Project: Movie Recommendation System

Pablo Mollá Chárlez

Contents

1 Introduction

Recommendation algorithms are specialized algorithms designed to suggest relevant items to users in various online settings, including e-commerce platforms, streaming services, and social media networks. These algorithms aim to predict user preferences by analyzing their historical behavior, the preferences of similar users, or inherent item characteristics. By leveraging these insights, recommendation systems facilitate the discovery of products, movies, books, and other content that users might find appealing but have not yet encountered.

Recommendation systems play a crucial role in numerous applications:

1. **E-commerce**: Suggest products to customers based on their past purchases and browsing history.
2. **Media Streaming**: Recommend movies, music, or TV shows based on previous views or listens.
3. **Social Media**: Propose posts, friends, or groups to follow based on user activity and interactions.

There are three primary approaches to building recommendation systems: Content-Based Filtering, Collaborative Filtering, and Latent Factor Models. Collaborative Filtering (CF) is a widely adopted recommendation technique that predicts a user's interests by aggregating preferences from a large pool of users. The foundational assumption behind Collaborative Filtering is that users who agreed on one item in the past are more likely to agree on other items in the future.

For instance, consider a user named *Pablo*. Collaborative Filtering identifies other users with similar tastes to *Pablo* based on their past ratings. It then recommends movies that these similar users have enjoyed but that *Pablo* has not yet rated or interacted with. Collaborative Filtering can be categorized into two main types:

- User-User Collaborative Filtering Identifies users similar to the target user based on their past ratings and recommends items liked by these similar users.
- Item-Item Collaborative Filtering Finds items similar to those the user has already liked and recommends these similar items.

The objective of this report is to provide a comprehensive description of the implementation of a Movie Recommendation System using the User-User Collaborative Filtering approach in Python. Additionally, it outlines the experimental methodology employed to evaluate the quality and effectiveness of the recommendations generated by the system.

2 Implementation Analysis

2.1 Algorithm Choices

In this project, a User-User Collaborative Filtering approach is implemented to develop the recommendation system. Initially, an Item-Item Collaborative Filtering method was considered. However, after thorough evaluation, the decision was made to adopt the User-User approach due to its more intuitive mechanism for predicting ratings and its relative simplicity in dataset splitting for evaluating prediction quality.

2.2 Dataset Description

The provided dataset, named **train_ratings.csv**, comprises a total of **800,168** entries, structured as follows:

```
user_id, movie_id, rating, timestamp
5412,2683,2,960243649
5440,904,5,959995181
368,3717,4,976311423
425,1721,4,976283587
4942,3697,1,962642480
... (more rows)
```

Where the **columns** are the **user_id** (unique identifier for each user), the **movie_id** (unique identifier for each movie), the **rating** (user's rating for the movie are on a scale from 1 to 5) and **timestamp** (the time at which the rating was given).

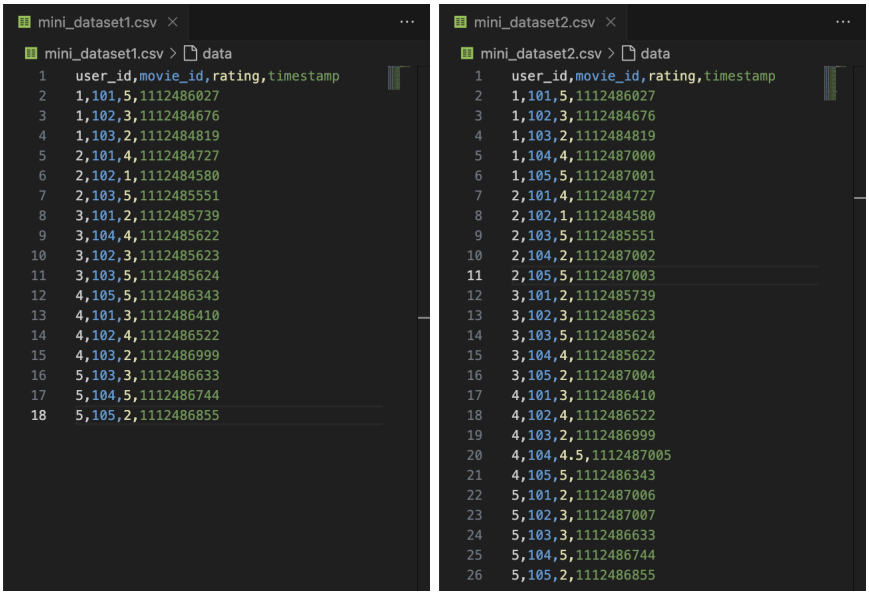

Figure 1: Testing Datasets: mini_dataset1.csv & mini_dataset2.csv

However, for testing and debugging purposes, two smaller datasets, mini_dataset1.csv and mini_dataset2.csv, were utilized. The mini_dataset1.csv was employed to assess the consistency of similarity computations and to predict ratings, specifically verifying the feasibility of the train/test split procedure with some unknown ratings. In contrast, the more comprehensive mini_dataset2.csv was used to evaluate the quality of rating recommendations using metrics such as MAE, RMSE, Precision, and Recall. This dataset was meticulously filled to allow for the assessment of recall and precision with full knowledge of user behavior, while acknowledging that, in real-life scenarios, no user has seen all movies. Below are the contents of the smaller datasets presented in table format for easier visualization.

User ID	Movie 101	Movie 102	Movie 103	Movie 104	Movie 105
1	5	3	2	✗	✗
2	4	1	5	✗	✗
3	2	3	5	4	✗
4	3	4	2	✗	5
5	✗	✗	3	5	2

Table 1: Sample Ratings from mini_dataset1.csv

User ID	Movie 101	Movie 102	Movie 103	Movie 104	Movie 105
1	5	3	2	4	5
2	4	1	5	2	5
3	2	3	5	4	2
4	3	4	2	4.5	5
5	2	3	3	5	2

Table 2: Sample Ratings from mini_dataset2.csv

2.2.1 Similarity Measures

The implementation considers two primary similarity measures: **Cosine Similarity** and **Jaccard Similarity**. Initially, the **Pearson Correlation Coefficient** was also evaluated as a potential similarity metric. However, its domain range of $[-1, 1]$ introduced complexities in handling negative correlations and required additional normalization steps. Given the project's focus on simplifying similarity computations and the adequate performance of Cosine and Jaccard similarities, Pearson Correlation was considered unnecessary and subsequently omitted.

2.2.1.1 Cosine Similarity

Cosine Similarity measures the cosine of the angle between two non-zero vectors in a multi-dimensional space. It is particularly effective in determining the orientation similarity between users based on their rating vectors, irrespective of their magnitude.

$$\text{Cosine Similarity} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \times \sqrt{\sum_{i=1}^{n} B_i^2}}$$

2.2.1.2 Jaccard Similarity

Jaccard Similarity, also known as the Jaccard Index, measures the similarity between two sets by evaluating the size of their intersection divided by the size of their union. It is particularly useful for binary or set-based data representations.

$$\text{Jaccard Similarity} = \frac{|A \cap B|}{|A \cup B|}$$

2.2.2 Algorithm Workflow

The recommendation system follows a structured workflow:

1. **Data Preprocessing**: Normalize ratings to ensure uniformity and eliminate biases arising from varying rating scales.
2. **Similarity Computation**: Calculate similarity scores between users using the chosen similarity measures (Cosine and Jaccard).
3. **Rating Prediction**: Predict ratings for unrated movies by aggregating ratings from similar users based on the computed similarity scores.
4. **Recommendation Generation**: Generate top-N movie recommendations for each user by combining actual and predicted ratings.
5. **Evaluation**: Assess the quality of recommendations using performance metrics such as MAE, RMSE, Precision, and Recall.

2.3 Code Structure and Modularity

The code is meticulously organized into modular functions, each responsible for distinct aspects of the recommendation system. This modularity ensures separation of concerns, enhances readability, and facilitates easier maintenance and scalability.

To ease the understanding of the code, I will describe simultaneously how the user interacts with the recommendation system via terminal and each function's goal details. Besides, the functions will be included witin the report to facilitate the understanding.

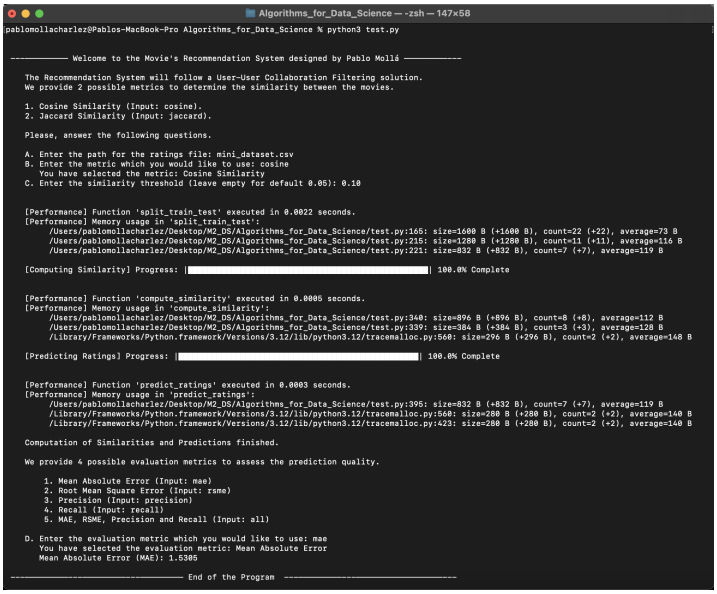

Figure 2: Program Workflow

2.3.1 User Workflow for the Movie's Recommendation System

Let's first start with a screenshot of what it looks like launching the program, to have an idea of the path that will be followed:

Upon starting the program, the user is greeted with a welcome message that introduces the Movie's Recommendation System and outlines its foundational approach. The system explains that a User-User Collaborative Filtering method will be considered to generate movie recommendations. The system requests the user to input the path to the ratings file, which contains the dataset of user ratings for various movies, and the existence of the provided file path is verified. If the file is not found, an error message is displayed, prompting the user to ensure the path is correct and ending the program.

Figure 3: Welcoming Message

On the other hand, if the file is found, then the system asks the user to choose a similarity metric:

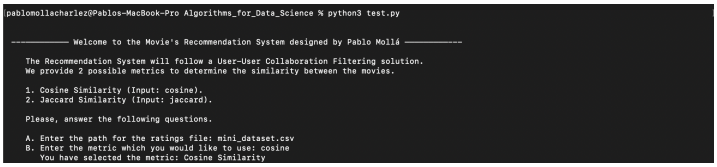

Figure 4: Similarity Metric Choice

The user inputs their choice by typing either *cosine* or *jaccard*. The system acknowledges the selected metric, checking whether the input is one of the available similarity metrics, and confirms the choice. The user is prompted to enter a similarity threshold, which determines the minimum similarity score required to consider two movies as similar. The user can leave this input empty to accept the default value of 0.05.

The program validates the input to ensure it's a numerical value within the specified range. If the input is invalid, the user is prompted to re-enter a correct value. Upon successful entry, the system proceeds with the provided threshold. After collecting the necessary inputs, the system performs several backend operations to process the data involving functions such as:

Figure 5: Similarity Threshold

- The split_train_test function is designed to divide a dataset of movie ratings into training and testing sets to facilitate the evaluation of the recommendation system. Initially, it loads all ratings from a specified CSV file, organizing them into three structures, including a list of all ratings (**all_ratings**), a dictionary mapping each movie to the users who rated it (**movie_to_users**) and a dictionary mapping each user to the movies they have rated (**user_to_movies**). To determine which ratings are eligible for the test set, the function applies two criteria:

  1. Each movie must be rated by at least a minimum number of other users (**min_users**)
  2. Those users must share a minimum number of common movies (**min_common_movies**) with the current user

  Eligible ratings are then randomly shuffled to ensure a unbiased selection. The function splits the shuffled eligible ratings into test and training sets based on the specified **test_ratio**, normalizes the ratings to a consistent scale, and organizes them into two nested dictionaries (**train_ratings** and **test_ratings**) where each user maps to their respective movie ratings. Finally, it returns the training ratings, testing ratings, and the complete list of all ratings for further processing.

- The compute_similarity function is designed to calculate similarity scores between users based on their movie ratings, facilitating the recommendation process in a collaborative filtering system. It takes in a dictionary of user ratings (**user_ratings**), where each user maps to their rated movies with normalized ratings. The function iterates through all possible pairs of users, ensuring that each pair shares at least a specified minimum number of common movies (**min_common**). For each eligible pair, it computes the similarity using the chosen metric (**cosine** or **jaccard**). The computed similarities are stored in a nested dictionary (**user_similarity**), where each user maps to other users along with their corresponding similarity scores. Throughout the computation, a progress bar visually indicates the calculation's advancement. Finally, the function returns the **user_similarity** dictionary, which encapsulates the similarity relationships between all user pairs, thereby enabling the system to identify and leverage similar users for making accurate movie recommendations.

- The predict_ratings function is responsible for estimating ratings for movies that users have not yet rated, utilizing the similarities between users to inform these predictions. It takes in three primary inputs:

  * A dictionary containing the training dataset of users and their normalized movie ratings (**user_ratings**)
  * A dictionary that holds the similarity scores between each pair of users (**user_similarity**)
  * A float value stating the minimum similarity required for another user's rating to influence the prediction (**similarity_threshold**)

  The function operates by iterating through each user in the training dataset and identifying movies that the user has not rated. For each unrated movie, it searches for other users who have rated the same movie and whose similarity score with the target user meets or exceeds the specified threshold. It then calculates the predicted rating for the unrated movie as a weighted average of these similar user's ratings, where the weights are the similarity scores themselves. This predicted rating is normalized to ensure it falls within the predefined range. The predictions are stored in a nested dictionary structure (**predicted_ratings**) where each user ID maps to another dictionary of movie IDs and their corresponding predicted ratings. Additionally, the function provides real-time feedback through a progress bar, indicating the advancement of the prediction process. Upon completion, it returns the **predicted_ratings** dictionary.

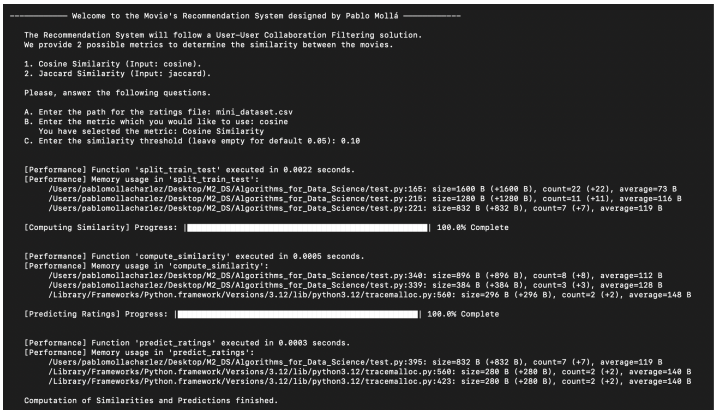

Figure 6: Finished Computations: Similarity & Predictions

As mentioned, throughout these processes, the system displays performance metrics, including execution time and memory usage for each function, ensuring transparency and aiding in performance evaluation. Then, the system presents four evaluation metrics to assess the quality of the predictions and asks the user to select one, including Mean Absolute Error (MAE), Root Mean Square Error (RSME), Precision, Recall or all at once.

Figure 7: Evaluation Metric

The user selects an evaluation metric by typing the corresponding input. For the cases **MAE** and **RSME**, the system confirms the selected evaluation metric, displays the calculated value and terminates after providing it. While the primary workflow concludes at the previous step, the program includes additional functionalities that become accessible based on the user's evaluation metric choice. These steps involve deeper insights into **Precision** and **Recall** metrics or **All** (which ultimately returns all metrics), and need further questions to be answered.

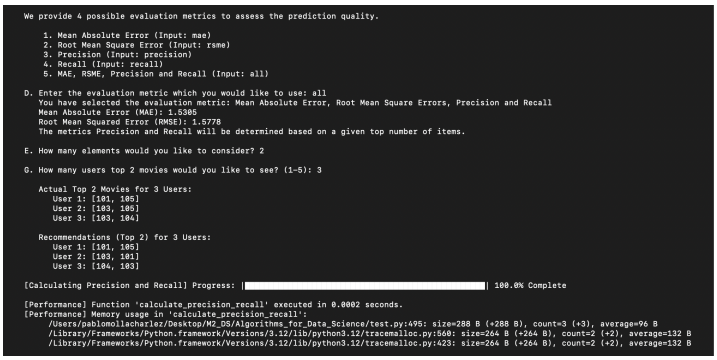

Figure 8: Actual vs. Recommended Ratings

For instance, by selecting the option **all**, the system prompts the user to specify how many top elements (movies) to consider when calculating precision and recall. The default value is 3 if no input is provided. Immediately after, the system allows the user to visualize the first user's actual top movies (number previously selected) against the recommendations (predictions) blended with the training dataset ratings (the true ratings are not included, just training and the predictions made). Finally, the system allows the user to select up to 5 specific users to view their precision and/or recall metrics in detail, and displaying the average precision and recall across all users to provide an overall performance measure.

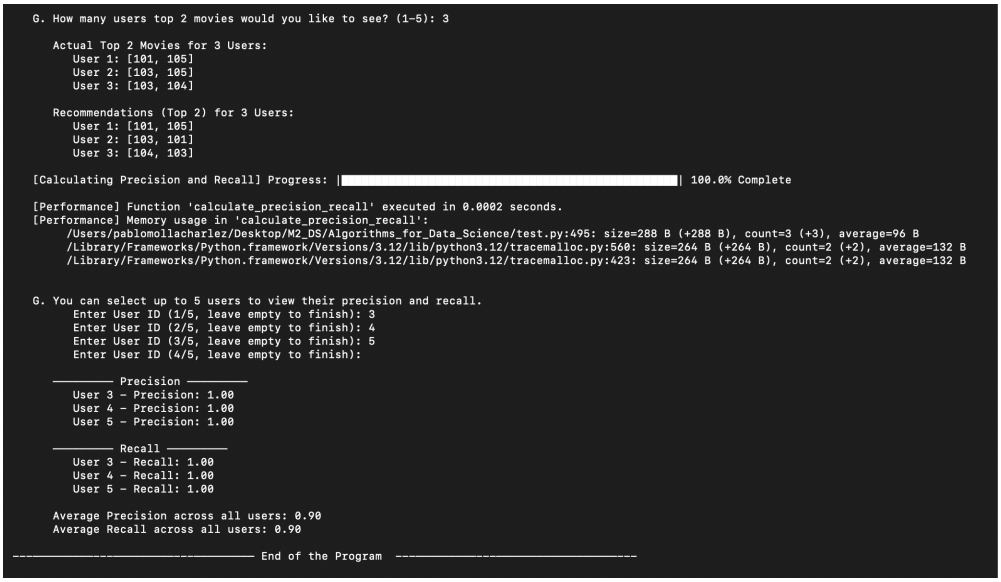

Figure 9: End of Program

2.3.2 Code Modules

As previously mentioned, the code is separated into multiple sections which include:

- **User vs. System Interactions**
- **Normalization & Denormalization**
- **Printing functions for user**
- **Data loading & Train/Test Split**
- **Similarity Metrics & Computations, Predictions and Recommendations**
- **Evaluation Metrics: MAE, RSME, Precision and Recall**

2.3.2.1 User vs. System Interactions

This module handles all interactions between the user and the system, including selecting similarity and evaluation metrics, setting thresholds, and choosing specific users for detailed performance metrics. It ensures that user inputs are validated and appropriately processed to customize the recommendation system's behavior. The module includes the following functions:

- **get_metric()** prompts the user to choose a similarity metric—either cosine or Jaccard similarity—for determining the relationship between movies. It continuously requests input until a valid option is provided, then returns both the metric identifier and its descriptive name.

```
def get_metric():
    metrics = ["cosine", "jaccard"]
    while True:
        metric_name = input("    B. Enter the metric which you would like to use: ").strip().lower()
        if metric_name in metrics:
            metric = {"cosine": "Cosine Similarity", "jaccard": "Jaccard Similarity"}[metric_name]
            print(f"       You have selected the metric: {metric}")
            return metric_name, metric
        else:
            print("       Metric not found. Please try again.")
```

- **get_eval_metric()** asks the user to select an evaluation metric to assess the recommendation system's performance. The available options include Mean Absolute Error (MAE), Root Mean Square Error (RMSE), Precision, Recall, or all metrics combined. The function validates the user's choice and returns the selected metric identifier.

```
def get_eval_metric():
    metrics = ["mae", "rsme", "precision", "recall", "all"]
    print("    We provide 4 possible evaluation metrics to assess the prediction quality.\n")
    print("        1. Mean Absolute Error (Input: mae)")
    print("        2. Root Mean Square Error (Input: rsme)")
    print("        3. Precision (Input: precision)")
    print("        4. Recall (Input: recall)")
    print("        5. MAE, RSME, Precision and Recall (Input: all)")
    while True:
        metric_name = input("\n    D. Enter the evaluation metric which you would like to use: ").strip().lower()
        if metric_name in metrics:
            metric = {"mae": "Mean Absolute Error", "rsme": "Root Mean Square Error",
                      "precision": "Precision", "recall": "Recall",
                      "all": "Mean Absolute Error, Root Mean Square Errors, Precision and Recall"
                     }[metric_name]
            print(f"       You have selected the evaluation metric: {metric}")
            return metric_name
        else:
            print("       Metric not found. Please try again.")
```

- **get_top_n()** determines the number of top-rated movies to consider when calculating precision and recall. It prompts the user to enter a number between 2 and 5, defaulting to 3 if no input is provided. The function ensures the input is within the specified range before returning the chosen value.

```
def get_top_n():
    print("       The metrics Precision and Recall will be determined based on a given top number of items.")
    while True:
        top_n_input = input("\n    E. How many elements would you like to consider? ").strip()
        # If the input is empty, the default value is 3
        if not top_n_input:
            return 3
        # Try converting the input to a int
        try:
            top_n = int(top_n_input)
            if 2 <= top_n <= 5:
                return top_n
            else:
                print("       The number must be between 2 and 5. Please try again.")
        except ValueError:
            print("       Invalid input. Please enter a numerical value for the number.")
```

- **get_similarity(user_similarity, user1, user2)** is a utility function that retrieves the similarity score between two specified users from the **user_similarity** dictionary. If no similarity score exists for the given user pair, it defaults to returning 0.0.

```
def get_similarity(user_similarity, user1, user2):
    return user_similarity.get(user1, {}).get(user2, 0.0)
```

- **get_similarity_threshold()** retrieves the similarity threshold from the user, which dictates the minimum similarity score required for another user's rating to influence the prediction. The user can input a value between 0 and 1 or leave it empty to accept the default threshold of 0.05. The function validates the input and returns the threshold value.

```
def get_similarity_threshold():
    while True:
        similarity_threshold_input = input(
                "    C. Enter the similarity threshold (leave empty for default 0.05): "
                                           )
        # If the input is empty, use the default value
        if not similarity_threshold_input:
            print() # Skipping a line for esthetics in terminal
            return 0.05
        # Try converting the input to a float
        try:
            similarity_threshold = float(similarity_threshold_input)
            if 0 <= similarity_threshold <= 1:
                print() # Skipping a line for esthetics in terminal
                return similarity_threshold
            else:
                print("       Threshold must be between 0 and 1. Please try again.")
        except ValueError:
            print("       Invalid input. Please enter a numerical value for the threshold.")
```

- **get_nb_users_precision_recall(precision_recall, metric_type='precision', max_users=5)** allows the user to select up to a specified number of users (default is five) to view their precision or recall metrics. It prompts the user to enter valid user IDs, ensuring each selection is unique and exists within the provided **precision_recall** dictionary. The function returns a list of the selected user IDs, which later on will allow the system to apply the formulas.

```
def get_nb_users_precision_recall(precision_recall, metric_type='precision', max_users=5):
    selected_users = []
    print(f"\n    G. You can select up to {max_users} users to view their {metric_type}.")
    while len(selected_users) < max_users:
        user_input = input(
            f"          Enter User ID ({len(selected_users)+1}/{max_users}, leave empty to finish): "
                           ).strip()
        if not user_input:
            break
        try:
            user_id = int(user_input)
            if user_id in precision_recall:
                if user_id not in selected_users:
                    selected_users.append(user_id)
                else:
                    print("          User already selected. Choose a different user.")
            else:
                print("          User ID not found. Please enter a valid User ID.")
        except ValueError:
            print("          Invalid input. Please enter a numerical User ID.")

    return selected_users
```

- **get_top_n_actual(all_ratings, top_n=5)** extracts the top **N** rated movies for each user from the complete dataset. It processes the **all_ratings** list to compile a dictionary where each user ID maps to their highest-rated movie IDs, sorted in descending order of rating. This function is essential for evaluating the accuracy of the recommendation system by comparing actual top-rated movies against the system's recommendations.

```
def get_top_n_actual(all_ratings, top_n=5):
    user_actual_ratings = defaultdict(list)
    for user_id, movie_id, rating in all_ratings:
        user_actual_ratings[user_id].append((movie_id, rating))

    # Sort the movies for each user by rating in descending order and take top N
    actual_top_n = {}
    for user, movies in user_actual_ratings.items():
        sorted_movies = sorted(movies, key=lambda x: x[1], reverse=True)
        top_movies = [movie for movie, rating in sorted_movies[:top_n]]
        actual_top_n[user] = top_movies

    return actual_top_n
```

The module plays a crucial role in tailoring the recommendation system to user preferences. By managing metric selections, threshold settings, and specific user inquiries, it ensures that the system operates according to the user's desired configurations. Each function within this module is designed to validate and process user inputs efficiently, thereby enhancing the overall user experience and the system's adaptability.

2.3.2.2 Normalization & Denormalization

This module is essential for scaling movie ratings to a standardized range, facilitating consistent computations throughout the recommendation system. The module provides utility functions to scale movie ratings between a defined minimum and maximum range. Normalizing ratings ensures that all values are on a consistent scale, which is crucial for accurate similarity calculations and predictions. Conversely, denormalizing allows the system to revert normalized ratings back to their original scale for meaningful interpretation and evaluation. The module includes the following functions:

- **normalize(rating)** takes a raw movie rating as input and scales it to a normalized value between 0 and 1. It does this by subtracting the global minimum rating (GLOBAL_MIN_RATING) from the input rating and then dividing by the range between the global maximum and minimum ratings (GLOBAL_MAX_RATING - GLOBAL_MIN_RATING). This function ensures that all ratings are proportionally adjusted within the standardized range, facilitating uniform processing in subsequent computations.

```
def normalize(rating):
    return (rating - GLOBAL_MIN_RATING) / (GLOBAL_MAX_RATING - GLOBAL_MIN_RATING)
```

- **denormalize(normalized_rating)** performs the inverse operation of the **normalize** function. It accepts a normalized rating (a float between 0 and 1) and converts it back to its original scale by multiplying by the range (GLOBAL_MAX_RATING - GLOBAL_MIN_RATING) and then adding the global minimum rating (GLOBAL_MIN_RATING). This function is vital for interpreting the normalized predictions in their original context, making the results meaningful and comparable to the initial dataset.

```
def denormalize(normalized_rating):
    return normalized_rating * (GLOBAL_MAX_RATING - GLOBAL_MIN_RATING) + GLOBAL_MIN_RATING
```

By standardizing ratings through normalization, it ensures consistency and accuracy in similarity assessments and predictive algorithms. Denormalization complements this process by translating the standardized results back into their original scale, enabling users to understand and evaluate the recommendations effectively.

2.3.2.3 Testing and Debugging Strategies

The module includes utility functions that facilitate the display of progress indicators and the presentation of precision and recall metrics to the user. These functions ensure that the user receives real-time feedback during lengthy computations and can easily interpret the evaluation results of the recommendation system. The module defines the following functions:

- **print_progress_bar(iteration, total, function_name, length=50)** serves to visually represent the progress of ongoing processes within the system. It takes the current iteration count, the total number of iterations, the name of the function being executed, and an optional length parameter to define the width of the progress bar. By dynamically updating the progress bar in the console, this function provides users with a clear indication of how much of a task has been completed and how much remains, enhancing the transparency and responsiveness of the system during operations such as similarity computations and rating predictions.

```
def print_progress_bar(iteration, total, function_name, length=50):
    percent = f"{100 * (iteration / float(total)):.1f}"
    filled_length = int(length * iteration // total)
    bar = "" * filled_length + "-" * (length - filled_length)

    # Displaying the progress bar with the function name
    sys.stdout.write(f"\r    [{function_name}] Progress: |{bar}| {percent}% Complete")
    sys.stdout.flush()
```

- **print_precision_selected(precision_recall, selected_users)** is designed to display the precision metrics for a subset of users chosen by the user. It accepts a dictionary containing precision and recall values for each user (**precision_recall**) and a list of user IDs (**selected_users**). The function iterates through the selected users and prints their respective precision scores in a formatted manner. This targeted display allows users to focus on specific individual's performance metrics, facilitating a deeper understanding of how well the recommendation system is performing for those particular users.

```
def print_precision_selected(precision_recall, selected_users):
    print("\n       --------- Precision ---------")
    for user in selected_users:
        precision = precision_recall[user]['precision']
        print(f"          User {user} - Precision: {precision:.2f}")
```

- **print_recall_selected(precision_recall, selected_users)** functions similarly to **print_precision_selected** but focuses on displaying recall metrics instead. It takes the same inputs a dictionary of precision and recall values and a list of selected user IDs and prints the recall scores for each chosen user. By providing recall metrics alongside precision, this function offers a comprehensive view of the recommendation system's effectiveness, allowing users to assess both the relevance and completeness of the recommendations provided to specific users.

```
def print_recall_selected(precision_recall, selected_users):
    print("\n       ||||||||| Recall |||||||||")
    for user in selected_users:
        recall = precision_recall[user]['recall']
        print(f"          User {user} - Recall: {recall:.2f}")
```

2.3.2.4 Data loading & Train/Test Split

The **split_train_test** function is essential in the implementation. It is designed to partition a dataset of movie ratings into training and testing sets, ensuring that the test set comprises ratings from users with sufficient interaction and commonality with others. The function begins by initializing the random seed for reproducibility. It then loads all ratings from the specified CSV file, parsing each entry to extract **user_id**, **movie_id**, and **rating**. During this loading process, it constructs two mappings, **movie_to_users** which maps each movie to the set of users who have rated it and **user_to_movies** which as well maps each user to the set of movies they have rated. To identify eligible ratings for the test set, the function applies two criteria:

- A movie must be rated by at least a minimum number of other users (**min_users**).
- Those other users must share a minimum number of common movies (**min_common_movies**) with the current user, excluding the movie in question.

Ratings that meet both criteria are collected into the **eligible_ratings** list. This list is then shuffled randomly to ensure an unbiased selection process. The function calculates the number of test ratings based on the specified **test_ratio** (throughout the whole implementation we use a **20% test set** and **80%** train set) and selects the corresponding portion from the shuffled **eligible_ratings** to form the test set (**test_ratings_list**). The remaining ratings constitute the training set (**train_ratings_list**).

Both training and testing ratings are then normalized to a consistent scale between 0 and 1 using the **normalize** function, ensuring that all ratings fall within the defined global minimum and maximum ratings (GLOBAL_MIN_RATING and GLOBAL_MAX_RATING). These normalized ratings are organized into nested dictionaries: **train_ratings** and **test_ratings**, where each user ID maps to another dictionary of movie IDs and their corresponding normalized ratings. Finally, the function returns three objects:

- The normalized training dataset as the variable **train_ratings**.
- The normalized testing dataset as **test_ratings**.
- The complete list of all ratings loaded from the CSV file **all_ratings**.

This structured approach ensures that the training and testing sets are both representative and suitable for evaluating the performance of the recommendation system, and by returning well-structured dictionaries for both training and testing datasets alongside the complete ratings list, the function lays a robust foundation for subsequent similarity computations and rating predictions within the recommendation system. The function can be found in the Python file.

2.3.2.5 Similarity Metrics & Computations, Predictions and Recommendations

This module is in charge of calculating similarities between users, predicting ratings for unrated movies, and generating personalised movie recommendations based on these predictions. The module includes the following functions:

- **cosine_similarity(ratings1, ratings2)** computes the cosine similarity between two user's rating vectors. Following the original formula, it calculates the dot product of the ratings, divides it by the product of their magnitudes, and returns the resulting similarity score.

```
def cosine_similarity(ratings1, ratings2):
    # Calculate the dot product of the two rating lists
    dot_product = sum([r1 * r2 for r1, r2 in zip(ratings1, ratings2)])

    # Calculate the norms (magnitudes) of each rating list
    norm1 = math.sqrt(sum([r1  2 for r1 in ratings1]))
    norm2 = math.sqrt(sum([r2  2 for r2 in ratings2]))

    # Return the cosine similarity, handling the case where norms are zero
    return dot_product / (norm1 * norm2) if norm1 and norm2 else 0.0
```

- **jaccard_similarity(vec1, vec2)** calculates the Jaccard similarity between two sets of movie ratings. It converts the rating vectors into sets, determines the size of their intersection and union, and returns the ratio of intersection size to the union size. This metric measures the similarity based on shared rated movies, avoiding issues related to rating magnitudes.

```
def jaccard_similarity(vec1, vec2):
    # Convert lists to sets
    set1 = set(vec1)
    set2 = set(vec2)

    # Calculate the intersection and union of the sets
    intersection = len(set1 & set2)
    union = len(set1 | set2)

    # Return the Jaccard similarity, avoiding division by zero
    return intersection / union if union != 0 else 0.0
```

- **compute_similarity(user_ratings, similarity_type='cosine', min_common=2)** evaluates the similarity between all pairs of users based on the specified similarity metric (**cosine** or **jaccard**). It iterates through each user pair, identifies common rated movies, and computes their similarity if they share at least **min_common** movies. The results are stored in a nested dictionary structure, facilitating quick lookup of similar scores between any two users (in both directions). A progress bar provides real-time feedback on the computation's advancement.

```
def compute_similarity(user_ratings, similarity_type='cosine', min_common=2):
        # Creating empty dictionary
        user_similarity = defaultdict(dict)
        # Calculating total number of comparisons (pairs of users) for the progress bar
        n = len(user_ratings)
        total_comparisons = n * (n - 1) // 2  # Integer division for whole number
        current_progress = 0

        # Computing similarity for each pair of users
        users = list(user_ratings.keys())
        for i in range(len(users)):
            user1 = users[i]
            for j in range(i + 1, len(users)):
                user2 = users[j]
                # Get common movies rated by both users
                common_movies = set(user_ratings[user1]).intersection(user_ratings[user2])
                if len(common_movies) < min_common:
                    similarity = 0.0
                else:
                    ratings1 = [user_ratings[user1][movie] for movie in common_movies]
                    ratings2 = [user_ratings[user2][movie] for movie in common_movies]
                    # Applying similarity based on the specified type
                    if similarity_type == 'cosine':
                        similarity = cosine_similarity(ratings1, ratings2)
                    elif similarity_type == 'jaccard':
                        similarity = jaccard_similarity(ratings1, ratings2)
                    else:
                        similarity = 0.0
                # Store the similarity score in both directions
                user_similarity[user1][user2] = similarity
                user_similarity[user2][user1] = similarity
                # Update progress bar
                current_progress += 1
                print_progress_bar(current_progress, total_comparisons, "Computing Similarity")

        return user_similarity
```

- **predict_ratings(user_ratings, user_similarity, similarity_threshold)** estimates ratings for movies that users have not yet rated by leveraging the similarity scores between users. For each unrated movie, it identifies similar users whose similarity exceeds the **similarity_threshold** and have rated the movie. It then calculates a weighed average of these ratings based on their similarity scores, normalizes the predicted rating, and stores it in a nested dictionary. Once again, a progress bar indicates the prediction process's progress, corresponding this time to the function that usually takes the longest times to process the data.

```
def predict_ratings(user_ratings, user_similarity, similarity_threshold):
    predicted_ratings = defaultdict(dict)
    # Calculating total number of users and movies to predict for the progress bar
    total_predictions = sum(
        len({movie for other_user in user_ratings for movie in user_ratings[other_user]
        if movie not in user_ratings[user]})
        for user in user_ratings
    )
    current_progress = 0

    for user in user_ratings:
        # Movies that this user hasn't rated
        unrated_movies = {movie for other_user in user_ratings for movie in user_ratings[other_user]
            if movie not in user_ratings[user]
        }
        for movie in unrated_movies:
            # Finding similar users
            similar_users = []
            for other_user in user_ratings:
                if other_user == user:
                    continue
                sim = get_similarity(user_similarity, user, other_user)
                if sim >= similarity_threshold and movie in user_ratings[other_user]:
                    similar_users.append((sim, other_user))
            if similar_users:
                numerator = sum(sim * user_ratings[other_user][movie] for sim, other_user in similar_users)
                denominator = sum(sim for sim, _ in similar_users)
                predicted_rating = numerator / denominator if denominator != 0 else 0
            else:
                predicted_rating = 0.0
            # Clipping the predicted rating to the range [0, 1] because we normalized ratings beforehand
            predicted_rating = min(max(predicted_rating, 0), 1)
            predicted_ratings[user][movie] = predicted_rating
            # Update progress bar
            current_progress += 1
            print_progress_bar(current_progress, total_predictions, "Predicting Ratings")
    return predicted_ratings
```

- **recommend_movies(train_ratings, predicted_ratings, top_n=5)** generates personalized movie recommendations for each user by combining their actual rated movies with the predicted ratings. It sorts the combined list in descending order of ratings and selects the top **N** movies as recommendations. The recommendations are set in a dictionary where each user ID maps to their list of top recommended movies.

```
def recommend_movies(train_ratings, predicted_ratings, top_n=5):
    recommendations = {}
    for user in train_ratings.keys():
        rated_movies = train_ratings[user] # Get rated movies with actual ratings
        predicted = predicted_ratings.get(user, {}) # Get predicted ratings
        # Combination into a list of tuples (movie_id, rating)
        combined = list(rated_movies.items()) + list(predicted.items())
        sorted_combined = sorted(combined, key=lambda x: x[1], reverse=True) # Sort by rating descending
        top_movies = sorted_combined[:top_n] # Select top N movies
        recommendations[user] = top_movies # Store recommendations
    return recommendations
```

2.3.2.6 Evaluation Metrics: MAE, RSME, Precision and Recall Computations

This module focuses on evaluating the accuracy and effectiveness of the recommendation system by calculating various performance metrics, including Mean Absolute Error (MAE), Root Mean Square Error (RMSE), Precision, and Recall. The functions within the module are:

- **calculate_mae_rmse(test_ratings, predicted_ratings)**computes the Mean Absolute Error (MAE) and Root Mean Square Error (RMSE) between the actual ratings in the test set and the predicted ratings generated by the system. It iterates through each user and their rated movies, calculates the absolute and squared differences between actual and predicted ratings, sums these errors, and then computes the average MAE and RMSE. These metrics provide insights into the prediction accuracy, with lower values indicating better performance.

```
def calculate_mae_rmse(test_ratings, predicted_ratings):
    total_error_mae = 0
    total_error_rmse = 0
    total_count = 0

    for user_id, movies in test_ratings.items():
        for movie_id, actual_norm in movies.items():
            actual = denormalize(actual_norm)
            predicted_norm = predicted_ratings.get(user_id, {}).get(movie_id, 0.0)
            predicted = denormalize(predicted_norm)
            error = abs(actual - predicted)
            total_error_mae += error
            total_error_rmse += error  2
            total_count += 1

    mae = total_error_mae / total_count if total_count != 0 else 0.0
    rmse = math.sqrt(total_error_rmse / total_count) if total_count != 0 else 0.0
    return mae, rmse
```

- **calculate_precision_recall(actual_top_n, recommended_top_n)** assesses the recommendation system's effectiveness by calculating precision and recall for each user. For each user, it determines the number of true positive recommendations—movies that appear in both the actual top **N** and the recommended top **N** lists. Precision is calculated as the ratio of true positives to the total recommended movies, while recall is the ratio of true positives to the total actual top movies. The results are stored in a dictionary mapping each user to their precision and recall scores. A progress bar provides real-time feedback during the computation process.

```
def calculate_precision_recall(actual_top_n, recommended_top_n):
    precision_recall = {}
    total_users = len(actual_top_n)
    current_iteration = 0
    for user, actual_movies in actual_top_n.items():
        recommended_movies = recommended_top_n.get(user, [])
        # Calculate true positives: movies that are both in actual and recommended top N
        true_positives = len(set(actual_movies) & set(recommended_movies))
        # Calculate precision and recall, avoiding division by zero
        precision = true_positives / len(recommended_movies) if recommended_movies else 0
        recall = true_positives / len(actual_movies) if actual_movies else 0
        # Store the precision and recall for the current user
        precision_recall[user] = {'precision': precision, 'recall': recall}
        # Update progress bar
        current_iteration += 1
        print_progress_bar(current_iteration, total_users, "Calculating Precision and Recall")
    print()
    return precision_recall
```

2.3.2.7 Time & Memory Usage Monitoring

This module is dedicated to monitoring and reporting the performance of various functions within the system, specifically tracking their execution time and memory usage. It employs a decorator to seamlessly integrate performance tracking into any function. The module contains a single function, **performance_monitor(func)**, which is a decorator designed to measure both the execution time and memory usage of the function it wraps. When applied to a function, it initiates memory tracking and starts a timer before the function executes. After the function completes, it captures a memory snapshot and calculates the differences to determine memory usage. It then prints out the execution time, the top three memory-consuming lines within the function, and the cumulative memory usage of these top three lines as an approximation. This additional aggregation provides a quick overview of the most significant memory allocations, aiding in the identification of potential optimization areas. This decorator allowed me to easily monitor and optimize the performance of critical functions without altering their core logic.

```
def performance_monitor(func):
    @functools.wraps(func)
    def wrapper_performance_monitor(*args, kwargs):
        tracemalloc.start() # Start memory tracking
        snapshot1 = tracemalloc.take_snapshot()
        start_time = time.perf_counter() # Start timing
        try:
            # Execute the function
            result = func(*args, kwargs)
        except Exception as e:
            tracemalloc.stop()
            print(f"\n    [Error] Function '{func.__name__}' raised an exception: {e}")
            raise

        end_time = time.perf_counter() # End timing
        elapsed_time = end_time - start_time # Time difference

        snapshot2 = tracemalloc.take_snapshot()
        tracemalloc.stop() # End memory tracking
        top_stats = snapshot2.compare_to(snapshot1, 'lineno') # Calculate memory usage

        # Display results
        print(f"\n    [Performance] Function '{func.__name__}' executed in {elapsed_time:.4f} seconds.")
        print(f"    [Performance] Memory usage in '{func.__name__}':")

        sizes = [] # List to store sizes of top 3 memory-consuming lines
        # Print top 3 memory-consuming lines and collect their sizes
        for stat in top_stats[:3]:  # Show top 3 memory-consuming lines
            print("        ", stat)
            sizes.append(stat.size)
        # Calculate the sum of the top 3 sizes
        total_size = sum(sizes)
        # Create a string that joins the sizes with a plus sign
        sizes_sum = " + ".join(map(str, sizes))
        # Print the sum of the top 3 memory usages
        print(f"        Approx Memory Usage: {sizes_sum} = {total_size} B\n")
        return result
    return wrapper_performance_monitor
```

2.3.2.8 Main Execution Function

This module serves as the entry point of the recommendation system, orchestrating the workflow by integrating all other modules. It handles user inputs, data processing, similarity computations, rating predictions, recommendation generation, and evaluation metric calculations.

The function **main()** orchestrates the entire recommendation process. It begins by greeting the user and

prompting them to input the path to the ratings file. It then guides the user through selecting a similarity metric and setting a similarity threshold. The function proceeds to split the dataset into training and testing sets using the **split_train_test** function, computes user similarities with compute_similarity, and predicts ratings with predict_ratings. After generating recommendations via **recommend_movies**, it prompts the user to choose an evaluation metric (MAE, RMSE, Precision, Recall, or all) and calculates the corresponding metrics using **calculate_mae_rmse** and **calculate_precision_recall**. Depending on the selected evaluation metric, it may further prompt the user to specify additional parameters, such as the number of top items to consider or specific users to inspect. Throughout the process, progress bars and performance metrics are displayed to keep the user informed. Finally, the function outputs the evaluation results and signals the end of the program. The function can be found in the code implementation.

2.3.3 Use of Libraries and External Tools

The implementation leverages several Python libraries and external tools to facilitate efficient development and execution:

- **Standard Libraries**:
  - **csv**: For reading and parsing the ratings dataset.
  - **collections.defaultdict**: Enables efficient storage and retrieval of user and movie mappings.
  - **math**: Provides mathematical functions, including those necessary for similarity calculations.
  - **os** and **sys**: Handle file system operations and system-specific parameters.
  - **random**: Facilitates data shuffling to ensure randomness in train-test splitting.
- **Performance Measurement**:
  - **time**: Measures execution time of functions to assess performance.
  - **tracemalloc**: Tracks memory allocations, enabling analysis of memory usage patterns.
  - **functools**: Supports the creation of decorators like **performance_monitor** by preserving function metadata.
- **Code Presentation**:
  - **minted**: Provides syntax-highlighted code listings within the report.

These libraries were selected for their efficiency, reliability, and suitability for handling data processing, algorithm implementation, and performance monitoring tasks essential to the recommendation system.

2.4 Challenges

2.4.1 Interpreting Predicted Data Ratings

One significant challenge encountered was the interpretation of predicted ratings, which occasionally yielded results that were considered counterintuitive. For instance, the system predicted a rating of 5 for **Movie 1** by **User 5412**, despite no other user having rated **Movie 1** that highly. This anomaly arises from the prediction formula:

$$\text{Predicted Rating}_{k,i} = \frac{\sum_{\text{all } j}(\text{similarity}(i,j) \times \text{rating}_{k,j})}{\sum_{\text{all } j}|\text{similarity}(i,j)|}$$

In this scenario, the high similarity score between **Movie 1** and **Movie 2** (0.8437) heavily influences the prediction. Since **User 5412** rated **Movie 2** as 5, the weighted average calculation results in a predicted rating of 5 for **Movie 1**, despite limited data points. This outcome highlights the system's reliance on high

similarity weights, which can disproportionately affect predictions when data is sparse, which is the case when using mini_dataset1.csv and mini_dataset2.csv. To mitigate such issues, I implemented techniques like rating normalization or ensuring a minimum number of contributing ratings which can enhance prediction reliability and prevent unrealistic rating outcomes.

2.4.2 Similarity Calculations with Limited Overlapping Data

Another challenge involved the computation of similarity scores between users who had minimal overlapping ratings. Specifically, when two users share only one common rated movie, the cosine similarity inherently calculates a similarity score of 1.0 if both ratings are positive, regardless of the actual rating values. For example, consider **User 2** and **User 3** who both rated **Movie 101**:

- User 2 Ratings: {101: 4.0, 102: 1.0, 103: 5.0}
- User 3 Ratings: {101: 2.0, 104: 4.0, 102: 3.0}

Their only common movie is **Movie 101** with ratings 4.0 and 2.0, respectively. Applying the cosine similarity formula:

$$\textbf{Similarity(User2, User3)} = \frac{(0.75 \times 0.25)}{(\sqrt{0.75^2} \times \sqrt{0.25^2})} = 1.0$$

Mathematically, this **result is accurate**; however, it does not reflect a meaningful similarity in practical terms. A similarity score of 1.0 suggests perfect alignment, which is misleading when based on a single data point. This limitation underscores the necessity of enforcing a minimum number of common rated movies to ensure that similarity scores are indicative of genuine user alignment. By setting a threshold for the minimum number of overlapping ratings, the system can avoid inflated similarity scores that do not accurately represent user preferences, thereby enhancing the quality and reliability of recommendations. This realization lead me to implement the **similarity_threshold** as an essential parameter for the **predict_ratings** function and logic behind the recommendation system.

2.4.3 Perfect Evaluation Metrics: MAE and RSME

When executing the code with incomplete subdatasets such as the original **train_ratings.csv** with just the first 1000 rows, we obtain **MAE** and **RMSE** values of 0.0 in the recommendation system. The highly likely possible reasons behind those results are:

- **Small and Sparse Dataset:** With only **1000 ratings**, many users have only one rating in the dataset. Most users appear once, resulting in insufficient overlapping ratings among users to compute meaningful similarities.
- **Train-Test Split Criteria:** The **split_train_test** function requires each movie in the test set to be rated by at least 2 users and have at least 2 common movies with other users. Given the dataset's sparsity, few or no ratings meet these criteria, leading to an empty or extremely small test set.
- **Predictions on Non-Existent or Trivially Simple Test Set:** If the test set is empty, the MAE and RMSE calculations default to 0 because there are no discrepancies between actual and predicted ratings. In cases where a test set exists but contains only a single rating per user, the recommendation system might trivially predict the exact rating, resulting in zero errors.

The previous reasonings lead to two main conlusions, first the zero **MAE** and **RMSE** falsely indicate perfect model performance, which isn't reflective of the model's true predictive capabilities. Secondly, without a robust test set, it's impossible to assess how well the model generalizes to unseen data.

2.4.4 Similar Evaluation Metrics: Precision and Recall

In the following sections, it can be noticed that the evaluation metrics **Precision** and **Recall** have very similar or almost identical values when executing the recommendation system on the small datasets mini_dataset1.csv and mini_dataset2.csv. This is due to the fact that in small datasets like those, the limited number of data points causes similarity metrics to produce comparable scores and similar recommendation lists.

Additionally, the restricted pool of items increases the likelihood that the recommended movies closely match the actual relevant items, resulting in consistent Precision and Recall values. However, this trend is not followed when running the recomendation system on the original (or sub-datasets from it) **train_ratings.csv** dataset.

3 Experimental Analysis

These experiments are designed to evaluate the performance and impact of various parameters of the system using the smaller datasets (mini_dataset1.csv and mini_dataset2.csv) and multiple subdatasets from the original **train_ratings.csv**.

3.1 Performance Metrics

3.1.1 Execution Time & Memory Usage: Cosine Similarity

The primary performance metric analyzed is the execution time of key functions such as **split_train_test**, **compute_similarity**, **predict_ratings**, and **calculate_precision_recall** to evaluate the recommendation system's scalability and efficiency across various dataset sizes.

We conducted multiple experiments using small datasets (mini_dataset1.csv and mini_dataset2.csv) and progressively larger subsets from the original **train_ratings.csv**, including the first **1,000**, **10,000**, **100,000**, **500,000** rows, and the **complete train_ratinsgs dataset**.

As anticipated, **increasing the number of rows led to longer execution times and more consistent evaluation metrics**. Concurrently, we assessed the memory consumption of these critical functions to identify potential bottlenecks.

For these experiments, we focused exclusively on Cosine Similarity with a fixed similarity threshold to streamline the evaluation process, and in order to evaluate the evaluation metrics **Precision** and **Recall**, we considered the top **5** recommended movies for every experiment with data from the **train_ratings.csv** and the top **3** for the experiments related to the small datasets (mini_dataset1.csv and mini_dataset2.csv). The results of these experiments are presented below.

From the table 3, we observe the total execution time for varying row counts from the **train_ratings.csv** dataset and smaller datasets. Additionally, the table includes evaluation metrics such as **MAE**, **RMSE**, **Precision** and **Recall**, to assess the recommendation system's performance and quality. The zero scores in **MAE** and **RSME** are discussed in the previous subsection **Challenges**.

Function	Dataset	Execution Time (s)	Memory Usage (KB)
split_train_test	mini_dataset1	0.0014	2.912
	mini_dataset2	0.0010	3.712
	1000r	0.0123	324.340
	10000r	0.2753	2053.832
	100000r	39.9701	13809.448
	500000r	2354.4748	65757.032
	Complete	10188.5095	104820.868
compute_similarity	mini_dataset1	0.0001	1.576
	mini_dataset2	0.002	1.576
	1000r	2.0036	31039.912
	10000r	42.0986	553146.400
	100000r	139.2160	1770799.480
	500000r	332.4478	2003654.952
	Complete	389.3004	2105001.336
predict_ratings	mini_dataset1	0.0002	1.840
	mini_dataset2	0.0001	1.392
	1000r	42.7433	31040.176
	10000r	2330.3002	276826.752
	100000r	9132.9596	1056458.480
	500000r	15006.8767	1394842.056
	Complete	17468.1702	1404399.400
calculate_precision_recall	mini_dataset1	0.0003	0.816
	mini_dataset2.	0.0001	0.816
	1000r	0.0056	229.904
	10000r	0.0252	1017.016
	100000r	0.0483	1679.968
	500000r	0.0514	1696.208
	Complete	0.0622	1696.208

Table 3: Execution Time and Memory Usage with Cosine Similarity and $T = 0.05$

Experiment	Dataset	Execution Time	Memory Usage (MB)	MAE	RMSE	Precision	Recall
Experiment 1	mini_dataset1	0.002s	0,007144	1.7902	2.1606	0.67	0.67
Experiment 2	mini_dataset2	0.0032s	0,007944	1.5305	1.5778	0.93	0.93
Experiment 3	1000r	44.7648s	63	0	0	0.24	1.0
Experiment 4	10000r	39min 33s	833	0	0	0.44	0.98
Experiment 5	100000r	2h 35min	2843	0.9485	1.2528	0.45	0.52
Experiment 6	500000r	4h 55min	3466	0.7844	0.9829	0.72	0.72
Experiment 7	Complete	7h 47min	3616	0.7802	0.9767	0.77	0.77

Table 4: Complete Execution Time and Memory Usage with Cosine Similarity and $T = 0.05$

3.1.2 Execution Time & Memory Usage: Jaccard Similarity

Repeating the experiments with the Jaccard Similarity metric and a similarity threshold of $T = 0.05$, we obtained the following results:

Analogously, the overall execution time and memory usage are:

Function	Dataset	Execution Time (s)	Memory Usage (KB)
split_train_test	mini_dataset1	0.0005	2.912
	mini_dataset2	0.0006	3.712
	1000r	0.0131	324.340
	10000r	0.2445	2053.832
	100000r	43.0013	13809.448
	500000r	2676.7219	65757.032
	Complete	-	-
compute_similarity	mini_dataset1	0.0001	1.576
	mini_dataset2	0.0001	1.576
	1000r	1.9817	31039.912
	10000r	42.7414	553146.400
	100000r	121.0636	1770852.424
	500000r	221.6659	2004221.088
	Complete	-	-
predict_ratings	mini_dataset1	0.0002	1.840
	mini_dataset2	0.0001	1.392
	1000r	42.7167	31040.176
	10000r	2336.2457	276812.880
	100000r	8975.2674	1034831.728
	500000r	15612.5792	1392219.448
	Complete	-	-
calculate_precision_recall	mini_dataset1	0.0001	0.816
	mini_dataset2	0.0001	0.816
	1000r	0.0055	229.904
	10000r	0.0306	1017.016
	100000r	0.0481	1679.968
	500000r	0.0542	1696.208
	Complete	-	-

Table 5: Execution Time and Memory Usage with Jaccard Similarity and $T = 0.05$

Experiment	Dataset	Execution Time	Memory Usage (MB)	MAE	RMSE	Precision	Recall
Experiment 1	mini_dataset1	0.0009s	0,007144	1.6250	2.1287	0.60	0.60
Experiment 2	mini_dataset2	0.0009s	0,007496	1.3933	1.5584	0.93	0.93
Experiment 3	1000r	44.717s	62,63433	0	0	0.24	1.0
Experiment 4	10000r	39min 39s	833,0301	0	0	0.45	0.99
Experiment 5	100000r	2h 32min	2821	1.0103	1.3468	0.45	0.53
Experiment 6	500000r	5h 9min	3464	0.7791	0.9765	0.72	0.72
Experiment 7	Complete	7h 45min	3610	0.7695	0.9704	0.76	0.76

Table 6: Complete Execution Time and Memory Usage with Jaccard Similarity and $T = 0.05$

3.1.3 Cosine vs. Jaccard Results

By analyzing the previous tables, we can deduce the following information:

- **Mean Absolute Error (MAE)** The **MAE** is higher for the mini datasets (mini_dataset1 and mini_dataset2) across both similarity measures. Cosine Similarity results in **slightly higher MAE** values than Jaccard Similarity (1.7902 vs. 1.6250 for mini_dataset1 and 1.5305 vs. 1.3933 for mini_dataset2). For larger datasets

(sub-datasets from **train_ratings**) **MAE** generally decreases as the dataset size increases for both similarity measures, with the complete dataset showing the lowest MAE (0.7802 for Cosine and 0.7695 for Jaccard). In general, Jaccard Similarity has slightly lower MAE than Cosine Similarity across all datasets, indicating that **Jaccard might be slightly more accurate in terms of absolute error**.

- **Root Mean Squared Error (RMSE)** For the mini datasets, **RMSE** follows a similar trend as **MAE**, with Cosine Similarity showing a slightly higher **RMSE** for mini_dataset1 and mini_dataset2 (2.1606 vs. 2.1287 for mini_dataset1 and 1.5778 vs. 1.5584 for mini_dataset2). In relation with, larger datasets, as dataset size increases, **RMSE** generally decreases, reaching similar values for the complete dataset (0.9767 for Cosine and 0.9704 for Jaccard). Therefore, Jaccard Similarity consistently shows marginally lower RMSE than Cosine Similarity.

- **Precision** **Precision** for mini_dataset1 is higher for both Cosine and Jaccard, with Cosine achieving 0.93, the highest **Precision** for both measures in the mini datasets. For larger datasets, **Precision** exhibits a strong correlation with dataset size, increasing with a higher number of rows, which is a logical and expected deduction. Cosine Similarity achieves a maximum **Precision** of 0.77 for the complete dataset, while Jaccard reaches 0.76. Cosine Similarity tends to have a slight edge in **Precision** over Jaccard, especially for the complete dataset, indicating that it may produce more relevant recommendations.

- **Recall** Recall for mini_dataset2 is also high in both similarity measures (0.93 for both), showing that even small datasets can yield high recall. For larger datasets, **Recall** values show variability but tend to stabilize at around $0.76 - 0.77$ for the Complete dataset (0.77 for Cosine and 0.76 for Jaccard). **Recall** is relatively comparable between the two similarity measures across all datasets, but generally they deliver practically the same the same results, although, Cosine performs slightly better.

Cosine Similarity shows slightly higher **MAE** and **RMSE** but tends to perform better in **Recall**, especially on larger datasets, while textcolormagentaJaccard Similarity generally yields lower **MAE**, **RMSE**, and slightly better **Precision**, indicating it may be more accurate and relevant in recommendations.

3.1.4 Similarity Threshold: mini_dataset1.csv vs. mini_dataset2.csv

Another essential experiment to conduct is to analyze how the different evaluation metrics are influenced by the similarity threshold. To determine **Precision** and **Recall**, we considered the top 2 recommended movies for each user. Due to the extensive execution time of approximately **7 hours and 50 minutes** when processing the complete **train_ratings.csv** dataset, we conducted our evaluations on the smaller datasets mini_dataset1.csv and mini_dataset2.csv.

Threshold	Metric	MAE	RMSE	Precision	Recall
0.05, 0.10, 0.15, 0.20, 0.30	Cosine	1.7902	2.1606	0.60	0.60
	Jaccard	1.6250	2.1287	0.60	0.60
0.40	Cosine	1.7902	2.1606	0.60	0.60
	Jaccard	2.0	2.8284	0.80	0.80
0.50	Cosine	2.0	2.2361	0.60	0.60
	Jaccard	2.0	2.8284	0.80	0.80
0.60	Cosine	2.0	2.2361	0.70	0.70
	Jaccard	2.0	2.8284	0.80	0.80
0.70 - 0.80	Cosine	2.5	2.5495	0.90	0.90
	Jaccard	2.0	2.8284	0.80	0.80
0.90	Cosine	3.0	3.1623	0.90	0.90
	Jaccard	2.0	2.8284	0.80	0.80

Table 7: Impact of Similarity Threshold on Evaluation Metrics in mini_dataset1.csv

In mini_dataset1.csv 7, increasing the similarity threshold leads to a slight rise in **MAE** and **RMSE** for both Cosine and Jaccard similarity measures, indicating a modest decline in prediction accuracy. Simultaneously, **Precision** and **Recall** metrics improve as the threshold escalates. Notably, for thresholds between 0.70 and 0.90, the Cosine Similarity metric achieves higher **Precision** and **Recall** compared to Jaccard Similarity, reflecting better performance in identifying relevant recommendations at higher thresholds.

Conversely, within the 0.40 to 0.60 range, Jaccard Similarity outperforms Cosine Similarity in these metrics, demonstrating its effectiveness in this intermediate threshold range. Below a threshold of 0.40, both similarity measures exhibit similar Precision and Recall values. These patterns suggest that the choice of similarity metric and threshold critically influences the balance between accuracy and recommendation relevance. The underlying reasons for these trends are explored in the **Challenges** section, providing plausible explanations for the observed performance variations.

Threshold	Metric	MAE	RMSE	Precision	Recall
0.05	Cosine	1.5305	1.5778	0.93	0.93
	Jaccard	1.3933	1.5584	0.93	0.93
0.10, 0.15, 0.20, 0.30, 0.40	Cosine	1.5305	1.5778	0.90	0.90
	Jaccard	1.3933	1.5584	0.80	0.80
0.50	Cosine	1.5305	1.5778	0.90	0.90
	Jaccard	1.5061	1.7255	0.80	0.80
0.60	Cosine	1.2681	1.5080	0.90	0.90
	Jaccard	1.7000	2.2913	0.80	0.80
0.70, 0.80, 0.90	Cosine	1.9724	2.3397	0.80	0.80
	Jaccard	2.5000	2.9069	0.70	0.70

Table 8: Impact of Similarity Threshold on Evaluation Metrics in mini_dataset2.csv

In mini_dataset2.csv 8, as the similarity threshold increases, both **MAE** and **RMSE** exhibit a slight upward trend for Cosine and Jaccard similarity measures, indicating a modest decline in prediction accuracy. Simultaneously, **Precision** and **Recall** metrics improve with higher thresholds, reflecting enhanced recommendation relevance. Notably, within the whole similarity threshold range, Jaccard Similarity is outperformed by Cosine Similarity in both **Precision** and **Recall**.

3.2 Visualizations

3.2.1 Execution Time vs. Number of Rows

The first plot compares the execution time obtained with the similarity metric Cosine Similarity, with a similarity threshold $T = 0.05$ with the number of rows considered from the different sub-datasets from the original **train_ratings**, mini_dataset1.csv and mini_dataset2.csv.

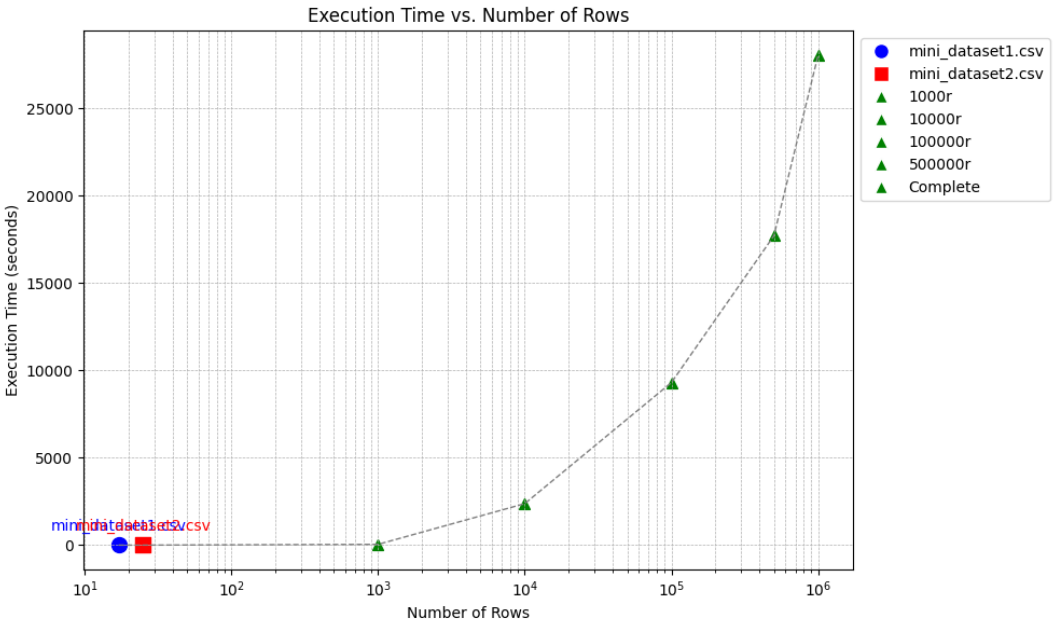

Figure 10: Execution Time vs. Number of Ratings

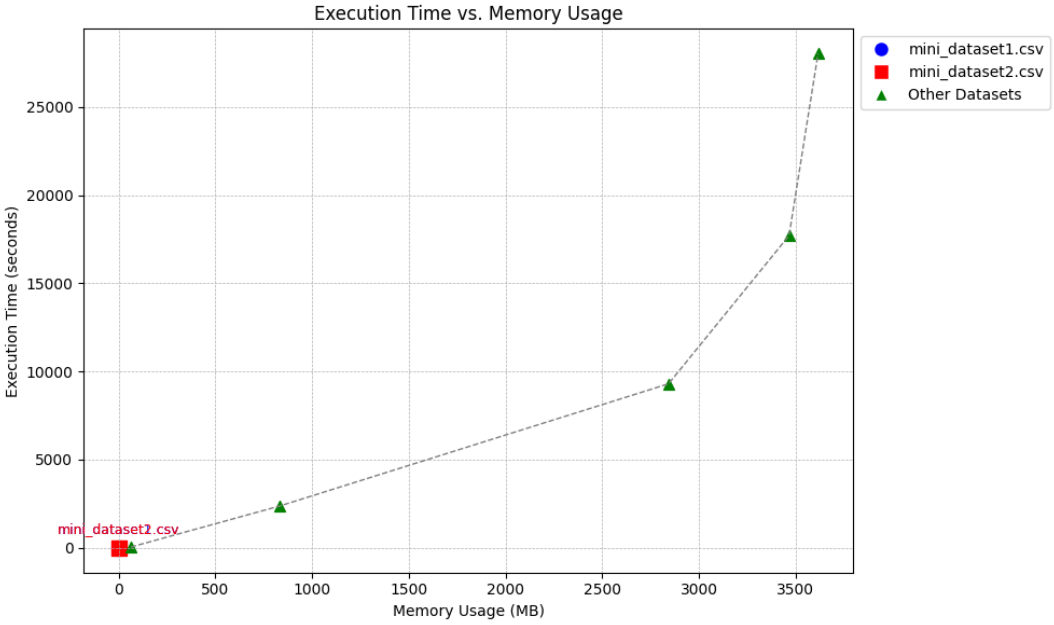

Figure 11: Execution Time vs. Memory Usage

3.2.2 Memory Usage vs. Number of Ratings

Similarly, the following graphic compares the memory usage (in MB) of the similarity metric Cosine Similarity and a similarity threshold $T = 0.05$, with the number of ratings considered from the previous datasets.

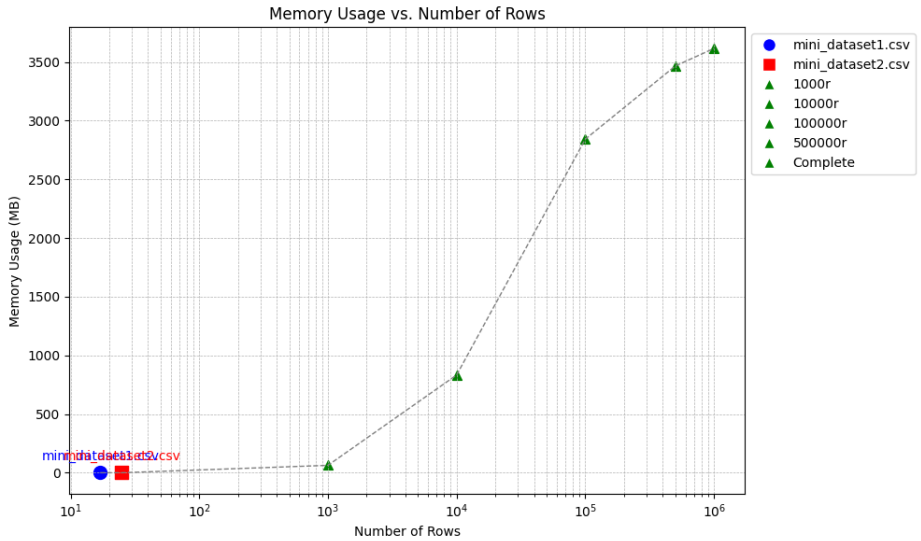

Figure 12: Memory Usage vs. Number of Rows

3.2.3 Accuracy Metrics vs. Similarity Thresholds

In the following quadruple plot, we can acknowledge the conclusions we discussed whena analyzing directly the tables 7 and 8.

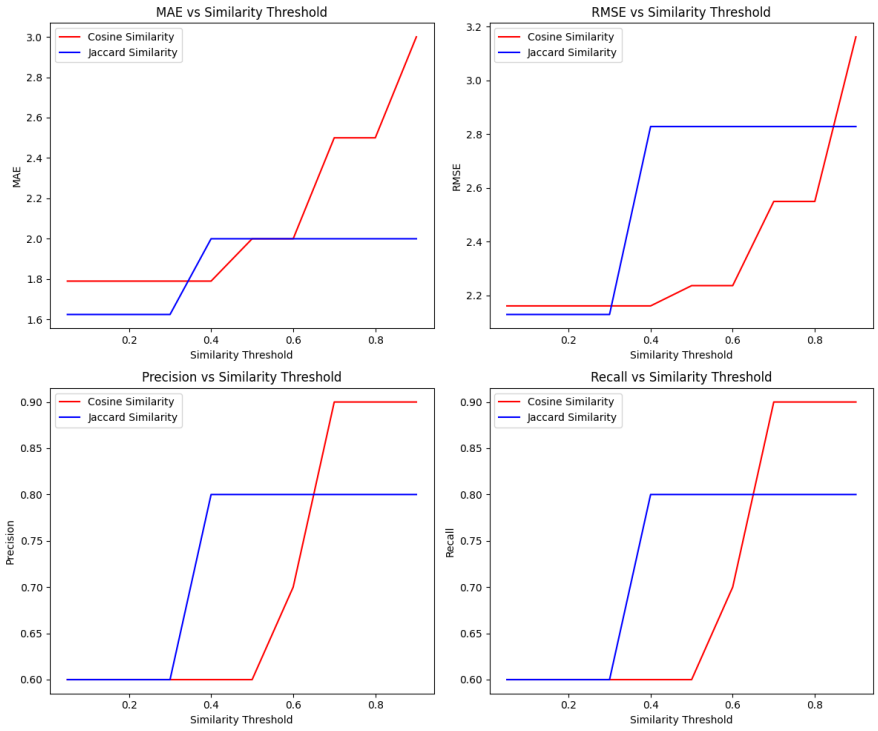

Figure 13: Accuracy Metrics vs. Similarity Threshold & Metrics

4 Conclusion

This project successfully implemented a **Movie Recommendation System** using the **User-User Collaborative Filtering** approach in **Python**. The system was designed to predict user preferences and generate personalized movie recommendations based on historical rating data. Through meticulous development and comprehensive experimental analysis, several key insights and findings emerged, highlighting both the strengths and limitations of the chosen methodologies.

1. **Similarity Metrics Performance**: Cosine Similarity consistently demonstrated superior performance in terms of **Precision** and **Recall** at higher similarity thresholds ($0.70 - 0.90$). This indicates that Cosine Similarity is more effective in identifying highly relevant recommendations when the similarity criterion is rigorous. Conversely, Jaccard Similarity outperformed Cosine Similarity within the intermediate threshold range ($0.40 - 0.60$), showcasing its effectiveness in balancing recommendation relevance and diversity. Below a threshold of 0.40, both metrics yielded comparable **Precision** and **Recall** values, suggesting similar performance when similarity requirements are relaxed. In terms of **MAE** and **RMSE**, Jaccard Similarity generally exhibited slightly lower error metrics compared to Cosine Similarity, indicating a marginally higher accuracy in prediction.

2. **Impact of Similarity Thresholds**: Increasing the similarity threshold led to a slight rise in both **MAE** and **RMSE** for both similarity measures, reflecting a modest decline in prediction accuracy as the threshold becomes more rigorous. Simultaneously, **Precision** and **Recall** improved with higher thresholds, indicating that recommendations became more relevant and accurate. This trade-off underscores the importance of carefully selecting an appropriate threshold based on the desired balance between accuracy and relevance.

3. **Performance Metrics**: The system's execution time and memory usage scaled predictably with the size of the dataset. Smaller datasets (mini_dataset1.csv and mini_dataset2.csv) exhibited minimal computational overhead, while larger datasets required significantly more resources, highlighting scalability challenges inherent in collaborative filtering approaches. The implementation of performance monitoring tools facilitated the identification of bottlenecks, particularly during similarity computations and rating predictions, providing valuable insights for future optimizations.

Several challenges were encountered during the project's execution:

- **Data Sparsity**: The recommendation system struggled with sparsely populated datasets, leading to inflated similarity scores and unreliable predictions. Implementing thresholds and minimum overlap criteria mitigated some of these issues but did not fully resolve the inherent limitations of sparse data.

- **Scalability**: As the dataset size increased, the computational demands of similarity calculations and rating predictions grew exponentially, resulting in prolonged execution times and high memory consumption. This limitation emphasizes the need for more efficient algorithms, the adoption of dimensionality reduction techniques in future iterations or the use of new more powerful servers rather than the personal laptop.

- **Interpretability of Predicted Ratings**: Instances of counterintuitive predictions, such as unrealistic high ratings for certain movies, highlighted the system's sensitivity to similarity weights and the influence of limited data points. Addressing this requires more sophisticated normalization methods and possibly the integration of additional contextual information.

For future avenues, the recommendation system could be improved by leveraging multi-threading or distributed computing frameworks which can accelerate similarity computations and rating predictions for large-scale datasets. Moreover, it would be interesting to combine collaborative filtering with content-based methods to enrich recommendations by considering both user preferences and item attributes, this could lead to some kind of hybrid recommendation model that blends multiple algorithms to enhance both accuracy and diversity of recommendations. Finally, developing strategies to effectively recommend items to new users or introduce new items into the system can address data sparsity problem that we faced.