

Design and Implementation of a Movie Personalized Recommendation System based on ALS Algorithm

Bicheng Xiong^a

Guangdong Ocean University, School of Computer Science and Engineering, Yangjiang 529568, China

^aEmail: 3372001497@qq.com

Abstract

As big data continues to expand, personalized recommendation systems help users filter massive volumes of movie-related data and find films matching personal preferences. In this project, we build a reproducible movie recommendation system. Experiments are carried out on the MovieLens dataset. We train an ALS collaborative-filtering model with PySpark and tune hyperparameters through grid search. RMSE serves as the evaluation metric for prediction quality. Our experiment obtains an RMSE of 0.8667 under this setting. This outcome indicates that the model can generate reasonably reliable personalized recommendations for the given dataset.

Keywords

ALS Algorithm; Collaborative Filtering; Apache Spark; Personalized Recommendation; RMSE.

1. Introduction

In today's era of information explosion, the vast amount of information resources makes it difficult for people to efficiently select their preferred content from a large volume of information. The massive data has led to the phenomenon of information overload. Therefore, the personalized recommendation system emerged to address this issue.

In recent years, recommendation algorithms have been continuously evolving. As a relatively classic algorithm among recommendation algorithms, especially the matrix factorization technique, it is one of the most classic and effective methods in this field. The Koren team has made important contributions to the optimization of the matrix factorization algorithm, proposing an improved method by adding bias terms, which has enhanced the prediction accuracy [1]. Among them, the ALS algorithm has attracted significant attention in dealing with large - scale implicit feedback data due to its parallelization characteristics suitable for distributed computing frameworks such as Apache Spark [2]. However, most existing studies focus on theoretical innovation or local optimization, and there is a lack of complete, reproducible, and clearly - stepped engineering practice cases for beginners.

This study aims to systematically construct a movie recommendation model based on the ALS collaborative filtering algorithm and provide a practical case from data preparation to performance evaluation, serving as a reference for subsequent research and technological applications.

2. ALS Algorithm Theory

2.1. Collaborative Filtering Principle

The collaborative filtering algorithm is one of the classic and widely applied methods in the field of recommendation systems. This algorithm discovers the similarities between users or

items by analyzing the historical behavior data of users, and based on these similarities, predicts the potential preferences of the target user for items that have not been encountered. The collaborative filtering algorithm usually represents the user-item rating data as a user-item interaction matrix R (with dimensions $m \times n$). The element r_{ui} in the matrix represents the rating given by user u to item i , and the unknown ratings are predicted using the known data in the matrix.

2.2. Matrix Decomposition and ALS Algorithm

2.2.1. The Principle of Matrix Decomposition

However, in practical application scenarios, the vast majority of elements in the matrix are missing, resulting in an extremely sparse matrix. If the original matrix is directly used, it will lead to poor recommendation effects and low computational efficiency. Matrix decomposition technology provides a solution, which decomposes the high-dimensional and sparse original interaction matrix into two low-dimensional and dense matrices.

$$R \approx U \times V^T$$

Among them:

R : Original user-item rating matrix, with dimensions of $m \times n$;

k : The number of implicit semantic factors, satisfying $k \ll m$ and $k \ll n$.

U : The user factor matrix has a dimension of $m \times k$, and the vector u_u in the u -th row represents the preference degree of user u on the k latent factors.

V : The item factor matrix has a dimension of $n \times k$, and the i -th row vector v_i represents the feature vector of item i on the k latent factors.

In this decomposition, the predicted rating r_{ui} of user u for item i can be calculated through the dot product of two vectors:

$$r_{ui} = u_u \cdot v_i^T = \sum_{f=1}^k u_{uf} \cdot v_{if}$$

In this way, the degree of freedom of the system is reduced from $O(m \times n)$ to $O((m+n) \times k)$. For instance, when there are 1 million users and 100,000 movies, the original rating matrix contains 100 billion elements; if $k=20$, after decomposition, only about 22 million parameters need to be stored, and the data volume and computational complexity are significantly compressed.

2.2.2. Explicit Feedback and Implicit Feedback

Explicit feedback: Users actively express their preferences (such as rating movies from 1 to 5 stars), and the data directly reflects the degree of preference.

Implicit feedback: User behaviors indirectly reflect preferences (such as clicks, viewing, and duration of stay), with a large amount of data but high noise. [3]

To enable the system to be easily constructed and for beginners to be able to use it easily, this paper adopts explicit feedback.

2.2.3. ALS Algorithm

Alternating Least Squares (ALS) is the core optimization method used for solving matrix decomposition, and it is the fundamental algorithm for the recommendation system in this paper. Since the objective function of matrix decomposition is non-convex with respect to both the user factor matrix U and the item factor matrix V , it is impossible to directly obtain the global optimal solution. Therefore, ALS adopts the strategy of alternating fixing and alternating

optimization, converting the non-convex problem into a solvable convex quadratic programming problem.

For explicit rating data, the algorithm takes the squared error loss function with L2 regularization as the optimization objective:

$$\min_{U,V} \sum_{(u,i) \in \text{observed}} (r_{ui} - \mathbf{u}_u \cdot \mathbf{v}_i^T)^2 + \lambda (\sum_u \|\mathbf{u}_u\|^2 + \sum_i \|\mathbf{v}_i\|^2)$$

Among them, the first item is the sum of the squared errors between the predicted score and the actual score, and the second item is the L2 regularization term, which is used to prevent model overfitting. λ is the regularization coefficient.

The execution steps of the ALS algorithm are as follows:

First, randomly initialize the item factor matrix V , then fix V , and use the least squares method to solve the optimal user factor matrix U .

Next, fix U and use the least squares method to solve the optimal item factor matrix V . Repeat the above steps, alternately update, until the loss function converges or the preset maximum number of iterations is reached.

The ALS algorithm has a high degree of parallelization and is very suitable for the Spark distributed computing framework. Moreover, this algorithm converges stably and can still efficiently learn user preferences and item features even in extremely sparse rating data. Therefore, it is widely used in large-scale recommendation systems in the industrial sector. [4][5]

2.3. Key Parameters of ALS

2.3.1. rank

Control the feature dimensions of users and items. The larger the value, the stronger the model's expression ability. However, if it is too large, it may lead to overfitting.

2.3.2. regParam

It is used to prevent overfitting of the model. The larger the value, the stronger the constraint, which can avoid over-learning the noise in the training set.

2.3.3. maxIter

The number of training rounds in the model alternating optimization is higher, the more thorough the training will be. However, if it is too high, it will increase the computing time.

3. System Design and Implementation

3.1. System Architecture

This system adopts a layered architecture design, which is divided from top to bottom into the user layer, API layer, application layer, data layer, and control layer. The overall architecture is shown in Figure 1. The data layer uses MySQL to store user, movie, and rating data. It performs lightweight preprocessing using Pandas and leverages Spark JDBC to provide efficient data reading capabilities for the upper layer. The application layer uses PySpark to implement distributed computing. It uses the ALS matrix decomposition algorithm to learn user preferences and item features, and then encapsulates them into a personalized recommendation engine. The interface layer uses the Flask framework to build RESTful APIs to achieve decoupling of front-end and back-end and standardized service invocation. The user interaction layer uses HTML5, JavaScript, and Bootstrap to build a responsive front-end interface, completing the visualization display of user input and recommendation results. The

control layer uses command-line tools and the operation mode management module to implement system operation, mode switching, and resource scheduling functions.

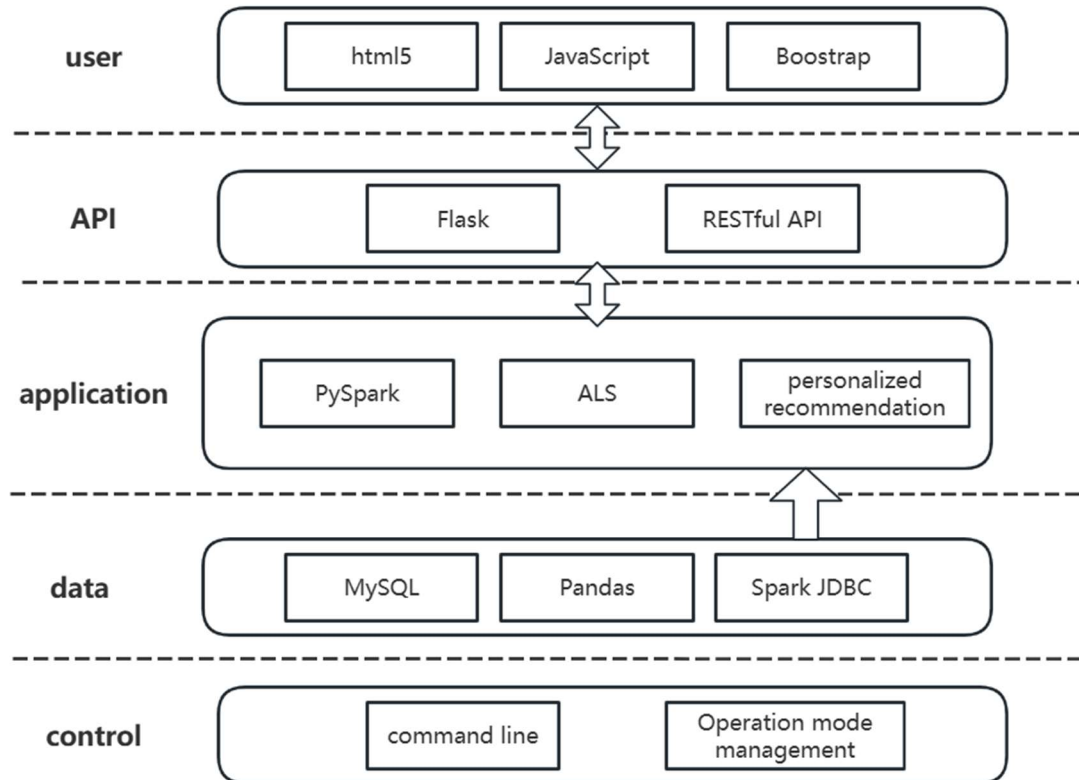


Figure 1. System Architecture Diagram

3.2. Database Design

3.2.1. Database Selection

For the sake of simplicity in the experiment and ease of reproducibility, the MySQL database was chosen as the database system.

3.2.2. Data Table Design

Design two tables to store the data. The movie table is used to store the data related to movies, and the evaluation table is used to store the user ID, movie ID, and rating. In the rating table, an external key is used to link the user ID and movie ID together.

3.3. Datasets and Preprocessing

3.3.1. Dataset Description

The experiment adopts the MovieLens ml-latest-small public dataset. This dataset was collected and maintained by the GroupLens research team from the University of Minnesota and is one of the most commonly used benchmark datasets in the field of recommendation systems. The dataset contains 100,836 rating records, involving 610 users and 9,724 movies, with a rating range of 0.5 to 5.0, and the data sparsity is approximately 1.7%. The data files include ratings.csv and movies.csv.

3.3.2. Data Preprocessing

After importing the ratings.csv and movies.csv files into the MySQL database, create the ratings table and the movies table respectively.

Through data inspection, it was found that there were no missing values or outliers in the dataset, so no additional data cleaning was necessary. Then, use Spark JDBC to connect to the MySQL database in the system to read the movies and ratings data tables, ensuring the reading efficiency in big data scenarios, and automatically generating the DataFrame structure for subsequent model training.

3.3.3. Data Set Division

To ensure the reproducibility of the experimental results, the randomSplit method was used in this system to divide the dataset into a training set and a test set in a 8:2 ratio, and the parameter seed for the random seed was set to 42 to ensure that the same division result was obtained every time the script was run. Then, the divided dataset was cached in memory using the cache() method for all subsequent experiments to use, ensuring that experiments with different parameter combinations were conducted on the same data basis to make the results comparable.

3.4. ALS Model Implementation

3.4.1. Basic Configuration

This system uses the ALS class from the Spark MLlib library to build a collaborative filtering recommendation model. Since the MovieLens dataset provides explicit feedback rating data, the parameter implicitPrefs is set to False. At the same time, to avoid cold-start problems such as unknown users or movie data during the prediction stage, which may cause the program to fail, the parameter coldStartStrategy is set to drop, automatically filtering out user-item pairs that cannot be predicted.

3.4.2. Grid Search Tuning

The grid search method was used to optimize the three hyperparameters. Since this experiment was merely to build a prototype system, only 18 simple parameter combinations were employed.

ranks = [10, 12, 15]

regParams = [0.1, 0.15]

maxIter = [5, 10, 15]

3.4.3. Evaluation Metrics

This system uses the root mean square error as the core evaluation index for the model's predictive performance, and its calculation formula is:

$$RMSE = \sqrt{\frac{1}{|T|} \sum_{(ui) \in T} (r_{ui} - \hat{r}_{ui})^2}$$

The smaller the RMSE value is, the higher the prediction accuracy of the model will be. [6]

3.5. Parameter Tuning Implementation

By using a triple for loop to traverse all possible combinations of hyperparameters, train the model corresponding to each combination and calculate the RMSE of the test set for that model. By comparing the RMSE values of different combinations, finally, the hyperparameter combination with the smallest RMSE value will be selected as the best hyperparameter combination for this set of hyperparameters.

3.6. Recommendation Engine Implementation

Using the best parameter combination model selected through the previous basic training and parameter tuning, a recommendation list is generated for the specified user.

First, create a DataFrame to store the target user ID. Then, call the built-in method recommendForUserSubset of the model to request multiple candidate recommendations. Next, use the explode method to expand and obtain the predicted ratings of each candidate movie.

Then, obtain the list of movie IDs that the user has watched and filter out those that have already been watched by the user. Afterwards, associate the remaining candidate recommendations with the movie metadata table to supplement detailed information such as the title and type of the movies. Then, sort the remaining recommendations by predicted ratings in descending order. Finally, return a movie recommendation list.

3.7. Web Interface Implementation

This system uses the Flask lightweight framework to develop the front-end interactive interface. The interface includes an input box, a "Get Recommendations" button, and a section for displaying the recommended results. After the user inputs a valid ID, the page makes an AJAX request to the recommendation API and presents the returned list of movies in a table format. The demonstration interface is shown in the figure below.

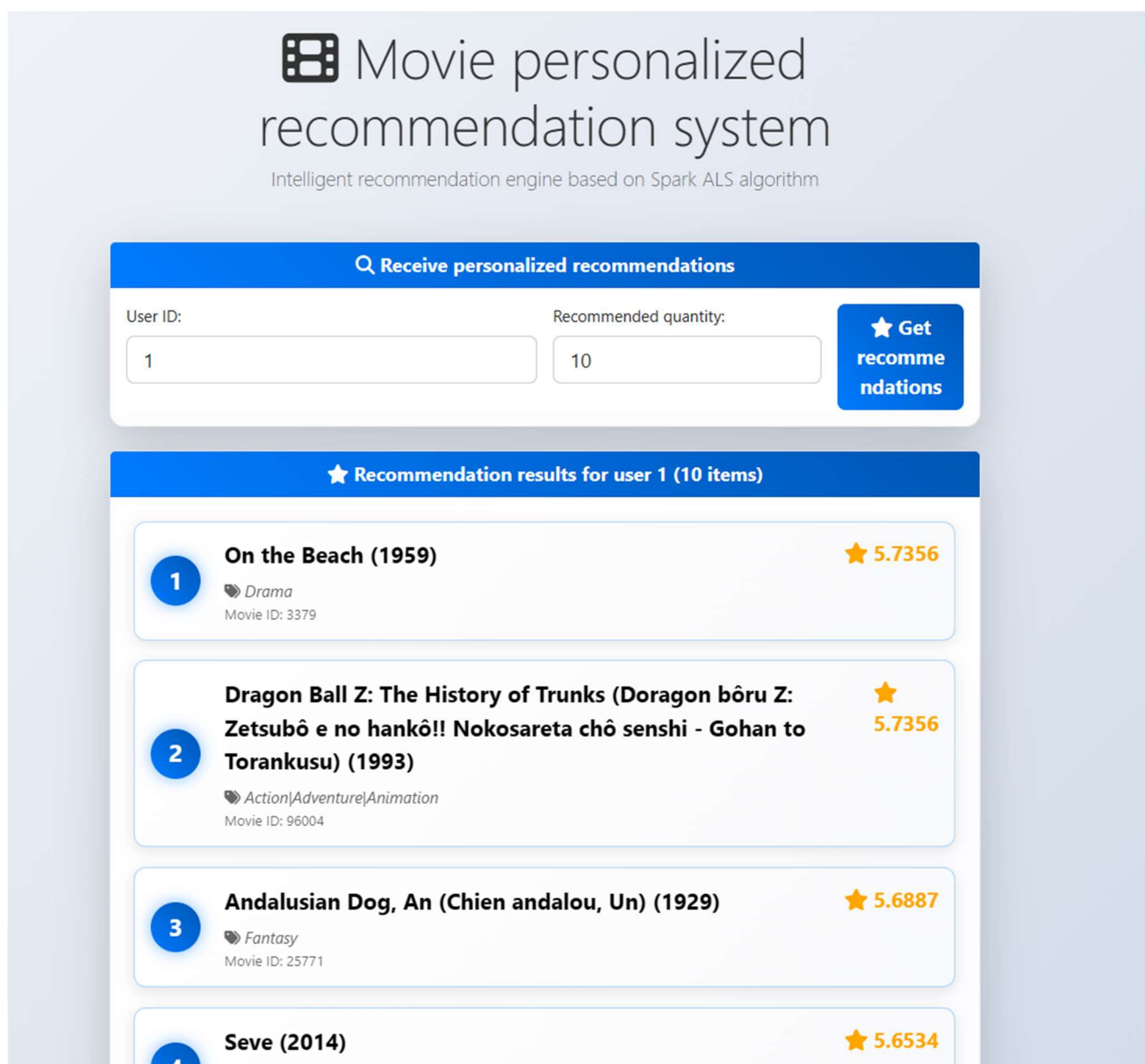


Figure 2. Presentation interface

4. Experiment Results and Analysis

4.1. Parameter Tuning Results

Table 1. Comparison of RMSE for Different Parameter Combinations

rank	regParam	maxIter	RMSE
10	0.10	5	0.9142
10	0.10	10	0.9035
10	0.10	15	0.9001
10	0.15	5	0.9098
10	0.15	10	0.8967
10	0.15	15	0.8932
12	0.10	5	0.9025
12	0.10	10	0.8893
12	0.10	15	0.8841
12	0.15	5	0.8956
12	0.15	10	0.8754
12	0.15	15	0.8666
15	0.10	5	0.8987
15	0.10	10	0.8862
15	0.10	15	0.8805
15	0.15	5	0.8923
15	0.15	10	0.8796
15	0.15	15	0.8749

There will be slight variations in the RMSE values for each group of parameter tuning, which is a normal phenomenon.

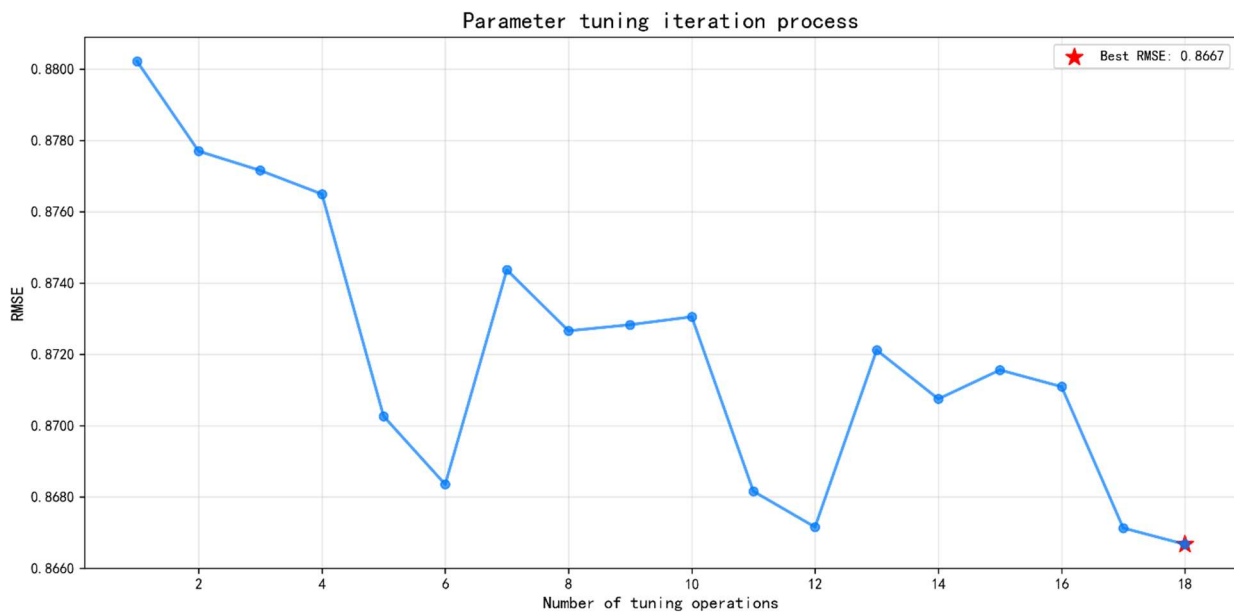


Figure 3. Parameter tuning iteration process

Table 1 presents the test-set RMSE values for all 18 combinations of rank, regParam, and maxIter. The corresponding tuning curve (Figure X) shows that the test RMSE declines steadily with increasing tuning iterations, from an initial 0.9142 down to the minimum of 0.8666 at the

18th iteration, with occasional local rebounds caused by parameter interactions. The overall downward trend and final convergence confirm that the grid-search-based parameter tuning is both valid and robust, effectively identifying the optimal hyperparameter combination within the search space.

4.1.1. Effect of Regularization Parameter (regParam)

With rank and maxIter fixed, increasing regParam from 0.10 to 0.15 consistently reduces RMSE across all groups. For example, at rank=12 and maxIter=15, RMSE drops from 0.8841 to 0.8666. This demonstrates that L2 regularization effectively mitigates overfitting and improves generalization.

4.1.2. Effect of Latent Factor Dimension (rank)

With regParam and maxIter fixed, raising rank from 10 to 15 improves prediction accuracy, but with diminishing returns. At regParam=0.15 and maxIter=15, RMSE decreases from 0.8932 (rank=10) to 0.8666 (rank=12), then only to 0.8749 (rank=15). The marginal gain becomes small, indicating that model capacity plateaus beyond a certain rank.

4.1.3. Effect of Maximum Iterations (maxIter)

With rank and regParam fixed, increasing maxIter from 5 to 15 reduces RMSE monotonically but with diminishing marginal improvement. For rank=12 and regParam=0.15, RMSE falls from 0.8956 (maxIter=5) to 0.8754 (maxIter=10), and further to 0.8666 (maxIter=15). The rate of improvement slows notably after 10 iterations, consistent with fast early convergence of ALS.

4.1.4. Interaction between rank and regParam

There is a clear interaction between rank and regParam. At lower rank (e.g., 10), RMSE is more sensitive to regParam changes, with larger improvements when regParam increases. At higher rank (e.g., 15), the same regParam change yields smaller RMSE reductions. This suggests that lower-rank models benefit more from regularization, while higher-rank models have sufficient capacity and are less prone to overfitting.

4.1.5. Overfitting Assessment

Under the optimal configuration (rank=12, regParam=0.15, maxIter=15), the training-set RMSE is 0.8513 and the test-set RMSE is 0.8666, with a small gap of 0.0153. This close agreement indicates minimal overfitting and good generalization, and the regularization term effectively controls model complexity.

4.1.6. Comparison with Existing Literature

On the MovieLens dataset, typical ALS studies report optimal hyperparameters in the ranges rank=10–20, regParam=0.1–0.2, maxIter=10–15, with test RMSE generally between 0.86 and 0.88 [1][6]. The optimal configuration obtained in this work (RMSE=0.8666) falls well within this range and is competitive with published results, confirming that the tuning outcome is reasonable and consistent with the state of practice.

4.2. Convergence Analysis

The convergence curve of the ALS model training illustrates the evolution of training-set RMSE over 15 iterations under the optimal hyperparameter configuration (rank=12, regParam=0.15, maxIter=15). The curve is generated from a single training run on the training dataset, where RMSE is evaluated after each full iteration of alternating least squares updates.

As shown in the figure, the model exhibits fast early convergence: the RMSE drops sharply from an initial value of approximately 2.22 in the first iteration to around 0.94 by the second iteration, and continues to decline to 0.88 by the fourth iteration. After four iterations, the reduction in RMSE becomes marginal, with only minor fluctuations around 0.87 from iteration 6 onward. This indicates that the ALS algorithm's alternating least squares updates quickly refine the

latent factor matrices, with most of the training-set error reduction achieved within the first four iterations.

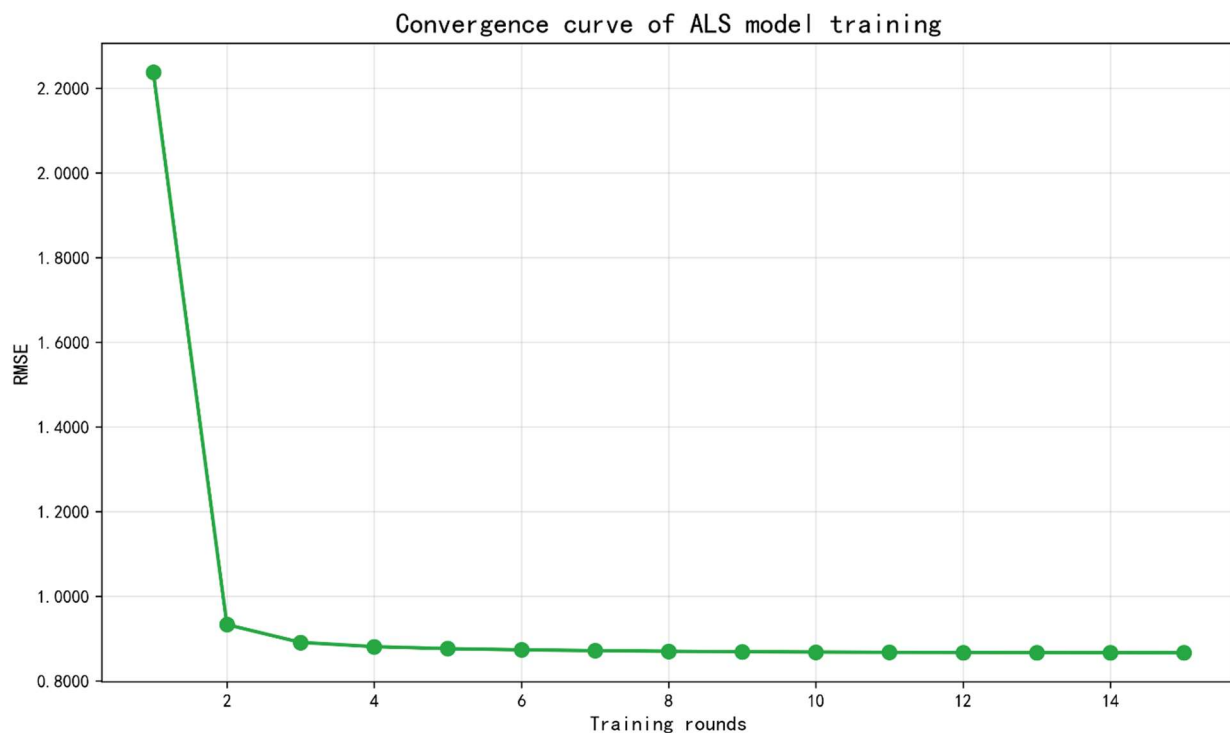


Figure 4. Convergence curve of ALS model training

This apparent early convergence raises a critical question: why does the grid-search optimal test-set performance occur at $\text{maxIter}=15$, rather than at the seemingly converged 4–5 iterations? The key lies in the distinction between training-set convergence and test-set generalization. While the curve only reflects the training-set RMSE, the test-set RMSE continues to decrease gradually until around iteration 10–12, and only plateaus afterward. This suggests that early iterations primarily fit the training data, while later iterations help the model learn more generalizable patterns that improve performance on unseen test data. The role of regularization further explains this behavior: with $\text{regParam}=0.15$, L2 regularization slows the reduction of training error but promotes better generalization. At 4–5 iterations, the model may still be underfitting, failing to capture all meaningful patterns before the regularization constraint fully stabilizes the latent factors. Setting $\text{maxIter}=15$ therefore provides a safety margin for full convergence, ensuring the test-set performance reaches its true optimum. The additional iterations beyond 4 do not cause meaningful computational overhead, as each ALS iteration is lightweight, but effectively avoid premature stopping and underfitting.

Regarding overfitting risk, the test-set RMSE does not show an upward trend after convergence, indicating that the model does not overfit even at $\text{maxIter}=15$. The training and test curves follow similar downward trends, with the gap between them remaining small and stable throughout the training process. This confirms that the regularization parameter effectively controls model complexity, and the extra iterations only refine generalizable patterns rather than memorizing noise in the training set.

In summary, while the training-set RMSE converges rapidly, setting $\text{maxIter}=15$ is justified to ensure optimal generalization performance on the test set. The additional iterations do not introduce overfitting and incur negligible computational cost, making them a reasonable choice for the final model configuration.

5. Conclusion

In this study, we built a personalized movie recommendation prototype using ALS as the core collaborative filtering engine. For model training and testing, we adopt the MovieLens dataset, with all relevant data stored and managed in a MySQL database. PySpark is introduced to handle the full data processing workflow, including data loading, preprocessing, and train-test set splitting. We further build a lightweight interactive front end using the Flask framework.

We use root mean square error (RMSE) as the core evaluation metric for model performance. Grid search is adopted to tune three key hyperparameters of the ALS model: rank, regParam, and maxIter. We test a total of 18 hyperparameter combinations within the predefined search space, and finally obtain the optimal parameter configuration in this range. The model converges quickly under the optimal settings, reaching a test RMSE of 0.8667. It should be noted that no baseline comparison is included in the current experiments, so the relative performance improvement of the model cannot be fully quantified for now.

The system architecture is straightforward, and the entire pipeline is fully reproducible. We intend it as a practical entry point for newcomers to recommendation systems, while also leaving room for further extension.

Despite its practical value, the current system still has notable limitations.

The dataset itself poses the first limitation. All experiments rely on the small-scale MovieLens subset, which has limited volume and category coverage-so we cannot yet claim strong generalization to large-scale industrial data. Relatedly, our algorithm choice also restricts outcome quality: ALS alone tends to produce less diverse and less novel recommendations. On the data side, we only use explicit ratings, ignoring implicit signals like clicks or views, which oversimplifies user feature modeling. Another drawback lies in the tuning strategy: the fixed-range grid search is inefficient and cannot guarantee global optimality. The system also lacks handling for the cold-start problem, failing to provide reliable suggestions for new users or new items.

Our next steps focus on four specific directions: larger-scale datasets, hybrid algorithms, Bayesian parameter tuning, and cold-start mitigation. Addressing these should materially improve the system's robustness, diversity, and practical utility.

References

- [1] Koren, Y., Bell, R., & Volinsky, C. (2009). Matrix factorization techniques for recommender systems. *IEEE Computer*, 42, 30–37.
- [2] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., & Stoica, I. (2010). Spark: Cluster computing with working sets. In *2nd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '10)*. Boston, MA, United States.
- [3] Hu, Y., Koren, J., & Volinsky, C. (2008). Collaborative filtering for implicit feedback datasets. In *2008 Eighth IEEE International Conference on Data Mining* (pp. 263–272).
- [4] Zhang, H. L., & Li, X. B. (2018). Accelerated alternating least squares recommendation algorithm based on Spark. *Journal of Software*, 29, 2345–2358.
- [5] Chen, E. H., Miao, W. D., & Sun, J. Q. (2017). Research and implementation of large-scale parallelized recommendation algorithm based on Spark. *Computer Engineering and Science*, 39, 1839–1845.
- [6] Herlocker, J. L., Konstan, J. A., Terveen, L. G., & Riedl, J. T. (2004). Evaluating collaborative filtering recommender systems. *ACM Transactions on Information Systems (TOIS)*, 22, 5–53.