A Composite Technique for Creating Contemporary MRS using Association Rule Mining & CF

A

Project Report Submitted

In partial fulfillment of the requirements for the award of the Degree of

BACHELOR OF TECHNOLOGY

In

COMPUTER SCIENCE & ENGINEERING

By

Muppalla Subba Rao	**19761A0540**
Pothireddy Hemalatha Reddy	**19761A0550**
Kadiyala Pavani	**19761A0524**

Under the esteemed guidance of

Mr. Md. Amanatulla

Assistant Professor

DEPARTMENT OF COMPUTER SCIENCE &ENGINEERING

LAKIREDDY BALIREDDY COLLEGE OF ENGINEERING

(AUTONOMOUS)

Accredited by NAAC with 'A' Grade & NBA (Under Tier - I), ISO 9001:2015 Certified
Institution Approved by AICTE, New Delhi and Affiliated to JNTUK, Kakinada
L.B. REDDY NAGAR, MYLAVARAM, NTR DIST., A.P.-521 230.

2019-2023

LAKIREDDY BALI REDDY COLLEGE OF ENGINEERING (AUTONOMOUS)

Accredited by NAAC with 'A' Grade & NBA (Under Tier - I), ISO 9001:2015 Certified
Institution Approved by AICTE, New Delhi and Affiliated to JNTUK, Kakinada

L.B. REDDY NAGAR, MYLAVARAM, NTR DIST., A.P.-521 230.
Department of
COMPUTER SCIENCE & ENGINEERING

CERTIFICATE

This is to certify that the project entitled "**A Composite Technique for Creating Contemporary MRS using Association Rule Mining & CF**" is being submitted by

Muppalla Subba Rao	**19761A0540**
Pothireddy Hemalatha Reddy	**19761A0550**
Kadiyala Pavani	**19761A0524**

in partial fulfillment of the requirements for the award of degree of B. **Tech** in **Computer Science & Engineering** from **Jawaharlal Nehru Technological University Kakinada** is a record of bonafide work carried out by them at **Lakireddy Bali Reddy College of Engineering.**

The results embodied in this Project report have not been submitted to any other University or Institute for the award of any degree or diploma.

PROJECT GUIDE
Mr. Md. Amanatulla

HEAD OF THE DEPARTMENT
Dr. D. Veeraiah

EXTERNAL EXAMINER

ACKNOWLEDGEMENT

We take great pleasure to express our deep sense of gratitude to our project guide **Mr. Md. Amanatulla,** Assistant Professor, for her valuable guidance during the course of our project work.

We would like to thank **Dr. D. Veeraiah**, Professor & Head of the Department of Computer Science & Engineering for his encouragement.

We would like to express our heart-felt thanks to **Dr K. Appa Rao,** Principal, Lakireddy Bali Reddy College of Engineering for providing all the facilities for our project.

Our utmost thanks to all the faculty members and Non-Teaching Staff of the Department of Computer Science & Engineering for their support throughout our project work.

Our Family Members and Friends receive our deepest gratitude and love for their support throughout our academic year.

Muppalla Subba Rao	**19761A0540**
Pothireddy Hemalatha Reddy	**19761A0550**
Kadiyala Pavani	**19761A0524**

DECLARATION

We are here by declaring that the project entitled **"A Composite Technique for Creating Contemporary MRS using Association Rule Mining & CF"** work done by us. We certify that the work contained in the report is original and has been done by us under the guidance of our supervisor. The work has not been submitted to any other institute in preparing for any degree or diploma. We have followed the guidelines provided by the institute in preparing the report. We have confirmed to the norms and guidelines given in the Ethical Code of Conduct of the Institute. Whenever we have used materials (data, theoretical analysis, figures and text) from other sources, we have given due credit to them by citing them in the text of the report and giving their details in the references. Further, we have taken permission from the copyright's owner of the sources, whenever necessary.

Signature(s) of the students(s)

Muppalla Subba Rao	**19761A0540**
Pothireddy Hemalatha Reddy	**19761A0550**
Kadiyala Pavani	**19761A0524**

ABSTRACT

The vast amount of information on the internet has made information available, but it has also made it difficult for users to choose the information that is necessary or interesting to them. To address this issue, recommender systems (RS) were developed to find relevant information using information filtering. Using RS, users may find the appropriate data from a vast collection. There are several types of RS, but those developed using collaborative filtering techniques have proven to be the most effective for a variety of issues. One of the most popular RS accessible is called the Movie Recommendation System (MRS). In this paper, suggestions will be made based on the shared features of user items. Both user objects and item objects are frequent in the movie recommendation system. In order to provide stronger suggestions, this paper integrates the collaborative filtering technique with association rule mining. By integrating collaborative filtering with association rule mining, a hybrid strategy that takes use of both techniques' advantages can boost the recommendation system's performance. Consequently, the recommendations that were generated can be regarded as strong recommendations. Collaborative filtering uses the past behavior of users to make recommendations, while association rule mining looks for patterns in the data to identify items that are frequently bought together. Combining these two approaches can help overcome the limitations of each individual method, such as the need for a large amount of data for collaborative filtering or the lack of personalization in association rule mining. This paper combines data mining and conventional filtering techniques to provide movie recommendation suggestions.

LIST OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

1. MRS: Movie Recommendation System
2. RS: Recommender Systems
3. ML: Machine Learning
4. CF: Collaborative Filtering
5. SVD++: Singular Value Decomposition with implicit feedback
6. ARM: Association Rule Mining
7. KNN: K-Nearest Neighbors
8. RMSE: Root Mean Squared Error
9. MAPE: Mean Absolute Percentage Error
10. SDLC: Software Development Life Cycle
11. XGBoost: Extreme Gradient Boosting
12. BSL: Baseline
13. SU: Similar User
14. SM: Similar Movie
15. OS: Operating System

1. INTRODUCTION

1.1 Overview of The Project

A movie recommendation system is an application that provides personalized movie suggestions to users based on their past behavior and preferences. These systems analyze user data, such as ratings, viewing history, and searches, to generate recommendations that match their tastes. The primary advantage of movie recommendation systems is that they help users save time and effort by suggesting movies that are relevant and interesting to them. With the vast amount of movie options available, it can be challenging to choose what to watch, and a recommendation system can simplify this process. Another advantage of a movie recommendation system is that it can introduce users to new movies that they may not have discovered otherwise. By analyzing the user's viewing history and preferences, the system can suggest movies that they may enjoy but may not have heard of. This can increase the user's overall movie-watching experience and expand their movie knowledge.

Figure 1: Family Watching Movie

The above figure depicts a family sitting together and enjoying a movie, with the parents and children all engrossed in the film on the screen.

Additionally, a movie recommendation system can help increase user engagement and retention. When users receive accurate and personalized recommendations, they are more likely to return to the application to find more movies to watch. This can increase user engagement and keep

users using the application for longer periods. Overall, a movie recommendation system is an excellent tool for both users and businesses. It simplifies the process of finding relevant and interesting movies, introduces users to new movies, and increases user engagement and retention.

The vast amount of information on the internet has made information available, but it has also made it difficult for users to choose the information that is necessary or interesting to them. To address this issue, recommender systems (RS) were developed to find relevant information using information filtering. Using RS, users may find the appropriate data from a vast collection. There are several types of RS, but those developed using collaborative filtering techniques have proven to be the most effective for a variety of issues. One of the most popular RS accessible is called the Movie Recommendation System (MRS). In this project, suggestions will be made based on the shared features of user items. Both user objects and item objects are frequent in the movie recommendation system. In order to provide stronger suggestions, this paper integrates the collaborative filtering technique with association rule mining. By integrating collaborative filtering with association rule mining, a hybrid strategy that takes use of both techniques' advantages can boost the recommendation system's performance. Consequently, the recommendations that were generated can be regarded as strong recommendations. Collaborative filtering uses the past behavior of users to make recommendations, while association rule mining looks for patterns in the data to identify items that are frequently bought together. Combining these two approaches can help overcome the limitations of each individual method, such as the need for a large amount of data for collaborative filtering or the lack of personalization in association rule mining. This project combines data mining and conventional filtering techniques to provide movie recommendation suggestions.

Collaborative filtering involves analyzing past user behavior to identify patterns and trends in their interactions with items. This approach is based on the assumption that users who have exhibited similar behavior in the past will have similar preferences in the future. Collaborative filtering algorithms use this principle to suggest items to users based on the past behavior of other users who have similar preferences. Collaborative filtering can be based on either user-user or item-item similarities.

Association rule mining is a technique used to identify frequent item sets, or combinations of items that are frequently purchased or used together. The idea is that if a user has purchased or used one item, they are likely to be interested in other items that are frequently bought or used in combination with it. Association rule mining algorithms use this principle to make recommendations by identifying and recommending items that are frequently bought or used

together.

Both collaborative filtering and association rule mining have their limitations. Collaborative filtering can suffer from the cold-start problem, where it is difficult to make recommendations for new users or items without sufficient data. Association rule mining can result in recommendations that lack personalization because it relies solely on patterns in the data and does not consider individual user preferences. To overcome these limitations, hybrid approaches that combine collaborative filtering and association rule mining have been developed. These approaches leverage the strengths of each technique while mitigating their weaknesses, resulting in more accurate and personalized recommendations.

1.2 Feasibility Study

In this we study various possibilities where existing and software equipment were sufficient for completing the project. The economic Feasibility determines whether doing the project is economically beneficial. The outcome of the first phase was that the request and various studies were approved and it was decided that the project taken up will serve the end user. On developing and implementation this software saves a lot of amounts and sharing of valuable time.

- Economical Feasibility
- Technical Feasibility
- Social Feasibility

1.2.1 Economical Feasibility

The study confirms the financial viability of a hybrid recommendation approach using collaborative filtering and association rule mining for movies. Despite limited funding, the expected benefits of increased user engagement and loyalty outweigh the costs. The system has potential to generate significant revenue by providing personalized and accurate recommendations to users.

1.2.2 Technical Feasibility

This is carried out to check the technical feasibility that is, the technical requirements of the system. Any system developed must not have a high demand on available technical resources. The developed System must have modest requirements and are required for implementing system. Our project has modest technical requirements.

1.2.3 Social Feasibility

The aspect of study is to check the level of acceptance of the system by the user. This includes the process of training the user to use the system efficiently. The user must not be threatened by the system. His/her level of confidence must be increased so that he/she is able to make some constructive criticism which is welcomed.

1.3 Scope

The main motive behind our project is to develop a robust and accurate recommendation system that helps users find relevant and personalized movie recommendations. With the vast amount of information available on the internet, it has become increasingly difficult for users to navigate and find the information that is relevant to them. Recommender systems (RS) have been developed to address this issue, and collaborative filtering techniques have emerged as the most effective method for building recommendation systems. However, collaborative filtering has its limitations, which can affect the accuracy and personalization of the recommendations. Therefore, in this project, we propose the integration of association rule mining with collaborative filtering to develop a hybrid recommendation system that overcomes the limitations of collaborative filtering and enhances the accuracy and personalization of the recommendations. The proposed hybrid strategy will be implemented in a Movie Recommendation System (MRS) to provide stronger movie recommendations based on shared features of user items. By developing a robust and accurate recommendation system, our project aims to help users find relevant and personalized movie recommendations and improve their overall movie-watching experience.

2. LITERATURE SURVEY

Movie recommendation systems have become an essential tool for many online platforms, and several studies have been conducted by researchers to evaluate and compare various recommendation techniques.

"A Comparative Study of Collaborative Filtering Techniques for Movie Recommendation" by S. Jain and A. Sharma provides an evaluation and comparison of various collaborative filtering techniques for movie recommendation systems. The study includes a comprehensive review of the collaborative filtering techniques, including user-based, item-based, and hybrid approaches. The authors evaluate the performance of these techniques using the MovieLens dataset and various evaluation metrics such as RMSE, MAE, and Precision-Recall. The results show that the item-based approach outperforms the user-based approach in terms of RMSE and MAE, while the hybrid approach provides the best performance in terms of Precision-Recall. The authors also highlight the importance of using appropriate preprocessing techniques and regularization methods to improve the performance of collaborative filtering techniques.

"Movie Recommendation System Using Hybrid Filtering Techniques" by M. Singh and M. Sharma proposes a hybrid recommendation system that combines collaborative filtering and content-based filtering techniques for movie recommendations. The proposed system first uses collaborative filtering to recommend movies based on the user's history and preferences. Then, it incorporates content-based filtering to provide recommendations based on the movie's genre, director, cast, and other attributes. The hybrid approach combines the advantages of both techniques to provide more accurate and diverse recommendations, especially for new or less popular movies.

"Movie Recommendation System Using Sentiment Analysis and Neural Networks" by M. Kumar and N. Singh proposes a movie recommendation system that utilizes sentiment analysis and neural networks to provide more relevant recommendations to users. The proposed system uses sentiment analysis to analyze the user's emotions and feelings towards movies they have watched. The sentiment analysis is performed on the textual reviews or comments provided by the user. The system then uses a neural network model to learn the user's preferences and make personalized movie recommendations. The study demonstrates the potential of sentiment analysis and neural networks in improving the performance of movie recommendation systems. The proposed system can provide more personalized and relevant recommendations to users by considering their emotional responses and preferences towards movies.

2.1 Existing System & Drawbacks

Existing systems for movie recommendations use various techniques to provide suggestions to users based on their preferences, past behavior, or other factors. Collaborative filtering is a commonly used technique that analyzes the behavior of similar users to suggest items that the user might like. Another technique is content-based filtering, which uses item attributes, such as genre or cast, to suggest similar movies to what the user has already watched. Hybrid systems that combine these techniques, such as collaborative filtering with content-based filtering, can provide more accurate and diverse recommendations. However, existing systems may still have limitations, such as data sparsity, overspecialization, or limited novelty, which can affect the quality and personalization of recommendations.

Limitations:

Lack of personalization: Many existing systems use only a user's past behavior to make recommendations, which may not reflect their current preferences or tastes.
Data sparsity (or) Limited data: Collaborative filtering relies on a large amount of data to make accurate recommendations, which may not always be available.
Over-reliance on popularity: Some existing systems make recommendations based solely on the popularity of items, which may not reflect a user's unique preferences.
Limited variety: Many existing systems may suggest similar items repeatedly, which can be frustrating for users who are looking for new and diverse recommendations.
Lack of explanation: Some existing systems may not provide explanations for why a particular recommendation is made, which can lead to confusion or mistrust.

2.2 Proposed System

The proposed model aims to enhance the performance of movie recommendation systems by integrating collaborative filtering and association rule mining techniques. Collaborative filtering is a widely used approach that utilizes the past behavior of users to make personalized recommendations for movies. On the other hand, association rule mining seeks to identify patterns in the data to identify movies that are frequently watched together. By combining these two techniques, the proposed model can overcome their limitations and provide strong recommendations for movies to users.

The model starts by collecting user-movie interaction data, including the user's movie ratings and movie metadata. Collaborative filtering is then used to find similar users and generate

recommendations based on their past behaviours. Association rule mining is applied to the movie metadata to discover frequent item sets and association rules. These association rules can be used to generate recommendations by suggesting movies that are frequently watched together. The proposed model provides a hybrid strategy that combines collaborative filtering and association rule mining to generate strong recommendations for movie recommendation systems. The integration of these techniques can help overcome the limitations of each individual method and provide more accurate and personalized movie recommendations to users.

2.3 Dataset

The MovieLens dataset (ml-20m) is a widely used public dataset that contains 5-star ratings and free-text tagging activity data from the MovieLens movie recommendation service. The dataset consists of 20 million ratings and 465,564 tag applications for 27,278 movies created by 138,493 users between January 09, 1995 and March 31, 2015. The data was generated on October 17, 2016, and is available for download from the GroupLens website.

The data files included in the dataset are genome-scores.csv, genome-tags.csv, links.csv, movies.csv, ratings.csv, and tags.csv. For the objective stated, the ratings.csv and movies.csv files would be used.

The ratings.csv file contains 20 million ratings, each represented by a user ID, a movie ID, a rating, and a timestamp. The ratings range from 0.5 to 5 in increments of 0.5, with half-star ratings allowed. The timestamp represents the time when the user rated the movie, recorded as the number of seconds since midnight Coordinated Universal Time (UTC) of January 1, 1970.

1	userId	movieId	rating	timestamp
2	1	2	3.5	1112486027
3	1	29	3.5	1112484676
4	1	32	3.5	1112484819
5	1	47	3.5	1112484727
6	1	50	3.5	1112484580
7	1	112	3.5	1094785740
8	1	151	4	1094785734
9	1	223	4	1112485573
10	1	253	4	1112484940
11	1	260	4	1112484826

Figure 2: Sample data in ratings.csv

The movies.csv file contains information about the 27,278 movies, each represented by a movie ID, a title, a list of genres, and a list of IMDb IDs. The genres are represented as a pipe-separated string of up to three genres, such as "Action|Comedy|Thriller". The IMDb IDs can be used to link the MovieLens data to the corresponding IMDb pages.

movieId	title	genres
1	Toy Story (1995)	Adventure|Animation|Children|Comedy|Fantasy
2	Jumanji (1995)	Adventure|Children|Fantasy
3	Grumpier Old Men (1995)	Comedy|Romance
4	Waiting to Exhale (1995)	Comedy|Drama|Romance
5	Father of the Bride Part II (1995)	Comedy
6	Heat (1995)	Action|Crime|Thriller
7	Sabrina (1995)	Comedy|Romance
8	Tom and Huck (1995)	Adventure|Children
9	Sudden Death (1995)	Action
10	GoldenEye (1995)	Action|Adventure|Thriller
11	American President, The (1995)	Comedy|Drama|Romance

Figure 3: Sample data in movies.csv

2.4 Machine Learning

Machine learning is a subfield of artificial intelligence that focuses on developing algorithms that can learn from data and make predictions or decisions based on that learning. It involves building models and algorithms that can automatically improve their performance with experience or data inputs, without being explicitly programmed. Machine learning is commonly used in movie recommendation systems to provide personalized movie recommendations to users. Movie recommendation systems use machine learning algorithms to analyze user behavior, such as movies watched, ratings given, and genres favoured, to generate movie suggestions that are tailored to individual preferences.

There are three main types of machine learning: supervised learning, unsupervised learning, and reinforcement learning.

Supervised learning involves training a model on a labelled dataset, where each data point has a corresponding label or outcome variable. The goal is to learn a function that maps inputs to outputs, given the labelled data. This type of learning is commonly used in applications such as image recognition, speech recognition, and natural language processing.

Unsupervised learning, on the other hand, involves training a model on an unlabelled dataset, where there is no target variable to predict. The goal is to discover patterns or structure in the data, such as clusters or groupings of similar data points. This type of learning is commonly used in applications such as anomaly detection, recommendation systems, and data compression.

Reinforcement learning is a type of machine learning where an agent learns to interact with an environment by taking actions and receiving rewards or penalties based on the outcomes of those actions. The goal is to learn a policy or set of rules that maximize the cumulative reward over time. This type of learning is commonly used in applications such as robotics, game playing, and decision making.

Movie recommendation systems are becoming increasingly popular, and there are several types of systems that are used to provide recommendations to users.

2.5 Collaborative Filtering

Collaborative filtering is a popular technique used in recommendation systems, including movie recommendation systems. It is based on the idea that people who have similar preferences in movies will also have similar preferences in other areas. In a collaborative filtering system, the algorithm analyzes the behavior of multiple users and identifies patterns in the movies they watch, rate, and like. The system then recommends new movies to users based on the preferences of other users who have similar viewing habits.

Collaborative filtering can be done in two ways: user-based and item-based.

User-based collaborative filtering: In user-based collaborative filtering, the system identifies users who have similar preferences to the target user and recommends movies that they have enjoyed. The system compares the target user's ratings of different movies with other users' ratings and identifies users with similar patterns. The system then recommends movies that the similar users have enjoyed.

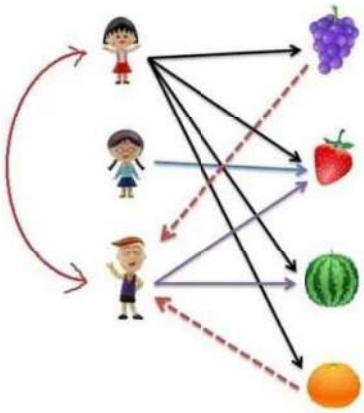

Figure 4: User-based collaborative filtering

Item-based collaborative filtering: In item-based collaborative filtering, the system identifies movies that are similar to the ones the target user has already enjoyed. The system looks at the patterns of ratings given to movies and identifies movies that have been highly rated by users who have also highly rated the movies the target user has watched. The system then recommends the similar movies to the target user.

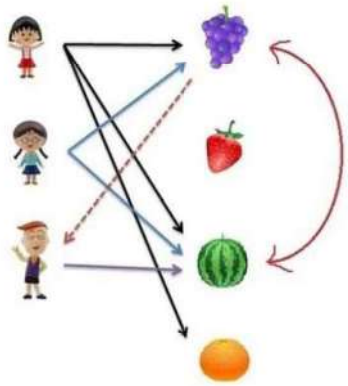

Figure 5: Item-based collaborative filtering

2.6 Association Rule Mining

Association rule mining is a data mining technique that identifies interesting relationships between variables in a large dataset. The goal of association rule mining is to discover patterns or associations that can be used to make predictions or recommendations. The technique is used to uncover relationships between variables in a transactional database, such as a retail store's point-of-sale data. The transactions are typically represented as a set of items, and the goal of association rule mining is to discover which items tend to be purchased together. For example, a retailer might use association rule mining to discover that customers who buy diapers also tend to buy baby wipes.

Association rule mining is often used in market basket analysis, which is a technique used by retailers to identify which products tend to be purchased together. By analyzing this data, retailers can optimize their product placement and marketing strategies to increase sales and revenue.

The process of association rule mining involves three main steps: data preparation, rule generation, and rule evaluation. In the data preparation step, the transactional data is formatted into a binary matrix where each row represents a transaction and each column represents an item. In the rule generation step, the algorithm searches for all possible combinations of items that occur frequently enough to be considered significant. Finally, in the rule evaluation step, the algorithm evaluates the generated rules based on metrics such as support, confidence, and lift, to determine which rules are the most interesting and actionable.

Association rules analysis related algorithms, such as Apriori and FP-growth tree algorithms

that can be defined as a developed techniques that been used for the transactional data type to compute how much association is between data values and their relative combinations, like if a customer is 70% to buy a milk how likely a customer will also buy bread. Here will use two main types of algorithms are being used: The Apriori and the FP Growth, where data are structured as shown in table 1.

TID	*Items*
1	{Bread, Milk}
2	{Bread, Diapers, Beer, Eggs}
3	{Milk, Diapers, Beer, Cola}
4	{Bread, Milk, Diapers, Beer}
5	{Bread, Milk, Diapers, Cola}

Table 1. Transactional data example

2.6.1 Major Computation in Association Rules

Association rules in general have the listed below major computations in the dataset.

Support: Is the percentage of the transaction that contains the items from the data set , which means how many times did the items occur in the dataset (Agrawal et al, 1993).

$$support(A \rightarrow C) = support(A \cup C) \rightarrow (range[0,1])$$

Confidence: The probability that items on left hand side and right-hand side of ruleset are occurring together, with a higher confidence will reflect a higher likelihood of the items that will be purchased together based on the given rule (Agrawal et al, 1993).

$$confidence(A \rightarrow C) = support(A \rightarrow C) support(A) \rightarrow (range\ [0,1])$$

Lift: Probability of the all items set all occurring together in the rule. lift value more than 1 reflects that the presence of the item will increase with the presence of the other items and its occurrence in the same transaction, so the lift summarizes the strength of association rule as a link between the items of both sides (Brin et al, 1997).

$$lift(A \rightarrow C) = confidence(A \rightarrow C) support(C) \rightarrow (range[0, \infty])$$

The left-hand side and right-hand side of the rules are identified also as "antecedents" and "consequents".

2.6.2 Apriori

Apriori algorithm is a classical algorithm for association rule mining, which is a data mining technique used to identify frequent itemsets in a dataset and generate association rules from them. The basic idea behind the Apriori algorithm is that a subset of a frequent itemset must also be frequent. This property is known as the "Apriori property". Based on this property, the algorithm generates frequent itemsets of increasing size by first finding frequent itemsets of size one, and then using these frequent itemsets to generate candidate itemsets of size two. These candidate itemsets are then scanned against the dataset to count their support (i.e., the number of transactions that contain the itemset). The algorithm then prunes the candidate itemsets that are not frequent and uses the remaining frequent itemsets to generate candidate itemsets of size three, and so on, until no more frequent itemsets can be generated.

Apriori algorithm

Algorithm 5.1 Frequent itemset generation of the *Apriori* algorithm.

1: $k = 1$.
2: $F_k = \{ i \mid i \in I \wedge \sigma(\{i\}) \geq N \times minsup\}$. {Find all frequent 1-itemsets}
3: **repeat**
4: $k = k + 1$.
5: C_k = candidate-gen(F_{k-1}). {Generate candidate itemsets.}
6: C_k = candidate-prune(C_k, F_{k-1}). {Prune candidate itemsets.}
7: **for** each transaction $t \in T$ **do**
8: C_t = subset(C_k, t). {Identify all candidates that belong to t.}
9: **for** each candidate itemset $c \in C_t$ **do**
10: $\sigma(c) = \sigma(c) + 1$. {Increment support count.}
11: **end for**
12: **end for**
13: $F_k = \{ c \mid c \in C_k \wedge \sigma(c) \geq N \times minsup\}$. {Extract the frequent k-itemsets.}
14: **until** $F_k = \emptyset$
15: Result = $\bigcup F_k$.

Figure 6: A pseudocode for frequent itemsets generation using Apriori algorithm

where Ck adopts a set of candidate k-itemsets, and the frequent set if itemsets are represented by Fk.

1. The algorithm initially makes a single pass over the data set to determine the support of each item. Upon completion of this step, the set of all frequent 1-itemsets K.

2. Algorithm generate new candidate k-itemsets and prune unnecessary candidates that are guaranteed to be infrequent given the frequent (k−1) itemsets found in the previous iteration.

3. Algorithm makes an additional pass to count the support of the generated candidates. The subset function is used to determine all the candidate itemsets in Ck that are contained in each transaction t.

4. After counting support, the algorithm eliminates all candidate itemsets whose support counts are less than N minimum support which is chosen by the user.

5. The algorithm terminates when there are no new frequent itemsets generated, ie Fk = ∅.

The Apriori algorithm consists of three main steps: support counting, candidate generation, and pruning. The support counting step involves scanning the dataset to count the support of each candidate itemset. The candidate generation step involves generating candidate itemsets of size k+1 from frequent itemsets of size k. This is done by joining pairs of frequent itemsets that have k-1 items in common. The pruning step involves removing candidate itemsets that are not frequent, based on the minimum support threshold specified by the user.

3.SYSTEM ANALYSIS

3.1 Overview of System Analysis

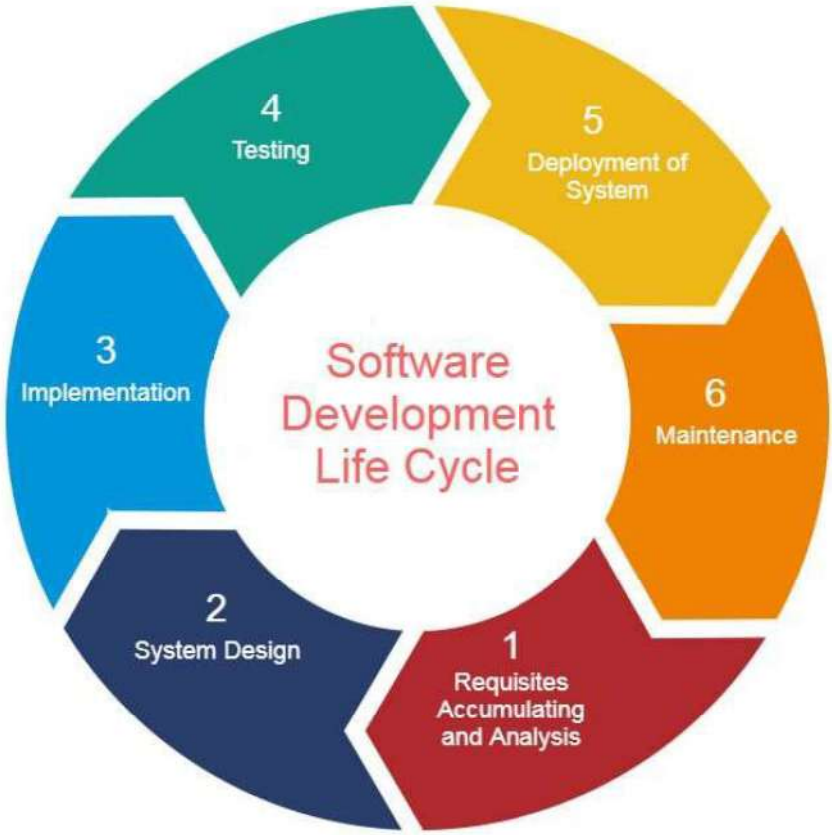

Figure 7: SDLC Architecture

- Requisites Accumulating and Analysis
- System Design
- Implementation
- Testing
- Deployment of system
- Maintenance

3.1.1 Requisites Accumulating and Analysis

The initial stage of any project is crucial, and in the case of an academic leave, it is especially important to gather all the necessary requisites for the project. In order to ensure a comprehensive understanding of the project's topic, we follow the guidelines and resources provided by reputable organizations, such as the IEEE. By using IEEE journals as a reference, we were able to amass numerous IEEE related papers that were relevant to your project. From these papers, we were able to narrow down our research to a final paper that was designated by its setting and substance importance input. In the analysis stage, we took referees from the paper and conducted a literature survey of some papers. This allowed us to further explore the topic and gain a deeper understanding of the subject matter.

3.1.2 System Design

System design involves dividing the design process into three types: GUI designing for creating an intuitive user interface, UML designing for structured development with actor and use case diagrams, and database design tailored to the project's requirements. UML design is crucial to ensure that the project is user-friendly and meets stakeholders' needs, while database design involves analyzing the project's modules to create an efficient data storage schema. Careful planning and execution of system design are critical to meet project requirements and ensure user satisfaction.

3.1.3 Implementation

During the implementation phase, we take the designs and specifications created in the previous stages and begins to turn them into a working product or solution. This is achieved by writing code and developing software components that can be integrated into the final product.
Overall, the implementation phase is a crucial and complex part of the project, requiring a high level of technical expertise and attention to detail. It is where most of the coding work takes place, and it is essential to ensure that the business logic is correctly implemented to ensure the project's success.

3.1.4 Testing

In the Software Development Life Cycle (SDLC), testing plays a critical role in ensuring that the final product is of high quality and meets all the requirements of the stakeholders involved. There are different types of testing that are typically performed throughout the SDLC to ensure that the

software product meets the desired quality standards.

- Unit Testing: This is typically the first level of testing and is performed by developers themselves. It involves testing individual components or modules of the software to ensure that they are functioning as intended.
- Integration Testing: Once the individual components are tested and found to be working correctly, they are combined into a larger system. Integration testing is performed to ensure that these different components work together as intended and that there are no compatibility issues.
- System Testing: This is performed on the entire system once all the components are integrated. It is aimed at verifying that the entire system meets the requirements and specifications that were set forth during the design phase.

3.1.5 Deployment of System

After completing the project, the next step is to deploy the client system into the real world. As this is an academic leave project, the deployment was done only in the college lab with all the necessary software and Windows operating system.

3.1.6 Maintenance

While maintenance is typically an ongoing process for many projects, our specific project is designed to require maintenance as a one-time process only. Once the necessary changes and updates have been made, the project will not require any further maintenance.

3.2 Software Used in The Project

Google Colab is a powerful software tool that is used to run Python code in a project. It is an online platform that provides a Jupyter notebook interface, where users can write, edit, and run Python code in a browser window. One of the primary benefits of using Google Colab to run Python code is that it is entirely free to use. Users can access a virtual machine environment that comes pre-installed with many popular Python libraries, such as NumPy, Matplotlib, Pandas, and TensorFlow, among others. This makes it easy for users to get started with their Python projects without worrying about software installation, configuration, and compatibility issues. Google Colab also provides users with access to a high-end GPU and TPU, which can significantly speed up the computation time required for machine learning and deep learning tasks. This feature is particularly useful for training large models that would take a long time on a standard CPU.

3.3 Libraries & Modules

3.3.1 NumPy

NumPy is a widely-used library in Python that provides support for large, multi-dimensional arrays and matrices, along with a set of mathematical functions to operate on these arrays. NumPy is short for "Numerical Python" and is an essential component of many scientific and data analysis workflows.

3.3.2 Pandas

Pandas is a popular open-source library in Python that provides data manipulation and analysis tools for handling structured data. Pandas is built on top of NumPy and is an essential tool for data scientists, analysts, and researchers who work with tabular data. The primary data structure in Pandas is the Data Frame, which is a 2-dimensional table-like structure that can store heterogeneous data types. A Data Frame is made up of rows and columns, where each column can have a different data type. Pandas also provides Series, which is a 1-dimensional labeled array that can hold any data type.

3.3.3 Matplotlib

Matplotlib is a popular open-source library in Python that provides tools for creating data visualizations such as charts, plots, and graphs. Matplotlib is built on top of NumPy and provides a flexible and easy-to-use interface for generating high-quality visualizations. Matplotlib provides a wide range of visualization tools, including line plots, scatter plots, bar plots, histograms, and many others. Matplotlib also provides tools for customizing the visual appearance of plots, such as changing colors, fonts, and labels.

3.3.4 Seaborn

Seaborn is a popular open-source library in Python that provides tools for creating statistical data visualizations. Seaborn is built on top of Matplotlib and provides a higher-level interface that is more concise and intuitive for creating complex visualizations. Seaborn provides a wide range of visualization tools, including scatter plots, line plots, heatmaps, and many others. Seaborn also provides tools for exploring relationships between variables, such as correlation plots and regression plots. One of the key benefits of Seaborn is its ability to create visually appealing plots with just a few lines of code. Seaborn provides a set of default styles and color palettes that can be easily customized to match your specific needs.

3.3.5 Fuzzywuzzy

Fuzzywuzzy is a Python library that provides tools for string matching and fuzzy string searching. It uses the Levenshtein distance algorithm to calculate the difference between two strings, which can be used to determine their similarity. Fuzzywuzzy is particularly useful when working with data that contains spelling errors, typos, or variations in formatting. It can also be used to match strings that are similar but not identical, such as different spellings of the same name.

3.3.6 Surprise

Surprise is a Python library that provides tools for building and analyzing recommender systems. It offers a wide range of collaborative filtering algorithms for making personalized recommendations based on user preferences and behavior. Collaborative filtering is a technique that is commonly used in recommender systems to predict user ratings for items based on the ratings of similar users.

The Surprise library provides several collaborative filtering algorithms, including:

- User-based Collaborative Filtering: This algorithm looks for users who have similar rating patterns to the target user and recommends items that these similar users have rated highly.
- Item-based Collaborative Filtering: This algorithm looks for items that are like the items that the target user has already rated highly and recommends these similar items.
- Matrix Factorization: This algorithm factorizes the user-item rating matrix into lower-dimensional matrices and uses these matrices to predict missing ratings.

Surprise also provides tools for evaluating the performance of recommender systems using various metrics such as RMSE (Root Mean Squared Error), MAE (Mean Absolute Error), and Precision-Recall.

3.3.7 Sklearn

Scikit-learn, commonly referred to as sklearn, is a popular open-source machine learning library for Python. It provides a wide range of tools for building and evaluating machine learning models, including classification, regression, clustering, and dimensionality reduction. Sklearn is built on top of other popular libraries in Python, such as NumPy, SciPy, and Matplotlib. It provides a consistent and easy-to-use interface for implementing machine learning algorithms, making it a popular choice among data scientists and machine learning practitioners.

3.3.8 Scipy

Scipy is a Python library for scientific computing and technical computing. It provides a wide range of tools for numerical optimization, integration, interpolation, linear algebra, signal processing, and more. Scipy is built on top of NumPy, another popular library for numerical computing in Python.

3.3.9 XGBoost

XGBoost (Extreme Gradient Boosting) is a popular open-source machine learning library designed to provide high accuracy and scalability for building gradient boosting models. It was developed by Tianqi Chen and released in 2016 under the Apache License. XGBoost is a powerful and flexible tool for building gradient boosting models, which are a type of ensemble learning method that combines multiple weak models to make accurate predictions. Gradient boosting models are particularly effective in tasks such as regression, classification, and ranking.

3.3.10 Random

In Python, the random module provides functions for generating pseudo-random numbers. Pseudo-random numbers are not truly random, but they appear random and can be used in simulations, games, and other applications that require random behavior.

3.3.11 Datetime

The datetime module in Python provides classes for working with dates and times. It offers a range of functions and methods for handling dates, times, time intervals, time zones, and more.

The datetime module includes the following main classes:

- datetime: Represents a date and time in a specific format. It includes the year, month, day, hour, minute, second, and microsecond.
- date: Represents a date in the format of year, month, and day.
- time: Represents a time in the format of hour, minute, second, and microsecond.
- timedelta: Represents a duration or time interval.

3.4 SYSTEM REQUIREMENTS

3.4.1 Software Requirements

- Google Colab
- Jupyter Notebook
- NumPy, Pandas, Matplotlib, Seaborn, Fuzzywuzzy, Surprise, Sklearn, Scipy, XGBoost, Random, Datetime.
- Operating System: Windows, Linux.

3.4.2 Hardware Requirements

- Processor: Intel core i5
- RAM: Minimum 8GB
- Hard Disk: Minimum 250GB of local storage

4. SYSTEM DESIGN

4.1 Overview of System Design

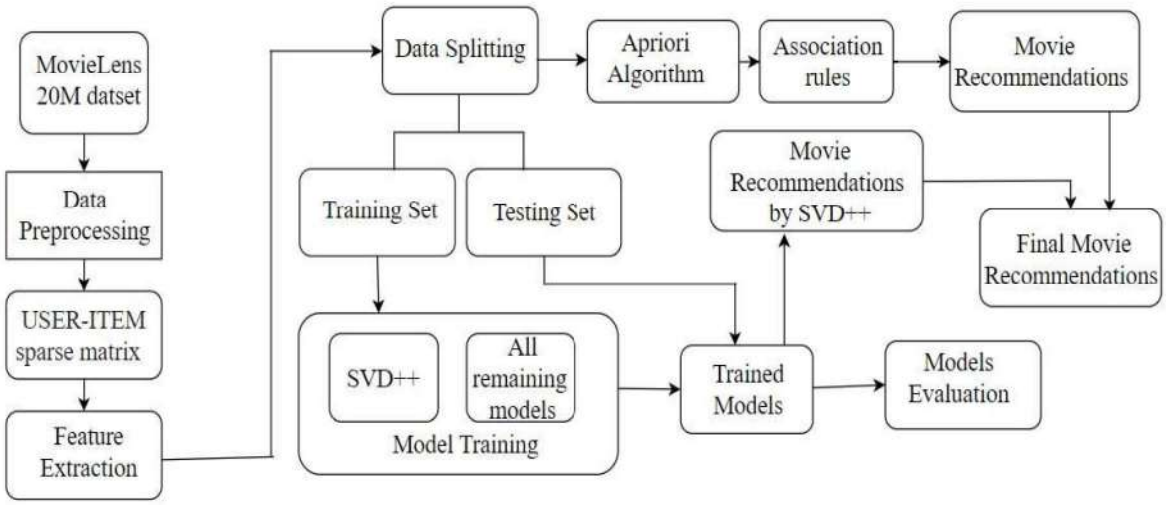

Figure 8: System Design

Firs data is collected from MovieLens website. The MovieLens 20M dataset is a collection of ratings, tags, and metadata for approximately 20 million movie ratings from over 138,000 users. The dataset contains information on movies ranging from older classics to more recent releases. The data in dataset is collected through several methods, including user ratings and movie metadata.

Data preprocessing is a critical step in developing a movie recommendation system. It involves cleaning, transforming, and reducing the dataset to improve the accuracy and effectiveness of the recommendation system. One of the first steps in data preprocessing is data cleaning. This step involves handling missing values and rare values. Missing values are usually represented by null values or NaN values. We will remove missing values from the dataset. Rare values refer to infrequent occurrences in the dataset that may not contribute much to the overall analysis and can be removed or grouped together.

After performing data pre-processing on the MovieLens dataset, the next step is to create a user-item sparse matrix. This matrix is used to represent the user's movie ratings and preferences in a structured format that can be used in the recommendation system. A sparse matrix is a matrix that contains a significant number of zeros, and in the case of a user-item

matrix, most of the entries will be zeros as users have only rated a small fraction of all possible movies. In this matrix, each row represents a user, and each column represents a movie. The non-zero entries in the matrix represent the ratings given by the user to the movie. Creating a sparse matrix involves mapping the cleaned and transformed data to a matrix format. The rows of the matrix represent users, and the columns represent movies. The values in the matrix represent the rating given by a user to a particular movie. Once the user-item sparse matrix is created, it is used in the recommendation system to generate recommendations based on the user's preferences and previous movie ratings.

After creating the user-item sparse matrix, we extract various features that can be used in the recommendation system. Some of these features include:

- User_ID: The ID of the user in the dataset.
- Movie_ID: The ID of the movie in the dataset.
- Global_Average: The global average rating given by all users to all movies in the dataset.
- User_Average: The average rating given by this user to all movies in the dataset.
- Movie_Average: The average rating given by all users to this movie.
- Ratings given to this Movie by top 5 similar users with this User: (SUR1, SUR2, SUR3, SUR4, SUR5).
- Ratings given by this User to top 5 similar movies with this Movie: (SMR1, SMR2, SMR3, SMR4, SMR5).
- Rating: Rating given by this User to this Movie.

After extracting the features, the next step is to split the data into training and testing sets. The training set is used to train the recommendation models, and the testing set is used to evaluate the performance of the models. The data splitting process involves randomly dividing the dataset into two parts: the training set and the testing set. The training set typically contains 80% of the data, while the testing set contains the remaining 20%.

Once the data is split, various recommendation models are trained on the training set using the extracted features. The performance of these models can be evaluated using various metrics such as root mean squared error (RMSE) and mean absolute percentage error (MAPE). Finally, the trained models are tested on the test set to assess their performance in predicting the movie ratings for the users. The test set contains a set of ratings that were not used during the training process, and the models' performance on this set gives an indication

of how well they can generalize to new data.

Next, movie recommendations are made by using SVD++ (a variant of Singular Value Decomposition). SVD++ decomposes the user-movie ratings matrix into latent features of users and movies. The latent features capture hidden patterns in user preferences and movie characteristics. By comparing user and movie latent features, recommendations can be made for movies a user might like.

Next, movie recommendations are made by using association rules generated from Apriori on the data. Apriori is an algorithm for mining frequent itemsets and generating association rules. It finds items (movies) that frequently co-occur in the data (are rated together by users). From the frequent itemsets, association rules of the form "if a user watches X movies, they will also likely watch Y movie" are generated. These rules capture relationships between movies in the data. The movie recommendations are generated based on association rules involving movies a user has already watched.

The movie recommendations from the Apriori association rules mining and SVD++ combined to generate final movie recommendations. Combining recommendations from multiple algorithms can give better results than using a single algorithm alone. Here, the association rules capture co-occurrence patterns between movies, while the latent features capture user preferences and movie characteristics. By aggregating recommendations from the two approaches, the final recommendations can leverage the strengths of both and may be more accurate. The combined recommendations are presented to the user as a personalized list of movies they may enjoy watching.

4.2 Evaluation Metrics

4.2.1 Root Mean Square Error (RMSE)

RMSE measures the difference between the predicted ratings and the actual ratings. It is calculated by taking the square root of the average of the squared differences between the predicted and actual ratings:

$$RMSE = sqrt(1/N * sum((predicted - actual)\text{^}2))$$

where N is the total number of ratings.

RMSE gives an idea of how much the predicted ratings deviate from the actual ratings. The

lower the RMSE, the better the model's performance.

4.2.1 Mean Absolute Percentage Error (MAPE)

MAPE measures the percentage difference between the predicted ratings and the actual ratings. It is calculated by taking the absolute difference between the predicted and actual ratings, dividing it by the actual rating, and then taking the average of these values:

$$MAPE = 1/N * sum(|(predicted - actual)/actual|) * 100\%$$

where N is the total number of ratings.

MAPE gives an idea of how much the predicted ratings deviate from the actual ratings in percentage terms. The lower the MAPE, the better the model's performance.

5.CODING & IMPLEMENTATION

```
# Connecting to Google drive
from google.colab import drive
drive.mount('/content/drive')
#Installing the necessary libraries
!pip install fuzzywuzzy
!pip install scikit-surprise
# Importing the necessary libraries
from scipy import sparse
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.metrics import mean_squared_error
import xgboost as xgb
from surprise import Reader, Dataset
from surprise import BaselineOnly
from surprise import KNNBaseline
from surprise import SlopeOne
from surprise import SVD
from surprise import SVDpp
from surprise.model_selection import GridSearchCV
from datetime import datetime
from fuzzywuzzy import fuzz
from fuzzywuzzy import process
import numpy as np
import pandas as pd
import matplotlib
import matplotlib.pyplot as plt
import seaborn as sns
import os
import random
import gc
import warnings
```

```
warnings.filterwarnings("ignore")
# Setting up some parameters for the workbook
pd.set_option('display.max_rows', 500)
pd.options.display.max_columns = None
%matplotlib inline
matplotlib.rcParams["figure.figsize"] = (25,5)
# Loading the dataset
file_path = "/content/drive/MyDrive/Colab Datasets"
movie_ratings = pd.read_csv(file_path + "/ratings.csv")
movies = pd.read_csv(file_path + "/movies.csv")
# Creating a newId for every movie to reduce the range of existing
movieId
movies["newId"] = range(1, movies["movieId"].nunique()+1)
# Converting the the UTC timestamp to Datetime
movie_ratings["timestamp"] = movie_ratings["timestamp"].apply(lambda x: datetime.utcfro
mtimestamp(x).strftime("%Y-%m-%d"))
# Merging the movies and ratings data files
movie_ratings = movie_ratings.merge(movies, how="left", on="movieId")
# Renaming the timestamp to date
movie_ratings.rename(columns={"timestamp": "date"}, inplace=True)
# Updating the movieId with the newId
movie_ratings["movieId"] = movie_ratings["newId"]
movies["movieId"] = movies["newId"]
# Dropping the newId from the datasets
movie_ratings.drop(["newId"], axis=1, inplace=True)
movies.drop(["newId"], axis=1, inplace=True)
# Sorting ratings based on date
movie_ratings.sort_values(by = "date", inplace = True)
movie_ratings.reset_index(drop=True, inplace=True)
# Checking the features and no. of records in the dataset
print("The number of records are : ", movie_ratings.shape[0])
print("The number of features are : ", movie_ratings.shape[1])
print("The list of features is : ", movie_ratings.columns)
```

```
movie_ratings.head()
# Checking for duplicates
print("No. of duplicates records in the dataset : ", movie_ratings.columns.duplicated().sum())
# Checking the columns' titles and datatypes
movie_ratings.info()
# Checking the number of missing values in data
movie_ratings.isna().sum()
# Checking the feature "userID"
total_users = len(np.unique(movie_ratings["userId"]))
print("The count of unique userID in the dataset is : ", total_users)
print("The top 5 userID in the dataset are : \n", movie_ratings["userId"].value_counts()[:5])
# Checking the feature "movieID"
total_movies = len(np.unique(movie_ratings["movieId"]))
print("The count of unique movieID in the dataset is : ", total_movies)
print("The top 5 movieID in the dataset are : \n", movie_ratings["movieId"].value_counts()[:5
])
# Helper function to Change the numeric label in terms of Millions
def changingLabels(number):
    return str(number/10**6) + "M"
# Set plot style
sns.set(style="darkgrid")
# Create figure and axes objects
fig, axes = plt.subplots(figsize=(25, 5))
# Create countplot
sns.countplot(x="rating", data=movie_ratings, ax=axes)
# Set ytick labels
ytick_labels = [changingLabels(num) for num in axes.get_yticks()]
axes.set_yticklabels(ytick_labels)
# Annotate bars with counts
for bar in axes.containers:
    axes.bar_label(bar, label_type="edge")
# Set tick label font size
plt.tick_params(labelsize=15)
```

```
# Set title and axis labels
plt.title("Distribution of Ratings in the dataset", fontsize=20)
plt.xlabel("Ratings", fontsize=10)
plt.ylabel("Counts (in Millions)", fontsize=10)
# Display plot
plt.show()
# Checking the feature "date"
print("The count of unique date in the dataset is : ", movie_ratings["date"].nunique())
print("The first rating was given on : ", movie_ratings["date"].min())
print("The latest rating was given on : ", movie_ratings["date"].max())
print("The top 5 date in the dataset are : \n", movie_ratings["date"].value_counts()[:5])
# Checking the feature "title"
movie_list = movie_ratings["title"].unique()
print("The count of unique title in the dataset is : ", movie_ratings["title"].nunique())
print("The top 5 title in the dataset are : \n", movie_ratings["title"].value_counts()[:5])
# Extract unique Genres along with their count
unique_genres = {}
def ExtractGenres(x):
    for g in x.split("|"):
        if g not in unique_genres.keys():
            unique_genres[g] = 1
        else:
            unique_genres[g] = unique_genres[g] + 1
movie_ratings["genres"].apply(ExtractGenres)
print("Genres Extracted from the dataset.")
# Visualizing the feature "Genres"
genres_df = pd.DataFrame(list(unique_genres.items()))
genres_df.columns = ["Genre", "Count"]
sns.set(style="darkgrid")
fig, axes = plt.subplots(1, 1, figsize=(25, 8), sharey=True)
sns.barplot(y="Count", x="Genre", data=genres_df, ax=axes)
axes.set_yticklabels([changingLabels(num) for num in axes.get_yticks()])
for p in axes.patches:
```

```
    axes.annotate('{}'.format(int(p.get_height())), (p.get_x(), p.get_height()+100))
plt.tick_params(labelsize = 15)
plt.title("Distribution of Genres in the dataset", fontsize = 20)
plt.xlabel("Genres", fontsize = 15)
plt.xticks(rotation=60, fontsize=10)
plt.yticks(fontsize=10)
plt.ylabel("Counts (in Millions)", fontsize = 15)
plt.show()
# Creating the train test set
file_path = "/content/drive/MyDrive/Colab Datasets"
if not os.path.isfile(file_path + "/TrainData.pkl"):
    print("Creating Train Data and saving it..")
    movie_ratings.iloc[:int(movie_ratings.shape[0] * 0.80)].to_pickle(file_path +
"/TrainData.pkl")
    Train_Data = pd.read_pickle(file_path + "/TrainData.pkl")
    Train_Data.reset_index(drop = True, inplace = True)
else:
    print("Loading Train Data..")
    Train_Data = pd.read_pickle(file_path + "/TrainData.pkl")
    Train_Data.reset_index(drop = True, inplace = True)
if not os.path.isfile(file_path + "/TestData.pkl"):
    print("Creating Test Data and saving it..")
    movie_ratings.iloc[int(movie_ratings.shape[0] * 0.80):].to_pickle(file_path +
"/TestData.pkl")
    Test_Data = pd.read_pickle(file_path + "/TestData.pkl")
    Test_Data.reset_index(drop = True, inplace = True)
else:
    print("Loading Test Data..")
    Test_Data = pd.read_pickle(file_path + "/TestData.pkl")
  Test_Data.reset_index(drop = True, inplace = True)
# Creating list of unique movies from Train Set
movie_list_in_training = Train_Data.drop_duplicates(subset=["title"],
keep="first")[["movieId", "title", "genres"]]
```

```
movie_list_in_training = movie_list_in_training.reset_index(drop=True)
movie_list_in_training.head()
# Checking the basic statistics for the training data
print("Total Train Data..")
print("Total number of movie ratings in train data : ", str(Train_Data.shape[0]))
print("Number of unique users in train data : ", str(len(np.unique(Train_Data["userId"]))))
print("Number of unique movies in train data : ",
str(len(np.unique(Train_Data["movieId"]))))
# Checking basic statistics for "rating"
print("The basic statistics for the feature is : \n", Train_Data["rating"].describe())
# Set plot style
sns.set(style="darkgrid")
# Create figure and axes objects
fig, axes = plt.subplots(figsize=(25, 5))
# Create countplot
sns.countplot(x="rating", data=Train_Data, ax=axes)
# Set ytick labels
ytick_labels = [changingLabels(num) for num in axes.get_yticks()]
axes.set_yticklabels(ytick_labels)
# Annotate bars with counts
for p in axes.patches:
   axes.annotate('{}'.format(p.get_height()), (p.get_x()+0.2, p.get_height()+100))
# Set tick label font size
plt.tick_params(labelsize=15)
# Set title and axis labels
plt.title("Distribution of Ratings in the dataset", fontsize=20)
plt.xlabel("Ratings", fontsize=10)
plt.ylabel("Counts (in Millions)", fontsize=10)
# Display plot
plt.show()
# Extracting the day of week from the date when rating was provided
Train_Data["date"] = pd.to_datetime(Train_Data["date"], errors='coerce')
Train_Data["DayOfWeek"] = Train_Data["date"].dt.strftime('%A')
```

```
Train_Data["Weekday"] = Train_Data["date"].apply(lambda x : 1 if x.dayofweek > 5 else 0)
# Converting the number into 'Ks.
def ChangingLabelsInK(number):
    return str(int(number/10**3)) + "K"
# Visualizing the count of total ratings made per month
sns.set(style="darkgrid")
fig, axes = plt.subplots(1, 1, figsize=(25, 5), sharey=True)
axes = Train_Data.resample("m", on = "date")["rating"].count().plot()
axes.set_yticklabels([ChangingLabelsInK(num) for num in axes.get_yticks()])
axes.set_title("Count of Total Ratings per Month", fontsize = 20)
axes.set_xlabel("Date", fontsize = 15)
axes.set_ylabel("Number of Ratings (in Millions)", fontsize = 15)
plt.tick_params(labelsize = 15)
plt.show()
# Visualizing the count of ratings by weekday
sns.set(style="darkgrid")
fig, axes = plt.subplots(1, 1, figsize=(25, 5), sharey=True)
sns.barplot(x="Weekday", y="rating" , data=Train_Data.groupby(by=["Weekday"],
as_index=False)["rating"].count(), ax=axes)
for p in axes.patches:
    axes.annotate('{}'.format(int(p.get_height())), (p.get_x(), p.get_height()+100))
axes.set_yticklabels([changingLabels(num) for num in axes.get_yticks()])
plt.tick_params(labelsize = 15)
plt.title("Distribution of number of ratings by Weekday", fontsize = 20)
plt.xlabel("Weekday", fontsize = 15)
plt.xticks(fontsize=10)
plt.yticks(fontsize=10)
plt.ylabel("Counts (in Millions)", fontsize = 15)
plt.show()
# Visualizing the count of ratings by individual days of the week
sns.set(style="darkgrid")
fig, axes = plt.subplots(1, 1, figsize=(25, 5), sharey=True)
sns.barplot(x="DayOfWeek", y="rating" , data=Train_Data.groupby(by=["DayOfWeek"],
```

```
as_index=False)["rating"].count(), ax=axes)
for p in axes.patches:
    axes.annotate('{}'.format(int(p.get_height())), (p.get_x(), p.get_height()+100))
axes.set_yticklabels([changingLabels(num) for num in axes.get_yticks()])
plt.tick_params(labelsize = 15)
plt.title("Distribution of number of ratings by individual days", fontsize = 20)
plt.xlabel("Days", fontsize = 15)
plt.xticks(fontsize=10)
plt.yticks(fontsize=10)
plt.ylabel("Counts (in Millions)", fontsize = 15)
plt.show()
# Calculating the number of ratings given by individual users
no_of_rated_movies_per_user = Train_Data.groupby(by=["userId"],
as_index=False)["rating"].count().sort_values(by="rating", ascending=False)
no_of_rated_movies_per_user.reset_index(drop=True, inplace=True)
# Visualizing the count of ratings by individual users
sns.set(style="darkgrid")
fig, axes = plt.subplots(1, 1, figsize=(25, 5), sharey=True)
sns.barplot(x="userId", y="rating" , data=no_of_rated_movies_per_user[:15], ax=axes)
for p in axes.patches:
    axes.annotate('{}'.format(int(p.get_height())), (p.get_x(), p.get_height()+100))
axes.set_yticklabels([ChangingLabelsInK(num) for num in axes.get_yticks()])
plt.tick_params(labelsize = 15)
plt.title("Number of ratings for Top 15 Users", fontsize = 20)
plt.xlabel("UserID", fontsize = 15)
plt.xticks(fontsize=10)
plt.yticks(fontsize=10)
plt.ylabel("Counts", fontsize = 15)
plt.show()
# Checking the basic statistics for the number of ratings per user
print("Information about no. of ratings by users : \n",
no_of_rated_movies_per_user["rating"].describe())
```

```
# Calculating average ratings given by individual users
avg_ratings_per_user = Train_Data.groupby(by = ["userId"],
as_index=False)["rating"].mean()
avg_ratings_per_user = avg_ratings_per_user.reset_index(drop=True)
avg_ratings_per_user =
avg_ratings_per_user.merge(no_of_rated_movies_per_user[["userId", "rating"]], how="left",
on="userId")
avg_ratings_per_user.rename(columns={"rating_x":"avg_rating", "rating_y":
"num_of_rating"}, inplace=True)
avg_ratings_per_user = avg_ratings_per_user.sort_values("num_of_rating",
ascending=False)
# Visualizing the average ratings by individual Days of the Week
sns.set(style="darkgrid")
fig, axes = plt.subplots(1, 1, figsize=(25, 5), sharey=True)
sns.barplot(x="userId", y="avg_rating", data=avg_ratings_per_user[:15], ax=axes)
for p in axes.patches:
   axes.annotate('{}'.format(round(p.get_height(), 2)), (p.get_x()+0.3, p.get_height()))
plt.tick_params(labelsize = 15)
plt.title("Average Ratings by top 15 Users", fontsize = 20)
plt.xlabel("User ID", fontsize = 15)
plt.xticks(fontsize=10)
plt.ylabel("Rating", fontsize = 15)
plt.yticks(fontsize=10)
plt.show()
# Calculating count of ratings received for movies
no_of_ratings_per_movie = Train_Data.groupby(by = ["movieId", "title"],
as_index=False)["rating"].count().sort_values(by=["rating"], ascending = False)
no_of_ratings_per_movie = no_of_ratings_per_movie.reset_index(drop=True)
# Visualizing top 5 movies heavily rated movies.
sns.set(style="darkgrid")
fig, axes = plt.subplots(1, 1, figsize=(25, 5), sharey=True)
sns.barplot(x="title", y="rating", data=no_of_ratings_per_movie[:15], ax=axes)
for p in axes.patches:
```

```
    axes.annotate('{}'.format(int(p.get_height())), (p.get_x(), p.get_height()+100))
axes.set_yticklabels([ChangingLabelsInK(num) for num in axes.get_yticks()])
plt.tick_params(labelsize = 15)
plt.title("Number of ratings for Top 15 Movies", fontsize = 20)
plt.xlabel("Movie", fontsize = 15)
plt.xticks(rotation=70, fontsize=10)
plt.ylabel("Counts", fontsize = 15)
plt.yticks(fontsize=10)
plt.show()
# Calculating average ratings for movies
avg_ratings_per_movie = Train_Data.groupby(by = ["movieId", "title"],
as_index=False)["rating"].mean()
avg_ratings_per_movie = avg_ratings_per_movie.reset_index(drop=True)
avg_ratings_per_movie =
avg_ratings_per_movie.merge(no_of_ratings_per_movie[["movieId", "rating"]], how="left",
on="movieId")
avg_ratings_per_movie.rename(columns={"rating_x":"avg_rating", "rating_y":
"num_of_rating"}, inplace=True)
avg_ratings_per_movie = avg_ratings_per_movie.sort_values("num_of_rating",
ascending=False)
# Visualizing the average ratings by individual Days of the Week
sns.set(style="darkgrid")
fig, axes = plt.subplots(1, 1, figsize=(25, 5), sharey=True)
sns.barplot(x="title", y="avg_rating", data=avg_ratings_per_movie[:15], ax=axes)
for p in axes.patches:
    axes.annotate('{}'.format(round(p.get_height(), 2)), (p.get_x()+0.3, p.get_height()))
plt.tick_params(labelsize = 15)
plt.title("Average Ratings For top 15 movies", fontsize = 20)
plt.xlabel("Top 15 Movie", fontsize = 15)
plt.xticks(rotation=70, fontsize=10)
plt.ylabel("Rating", fontsize = 15)
plt.yticks(fontsize=10)
plt.show()
```

```
# Path for loading/saving files
file_path = "/content/drive/MyDrive/Colab Datasets"
# Creating/loading user-movie sparse matrix for train data
startTime = datetime.now()
print("Creating USER_ITEM sparse matrix for train Data..")
if os.path.isfile(file_path + "/TrainUISparseData.npz"):
    print("Sparse Data is already present in your disk, no need to create further. Loading
Sparse Matrix")
    TrainUISparseData = sparse.load_npz(file_path + "/TrainUISparseData.npz")
    print("Shape of Train Sparse matrix = "+str(TrainUISparseData.shape))
else:
    print("We are creating sparse data..")
    TrainUISparseData = sparse.csr_matrix((Train_Data.rating, (Train_Data.userId,
Train_Data.movieId)))
    print("Creation done. Shape of sparse matrix : ", str(TrainUISparseData.shape))
    print("Saving it into disk for furthur usage.")
    sparse.save_npz(file_path + "/TrainUISparseData.npz", TrainUISparseData)
    print("Done\n")
print("Time taken : ", datetime.now() - startTime)
rows,cols = TrainUISparseData.shape
presentElements = TrainUISparseData.count_nonzero()
print("Sparsity Of Train matrix : {}% ".format((1-(presentElements/(rows*cols)))*100))
# Creating/loading user-movie sparse matrix for test data
startTime = datetime.now()
print("Creating USER_ITEM sparse matrix for test Data..")
if os.path.isfile(file_path + "/TestUISparseData.npz"):
    print("Sparse Data is already present in your disk, no need to create further. Loading
Sparse Matrix")
    TestUISparseData = sparse.load_npz(file_path + "/TestUISparseData.npz")
    print("Shape of Test Sparse Matrix : ", str(TestUISparseData.shape))
else:
    print("We are creating sparse data..")
```

```
    TestUISparseData = sparse.csr_matrix((Test_Data.rating, (Test_Data.userId,
Test_Data.movieId)))
    print("Creation done. Shape of sparse matrix : ", str(TestUISparseData.shape))
    print("Saving it into disk for furthur usage.")
    sparse.save_npz(file_path + "/TestUISparseData.npz", TestUISparseData)
    print("Done\n")
print("Time Taken : ", datetime.now() - startTime)
rows,cols = TestUISparseData.shape
presentElements = TestUISparseData.count_nonzero()
print("Sparsity Of Test matrix : {}% ".format((1-(presentElements/(rows*cols)))*100))
# Function to Calculate Average rating for users or movies from User-movie sparse matrix
def getAverageRatings(sparseMatrix, if_user):
    #axis = 1 means rows and axis = 0 means columns
    ax = 1 if if_user else 0
    sumOfRatings = sparseMatrix.sum(axis = ax).A1
    noOfRatings = (sparseMatrix!=0).sum(axis = ax).A1
    rows, cols = sparseMatrix.shape
    averageRatings = {i: sumOfRatings[i]/noOfRatings[i] for i in range(rows if if_user else
cols) if noOfRatings[i]!=0}
    return averageRatings
AvgRatingUser = getAverageRatings(TrainUISparseData, True)
AvgRatingMovie = getAverageRatings(TrainUISparseData, False)
train_users = len(AvgRatingUser)
uncommonUsers = total_users - train_users
print("Total no. of Users : ", total_users)
print("No. of Users in Train data : ", train_users)
print("No. of Users not present in Train data : {}({}%)".format(uncommonUsers,
np.round((uncommonUsers/total_users)*100), 2))
train_movies = len(AvgRatingMovie)
uncommonMovies = total_movies - train_movies
print("Total no. of Movies : ", total_movies)
print("No. of Movies in Train data : ", train_movies)
print("No. of Movies not present in Train data = {}({}%)".format(uncommonMovies,
```

```
np.round((uncommonMovies/total_movies)*100), 2))
# Computing user-user similarity matrix for the train data
# We have 138K sized sparse vectors using which a 14K x 14K movie similarity matrix
would be calculated
start = datetime.now()
if not os.path.isfile(file_path + "/m_m_similarity.npz"):
    print("Movie-Movie Similarity file does not exist in your disk. Creating Movie-Movie
Similarity Matrix...")
    m_m_similarity = cosine_similarity(TrainUISparseData.T, dense_output = False)
    print("Dimension of Matrix : ", m_m_similarity.shape)
    print("Storing the Movie Similarity matrix on disk for further usage")
    sparse.save_npz(file_path + "/m_m_similarity.npz", m_m_similarity)
else:
    print("File exists in the disk. Loading the file...")
    m_m_similarity = sparse.load_npz(file_path + "/m_m_similarity.npz")
    print("Dimension of Matrix : ", m_m_similarity.shape)
print("The time taken to compute movie-movie similarity matrix is : ", datetime.now() - start)
# Creating a function to take Movie Name and generate the top matched name and generate
its N similar movies based on M-M Similary
def GetSimilarMoviesUsingMovieMovieSimilarity(movie_name, num_of_similar_movies):
    matches = process.extract(movie_name, movie_list_in_training["title"],
scorer=fuzz.partial_ratio)
    if len(matches) == 0:
        return "No Match Found"
    movie_id = movie_list_in_training.iloc[matches[0][2]]["movieId"]
    similar_movie_id_list = np.argsort(-
m_m_similarity[movie_id].toarray().ravel())[0:num_of_similar_movies+1]
    sm_df =
movie_list_in_training[movie_list_in_training["movieId"].isin(similar_movie_id_list)]
    sm_df["order"] = sm_df.apply(lambda x: list(similar_movie_id_list).index(x["movieId"]),
axis=1)
    return sm_df.sort_values("order")
```

```
# Picking random movie and checking it's top 10 most similar movies
GetSimilarMoviesUsingMovieMovieSimilarity("Superman", 10)
# Getting highest uder id
row_index, col_index = TrainUISparseData.nonzero()
unique_user_id = np.unique(row_index)
print("Max User id is :", np.max(unique_user_id))
# Here, we are calculating user-user similarity matrix only for first 100 users in our sparse
matrix. And we are calculating
# Top 100 most similar users with them.
def getUser_UserSimilarity(sparseMatrix, top = 100):
    startTimestamp20 = datetime.now()
    row_index, col_index = sparseMatrix.nonzero()
    rows = np.unique(row_index)
    similarMatrix = np.zeros(13849300).reshape(138493,100)    # 138493*100 = 13849300.
As we are building similarity matrix only
    #for top 100 most similar users.
    timeTaken = []
    howManyDone = 0
    for row in rows[:top]:
        howManyDone += 1
        startTimestamp = datetime.now().timestamp()  #it will give seconds elapsed
        sim = cosine_similarity(sparseMatrix.getrow(row), sparseMatrix).ravel()
        top100_similar_indices = sim.argsort()[-top:]
        top100_similar = sim[top100_similar_indices]
        similarMatrix[row] = top100_similar
        timeforOne = datetime.now().timestamp() - startTimestamp
        timeTaken.append(timeforOne)
        if howManyDone % 20 == 0:
            print("Time elapsed for {} users = {}sec".format(howManyDone, (datetime.now() -
startTimestamp20)))
    print("Average Time taken to compute similarity matrix for 1 user =
"+str(sum(timeTaken)/len(timeTaken))+"seconds")
```

```
    sns.set(style="darkgrid")
    fig = plt.figure(figsize = (25, 5))
    plt.plot(timeTaken, label = 'Time Taken For Each User')
    plt.plot(np.cumsum(timeTaken), label='Cumulative Time')
    plt.legend(loc='upper left', fontsize = 15)
    plt.xlabel('Users', fontsize = 20)
    plt.ylabel('Time(Seconds)', fontsize = 20)
    plt.tick_params(labelsize = 15)
    plt.show()
    return similarMatrix
simMatrix = getUser_UserSimilarity(TrainUISparseData, 100)
# Calculating user-user similarity only for particular users in our sparse matrix and return
user_ids
def Calculate_User_User_Similarity(sparseMatrix, user_id, num_of_similar_users=10):
    if user_id in unique_user_id:
        # Calculating the cosine similarity for user_id with all the "userId"
        sim = cosine_similarity(sparseMatrix.getrow(user_id), sparseMatrix).ravel()
        # Sorting the indexs(user_id) based on the similarity score for all the user ids
        top_similar_user_ids = sim.argsort()[::-1]
        # Sorted the similarity values
        top_similarity_values = sim[top_similar_user_ids]
    return top_similar_user_ids[1: num_of_similar_users+1]
# Path for saving/loading files
file_path = "/content/drive/MyDrive/Colab Datasets"
# Since the given dataset might not completely fit into computaton capacity that we have, we
will sample the data and work it
# Function for Sampling random movies and users to reduce the size of rating matrix
def get_sample_sparse_matrix(sparseMatrix, n_users, n_movies, matrix_name):
    np.random.seed(15)   #this will give same random number everytime, without replacement
    startTime = datetime.now()
    users, movies, ratings = sparse.find(sparseMatrix)
    uniq_users = np.unique(users)
    uniq_movies = np.unique(movies)
```

```
    userS = np.random.choice(uniq_users, n_users, replace = False)
    movieS = np.random.choice(uniq_movies, n_movies, replace = False)
    mask = np.logical_and(np.isin(users, userS), np.isin(movies, movieS))
    sparse_sample = sparse.csr_matrix((ratings[mask], (users[mask], movies[mask])), shape =
(max(userS)+1, max(movieS)+1))
    print("Sparse Matrix creation done. Saving it for later use.")
    sparse.save_npz(file_path + "/" + matrix_name, sparse_sample)
    print("Shape of Sparse Sampled Matrix = " + str(sparse_sample.shape))
    print("Time taken : ", datetime.now() - startTime)
    return sparse_sample
# Creating Sample Sparse Matrix for Train Data
if not os.path.isfile(file_path + "/TrainUISparseData_Sample.npz"):
    print("Sample sparse matrix is not present in the disk. We are creating it...")
    train_sample_sparse = get_sample_sparse_matrix(TrainUISparseData, 5000, 1000,
"TrainUISparseData_Sample.npz")
else:
    print("File is already present in the disk. Loading the file...")
    train_sample_sparse = sparse.load_npz(file_path + "/TrainUISparseData_Sample.npz")
    print("Shape of Train Sample Sparse Matrix = " + str(train_sample_sparse.shape))
# Creating Sample Sparse Matrix for Test Data
if not os.path.isfile(file_path + "/TestUISparseData_Sample.npz"):
    print("Sample sparse matrix is not present in the disk. We are creating it...")
    test_sample_sparse = get_sample_sparse_matrix(TestUISparseData, 2000, 200,
"TestUISparseData_Sample.npz")
else:
    print("File is already present in the disk. Loading the file...")
    test_sample_sparse = sparse.load_npz(file_path + "/TestUISparseData_Sample.npz"
    print("Shape of Test Sample Sparse Matrix = " + str(test_sample_sparse.shape))
# Checking the shape of Training and test data
print("Shape of Train Sparse Matrix : ", train_sample_sparse.shape)
print("Shape of Test Sparse Matrix : ", test_sample_sparse.shape)
# Calculating few GlobalAverageRating, AvgMovieRating, AvgUserRating and
TotalNoOfRatings
```

```
globalAvgRating=np.round((train_sample_sparse.sum()/train_sample_sparse.count_nonzero(
)), 2)
globalAvgMovies = getAverageRatings(train_sample_sparse, False)
globalAvgUsers = getAverageRatings(train_sample_sparse, True)
print("Global average of all movies ratings in Train Set is : ", globalAvgRating)
print("No. of ratings in the train matrix is : ", train_sample_sparse.count_nonzero())
# Function to extract features and create row using the sparse matrix
def CreateFeaturesForTrainData(SampledSparseData, TrainSampledSparseData):
    startTime = datetime.now()
    # Extracting userId list, movieId list and Ratings
    sample_users, sample_movies, sample_ratings = sparse.find(SampledSparseData)
    print("No. of rows in the returned dataset : ", len(sample_ratings))
    count = 0
    data = []
    for user, movie, rating in zip(sample_users, sample_movies, sample_ratings):
        row = list()
#---------------------------------Appending "user Id" average, "movie Id" average & global
average rating----------#
        row.append(user)
        row.append(movie)
        row.append(globalAvgRating)
#---------------------------------Appending "user" average, "movie" average & rating of
"user""movie"----------#
        try:
            row.append(globalAvgUsers[user])
        except (KeyError):
            global_average_rating = globalAvgRating
            row.append(global_average_rating)
        except:
            raise
        try:
            row.append(globalAvgMovies[movie])
        except (KeyError):
```

```
            global_average_rating = globalAvgRating
            row.append(global_average_rating)
        except:
            raise
#--------------------------------Ratings given to "movie" by top 5 similar users with "user"-----
-------------#
        try:
            similar_users = cosine_similarity(TrainSampledSparseData[user],
TrainSampledSparseData).ravel()
            similar_users_indices = np.argsort(-similar_users)[1:]
            similar_users_ratings = TrainSampledSparseData[similar_users_indices,
movie].toarray().ravel()
            top_similar_user_ratings = list(similar_users_ratings[similar_users_ratings != 0][:5])
            top_similar_user_ratings.extend([globalAvgMovies[movie]]*(5-
len(top_similar_user_ratings)))
            #above line means that if top 5 ratings are not available then rest of the ratings will be
filled by "movie" average
            #rating. Let say only 3 out of 5 ratings are available then rest 2 will be "movie"
average rating.
            row.extend(top_similar_user_ratings)
        #########Cold Start Problem, for a new user or a new movie#########
        except (IndexError, KeyError):
            global_average_rating = [globalAvgRating]*5
            row.extend(global_average_rating)
        except:
            raise
#--------------------------------Ratings given by "user" to top 5 similar movies with "movie"---
--------------#
        try:
            similar_movies = cosine_similarity(TrainSampledSparseData[:,movie].T,
TrainSampledSparseData.T).ravel()
            similar_movies_indices = np.argsort(-similar_movies)[1:]
            similar_movies_ratings = TrainSampledSparseData[user,
```

```
similar_movies_indices].toarray().ravel()
        top_similar_movie_ratings = list(similar_movies_ratings[similar_movies_ratings !=
0][:5])
        top_similar_movie_ratings.extend([globalAvgUsers[user]]*(5-
len(top_similar_movie_ratings)))
        #above line means that if top 5 ratings are not available then rest of the ratings will be
filled by "user" average
        #rating. Let say only 3 out of 5 ratings are available then rest 2 will be "user" average
rating.
        row.extend(top_similar_movie_ratings)
      ########Cold Start Problem, for a new user or a new movie#########
      except (IndexError, KeyError):
        global_average_rating = [globalAvgRating] * 5
        row.extend(global_average_rating)
      except:
        raise
#-----------------------------------Appending rating of "user""movie"-----------#
      row.append(rating)
      count += 1
      data.append(row)
      if count % 5000 == 0:
        print("Done for {}. Time elapsed: {}".format(count, (datetime.now() - startTime)))
  print("Total Time for {} rows = {}".format(len(data), (datetime.now() - startTime)))
  print("Completed..")
  return data
# Using sampled train data, creating Features for each row and saving it into the list
data_rows = CreateFeaturesForTrainData(train_sample_sparse, train_sample_sparse)
# Using sampled train data, creating Features for each row and saving it into the list
test_data_rows = CreateFeaturesForTrainData(test_sample_sparse, train_sample_sparse)
# Creating the pandas dataframe from the data rows extracted from the sparse matrix for train
and test set
names = ["User_ID", "Movie_ID", "Global_Average", "User_Average", "Movie_Average",
"SUR1", "SUR2", "SUR3", "SUR4", "SUR5", "SMR1", "SMR2", "SMR3", "SMR4",
```

```
"SMR5", "Rating"]
train_regression_data = pd.DataFrame(data_rows, columns=names)
test_regression_data = pd.DataFrame(test_data_rows, columns=names)
# Saving the df to drive for future use
train_regression_data.to_csv(file_path + "/Training_Data_For_Regression.csv")
test_regression_data.to_csv(file_path + "/Testing_Data_For_Regression.csv")
# Loading the train and test csv files
# Path for saving/loading files
file_path = "/content/drive/MyDrive/Colab Datasets"
print("File is already present in the disk. Loading the file...")
train_regression_data = pd.read_csv(file_path + "/Training_Data_For_Regression.csv")
train_regression_data = train_regression_data.drop(["Unnamed: 0"], axis=1)
test_regression_data = pd.read_csv(file_path + "/Testing_Data_For_Regression.csv")
test_regression_data = test_regression_data.drop(["Unnamed: 0"], axis=1)
print("Done..")
# Checking the shape and first few records for train data
print("The shape of the dataframe is : ", train_regression_data.shape)
print("Number of missing Values : ", train_regression_data.isnull().sum().sum())
train_regression_data.head()
# Checking the shape and first few records for test data
print("The shape of the dataframe is : ", test_regression_data.shape)
print("Number of missing Values : ", test_regression_data.isnull().sum().sum())
test_regression_data.head()
train_regression_data[['User_ID', 'Movie_ID', 'Rating']].head(5)
# Using Surprise library Data Structures to store train data
reader = Reader(rating_scale=(1, 5))
data = Dataset.load_from_df(train_regression_data[["User_ID", "Movie_ID", "Rating"]],
reader)
trainset = data.build_full_trainset()
# Creating tuple for test set
testset = list(zip(test_regression_data["User_ID"].values,
test_regression_data["Movie_ID"].values, test_regression_data["Rating"].values))
# Utilities to save the modelling results
```

```
error_cols = ["Model", "Train RMSE", "Train MAPE", "Test RMSE", "Test MAPE"]
error_table = pd.DataFrame(columns = error_cols)
model_train_evaluation = dict()
model_test_evaluation = dict()
# Function to save modelling results in a table
def make_table(model_name, rmse_train, mape_train, rmse_test, mape_test):
    global error_table
    error_table = error_table.append(pd.DataFrame([[model_name, rmse_train, mape_train,
rmse_test, mape_test]], columns = error_cols))
    error_table.reset_index(drop = True, inplace = True)
# Function to calulate RMSE and MAPE values
def error_metrics(y_true, y_pred):
    rmse = np.sqrt(mean_squared_error(y_true, y_pred))
    mape = np.mean(abs((y_true - y_pred)/y_true))*100
    return rmse, mape
# Apply Xgboost Regressor on the Train and Test Data
def train_test_xgboost(x_train, x_test, y_train, y_test, model_name):
    startTime = datetime.now()
    train_result = dict()
    test_result = dict()
    clf = xgb.XGBRegressor(n_estimators = 100, silent = False, n_jobs  = 10)
    clf.fit(x_train, y_train)
    print("-" * 50)
    print("TRAIN DATA")
    y_pred_train = clf.predict(x_train)
    rmse_train, mape_train = error_metrics(y_train, y_pred_train)
    print("RMSE : {}".format(rmse_train))
    print("MAPE : {}".format(mape_train))
    train_result = {"RMSE": rmse_train, "MAPE": mape_train, "Prediction": y_pred_train}
    print("-" * 50)
    print("TEST DATA")
    y_pred_test = clf.predict(x_test)
    rmse_test, mape_test = error_metrics(y_test, y_pred_test)
```

```
    print("RMSE : {}".format(rmse_test))
    print("MAPE : {}".format(mape_test))
    test_result = {"RMSE": rmse_test, "MAPE": mape_test, "Prediction": y_pred_test}
    print("-"*50)
    print("Time Taken : ", datetime.now() - startTime)
    plot_importance(xgb, clf)
    make_table(model_name, rmse_train, mape_train, rmse_test, mape_test)
    return train_result, test_result
# Function to plot feature importance for a model
def plot_importance(model, clf):
    sns.set(style="darkgrid")
    fig = plt.figure(figsize = (25, 5))
    ax = fig.add_axes([0, 0, 1, 1])
    model.plot_importance(clf, ax = ax, height = 0.3)
    plt.xlabel("F Score", fontsize = 20)
    plt.ylabel("Features", fontsize = 20)
    plt.title("Feature Importance", fontsize = 20)
    plt.tick_params(labelsize = 15)
    plt.show()
# in surprise prediction of every data point is returned as dictionary like this:
# "user: 196        item: 302        r_ui = 4.00   est = 4.06   {'actual_k': 40, 'was_impossible':
False}"
# In this dictionary, "r_ui" is a key for actual rating and "est" is a key for predicted rating
def get_ratings(predictions):
    actual = np.array([pred.r_ui for pred in predictions])
    predicted = np.array([pred.est for pred in predictions])
    return actual, predicted
def get_error(predictions):
    actual, predicted = get_ratings(predictions)
    rmse = np.sqrt(mean_squared_error(actual, predicted))
    mape = np.mean(abs((actual - predicted)/actual))*100
    return rmse, mape
my_seed = 15
```

```
random.seed(my_seed)
np.random.seed(my_seed)
# Running Surprise model algorithms
def run_surprise(algo, trainset, testset, model_name):
    startTime = datetime.now()
    train = dict()
    test = dict()
    algo.fit(trainset)
#----------------Evaluating Train Data----------------#
    print("-"*50)
    print("TRAIN DATA")
    train_pred = algo.test(trainset.build_testset())
    train_actual, train_predicted = get_ratings(train_pred)
    train_rmse, train_mape = get_error(train_pred)
    print("RMSE = {}".format(train_rmse))
    print("MAPE = {}".format(train_mape))
    train = {"RMSE": train_rmse, "MAPE": train_mape, "Prediction": train_predicted}
#----------------Evaluating Test Data----------------#
    print("-"*50)
    print("TEST DATA")
    test_pred = algo.test(testset)
    test_actual, test_predicted = get_ratings(test_pred)
    test_rmse, test_mape = get_error(test_pred)
    print("RMSE = {}".format(test_rmse))
    print("MAPE = {}".format(test_mape))
    test = {"RMSE": test_rmse, "MAPE": test_mape, "Prediction": test_predicted}
    print("-"*50)
    print("Time Taken = "+str(datetime.now() - startTime))
    make_table(model_name, train_rmse, train_mape, test_rmse, test_mape)
    return train, test
# Creating the train-test X and y variables for the ML algos
x_train = train_regression_data.drop(["User_ID", "Movie_ID", "Rating"], axis = 1)
x_test = test_regression_data.drop(["User_ID", "Movie_ID", "Rating"], axis = 1)
```

```
y_train = train_regression_data["Rating"]
y_test = test_regression_data["Rating"]
# Training the Xgboost Regression Model on with the 13 features
train_result, test_result = train_test_xgboost(x_train, x_test, y_train, y_test, "XGBoost_13")
model_train_evaluation["XGBoost_13"] = train_result
model_test_evaluation["XGBoost_13"] = test_result
# Applying BaselineOnly from the surprise library to predict the ratings
bsl_options = {"method":"sgd", "learning_rate":0.01, "n_epochs":25}
algo = BaselineOnly(bsl_options=bsl_options)
train_result, test_result = run_surprise(algo, trainset, testset, "BaselineOnly")
model_train_evaluation["BaselineOnly"] = train_result
model_test_evaluation["BaselineOnly"] = test_result
# Adding predicted ratings from Surprise BaselineOnly model to our Train and Test
Dataframe
train_regression_data["BaselineOnly"] =
model_train_evaluation["BaselineOnly"]["Prediction"]
test_regression_data["BaselineOnly"] =
model_test_evaluation["BaselineOnly"]["Prediction"]
# Fitting the Xgboost again with new BaselineOnly feature
x_train = train_regression_data.drop(["User_ID", "Movie_ID", "Rating"], axis = 1)
x_test = test_regression_data.drop(["User_ID", "Movie_ID", "Rating"], axis = 1)
y_train = train_regression_data["Rating"]
y_test = test_regression_data["Rating"]
train_result, test_result = train_test_xgboost(x_train, x_test, y_train, y_test, "XGB_BSL")
model_train_evaluation["XGB_BSL"] = train_result
model_test_evaluation["XGB_BSL"] = test_result
# Finding the suitable parameter for Surprise KNN-Baseline with User-User Similarity
param_grid  = {'sim_options':{'name': ["pearson_baseline"], "user_based": [True],
"min_support": [2], "shrinkage": [60, 80, 80, 140]}, 'k': [5, 20, 40, 80]}
gs = GridSearchCV(KNNBaseline, param_grid, measures=['rmse', 'mae'], cv=3)
gs.fit(data)
# best RMSE score
print(gs.best_score['rmse'])
```

```
# combination of parameters that gave the best RMSE score
print(gs.best_params['rmse'])
# Applying the KNN-Baseline with the searched parameters
sim_options = {'name':'pearson_baseline', 'user_based':True, 'min_support':2,
'shrinkage':gs.best_params['rmse']['sim_options']['shrinkage']}
bsl_options = {'method': 'sgd'}
algo = KNNBaseline(k = gs.best_params['rmse']['k'], sim_options = sim_options,
bsl_options=bsl_options)
train_result, test_result = run_surprise(algo, trainset, testset, "KNNBaseline_User")
model_train_evaluation["KNNBaseline_User"] = train_result
model_test_evaluation["KNNBaseline_User"] = test_result
# Similarly finding best parameters for Surprise KNN-Baseline with Item-Item Similarity
param_grid  = {'sim_options':{'name': ["pearson_baseline"], "user_based": [False],
"min_support": [2], "shrinkage": [60, 80, 80, 140]}, 'k': [5, 20, 40, 80]}
gs = GridSearchCV(KNNBaseline, param_grid, measures=['rmse', 'mae'], cv=3)
gs.fit(data)
# best RMSE score
print(gs.best_score['rmse'])
# combination of parameters that gave the best RMSE score
print(gs.best_params['rmse'])
# Applying KNN-Baseline with best parameters searched
sim_options = {'name':'pearson_baseline', 'user_based':False, 'min_support':2,
'shrinkage':gs.best_params['rmse']['sim_options']['shrinkage']}
bsl_options = {'method': 'sgd'}
algo = KNNBaseline(k = gs.best_params['rmse']['k'], sim_options = sim_options,
bsl_options=bsl_options)
train_result, test_result = run_surprise(algo, trainset, testset, "KNNBaseline_Item")
model_train_evaluation["KNNBaseline_Item"] = train_result
model_test_evaluation["KNNBaseline_Item"] = test_result
# Addding the KNNBaseline features to the train and test dataset
train_regression_data["KNNBaseline_User"] =
model_train_evaluation["KNNBaseline_User"]["Prediction"]
train_regression_data["KNNBaseline_Item"] =
```

```
model_train_evaluation["KNNBaseline_Item"]["Prediction"]
test_regression_data["KNNBaseline_User"] =
model_test_evaluation["KNNBaseline_User"]["Prediction"]
test_regression_data["KNNBaseline_Item"] =
model_test_evaluation["KNNBaseline_Item"]["Prediction"]
# Applying Xgboost with the KNN-Baseline newly added features
x_train = train_regression_data.drop(["User_ID", "Movie_ID", "Rating"], axis = 1)
x_test = test_regression_data.drop(["User_ID", "Movie_ID", "Rating"], axis = 1)
y_train = train_regression_data["Rating"]
y_test = test_regression_data["Rating"]
train_result, test_result = train_test_xgboost(x_train, x_test, y_train, y_test,
"XGB_BSL_KNN")
model_train_evaluation["XGB_BSL_KNN"] = train_result
model_test_evaluation["XGB_BSL_KNN"] = test_result
# Appling the SlopeOne algorithm from the Surprise library
so = SlopeOne()
train_result, test_result = run_surprise(so, trainset, testset, "SlopeOne")
model_train_evaluation["SlopeOne"] = train_result
model_test_evaluation["SlopeOne"] = test_result
# Adding the SlopOne predictions to the train and test datasets
train_regression_data["SlopeOne"] = model_train_evaluation["SlopeOne"]["Prediction"]
train_regression_data["SlopeOne"] = model_train_evaluation["SlopeOne"]["Prediction"]
test_regression_data["SlopeOne"] = model_test_evaluation["SlopeOne"]["Prediction"]
test_regression_data["SlopeOne"] = model_test_evaluation["SlopeOne"]["Prediction"]
# Matrix Factorization using SVD from Surprise Library
# here, n_factors is the equivalent to dimension 'd' when matrix 'A'
# is broken into 'b' and 'c'. So, matrix 'A' will be of dimension n*m. So, matrices 'b' and 'c'
will be of dimension n*d and m*d.
param_grid  = {'n_factors': [5,7,10,15,20,25,35,50,70,90]}
gs = GridSearchCV(SVD, param_grid, measures=['rmse', 'mae'], cv=3)
gs.fit(data)
# best RMSE score
print(gs.best_score['rmse'])
```

```
# combination of parameters that gave the best RMSE score
print(gs.best_params['rmse'])
# Applying SVD with best parameters
algo = SVD(n_factors = gs.best_params['rmse']['n_factors'], biased=True, verbose=True)
train_result, test_result = run_surprise(algo, trainset, testset, "SVD")
model_train_evaluation["SVD"] = train_result
model_test_evaluation["SVD"] = test_result
# Matrix Factorization SVDpp with implicit feedback
# Hyper-parameter optimization for SVDpp
param_grid = {'n_factors': [10, 30, 50, 80, 100], 'lr_all': [0.002, 0.006, 0.018, 0.054, 0.10]}
gs = GridSearchCV(SVDpp, param_grid, measures=['rmse', 'mae'], cv=3)
gs.fit(data)
# best RMSE score
print(gs.best_score['rmse'])
# combination of parameters that gave the best RMSE score
print(gs.best_params['rmse'])
#Applying SVDpp with best parameters
algo = SVDpp(n_factors = gs.best_params['rmse']['n_factors'], lr_all =
gs.best_params['rmse']["lr_all"], verbose=True)
train_result, test_result = run_surprise(algo, trainset, testset, "SVDpp")
model_train_evaluation["SVDpp"] = train_result
model_test_evaluation["SVDpp"] = test_result
# XGBoost 13 Features + Surprise BaselineOnly + Surprise KNN Baseline + SVD + SVDpp
train_regression_data["SVD"] = model_train_evaluation["SVD"]["Prediction"]
train_regression_data["SVDpp"] = model_train_evaluation["SVDpp"]["Prediction"]
test_regression_data["SVD"] = model_test_evaluation["SVD"]["Prediction"]
test_regression_data["SVDpp"] = model_test_evaluation["SVDpp"]["Prediction"]
# Applying Xgboost on the feature set
x_train = train_regression_data.drop(["User_ID", "Movie_ID", "Rating"], axis = 1)
x_test = test_regression_data.drop(["User_ID", "Movie_ID", "Rating"], axis = 1)
y_train = train_regression_data["Rating"]
y_test = test_regression_data["Rating"]
```

```
train_result, test_result = train_test_xgboost(x_train, x_test, y_train, y_test,
"XGB_BSL_KNN_MF")
model_train_evaluation["XGB_BSL_KNN_MF"] = train_result
model_test_evaluation["XGB_BSL_KNN_MF"] = test_result
# Applying Xgboost with Surprise's BaselineOnly + KNN Baseline + SVD + SVDpp +
SlopeOne
x_train = train_regression_data[["BaselineOnly", "KNNBaseline_User",
"KNNBaseline_Item", "SVD", "SVDpp", "SlopeOne"]]
x_test = test_regression_data[["BaselineOnly", "KNNBaseline_User", "KNNBaseline_Item",
"SVD", "SVDpp", "SlopeOne"]]
y_train = train_regression_data["Rating"]
y_test = test_regression_data["Rating"]
train_result, test_result = train_test_xgboost(x_train, x_test, y_train, y_test,
"XGB_KNN_MF_SO")
model_train_evaluation["XGB_KNN_MF_SO"] = train_result
model_test_evaluation["XGB_KNN_MF_SO"] = test_result
print(error_table)
# Visualizing the errors of all the models we tested out
error_table2 = error_table.drop(["Train MAPE", "Test MAPE"], axis = 1)
error_table2.plot(x = "Model", kind = "bar", figsize = (25, 8), grid = True, fontsize = 15)
plt.title("Train and Test RMSE and MAPE of all Models", fontsize = 20)
plt.ylabel("Error Values", fontsize = 10)
plt.xticks(rotation=60)
plt.legend(bbox_to_anchor=(1, 1), fontsize = 10)
plt.show()
# Tabular Values of Errors
error_table.drop(["Train MAPE", "Test MAPE"], axis = 1)
# Testing the recommendations made by SVDpp Algorithm
from collections import defaultdict
def Get_top_n(predictions, n=10):
    # First map the predictions to each user.
    top_n = defaultdict(list)
    for uid, mid, true_r, est, _ in predictions:
```

```
        top_n[uid].append((mid, est))
    # Then sort the predictions for each user and retrieve the k highest ones.
    for uid, user_ratings in top_n.items():
        user_ratings.sort(key=lambda x: x[1], reverse=True)
        top_n[uid] = user_ratings[:n]
    return top_n
# Creating instance of svd_pp
svd_pp = SVDpp(n_factors = 10, lr_all = 0.006, verbose=True)
svd_pp.fit(trainset)
predictions = svd_pp.test(testset)
# Saving the training predictions
train_pred = svd_pp.test(trainset.build_anti_testset())
top_n = Get_top_n(train_pred, n=10)
# Print the recommended items for each user
def Generate_Recommendated_Movies(u_id, n=10):
    recommend = pd.DataFrame(top_n[u_id], columns=["Movie_Id", "Predicted_Rating"])
    recommend = recommend.merge(movies, how="inner", left_on="Movie_Id",
right_on="movieId")
    recommend = recommend[["Movie_Id", "title", "genres", "Predicted_Rating"]]
    return recommend[:n]
# Saving the sampled user id list to help generate movies
sampled_user_id = list(top_n.keys())
# Generating recommendation using the user_Id
test_id = random.choice(sampled_user_id)
print("The user Id is : ", test_id)
Generate_Recommendated_Movies(test_id)
# Generating recommendation using the user_Id
test_id = random.choice(sampled_user_id)
print("The user Id is : ", test_id)
Generate_Recommendated_Movies(test_id)
import pandas as pd
from mlxtend.frequent_patterns import apriori, association_rules
```

```
def create_basket(df, threshold=3.5):
    # Filter the data based on the threshold
    df = df[df['rating'] >= threshold]
    # Create a basket for each user containing the movies they rated positively
    basket = (df.groupby(['userId', 'movieId'])['rating']
              .sum().unstack().reset_index().fillna(0)
              .set_index('userId'))
    # Convert ratings to binary values
    basket[basket > 0] = 1
    return basket
def recommend_movies(ratings_file, movies_file, min_support=0.1, min_threshold=1.0):
    # Load the data
    ratings = pd.read_csv(ratings_file, names=['userId', 'movieId', 'rating', 'timestamp'],
header=0)
    movies= pd.read_csv(movies_file, names=['movieId', 'title', 'genres'], header=0)
    # Create the basket
    basket = create_basket(ratings)
    # Perform association rule mining
    frequent_items = apriori(basket, min_support=min_support, use_colnames=True)
    rules = association_rules(frequent_items, metric="lift", min_threshold=min_threshold)
    # Get the top 10 recommended movies based on lift
    recommended_movies = (rules.sort_values(by=['lift'], ascending=False)
                          .head(10)[['antecedents', 'consequents']]
                          .apply(lambda x: list(x)[0], axis=1)
                          .explode()
                          .apply(lambda x: int(x))
                          .map(movies.set_index('movieId')['title'])
                          .tolist())
    return recommended_movies
# Test the function with the Movie Lens 25M dataset
recommended_movies = recommend_movies('ratings.csv', 'movies.csv')
for _ in recommended_movies:
  print(_)
```

6.SYSTEM TESTING

6.1 Overview of Testing

Testing is an essential component of the development process for any recommendation system. It involves evaluating the performance of the recommendation system by measuring how accurately it can predict user preferences and provide relevant recommendations.

6.2 Types of Tests

6.2.1 Unit Testing

This involves testing individual units or components of the system to ensure that they function correctly. In the case of recommendation systems, unit testing may involve testing individual algorithms or modules that make up the system to ensure that they are functioning as expected.

6.2.2 Integration Testing

This involves testing the integration of multiple components or modules of the system to ensure that they work together as expected. In the context of recommendation systems, integration testing may involve testing how different algorithms or models interact with each other and how data flows between them.

6.2.2 System Testing

This involves testing the system as a whole to ensure that it meets the functional and non-functional requirements specified by the stakeholders. In the case of recommendation systems, system testing may involve testing the performance, scalability, and usability of the system.

7.RESULTS

```
--------------------------------------------------
TRAIN DATA
RMSE = 0.7631256379833844
MAPE = 25.417482535813974
--------------------------------------------------
TEST DATA
RMSE = 0.999497987467072
MAPE = 36.45769475543719
--------------------------------------------------
Time Taken = 0:00:08.135545
```

Figure:9 Performance of SVDpp Model.

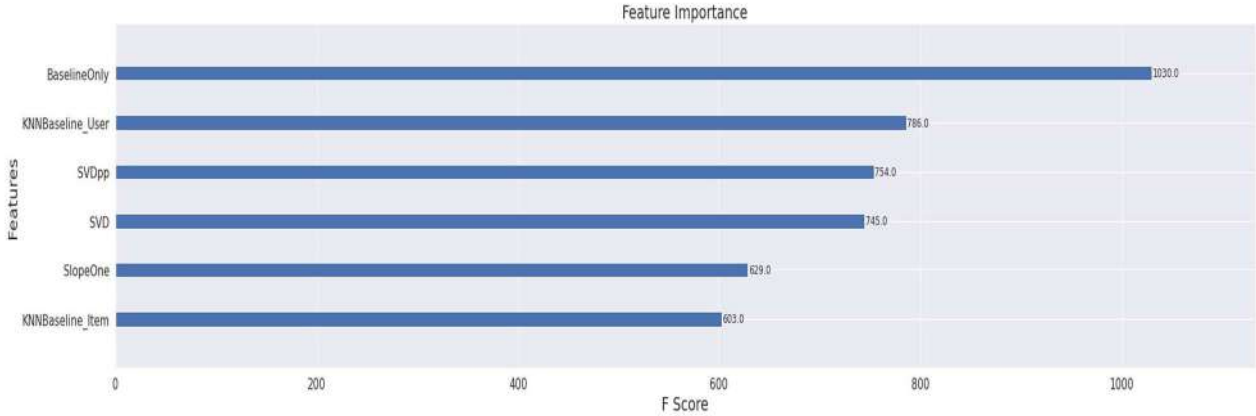

Figure:10 Plot for Feature Importance.

	Model	Train RMSE	Train MAPE	Test RMSE	Test MAPE
0	XGBoost_13	0.703410	21.925777	1.023873	36.546029
1	BaselineOnly	0.827371	27.610189	0.999581	36.467623
2	XGB_BSL	0.706319	22.095931	1.049667	36.612986
3	KNNBaseline_User	0.337843	10.313457	0.999480	36.460648
4	KNNBaseline_Item	0.274160	7.982777	0.999480	36.460648
5	XGB_BSL_KNN	0.702866	22.047667	1.015867	36.525591
6	SlopeOne	0.760572	24.411324	0.999146	36.447592
7	SVD	0.831105	28.103554	0.999488	36.460337
8	SVDpp	0.763126	25.417483	0.999498	36.457695
9	XGB_BSL_KNN_MF	0.696570	21.858855	1.012014	36.510069
10	XGB_KNN_MF_SO	0.992102	35.414786	0.996925	36.814946

Figure:11 Performance of Various Models for Movie Rating Prediction.

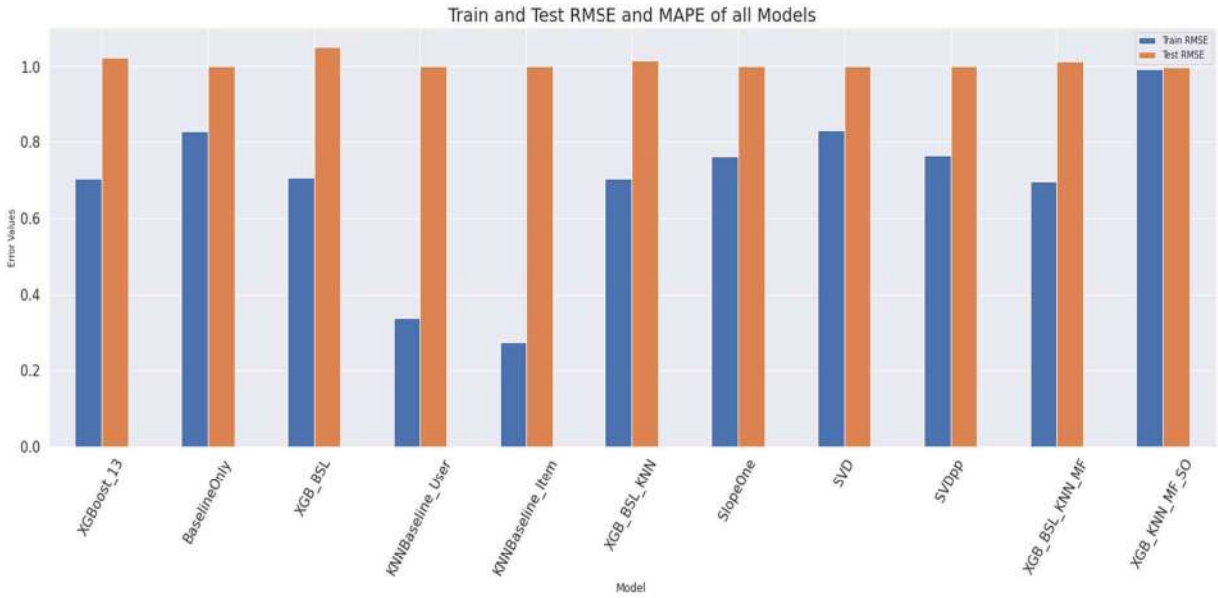

Figure:12 Plot for Performance of Various Models for Movie Rating Prediction.

```
# Generating recommendation using the user_Id

test_id = random.choice(sampled_user_id)
print("The user Id is : ", test_id)
Generate_Recommendated_Movies(test_id)
```

```
The user Id is :  55237
```

	Movie_Id	title	genres	Predicted_Rating
0	893	Apartment, The (1960)	Comedy|Drama|Romance	4.337311
1	2850	Lady Eve, The (1941)	Comedy|Romance	4.293874
2	213	Before the Rain (Pred dozhdot) (1994)	Drama|War	4.249252
3	2120	Shadow of a Doubt (1943)	Crime|Drama|Thriller	4.235916
4	3380	Hustler, The (1961)	Drama	4.224963
5	3711	Anatomy of a Murder (1959)	Drama|Mystery	4.219006
6	4312	Man Who Shot Liberty Valance, The (1962)	Crime|Drama|Western	4.190298
7	6902	Night of the Hunter, The (1955)	Drama|Film-Noir|Thriller	4.176537
8	4913	Witness for the Prosecution (1957)	Drama|Mystery|Thriller	4.132051
9	7045	Fog of War: Eleven Lessons from the Life of Ro...	Documentary|War	4.121651

Figure:13 Movie recommendations for the user_Id by SVDpp.

	antecedents	consequents	antecedent support	consequent support	support	confidence	lift	leverage	conviction
0	(Anchorman: The Legend of Ron Burgundy (2004))	(Step Brothers (2008))	0.068966	0.051724	0.034483	0.500000	9.666667	0.030916	1.896552
1	(Step Brothers (2008))	(Anchorman: The Legend of Ron Burgundy (2004))	0.051724	0.068966	0.034483	0.666667	9.666667	0.030916	2.793103
2	(Corpse Bride (2005))	(Step Brothers (2008))	0.051724	0.051724	0.017241	0.333333	6.444444	0.014566	1.422414
3	(Step Brothers (2008))	(Corpse Bride (2005))	0.051724	0.051724	0.017241	0.333333	6.444444	0.014566	1.422414

Figure:14 Association rules for the data

		lift
antecedents	consequents	
(Inception (2010))	(Gentlemen Broncos (2009))	14.5
	(Life Aquatic with Steve Zissou, The (2004))	14.5
	(400 Blows, The (Les quatre cents coups) (1959))	14.5
	(A Pigeon Sat on a Branch Reflecting on Existence (2014))	14.5
	(Arrival (2016))	14.5
	(Andalusian Dog, An (Chien andalou, Un) (1929))	14.5
	(Angel's Egg (Tenshi no tamago) (1985))	14.5
	(Illusionist, The (2006))	14.5
	(Dunkirk (2017))	14.5
	(Don't Breathe (2016))	14.5

Figure:15 Top 10 related movies to Inception based on lift.

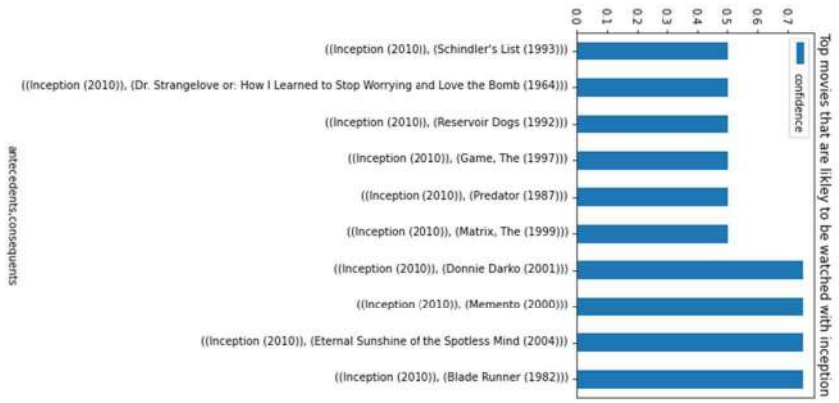

Figure:16 Plot for Top 10 related movies to Inception based on confidence.

8.CONCLUSION

This project suggests a hybrid recommendation system that integrates association rule mining with collaborative filtering to enhance the precision and dependability of movie suggestions. This system utilizes similarities in user and item characteristics to make recommendations, and the combination of these techniques allows for strong recommendations even when data is limited. This project also mentions that the system is suitable for solving the problem of information overload by filtering and curating relevant information from massive data banks, and that Collaborative Filtering techniques have proven to be the most suitable for many problems. This project describes two different methods of recommendation. Here, the objective is to raise the standard of recommendations and provide target consumers with better recommendations. It is essential to boost performance even more in order to remove constraints on the collaborative filtering process. Additionally, it may create a dynamic architecture that is new and much more effective to use and provide people with helpful recommendations. Overall, the proposed method appears to be a promising solution for improving the accuracy and reliability of movie recommendations.

9.REFERENCES

[1] Jiawei Han and Micheline Kamber,"Data Mining Concepts & Techniiques",Elsevier,2011.

[2] Masoumeh Mohammad and Mehregan Mahdavi, IJITCS Intelligent Systems, Vol. 21, No. 1, pp.35-41, IJITCS 2012.

[3] Adomavicius, G., Tuzhilin, A" Toward the next generation of recommender systems: a survey of the state of-the-art and possible extensions", IEEE Transactions on Knowledge and Data Engineering, Vol.17, No. 6, pp. 734-749, IJITCS 2012Tavel, P. 2007 Modeling and Simulation Design. AK Peters Ltd.

[4] Aggarwal, C. C., Procopiuc, C., and Yu, P. S." Finding localized associations in market basket data. IEEE Transactions on Knowledge and Data Engineering", 14, 1, 2002,pp.51-62, ELSEVIER 2012.Forman, G. 2003. An extensive empirical study of feature selection metrics for text classification. J. Mach. Learn. Res. 3 (Mar. 2003), 1289-1305.

[5] Yan Ying Chen, An-Jung Cheng and Winston H. Hsu" IEEE Transactions on Multimedia",15,6, pp,1283-1288,IEEE Oct 2013.

[6] Lee, C.-H., Kim, Y.-H., & Rhee, P.-K." Web personalization expert with combining collaborative filtering and association rule mining technique". Expert Systems and Applications, 21(3), 131–137, ELSEVIER 2013.

[7] Agrawal, R., Imielinski, T., & Swami, A." Mining association rules between sets of items in large databases. In P. Buneman& S. Jajodia (Eds.)," pp.207–216, ELSEVIER 2013.

[8] Ricci, F., Rokach, L., Shapira, B. (2011) "Recommender Systems Handbook", Springer,

[9] Liangxing, Y., Aihua, D. (2010) "Hybrid Product Recommender System for Apparel RetailingCustomers, In proceeding ICIE '10 "Proceedings of the 2010 WASE International Conference onInformation Engineering, Washington, DC, USA.

[10] Banati, H., Mehta, S. (2010)" Memetic Collaborative filtering-based Recommender System", Second Vaagdevi

International Conference on Information Technology for Real WorldProblems, Warangal, India.

[11] Salter, J., Antonopoulus, N." CinemaScreen recommender agent: Combining collaborative filtering and content-based filtering", IEEE Intelligent Systems, Vol. 21, No. 1, pp.35-41, IJITCS 2012.

[12] Shaw, Geva, S. "Investigating the use of association rules in improving the recommender system" Proc. 14th Australasian Document Computing, Sydney, Australia.

[13] B.,Amini, R.,Ibrahim, and M.S.,Othman (2011). Discovering the impact of knowledge in recommender systems: A comparative study. arXiv preprint arXiv:1109.0166.

[14] P.,Lops, M.,Gemmis, and G.,Semeraro (2011). Content-based recommender systems: State of the art and trends. Recommender Systems Handbook, 73-105.

[15] R.,Burke (2002). Hybrid recommender systems: Survey and experiments. User modeling and user-adapted interaction, 12(4), 331-370.