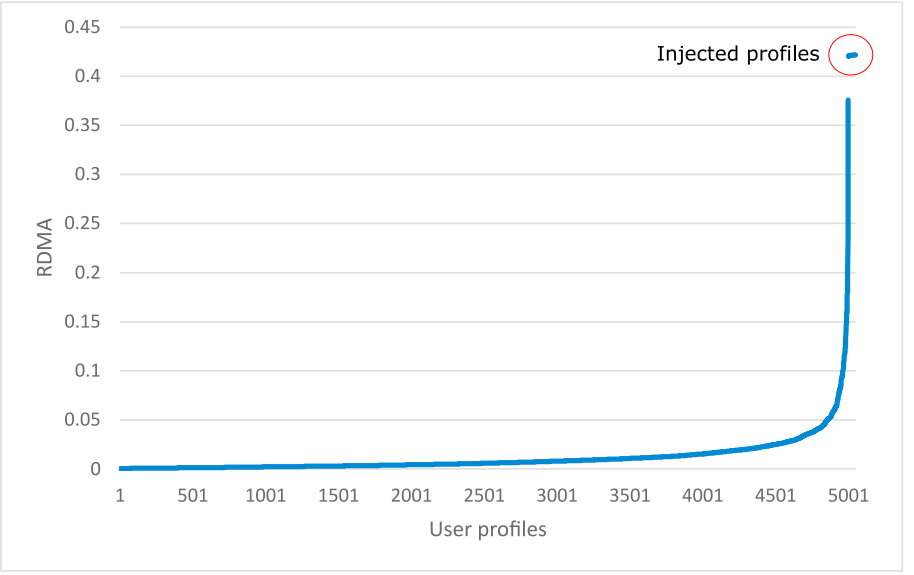


Fig. 4. Calculated RDMA value for every user profile in system. All injected profiles were detected.

## 6. Conclusion and future work

In this paper, we presented how the Differential Evolution algorithm can be accelerated by using a dedicated fitness function to generate personalized recommendations for a given user. The proposed algorithm was tested on the publicly available MovieLens 25m database. The experiments and their analysis show that our proposed fitness function significantly improves the speed of the Differential Evolution algorithm without any loss in the quality of the generated recommendations. Converting DE population and items features to matrix form, enables much easier parallelization of the process of acquiring individual values (dot products) on the basis of which ranking of recommended items can be done and this modification should also work for other metaheuristic algorithms. Additionally, our system correctly identified the injected profiles, but in the future we will conduct a more detailed study by implementing other types of attacks and compare the results with more complex algorithms for their detection. The main limitation of the proposed approach is that it operates mainly on dot products, so it is primarily suitable for problems where it is possible to present the problem into matrix multiplication form, which may not always be possible or appropriate. In future work, we will also focus on improving the quality of the generated recommendations.

## References

[1] Bennett, J., Lanning, S., Netflix, N., 2007. The netflix prize, in: In KDD Cup and Workshop in conjunction with KDD.

[2] Bhaumik, R., Mobasher, B., Burke, R., 2012. A clustering approach to unsupervised attack detection in collaborative recommender systems .

[3] Bollegala, D., Noman, N., Iba, H., 2011. Rankde: Learning a ranking function for information retrieval using differential evolution, pp. 1771–1778. doi:10.1145/2001576.2001814.

[4] Boryczka, U., Bałchanowski, M., 2020. Using differential evolution in order to create a personalized list of recommended items. Procedia Computer Science 176, 1940–1949. doi:https://doi.org/10.1016/j.procs.2020.09.233. knowledge-Based and Intelligent Information & Engineering Systems: Proceedings of the 24th International Conference KES2020.

[5] Boryczka, U., Juszczuk, P., Kłosowicz, L., 2009. A comparative study of various strategies in differential evolution, in: Evolutionary Computing and Global Optimization, pp. 19–26.

[6] Chakraborty, P., Karforma, S., 2013. Detection of profile-injection attacks in recommender systems using outlier analysis. Procedia Technology 10, 963–969. URL: https://www.sciencedirect.com/science/article/pii/S2212017313006063, doi:https://doi.org/10.1016/j.protcy.2013.12.444. first International Conference on Computational Intelligence: Modeling Techniques and Applications (CIMTA) 2013.

[7] Chirita, P.A., Nejdl, W., Zamfir, C., 2005. Preventing shilling attacks in online recommender systems, Association for Computing Machinery, New York, NY, USA. p. 67–74. URL: https://doi.org/10.1145/1097047.1097061, doi:10.1145/1097047.1097061.

[8] Cremonesi, P., Koren, Y., Turrin, R., 2010. Performance of recommender algorithms on top-n recommendation tasks, pp. 39–46. doi:10.1145/1864708.1864721.
[9] Deshpande, M., Karypis, G., 2004. Item-based top-n recommendation algorithms. ACM Trans. Inf. Syst. 22, 143–177. URL: https://doi.org/10.1145/963770.963776, doi:10.1145/963770.963776.
[10] Diaz-Aviles, E., Nejdl, W., Schmidt-Thieme, L., 2009. Swarming to rank for information retrieval, in: Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation, Association for Computing Machinery, New York, NY, USA. p. 9–16. doi:10.1145/1569901.1569904.
[11] Harper, F.M., Konstan, J.A., 2015. The movielens datasets: History and context. ACM Trans. Interact. Intell. Syst. 5. URL: https://doi.org/10.1145/2827872, doi:10.1145/2827872.
[12] Horvath, T., de Carvalho, A., 2016. Evolutionary computing in recommender systems: a review of recent research. Natural Computing doi:10.1007/s11047-016-9540-y.
[13] Karatzoglou, A., Baltrunas, L., Shi, Y., 2013. Learning to rank for recommender systems, pp. 493–494. doi:10.1145/2507157.2508063.
[14] Koren, Y., Bell, R., Volinsky, C., 2009. Matrix factorization techniques for recommender systems. Computer 42, 30–37.
[15] Kula, M., 2015. Metadata embeddings for user and item cold-start recommendations, in: Bogers, T., Koolen, M. (Eds.), Proceedings of the 2nd Workshop on New Trends on Content-Based Recommender Systems co-located with 9th ACM Conference on Recommender Systems (RecSys 2015), Vienna, Austria, September 16-20, 2015., CEUR-WS.org. pp. 14–21. URL: http://ceur-ws.org/Vol-1448/paper4.pdf.
[16] Liu, T.Y., 2009. Learning to rank for information retrieval. Found. Trends Inf. Retr. 3, 225–331. doi:10.1561/1500000016.
[17] Pei, C., Zhang, Y., Zhang, Y., Sun, F., Lin, X., Sun, H., Wu, J., Jiang, P., Ge, J., Ou, W., Pei, D., 2019. Personalized re-ranking for recommendation, Association for Computing Machinery, New York, NY, USA. p. 3–11. URL: https://doi.org/10.1145/3298689.3347000, doi:10.1145/3298689.3347000.
[18] 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, AUAI Press, Arlington, Virginia, USA. p. 452–461.
[19] Rezaimehr, F., Dadkhah, C., 2021. A survey of attack detection approaches in collaborative filtering recommender systems. Artificial Intelligence Review .
[20] Storn, R., Price, K., 1997. Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces. J. of Global Optimization 11, 341–359.
[21] Wasilewski, J., Hurley, N., 2016. Incorporating diversity in a learning to rank recommender system, in: FLAIRS Conference.