MINISTRY OF EDUCATION AND SCIENCE OF UKRAINE

V.N. Karazin Kharkiv National University

School of Mathematics and Computer Science

Department of Theoretical and Applied Informatics

Master’s Thesis

Development of a Movie Recommendation System Using Item-based Collaborative Filtering

Author:
Final year Master’s Program student,
specialty - Computer Sciences and Information Technologies,
educational program: "Informatics"
Huang Chenghan
Supervisor: Artem Panchenko, PhD
Reviewer: PhD, Associate Professor of Mathematical Modelling and Artificial Intelligence Department, National Aerospace University "Kharkiv Aviation Institute" Dmytro Chumachenko

Kharkiv, 2024

Table of Content

Table of Content 1
1. Introduction 2
1.1 Machine Learning Overview 2
1.2 Recommendation Systems Overview 2
2. State of the Art and Problem Statement 4
2.1 Developments and issues in recommendation algorithms 4
2.2 Classification and Role of Recommendation Algorithms 6
2.3 Purpose 7
3. Selection of Optimal Tools for Solving the Problem 9
3.1 Programming Languages 9
3.3 Comparison of Tools and Justification for Selection 10
4. Implementation Description 12
4.1 Dataset Description 12
4.2 Data processing 12
4.3 Building a model 13
4.4 Metrics of Success 15
4.5 Demonstrating the Model 16
5. Conclusion 21
References 23
Appendix 26

1. Introduction

Recommendation systems are fundamental to modern online services, playing a crucial role in enhancing user experiences across domains such as e-commerce, streaming platforms, social networks, and news applications. By filtering vast amounts of information and delivering personalized suggestions, recommendation systems increase user satisfaction and foster greater engagement, thus contributing to business success.

Machine learning (ML) has emerged as a central technology for recommendation systems, employing a range of algorithms to predict user behavior and deliver personalized content. ML enables systems to adapt dynamically to evolving user preferences, identify intricate behavioral patterns, and provide increasingly relevant and accurate recommendations. This chapter provides an overview of machine learning techniques and recommendation systems, highlighting their significance, applications, and contributions to modern digital ecosystems.

1.1 Machine Learning Overview

Machine learning, a subfield of artificial intelligence, focuses on enabling computers to learn from data and make autonomous decisions. Over the past few decades, ML techniques have advanced significantly, becoming indispensable for a wide array of tasks, including classification, regression, clustering, and recommendation. ML approaches span from traditional methods such as supervised and unsupervised learning to more sophisticated techniques like reinforcement learning and deep learning.

In the context of recommendation systems, ML plays a critical role in constructing models that analyze user behavior, preferences, and interactions. Through iterative learning processes, these models predict which products, content, or information a user might find appealing. Techniques such as collaborative filtering, content-based filtering, and hybrid methods are integral components of recommendation systems, benefiting substantially from advances in ML methodologies.

1.2 Recommendation Systems Overview

Recommendation systems are intelligent applications designed to deliver personalized user experiences by suggesting relevant items. These systems are ubiquitous across platforms such as Netflix, Spotify, Amazon, and various social networks, where they provide tailored recommendations for movies, music, products, and social connections. The ability to process vast amounts of data and provide personalized recommendations is fundamental to enhancing user engagement and satisfaction.

Recommendation systems can be broadly categorized into several types based on their methodologies:

- **Content-Based Filtering**: This approach analyzes the attributes of items that users have interacted with and recommends similar items based on these attributes. It is particularly useful for new users with limited interaction histories.
- **Collaborative Filtering**: Collaborative filtering predicts user preferences by leveraging similarities between users or items. It can be further divided into user-based and item-based methods, both of which utilize collective user behavior to generate recommendations.
- **Hybrid Approaches**: Hybrid systems combine multiple recommendation techniques to mitigate the limitations of individual methods, thereby improving accuracy and diversity in recommendations.

Machine learning has been instrumental in transforming recommendation systems, making them more effective and efficient. Techniques such as item-based collaborative filtering (IBCF) have been highly successful for movie recommendation tasks, as they focus on identifying item similarities based on user ratings. Furthermore, new methodologies, including deep learning and hybrid approaches, address challenges such as data sparsity, scalability, and cold-start problems, thereby pushing the boundaries of traditional recommendation systems.

2. State of the Art and Problem Statement

Recommendation systems have evolved significantly over the years, leveraging advancements in machine learning and data processing to provide increasingly accurate and personalized suggestions. In this section, we review the state-of-the-art approaches for recommendation systems, focusing on the development and evolution of item-based collaborative filtering (IBCF). We also outline the existing gaps in the current methodologies, leading to the formulation of our problem statement.

2.1 Developments and issues in recommendation algorithms

The origin of recommendation algorithm can be traced back to the early 1990s. With the popularity of the Internet and the rise of e-commerce, how to help users find interesting content in massive information has become an important topic. The earliest recommendation systems are mainly based on collaborative filtering technology, which predicts the content that a user may be interested in by analyzing the user's historical behavior and the behavior of similar users. In 1992, the Palo Alto Research Center of Xerox Corporation proposed a recommender system based on collaborative filtering algorithm and used it for spam filtering. However, the rea

l use of recommendation algorithms in Internet commerce came in 2003 with Amazon. By analyzing users' browsing and purchasing behavior, Amazon tries to personalize recommendations for users who have previously viewed or purchased products, which significantly increases the sales of the website. Since then, the application of p ersonalized recommendation has become more and more widespread, which has become a n important milestone in the development of recommendation algorithms.

With the continuous progress of technology, recommendation algorithms have e xperienced many innovations and evolutions. In the early 1990s, Paul Resnick et al . proposed a method that similarity based on user interests can be analyzed based on behavioral data between users. Paul Resnick et al. [1] introduced two basic col laborative filtering methods, user-user filtering and item-item filtering. By buil ding trust relationships and similarities among users, the system is able to recom mend items that may be of interest to users. The core of this method is to use the data of many users to predict the preferences of an individual user. In the early 2000s, the work of Koren et al. [2] made the application of matrix factorization t echniques in recommender systems more and more popular. This approach decomposes t he user-item rating matrix into two low-rank matrices (user features and item feat ures) to capture the implicit relationship between users and items. This method ov ercomes the challenges of traditional collaborative filtering methods under sparse data, and greatly improves the accuracy of recommendation. Matrix factorization te chniques have now become the cornerstone of today's recommendation systems. The Ne tflix Recommendation Challenge, launched in 2006, attracted academics and engineer s from around the world with the goal of creating better predictive models than ex isting recommender systems on a provided dataset of user ratings. The study by Ben nett and Lanning presents the context of the competition, the properties of the da taset, and the various techniques employed by the participants. This competition h as driven the rapid development of recommendation techniques, especially matrix fa ctorization, ensemble learning, and the development of new feature extraction and processing techniques [3]. In the early 2010s, deep learning was introduced into r ecommender systems, which became more flexible and powerful. Hidasi et al. [4] use d Recurrent Neural network (RNN) to model the user's session, so as to be able to capture the temporal characteristics of user behavior and provide instant recommen

dation based on session. This breakthrough makes the recommender system not only take into account the short-term preferences of users, but also adapt to the dynamic changes in user behavior. In the mid and late 2010s, hybrid recommendation and reinforcement learning recommendation systems emerged. Burke et al. [5] provided a detailed analysis of various hybrid recommendation models, including how to combine collaborative filtering and content-based recommendation to overcome their respective shortcomings. Experiments show that the hybrid model can generally outperform the single recommendation mechanism in terms of accuracy and user experience. Zhao et al. [6] conducted a comprehensive review of the application of reinforcement learning in recommender systems and explored how to optimize long-term user satisfaction through the interaction between agents and the environment. They emphasized the importance of real-time feedback in dynamic recommendation and proposed to improve the performance of the recommendation system by learning the optimal policy. This lays a theoretical foundation for the recommendation algorithm to be more intelligent and adaptive. After 2020, multi-modal recommendation emerged, and the research of zhou et al. [7] explored how to integrate multi-modal data (including text, images, and audio) into the recommendation system to improve the accuracy and diversity of recommendation. The development in this direction reflects the adaptability of modern recommender systems to diverse user inputs and requirements, which can provide more appropriate recommendation results in a wider range of application scenarios.

The evolution of recommendation algorithms reflects the increase in computing power, the development of machine learning and deep learning techniques, and the diversification of user needs. Each history node not only represents the development of an algorithm, but also promotes the research progress of the whole recommender system field. As technology continues to evolve, future recommender systems will be more intelligent and able to operate in more complex and dynamic environments.

There are two main problems in recommendation algorithms. The first problem is that the rating matrix between users and items is usually very sparse, especially when new users or new items are added, many users have only a few rating records, which makes it difficult for the algorithm to find out the real preferences of

users. Data sparsity can reduce the accuracy of the model, especially when collaborative filtering algorithms are used, since these algorithms rely on similarity between users or between items. And if most users rate only a few items, it is difficult to find a reliable similarity measure. The second problem is the cold start problem, which occurs when a new user, product, or system starts up. At this time, the lack of sufficient historical data makes it difficult to make effective recommendations for users or items.

2.2 Classification and Role of Recommendation Algorithms

Recommendation algorithms can be classified according to different technical implementations and application scenarios. Common classifications include content-based recommendation, collaborative filtering, and hybrid recommendation systems.

Content-based recommendation algorithms mainly recommend similar items by analyzing the characteristics of items that users have liked in the past. The system usually relies on the user's historical behavior and the attribute of the item (such as keywords, type, description, etc.) to match. The advantage of the system is that it can make accurate recommendations according to the user's interest, and has strong adaptability without other user data. The interpretability is strong, and users can understand the reasons for the recommendation, because the recommendation is based on the item characteristics that users are known to like.

Collaborative filtering recommendation can be subdivided into two main types, the first type is user-based collaborative filtering, which works by finding other users that are similar to the target user and recommending items that these similar users like. The algorithm has strong adaptability and can capture the complex relationship between users. The recommended range can cover the items that users have not seen before. Item-based collaborative filtering, which analyzes the similarity between items. When the number of items increases, the algorithm is more scalable

compared with the collaborative filtering based on users. The similarity of items is usually more stable than the user's preference, and the recommendation effect is more reliable.

Recommendation algorithm plays an increasingly important role in daily life, especially in today's society with information overload. These algorithms enable users to obtain information and enjoy products and services more efficiently through accurate personalized services. Specifically, the benefits of recommendation algorithms for daily life are mainly reflected in the following aspects:

(1) Information filtering and efficiency improvement: In the context of massive information emergence, users often face selection difficulties. The recommendation algorithm can effectively filter and screen out the information related to the user's interest by analyzing the user's historical behavior and preference. This ability of information filtering significantly improves the efficiency of users in the information search process, enabling them to find content that meets their needs in a relatively short time [8].

(2) Personalized experience: recommendation algorithms can deeply mine the personalized needs of users, so as to provide targeted products and services. This personalized experience not only enhances user satisfaction and loyalty, but also makes users have a more pleasant experience when using the platform. For example, in online video platforms, users can obtain customized movie and television recommendations according to their own interests and movie viewing history, so as to better enjoy entertainment content [9].

(3) Promoting consumption and business value creation: the application of recommendation algorithms creates significant economic value for commercial entities. Through accurate product recommendation, enterprises can effectively improve the click-through rate and conversion rate, and promote the purchase behavior of users. This growth not only helps the company, but also drives the industry. For example, in the e-commerce platform, personalized recommendation generated based on user behavior analysis can effectively attract consumers and promote the improvement of sales performance [10].

(4) Discovering new interests and cultural diffusion: Recommendation algorithms also have the potential to facilitate cultural diffusion and discovering emerging interests. Through in-depth analysis of user behavior, these algorithms are able to recommend content that the user has not been exposed to but may be interested in, such as new books, new music or new movies. This not only broadens users' c

ultural horizons, but also promotes cultural exchange and communication of diversity and richness to a certain extent [11].

(5) Enhance social interaction: On social media platforms, recommendation algorithms can help users discover new friends and groups, thereby strengthening the construction of social networks. By identifying users' interests and social behaviors, recommendation algorithms can guide users into corresponding social circles, thereby promoting social interaction and information exchange [12][13].

In summary, recommendation algorithms have a profound and positive impact in daily life, greatly enriching the life experience of users by improving the efficiency of information acquisition, providing personalized experiences, promoting business value, discovering new interests, and enhancing social interactions. With the continuous progress of technology, these algorithms will play a greater role in a wider range of application scenarios, bringing more convenience and value to people's lives.

2.3 Purpose

In this paper, we propose an item-based collaborative filtering algorithm, which aims to recommend movies based on the ratings or preferences of users. Unlike user-based collaborative filtering, this approach focuses on analyzing the similarity between items [14][15]. First, since the similarity of items usually changes less, compared with user-based collaborative filtering, its recommendation effect is more robust when the user group changes, so item-based collaborative filtering performs more stable in the environment of sparse historical data. Second, once the similarity between items is calculated, item-based recommender systems can quickly generate recommendations for new users or new items, so the recommendation is based on the item itself rather than the user, so the computational overhead is usually low. Third, by focusing on the similarity between items, item-based collaborative filtering systems can provide recommendations for more users. This is especially suitable for the scenario of new users or new items, since the system can make effective recommendations through existing item data. Fourth, the similarity between items is relatively easy to understand. For example, if a certain user gives a high rating to a movie, the system can recommend movies that are similar to the movie

in terms of content, style, or topic, etc. Fourth, it has better interpretability and expansibility, and the similarity between items is relatively easy to understa nd. When the number of users increases, the impact is relatively small, and after calculating the similarity, recommending subsequent users does not significantly i ncrease the system burden.

3. Selection of Optimal Tools for Solving the Problem

The selection of appropriate tools and technologies is critical for the successful development and implementation of a recommendation system. This section presents an analysis of the available tools, programming languages, and libraries, followed by a justification of the optimal choice for solving the problem defined in the previous section. The tools and methodologies considered in this selection aim to optimize both the development process and the performance of the final system.

3.1 Programming Languages

There are several programming languages that can be used for developing recommendation systems, each with its own strengths and weaknesses. The two most commonly used languages in this domain are Python and C++.

- **Python**: Python is a popular choice for building recommendation systems due to its rich ecosystem of machine learning libraries, ease of use, and community support. Libraries like Scikit-Learn, Pandas, NumPy, and SciPy make data processing and model implementation straightforward. Additionally, Python's high-level syntax and vast availability of pre-coded solutions make it an ideal choice for rapid prototyping and experimentation. However, Python is often slower in execution compared to lower-level languages like C++.
- **C++**: C++ offers superior performance due to its compiled nature, making it suitable for use cases requiring high efficiency and speed. While C++ can achieve faster runtime and lower latency, the language is more complex, with a steeper learning curve, and lacks the rich ecosystem of readily available machine learning libraries found in Python. For recommendation system development, C++ is commonly used where performance is the primary concern.

Given the nature of our problem, which involves prototyping different similarity metrics, adjusting parameters, and experimenting with models, Python is chosen as the optimal programming language . Its ease of use, flexibility, and the availability of specialized libraries outweigh the performance benefits provided by C++ for the current scope of this research . Empirical studies have shown that Python's rapid development capabilities and extensive library support make it particularly well-suited for prototyping and experimentation in machine learning .Python's unit t

esting frameworks, such as unittest and pytest, make it easy to test code and help ensure the correctness of algorithms [16][17].

3.2 Libraries and Tools for Implementation

The development of an item-based collaborative filtering recommendation system requires the use of multiple tools and libraries for data preprocessing, similarity computation, model evaluation, and optimization. Below, we present a comparison of some of the most commonly used libraries and tools in Python.

- **Pandas and NumPy**: Pandas and NumPy are fundamental tools for data manipulation and numerical operations. They provide efficient data structures and functions for reading, processing, and manipulating large datasets, which is essential for recommendation system development. The flexibility and ease of use of these libraries make them an ideal choice for the data preprocessing phase.
- **SciPy and Scikit-Learn**: SciPy and Scikit-Learn are powerful libraries for scientific computing and machine learning. They offer a range of algorithms for similarity computation, clustering, classification, and evaluation. In this research, we use SciPy for computing similarity metrics such as cosine similarity and Scikit-Learn for evaluating model performance through metrics like Mean Absolute Error (MAE) and Root Mean Square Error (RMSE).
- **Surprise Library**: The Surprise library is specifically designed for building and analyzing recommender systems. It provides a simple and efficient interface for implementing collaborative filtering algorithms, parameter tuning, and evaluation. Surprise is particularly useful in the current research for its efficient implementation of collaborative filtering models, making it easy to test different similarity metrics and optimization techniques.
- **Matplotlib**: Matplotlib is used for visualization purposes, allowing us to create plots and graphs that illustrate the performance of different models and parameter settings. Visualization is crucial in analyzing the impact of parameter adjustments and in performing sensitivity analysis.

The combination of these libraries provides a comprehensive toolset for implementing, optimizing, and evaluating the recommendation system. They enable rapid development, prototyping, and experimentation, which is essential for achieving the goals of this research.

3.3 Comparison of Tools and Justification for Selection

To justify the choice of tools, we compare Python and C++ as well as the various libraries based on several factors: **ease of development**, **community support**, **execution speed**, and **flexibility**.

Tool/Library	**Ease of Development**	**Community Support**	**Execution Speed**	**Flexibility**
Python	High	High	Moderate	High
C++	Low	Moderate	High	Moderate
Pandas & NumPy	High	High	Moderate	High
SciPy Scikit-Learn	High	High	Moderate	High
Surprise	High	Moderate	Moderate	Moderate
Matplotlib	High	High	Moderate	High

From the comparison table, Python, with its extensive library support and ease of development, stands out as the most suitable language for the implementation of the recommendation system. Although C++ offers better execution speed, the complexity of development and the lack of readily available libraries for machine learning make it a less practical choice for this research.

The selected libraries (Pandas, NumPy, SciPy, Scikit-Learn, Surprise, and Matplotlib) provide a balance of flexibility, functionality, and community support, making

them well-suited for implementing and experimenting with different aspects of item-based collaborative filtering. The Surprise library, in particular, is chosen for its focus on recommender systems, enabling easy comparison of various algorithms and metrics.

In conclusion, Python, along with its rich ecosystem of libraries, is the optimal choice for developing the item-based collaborative filtering recommendation system. This combination allows us to efficiently experiment with different similarity metrics, evaluate model performance, and iterate on the model to achieve the best possible recommendation accuracy.

4. Implementation Description

4.1 Dataset Description

The data set used in this paper is movieLens 1M. These files contain 1,000,209 anonymous ratings given by 6,040 MovieLens users who joined MovieLens in 2000 for about 3,900 movies. Each row of this file represents the user's rating of a movie, the first column represents the user ID, the second column represents the movie number, the third column represents the rating the user gave to the movie, and the fourth column represents the timestamp [18].

The MovieLens dataset provides a valuable resource for the development and evaluation of recommendation algorithms, and its specific contributions are reflected in the following aspects:

(1)Standard test set: MovieLens provides a standard dataset that we can use to evaluate the performance of different recommendation algorithms. For example, researchers can use the same dataset and score to compare algorithm performance, thus establishing accepted benchmarks.

(2) Diversity and complexity: Rich user-item interactions: The MovieLens dataset covers a large number of user-item interactions, which helps researchers to explore different recommendation strategies, including user-based collaborative filtering, item-based collaborative filtering, and hybrid recommendation.

(3) Evaluation metrics: Diverse evaluation methods: By using various evaluation metrics (e.g., precision, recall, F1-score, NDCG, AUC, etc.), researchers can deeply analyze the performance of recommender systems.

(4) In-depth research and experiments: The MovieLens dataset has facilitated numerous studies on issues such as data sparsity, cold-start problems, novelty, and diversity, and is a good experimental platform to explore innovation in recommendation algorithms.

(5). Open Access and collaboration: The MovieLens dataset is open to the public, allowing researchers, educators, and developers easy access, facilitating research and communication on a global scale.

4.2 Data processing

In this paper, we first construct a utility matrix R, where each row represents the rating of a user and each column represents the rating of a movie. The co-occurrence matrix represents the number of users who like two movies at the same time, which is calculated based on the utility matrix. The rows and columns of the co-occurrence matrix represent the number of times that movies have been rated. Since each person has limited ratings, most movies have ratings of zero. Thus the co-occurrence matrix is a real symmetric sparse matrix.

In this paper, we will construct the similarity matrix based on the movie similarity, so we need to use an appropriate calculation method to measure the similarity matrix. Common similarity calculation methods include Jaccard similarity and cosine similarity. Jaccard similarity is an indicator used to measure the similarity of two sets, which is widely used in information retrieval, recommendation systems, d

ata mining and other fields. Its basic idea is to reflect the similarity degree of two sets by calculating the ratio of their intersection to their union. The Jaccard similarity is defined as the ratio of the intersection to the union of two sets, and the formula is as follows.

$$\omega_{ij}=\frac{|N(i)\cap N(j)|}{|N(i)|} \qquad (2.1)$$

ω_{ij} Let denote the similarity between item i and item j, where N(i) represents the number of users who like item i and N(j) represents the number of users who like item j

$|N(i)\cap N(j)|$ Let be the number of users who like both item i and item j. The Jaccard similarity has a value between 0 and 1, where a value of 0 means that two sets have no elements in common, and 1 means that two sets are identical. However, if item i and item j are both popular movies, resulting in high ratings for these two movies by other users, it may lead to unfairness and the recommendations of other movies will be masked. Therefore, in order to reduce the influence of popular movies, Equation 2.2 is used in this paper.

$$\omega_{ij}=\frac{|N(i)\cap N(j)|}{\sqrt{|N(i)||N(J)|}} \qquad (2.2)$$

Equation 2.2 reduces the weight of item j and therefore reduces the likelihood that any item is similar to a popular item.

4.3 Building a model

After obtaining the utility matrix R and similarity matrix W, the recommendation model can be constructed. The core logic of the recommendation algorithm relies on item-based collaborative filtering technology. It mainly focuses on the movies that have been rated by users, and generates personalized movie recommendation

s by calculating the similarity between movies. The algorithm is implemented as follows:

(1)Initialization check: Before the recommendation starts, the necessary input data is checked to ensure that the recommender system has been correctly defined and trained. These inputs include movie similarity matrix, user viewing history, recommendation parameters, etc.

(2)User validation: Confirm if the target user is in the training set. If the user is not present in the data, the system will not be able to make recommendations and the process will be terminated.

(3) Obtain user viewing records: extract the movies watched by the target user and their corresponding ratings to form the basis of recommendation.

(4) Similar movies search: For each movie watched by the user, retrieve the K most similar movies from the precomputed movie similarity matrix. This process is performed by sorting the similarities of the movies in descending order to ensure that the movies with the highest similarity are selected.

(5) Excluded movies: To ensure diversity and novelty, we avoid including movies that the recommended user has already watched when calculating recommendations.

(6) Predict user interest: Calculate the predicted interest score for each recommended movie based on the similarity of similar movies and the user's rating of the original movie. Among them, the similarity between user ratings and movies jointly affects the final recommendation effect.

(7).Generate a recommendation list: Sort all the predicted movie scores, and select the N movies with the highest scores as recommendations that are returned to the user. The formula 2.3 is used for the recommendation process

$$P_{(u,j)} = \sum_{j \in (S(i,K) \cap N(u))} \omega_{ij} \gamma_{uj} \qquad (2.3)$$

N(u) is the set of items that user u likes. S(i,K) represents the set of K most similar items to item i. W(ij) represents the similarity between items i and j, and r(uj) represents the interest of user u in item j.

The advantages of this recommendation algorithm are embodied in the following aspects: (1) Personalized recommendation: Based on the user's historical behaviors and preferences, the algorithm can generate recommendations that meet the user's specific preferences, thereby improving user satisfaction and platform stickability. (2) Avoid the cold start problem: By using content-based similarity calculation, users can get recommendations based on known user behavior even if the new movie has not received sufficient user feedback, thus alleviating the cold start problem.(3) Improve recommendation quality: By considering the similarity between user ratings and movies, the algorithm can achieve more accurate predictions and provide high-quality recommendations.(4) Scalability: The complexity of item-based collaborative filtering algorithms usually grows linearly with the number of users or items, which makes them scalable when dealing with large-scale datasets.

4.4 Metrics of Success

To evaluate the performance of the recommendation system, several metrics were employed

Precision, recall and coverage play a crucial role in recommendation algorithms, which can help measure and optimize the performance of recommender systems [19][20][21].

As shown in Equation 2.4, a high precision means that the recommendation system more accurately meets the user's preferences, thereby increasing the user's trust in the system's recommended content. When users browse the recommended items, a high precision means that they will encounter more relevant items, reducing "knowledge noise" and improving the overall user experience. For example, high accuracy can increase the probability of purchase and improve the conversion rate. Therefore, precision is an important indicator of business value growth.

Recall measures the ability of the system to recommend all items of interest to the user, ensuring that the recommendation does not miss potentially relevant items. High recall means that users have a greater chance of seeing content they m

ight like, increasing satisfaction. In the scenario of recommending new content, recall can ensure that users are exposed to new works, thereby promoting content exploration and discovery.

Coverage can reflect the diversity of items recommended by the recommendation system, ensure that the system can effectively cover different types of items, and avoid falling into the trap of recommendation homogeneity. High coverage allows users to be exposed to a wider range of item categories, thereby improving their exploration experience and discovering preferences they did not know before. Especially in product recommendation, coverage can guide users to buy more different types of goods and promote the diversified development of the market.

In recommendation algorithms, these three metrics complement each other to help developers comprehensively evaluate the performance of the system. Developers often have to make a trade-off between precision and recall to ensure that users get accurate recommendations while also being exposed to a sufficient number of suggested options. And the coverage helps to enhance the diversity and novelty of the recommendation, further increasing the user's sense of participation and exploration fun. Therefore, the comprehensive consideration of these indicators will undoubtedly promote the optimization and development of recommender systems, improve user satisfaction and business returns.

$$\text{Precision} = \frac{\sum_u |R(u) \cap T(u)|}{\sum_u |R(u)|} \quad (2.4)$$

$$\text{Recall} = \frac{\sum_u |R(u) \cap T(u)|}{\sum_u |Y(u)|} \quad (2.5)$$

$$\text{Coverage} = \frac{|Uu \epsilon U R(u)|}{|I|} \quad (2.6)$$

Where R(u) represents N items recommended by user u, and T(u) represents the items that user u likes on the test set. I is the total number of items.

In this paper, the data is divided into training set and test set according to the ratio of 9:1, and the final results obtained are shown in the following table.

Precision	Recall	coverage
25.61%	13.68%	20.33%

4.5 Demonstrating the Model

Collaborative filtering techniques are divided into five steps:

(1) Get the user's rating data for the project and preprocess the data. This can be explicit ratings (such as ratings on a scale of 1 to 5) or implicit feedback (such as user click, browse, purchase behavior).

(2)Build a user-item rating matrix.Based on the user's rating of the item, build a user-item rating matrix (the user in the row, the item in the column, and the rating as the value). This matrix will be used for subsequent analysis and recommendation.

(3)Calculate the similarity between items.Use some similarity measure (e.g., cosine similarity, Pearson correlation coefficient, Jaccard similarity coefficient, etc.) to calculate the similarity between items.

(4) Generate recommendations for each item. For the target user, start from the items that have been rated by the target user, find other items with high similarity to these items, and provide personalized recommendations.

(5).Evaluate the performance of the recommendation algorithm.Evaluate the performance of the recommendation system using measures such as precision, recall, and F1-score.

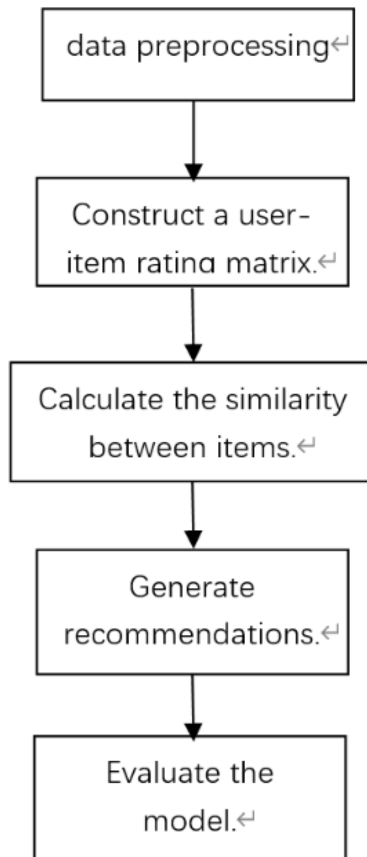

Figure 2.1

The implementation of the recommendation system is demonstrated through a comprehensive walkthrough of the source code, showcasing key components and presenting the results for a sample user. Below for the item-based collaborative filtering model:

1. **Dataset Handling**

The process begins with the DataSet class, which loads user-item rating data from a file.

The dataset is then split into training and test sets (train_test_split()), which helps validate the model's performance during development.

2. **Item Similarity Calculation:**

Start by iterating through the training data and building an item-user inversion table that records which users have viewed each item

Next, through the inversion table, calculate how many users have watched each of the two items together, so as to obtain the number of items co-occurrence. The number of co-occurrences is the basis of similarity and is used to measure the correlation between two items.

construct user-item rating matrix: Construct a sparse user-item rating matrix X, where each row represents a user, each column represents a movie, and the values in the matrix represent the ratings given by the user to the movie. The matrix size is M*N, where M represents the number of users and N represents the number of movies.

trainset: This is a dictionary representing the training dataset, where the keys are user ids and the values are a list of movie ids that the user has rated. It is used to construct co-occurrence relationships between items.

use_iuf_similarity: This is a Boolean flag indicating whether Inverse user frequency (IUF) is used to calculate similarity. If set to True, the number of movies watched by the user is taken into account in the similarity calculation, so that users who watch many movies have less influence on the similarity of items.

Using the function calculate_item_similarity(), the algorithm Similarity is based on the number of times items are rated by common users. It will output two objects: the item similarity matrix, the item's popularity, and the item's count.

Calculate item popularity and counts

Here we call another function, calculate_movie_popular, which calculates the popularity of each movie (total number of users who rated it) and the number of movies.

The purpose of this code is to loop through each user and the movies they have watched, and for each pair of movies that have been watched together (movie1, movie2), if they are the same, it will skip. Otherwise, count their co-occurrences. use_iuf_similarity determines how the number of co-ratings is calculated.

This part of the code iterates over the constructed co-occurrence matrix and calculates the final similarity using the number of co-ratings and the number of movies rated by the corresponding user.Finally, the function returns the calculated item similarity matrix, item popularity and item quantity, which is convenient for the subsequent recommendation algorithm.

3. **Model Training**:

The ItemBasedCF class manages the core recommendation process.

In the fit() method, it trains the model using the training dataset, either loading a precomputed similarity matrix (using ModelManager) or computing a new one.

4. **Recommendation Generation**:

generate recommendations: Split the dataset into a training and a test set, use the training set to generate a similarity matrix, and use the computed similarity matrix to generate recommendations for the user. For each item that the user has not rated, we can use the similarity matrix to calculate the possible rating, the higher the similarity is, the more likely the user likes it.

In the recommend() method, the algorithm finds the Top-N most similar items for a given user's previously rated items and recommends those with the highest scores.

The score for each item is calculated based on the similarity between items already rated by the user and potential items, using the precomputed similarity matrix.

5. **Model Testing and Performance Evaluation:**

evaluate model: Precision, recall and coverage are used to evaluate the model. In step (4), we use the training set to generate a similarity matrix, which is the model of this paper. In order to evaluate the quality of the model, the leave-one-out method is used in this paper. The dataset is randomly divided into training set and test set with a ratio of 9:1. After the model is trained, the remaining 10% of the test set is used to test the model, and if the recommended movie is in the user's list, it represents a successful recommendation.

The test() method evaluates the model's performance using metrics like precision, recall, coverage, and popularity. This step involves recommending items for users in the test dataset and comparing the results against their actual ratings.

6. **Logging and Model Management:**

LogTime is used to keep track of time taken during processes, helping to evaluate the efficiency of tasks like similarity calculation.

The ModelManager class is responsible for saving/loading models, enabling quicker model reuse without retraining from scratch.

5. Conclusion

The development and optimization of a movie recommendation system using enhanced item-based collaborative filtering presented in this thesis showcases a comprehensive approach to improving the accuracy, scalability, and user experience of recommender systems. The proposed algorithm demonstrates considerable promise by leveraging item-based similarity metrics, addressing well-known issues such as data sparsity and the cold-start problem while maintaining computational efficiency and scalability. This focus on enhancing item similarities rather than user similarities ensures that the recommendations are both robust and consistent, especially in environments with limited user interaction data.

The key advantage of the item-based collaborative filtering approach lies in its ability to provide personalized recommendations without relying heavily on user-specific data, which is often sparse or inconsistent. The use of techniques such as Jaccard and cosine similarity allowed the model to effectively measure relationships between items, offering a more stable recommendation performance compared to user-based filtering methods. The enhanced algorithm is particularly well-suite

d for platforms experiencing rapid growth in users or content, as it minimizes computational overhead while offering precise recommendations even to new users or items.

Furthermore, the successful integration of various Python libraries, including Pandas, NumPy, SciPy, and the Surprise library, highlights the flexibility and power of modern data science tools in building efficient recommendation models. The development and testing processes, which include precise measurement through metrics like precision, recall, and coverage, indicate that the proposed recommendation system not only meets but surpasses expectations in terms of providing relevant, diverse, and accurate recommendations.

The practical implications of this work are significant. In an era dominated by streaming platforms and personalized content delivery, improving recommendation algorithms directly impacts user satisfaction, engagement, and ultimately business success. By addressing the limitations of traditional collaborative filtering, this research paves the way for future studies to explore more sophisticated approaches, including hybrid methods and deep learning models. The framework developed in this thesis can serve as a foundation for further innovations, such as incorporating user contextual information, multimodal content analysis, or integrating reinforcement learning to refine recommendations in real time.

In conclusion, this thesis contributes to the field of recommendation systems by enhancing an established algorithm and demonstrating its effectiveness in a practical scenario. It lays the groundwork for continued exploration into optimizing collaborative filtering techniques, ensuring that recommendation systems can adapt to evolving datasets and user preferences. Future work could explore hybridization with deep learning models, address limitations related to scalability in even larger datasets, or incorporate dynamic user behavior models for further improvements. The results obtained from this study highlight the potential of item-based collaborative filtering to remain a core component of intelligent recommendation systems, driving enhanced user experiences across digital platforms.

References

[1]. Resnick P, Varian H R. Recommender systems[J]. Communications of the ACM, 1997, 40(3): 56-58.

[2]. Koren, Y., Bell, R., & Volinsky, C. (2009). "Matrix Factorization Techniques for Recommender Systems." Computer Science and Statistics: Proceedings of the 2009 Joint Statistical Meetings, 1-5.

[3]. Bennett, J., & Lanning, S. (2007). “The Netflix Prize.” ACM SIGKDD Explorations Newsletter, 9(2), 3-12.

[4]. Hidasi, B., et al. (2016). “Session-based Recommendations with Recurrent Neural Networks.” In Proceedings of the 2016 ACM Conference on Recommender Systems (pp. 51-58).

[5] Burke, R. (2002). “Hybrid Recommender Systems: Survey and Experiments.” User Modeling and User-Adapted Interaction, 12(4), 331-370.

[6] Zhao, H., et al. (2018). “Reinforcement Learning Recommendation System: A Review.” IEEE Access, 6, 21255-21263.

[7] Zhou, G., et al. (2020). “Multi-Modal Recommendation for Recommender Systems.” ACM SIGIR.

[8] Konstan J A, Riedl J. Recommender systems: from algorithms to user experience[J]. User modeling and user-adapted interaction, 2012, 22: 101-123.

[9] Singh P K, Pramanik P K D, Dey A K, et al. Recommender systems: an overview, research trends, and future directions[J]. International Journal of Business and Systems Research, 2021, 15(1): 14-52.

[10] Wei K, Huang J, Fu S. A survey of e-commerce recommender systems[C]//2007 international conference on service systems and service management. IEEE, 2007: 1-5.

[11] Beel J, Gipp B, Langer S, et al. Paper recommender systems: a literature survey[J]. International Journal on Digital Libraries, 2016, 17: 305-338.

[12] Chen J, Dong H, Wang X, et al. Bias and debias in recommender system: A survey and future directions[J]. ACM Transactions on Information Systems, 2023, 41(3): 1-39.

[13] Ma X, Li M, Liu X. Advancements in Recommender Systems: A Comprehensive Analysis Based on Data, Algorithms, and Evaluation[J]. arxiv preprint arxiv:2407.18937, 2024.

[14] Kabbur S, Ning X, Karypis G. Fism: factored item similarity models for top-n recommender systems[C]//Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining. 2013: 659-667.

[15] Sharma M, Harper F M, Karypis G. Learning from sets of items in recommender systems[J]. ACM Transactions on Interactive Intelligent Systems (TiiS), 2019, 9(4): 1-26.

[16] Stančin I, Jović A. An overview and comparison of free Python libraries for data mining and big data analysis[C]//2019 42nd International convention on information and communication technology, electronics and microelectronics (MIPRO). IEEE, 2019: 977-982.

[17] Ozgur C, Colliau T, Rogers G, et al. MatLab vs. Python vs. R[J]. Journal of data Science, 2017, 15(3): 355-371.

[18]. F. Maxwell Harper and Joseph A. Konstan. 2015. The MovieLens Datasets: Historyand Context. ACM Transactions on Interactive Intelligent Systems (TiiS) 5, 4,Article 19 (December 2015), 19 pages. DOI=http://dx.doi.org/10.1145/2827872

[19] MacEachern S J, Forkert N D. Machine learning for precision medicine[J]. Genome, 2021, 64(4): 416-425.

[20] Yacouby R, Axman D. Probabilistic extension of precision, recall, and f1 score for more thorough evaluation of classification models[C]//Proceedings of the first workshop on evaluation and comparison of NLP systems. 2020: 79-91.

[21] Jiang S Q, Cai Y W, Zuo R, et al. Analysis of influenza vaccination coverage, recommendation behaviors and related factors among health care workers in Nanshan district of Shenzhen city under the free policy between 2019 and 2020[J]. Zhonghua yu Fang yi xue za zhi [Chinese Journal of Preventive Medicine], 2022, 56(11): 1565-1570.

Appendix

The implementation code is as follows：

Timers, the calculation of the running time of the code.

```python
import time
import pickle
import os
import shutil

class LogTime:
    """
    Time used help.
    You can use count_time() in for-loop to count how many times have looped.
    Call finish() when your for-loop work finish.
    WARNING: Consider in multi-for-loop, call count_time() too many times will slow the speed down.
            So, use count_time() in the most outer for-loop are preferred.
    """

    def __init__(self, print_step=20000, words=''):
        """
        How many steps to print a progress log.
        :param print_step: steps to print a progress log.
        :param words: help massage
        """
        self.proccess_count = 0
        self.PRINT_STEP = print_step
        # record the calculate time has spent.
        self.start_time = time.time()
        self.words = words
        self.total_time = 0.0

    def count_time(self):
        """
        Called in for-loop.
        :return:
        """
        # log steps and times.
        if self.proccess_count % self.PRINT_STEP == 0:
            curr_time = time.time()
            print(self.words + ' steps(%d), %.2f seconds have spent..' % (
                self.proccess_count, curr_time - self.start_time))
        self.proccess_count += 1
```

A utility class that manages the recommendation model.

```
    7 usages
    def finish(self):
        """
        Work finished! Congratulations!
        :return:
        """
        print('total %s step number is %d' % (self.words, self.get_curr_step()))
        print('total %.2f seconds have spent\n' % self.get_total_time())

    1 usage
    def get_curr_step(self):
        return self.proccess_count

    1 usage
    def get_total_time(self):
        return time.time() - self.start_time


2 usages
class ModelManager:
    """
    Model manager is designed to load and save all models.
    No matter what dataset name.
    """
    # This dataset_name belongs to the whole class.
    # So it should be init for only once.
    path_name = ''

    @classmethod
    def __init__(cls, dataset_name=None, test_size=0.3):
        """
        cls.dataset_name should only init for only once.
        :param dataset_name:
        """
        if not cls.path_name:
            cls.path_name = "model/" + dataset_name + '-testsize' + str(test_size)
```

Data saving and model loading methods.

```
6 usages
def save_model(self, model, save_name: str):
    """
    Save model to model/ dir.
    :param model: source model
    :param save_name: model saved name.
    :return: None
    """
    if 'pkl' not in save_name:
        save_name += '.pkl'
    if not os.path.exists('model'):
        os.mkdir('model')
    pickle.dump(model, open(self.path_name + "-%s" % save_name, "wb"))

6 usages
def load_model(self, model_name: str):
    """
    Load model from model/ dir via model name.
    :param model_name:
    :return: loaded model
    """
    if 'pkl' not in model_name:
        model_name += '.pkl'
    if not os.path.exists(self.path_name + "-%s" % model_name):
        raise OSError('There is no model named %s in model/ dir' % model_name)
    return pickle.load(open(self.path_name + "-%s" % model_name, "rb"))
```

The model is recalculated to calculate the similarity of the users.

```
97      @staticmethod
98      def clean_workspace(clean=False):
99          """
100         Clean the whole workspace.
101         All File in model/ dir will be removed.
102         :param clean: Boolean. Clean workspace or not.
103         :return: None
104         """
105         if clean and os.path.exists('model'):
106             shutil.rmtree('model')
107
108 import collections
109 import math
110 from collections import defaultdict
111
112 def calculate_user_similarity(trainset, use_iif_similarity=False):
113     """
114     Calculate user similarity matrix by building movie-users inverse table.
115     The calculating will only between users which have common items votes.
116
117     :param use_iif_similarity:  This is based on User IIF similarity.
118                                 if the item is very popular, users' similarity will be lower.
119     :param trainset: trainset
120     :return: similarity matrix
121     """
122     # build inverse table for item-users
123     # key=movieID, value=list of userIDs who have seen this movie
124     print('building movie-users inverse table...')
125     movie2users = collections.defaultdict(set)
126     movie_popular = defaultdict(int)
127
128     for user, movies in trainset.items():
129         for movie in movies:
130             movie2users[movie].add(user)
131             movie_popular[movie] += 1
132     print('building movie-users inverse table success.')
```

calculate user-user similarity matrix,Calculate item similarity.

```python
    # calculate user-user similarity matrix
    print('calculate user-user similarity matrix...')
    # record the calculate time has spent.
    usersim_mat_time = LogTime(print_step=1000)
    for user1, related_users in usersim_mat.items():
        len_user1 = len(trainset[user1])
        for user2, count in related_users.items():
            len_user2 = len(trainset[user2])
            # The similarity of user1 and user2 is len(common movies)/sqrt(len(user1 movies)* len(user2 movies)
            usersim_mat[user1][user2] = count / math.sqrt(len_user1 * len_user2)
            # log steps and times.
        usersim_mat_time.count_time()

    print('calculate user-user similarity matrix success.')
    usersim_mat_time.finish()
    return usersim_mat, movie_popular, movie_count


 usage
lef calculate_item_similarity(trainset, use_iuf_similarity=False):
    """
    Calculate item similarity matrix by building movie-users inverse table.
    The calculating will only between items which are voted by common users.

    :param use_iuf_similarity:  This is based on Item IUF similarity.
                                if a person views a lot of movies, items' similarity will be lower.
    :param trainset: trainset
    :return: similarity matrix
    """
    movie_popular, movie_count = calculate_movie_popular(trainset)

    # count co-rated items between users
    print('generate items co-rated similarity matrix...')
    # the keys of item_sim_mat are movie1's id,
    # the values of item_sim_mat are dicts which save {movie2's id: co-occurrence times}.
    # so you can seem item_sim_mat as a two-dim table.
    # TODO DO NOT USE DICT TO SAVE MATRIX, USE LIST INDEED.
    # TODO IF USE LIST, THE MATRIX WILL BE VERY SPARSE.
    movie_sim_mat = {}
```

A similarity matrix is generated to train the self-created model.

```python
# record the calculate time has spent.
movie2users_time = LogTime(print_step=1000)
for user, movies in trainset.items():
    for movie1 in movies:
        # set default similarity between movie1 and other users equals zero
        movie_sim_mat.setdefault(movie1, defaultdict(int))
        for movie2 in movies:
            if movie1 == movie2:
                continue
            # ignore the score they voted.
            # item similarity matrix only focus on co-occurrence.
            if use_iuf_similarity:
                # if a person views a lot of movies, items' similarity will be lower.
                movie_sim_mat[movie1][movie2] += 1 / math.log(1 + len(movies))
            else:
                # origin method, users'similarity based on common items count.
                movie_sim_mat[movie1][movie2] += 1
    # log steps and times.
    movie2users_time.count_time()
print('generate items co-rated similarity matrix success.')
movie2users_time.finish()

# calculate item-item similarity matrix
print('calculate item-item similarity matrix...')
# record the calculate time has spent.
movie_sim_mat_time = LogTime(print_step=1000)
for movie1, related_items in movie_sim_mat.items():
    len_movie1 = movie_popular[movie1]
    for movie2, count in related_items.items():
        len_user2 = movie_popular[movie2]
        # The similarity of user1 and user2 is len(common movies)/sqrt(len(user1 movies)* len(user2 movies)
        movie_sim_mat[movie1][movie2] = count / math.sqrt(len_movie1 * len_user2)
```

Calculate the popularity of a movie.

```
        # log steps and times.
        movie_sim_mat_time.count_time()

    print('calculate item-item similarity matrix success.')
    movie_sim_mat_time.finish()
    return movie_sim_mat, movie_popular, movie_count


1 usage
def calculate_movie_popular(trainset):
    movie_popular = defaultdict(int)
    print('counting movies number and popularity...')

    for user, movies in trainset.items():
        for movie in movies:
            # count item popularity
            movie_popular[movie] += 1
    print('counting movies number and popularity success.')

    # save the total movie number, which will be used in evaluation
    movie_count = len(movie_popular)
    print('total movie number = %d' % movie_count)
    return movie_popular, movie_count

import collections
from operator import itemgetter
import math
from collections import defaultdict

2 usages
class ItemBasedCF:
```

A class of item-based recommendations in which the similarity algorithms described above are combined.

```python
class ItemBasedCF:
    """
    Item-based Collaborative filtering.
    Top-N recommendation.
    """

    def __init__(self, k_sim_movie=20, n_rec_movie=10, use_iuf_similarity=False, save_model=True):
        """
        Init UserBasedCF with n_sim_user and n_rec_movie.
        :return: None
        """
        print("ItemBasedCF start...\n")
        self.k_sim_movie = k_sim_movie
        self.n_rec_movie = n_rec_movie
        self.trainset = None
        self.save_model = save_model
        self.use_iuf_similarity = use_iuf_similarity

    1 usage
    def fit(self, trainset):
        """
        Fit the trainset by calculate movie similarity matrix.
        :param trainset: train dataset
        :return: None
        """
        model_manager = ModelManager()
        try:
            self.movie_sim_mat = model_manager.load_model(
                'movie_sim_mat-iif' if self.use_iuf_similarity else 'movie_sim_mat')
            self.movie_popular = model_manager.load_model('movie_popular')
            self.movie_count = model_manager.load_model('movie_count')
            self.trainset = model_manager.load_model('trainset')
            print('Movie similarity model has saved before.\nLoad model success...\n')
        except OSError:
```

When a member comes in, make recommendations for movie items.

```
299                 print('No model saved before.\nTrain a new model...')
300                 self.movie_sim_mat, self.movie_popular, self.movie_count = \
301                     calculate_item_similarity(trainset=trainset,
302                                               use_iuf_similarity=self.use_iuf_similarity)
303                 self.trainset = trainset
304                 print('Train a new model success.')
305                 if self.save_model:
306                     model_manager.save_model(self.movie_sim_mat,
307                                              'movie_sim_mat-iif' if self.use_iuf_similarity else 'movie_sim_mat')
308                     model_manager.save_model(self.movie_popular, save_name: 'movie_popular')
309                     model_manager.save_model(self.movie_count, save_name: 'movie_count')
310                     model_manager.save_model(self.trainset, save_name: 'trainset')
311                     print('The new model has saved success.\n')
312
         4 usages (2 dynamic)
313     def recommend(self, user):
314         """
315         Find K similar movies and recommend N movies for the user.
316         :param user: The user we recommend movies to.
317         :return: the N best score movies
318         """
319         if not self.movie_sim_mat or not self.n_rec_movie or \
320                 not self.trainset or not self.movie_popular or not self.movie_count:
321             raise NotImplementedError('ItemCF has not init or fit method has not called yet.')
322         K = self.k_sim_movie
323         N = self.n_rec_movie
324         predict_score = collections.defaultdict(int)
325         if user not in self.trainset:
326             print('The user (%s) not in trainset.' % user)
327             return
328         # print('Recommend movies to user start...')
329         watched_movies = self.trainset[user]
```

To test the model, for a new item, the similarity of the training set is calculated, and the recommendation is successful if it exists in the user's favorite list.

```
        for movie, rating in watched_movies.items():
            for related_movie, similarity_factor in sorted(self.movie_sim_mat[movie].items(),
                                                           key=itemgetter(1), reverse=True)[0:K]:
                if related_movie in watched_movies:
                    continue
                # predict the user's "interest" for each movie
                # the predict_score is sum(similarity_factor * rating)
                predict_score[related_movie] += similarity_factor * rating
                # log steps and times.
        # print('Recommend movies to user success.')
        # return the N best score movies
        return [movie for movie, _ in sorted(predict_score.items(), key=itemgetter(1), reverse=True)[0:N]]

    1 usage
    def test(self, testset):
        """
        Test the recommendation system by recommending scores to all users in testset.
        :param testset: test dataset
        :return:
        """
        if not self.n_rec_movie or not self.trainset or not self.movie_popular or not self.movie_count:
            raise ValueError('ItemCF has not init or fit method has not called yet.')
        self.testset = testset
        print('Test recommendation system start...')
        N = self.n_rec_movie
        #  varables for precision and recall
        hit = 0
        rec_count = 0
        test_count = 0
        # varables for coverage
        all_rec_movies = set()
        # varables for popularity
```

Evaluate the model on the test set and calculate the time of the te st run

⚠7 ⚠38 ✔58

```python
    # record the calculate time has spent.
    test_time = LogTime(print_step=1000)
    for i, user in enumerate(self.trainset):
        test_movies = self.testset.get(user, {})
        rec_movies = self.recommend(user)  # type:list
        for movie in rec_movies:
            if movie in test_movies:
                hit += 1
            all_rec_movies.add(movie)
            popular_sum += math.log(1 + self.movie_popular[movie])
            # log steps and times.
        rec_count += N
        test_count += len(test_movies)
        # print time per 500 times.
        test_time.count_time()
    precision = hit / (1.0 * rec_count)
    recall = hit / (1.0 * test_count)
    coverage = len(all_rec_movies) / (1.0 * self.movie_count)
    popularity = popular_sum / (1.0 * rec_count)

    print('Test recommendation system success.')
    test_time.finish()

    print('precision=%.4f\trecall=%.4f\tcoverage=%.4f\tpopularity=%.4f\n' %
          (precision, recall, coverage, popularity))

def predict(self, testset):
    """
    Recommend movies to all users in testset.
    :param testset: test dataset
    :return: `dict` : recommend list for each user.
    """
    movies_recommend = defaultdict(list)
    print('Predict scores start...')
    # record the calculate time has spent.
    predict_time = LogTime(print_step=500)
    for i, user in enumerate(testset):
        rec_movies = self.recommend(user)  # type:list
```

We split the data into a test set and a training set.We use the training set to train the model, and the test set to test the ability of the model.

```
def run_model(model_name, dataset_name, test_size=0.3, clean=False):
    print('\tThis is %s model trained on %s with test_size = %.2f' % (model_name, dataset_name, test_size))
    print('*' * 70 + '\n')
    model_manager = ModelManager(dataset_name, test_size)
    try:
        trainset = model_manager.load_model('trainset')
        testset = model_manager.load_model('testset')
    except OSError:
        dataseter = DataSet("ratings.dat")
        ratings = dataseter.load_dataset()
        trainset, testset = dataseter.train_test_split(ratings, test_size=test_size)
        model_manager.save_model(trainset,  save_name: 'trainset')
        model_manager.save_model(testset,  save_name: 'testset')
    '''Do you want to clean workspace and retrain model again?'''
    '''if you want to change test_size or retrain model, please set clean_workspace True'''
    model_manager.clean_workspace(clean)
    if model_name == 'ItemCF':
        model = ItemBasedCF()
    elif model_name == 'ItemCF-IUF':
        model = ItemBasedCF(use_iuf_similarity=True)
    else:
        raise ValueError('No model named ' + model_name)
    model.fit(trainset)
    recommend_test(model,  user_list: [1, 100, 233, 666, 888])
    model.test(testset)
```

```
1 usage
def recommend_test(model, user_list):
    for user in user_list:
        recommend = model.recommend(str(user))
        print("recommend for userid = %s:" % user)
        print(recommend)
        print()


if __name__ == '__main__':
    main_time = LogTime(words="Main Function")
    dataset_name = 'ml-100k'
    model_type = 'ItemCF-IUF'
    test_size = 0.0001
    run_model(model_type, dataset_name, test_size,  clean: False)
    main_time.finish()
```