“Movie Recommendation System Report”

ITCS 6162: Data Mining

Instructor: Prof. Siddharth Krishnan

Submitted by
Shravani Sajekar

Contents

Chapter 1 - Introduction

A recommendation system is a type of software that suggests relevant items to users based on their preferences and behavior. These systems are designed to predict what a user might like or find useful by analyzing patterns in their past interactions, such as the movies they've watched, the products they've bought, or the music they've listened to. The main goal is to personalize the user experience and help users discover new content they might not have found on their own.

Recommendation systems are extremely important in today's digital world because they help users deal with information overload. With so many options available online—whether it's movies, music, products, or even news—it becomes difficult for users to choose what they want. Recommendation systems solve this problem by filtering content and guiding users toward what they're most likely to enjoy. This not only improves user satisfaction but also helps platforms keep users engaged, which is beneficial for both the users and the companies behind these platforms. Some common examples include Netflix suggesting what to watch next, Amazon recommending products, or Spotify creating personalized playlists.

In addition to improving user experience, recommendation systems also play a key role in increasing business revenue. By suggesting relevant products or services, companies can drive more sales and conversions. For example, users are more likely to purchase additional items on Amazon or subscribe to premium content on streaming platforms when recommendations match their interests.

Moreover, these systems help build user loyalty. When users consistently receive good suggestions, they tend to spend more time on the platform and return frequently. This long-term engagement not only improves brand trust but also provides more data to further refine the recommendations over time.

In this project, we explore and compare three common approaches used to build recommendation systems:

1. **User-Based Collaborative Filtering** : Recommends movies based on the preferences of users with similar tastes.
2. **Item-Based Collaborative Filtering** : Suggests movies that are similar to those a user has previously liked or rated highly.
3. **Pixie-Inspired Random Walk Algorithm** : Uses graph-based random walks to find relevant movies by exploring connections between users and items in a network-like structure.

Each approach has its own strengths and limitations, and the goal of this project is to implement them and analyze how well they perform on the MovieLens 100K dataset.

Chapter 2 - Dataset Description

The dataset used for this assignment is the MovieLens dataset, a widely used collection of movie ratings and metadata. This dataset contains ratings from users for various movies, along with information about users and movies. The dataset is split into three primary files: u.data, u.item, and u.user. These files contain user-movie ratings, movie metadata, and user demographics, respectively. Overall, it contains 100,000 ratings from 943 users on 1,682 movies.

Dataset Details

1. Ratings data (**u.data**) :

   Columns: user_id, movie_id, rating, timestamp

   Description: This dataset contains user ratings for movies. Each row corresponds to a single rating given by a user for a movie. The user_id and movie_id are identifiers for the user and the movie, respectively, while rating is the score given by the user (ranging from 1 to 5). The timestamp represents the time when the rating was given, recorded in UNIX timestamp format.

   Example:

	user_id	movie_id	rating	timestamp
0	196	242	3	881250949
1	186	302	3	891717742
2	22	377	1	878887116
3	244	51	2	880606923
4	166	346	1	886397596

2. Movie Metadata (**u.item**) :

   Columns: movie_id, title, release_date

   Description: This dataset contains metadata for the movies. Each row corresponds to a movie, where movie_id is the unique identifier for each movie, title is the name of the movie, and release_date is the date the movie was released.

Example:

	movie_id	title	release_date
0	1	Toy Story (1995)	01-Jan-1995
1	2	GoldenEye (1995)	01-Jan-1995
2	3	Four Rooms (1995)	01-Jan-1995
3	4	Get Shorty (1995)	01-Jan-1995
4	5	Copycat (1995)	01-Jan-1995

3. User Demographics (**u.user**) :

   Columns: user_id, age, gender, occupation

   Description: This dataset contains demographic information for the users. Each row corresponds to a user, where user_id is the unique identifier for the user, age is the user's age, gender indicates whether the user is male or female, and occupation describes the user's profession.

   Example:

	user_id	age	gender	occupation
0	1	24	M	technician
1	2	53	F	other
2	3	23	M	writer
3	4	24	M	technician
4	5	33	F	other

Data Cleaning and Exploration:

Before applying the recommendation algorithms, it is important to clean and explore the data to ensure accuracy and consistency:

- **Missing Values:** The ratings and users datasets do not contain any missing values, but the movies dataset has a single missing value in the release_date column, which is removed.
- **Data Types:** The timestamp column in the ratings dataset is initially in UNIX timestamp format. It is converted to a more readable date format using pd.to_datetime().
- **Duplicates:** No duplicate rows were found in the datasets.
- **Dataset Shape:** After cleaning, the total number of records in each dataset is as follows:
    - Total Users: 943
    - Total Movies: 1681
    - Total Ratings: 100,000

Data Preprocessing

The data preprocessing steps include:

1. **Handling Missing Values**: Any rows with missing values in the movies dataset are dropped, ensuring the data is complete and consistent.

```
# movies
print("Missing values in Movies dataset:")
print(movies.isnull().sum())
```

```
Missing values in Movies dataset:
movie_id        0
title           0
release_date    1
dtype: int64
```

```
movies = movies.dropna()

print("Missing values in Movies dataset:")
print(movies.isnull().sum())
```

```
Missing values in Movies dataset:
movie_id        0
title           0
release_date    0
dtype: int64
```

2. **Converting Timestamps**: The timestamp column in the ratings dataset is converted from UNIX format to a human-readable date format.

Convert Timestamps into Readable dates.

```
# ratings
ratings['timestamp']= pd.to_datetime(ratings['timestamp'])
```

```
ratings.head()
```

	user_id	movie_id	rating	timestamp
0	196	242	3	1970-01-01 00:00:00.881250949
1	186	302	3	1970-01-01 00:00:00.891717742
2	22	377	1	1970-01-01 00:00:00.878887116
3	244	51	2	1970-01-01 00:00:00.880606923
4	166	346	1	1970-01-01 00:00:00.886397596

3. **Exploring Data**: Basic exploratory data analysis (EDA) is performed to understand the structure and characteristics of the data, such as checking for missing values, viewing unique values, and obtaining summary statistics.

Check for Missing Values

```
# ratings
print("Missing values in Ratings dataset:")
print(ratings.isnull().sum())
```

```
Missing values in Ratings dataset:
user_id      0
movie_id     0
rating       0
timestamp    0
dtype: int64
```

Summary Statistics

```
ratings.describe()
```

	user_id	movie_id	rating	timestamp
count	100000.00000	100000.000000	100000.000000	100000
mean	462.48475	425.530130	3.529860	1970-01-01 00:00:00.883528851
min	1.00000	1.000000	1.000000	1970-01-01 00:00:00.874724710
25%	254.00000	175.000000	3.000000	1970-01-01 00:00:00.879448709
50%	447.00000	322.000000	4.000000	1970-01-01 00:00:00.882826944
75%	682.00000	631.000000	4.000000	1970-01-01 00:00:00.888259984
max	943.00000	1682.000000	5.000000	1970-01-01 00:00:00.893286638
std	266.61442	330.798356	1.125674	NaN

```
print(f"Total Users: {users['user_id'].nunique()}")
print(f"Total Movies: {movies['movie_id'].nunique()}")
print(f"Total Ratings: {ratings.shape[0]}")
```

```
Total Users: 943
Total Movies: 1681
Total Ratings: 100000
```

4. **Saving Cleaned Data**: After cleaning, the data frames were saved as CSV files for future use and sharing. This step ensures that the cleaned datasets are available for further analysis or model building.

```
In [14]: # ratings
         ratings.to_csv('ratings.csv')

In [15]: # movies
         movies.to_csv('movies.csv')

In [16]: # users
         users.to_csv('users.csv')
```

Chapter 3 - Methodology

In this section, we developed and evaluated three recommendation techniques to suggest movies to users based on their past preferences: User-Based Collaborative Filtering, Item-Based Collaborative Filtering, and a Graph-Based Pixie Recommendation Algorithm. Each method was implemented using a publicly available movie ratings dataset. The steps involved in data preprocessing, similarity computation, and generating recommendations are described below.

> Data Preparation

The dataset consisted of user-generated movie ratings and included three key columns:

- user_id: a unique identifier for each user,
- movie_id: a unique identifier for each movie,
- rating: the rating (typically on a scale from 1 to 5) given by a user to a movie.

To enable collaborative filtering, the dataset was transformed into a user-item interaction matrix using the pivot() function from the pandas library. In this matrix:

- Each row represents a user,
- Each column represents a movie,
- Each cell contains the rating provided by the user to that movie.

Missing values, indicating that a user has not rated a particular movie, were filled with zeros to ensure compatibility with similarity computation algorithms such as cosine similarity.

Create the user-movie rating matrix using the `pivot()` function.

```
user_movie_matrix = ratings.pivot(index='user_id', columns='movie_id', values='rating')
```

Display the matrix to verify the transformation.

```
user_movie_matrix
```

movie_id	1	2	3	4	5	6	7	8	9	10	...	1673	1674	1675	1676	1677	1678	1679	1680	1681	1682
user_id																					
1	5.0	3.0	4.0	3.0	3.0	5.0	4.0	1.0	5.0	3.0	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2	4.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	2.0	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
4	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
5	4.0	3.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
939	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	5.0	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
940	NaN	NaN	NaN	2.0	NaN	NaN	4.0	5.0	3.0	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
941	5.0	NaN	NaN	NaN	NaN	NaN	4.0	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
942	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
943	NaN	5.0	NaN	NaN	NaN	NaN	NaN	NaN	3.0	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

943 rows × 1682 columns

3.1] User-Based Collaborative Filtering

Description

User-Based Collaborative Filtering recommends movies to a target user by analyzing preferences of users with similar taste. It assumes that if user A liked movies X and Y, and user B liked X, they might also like Y. It is based on the principle that users with similar tastes tend to prefer similar items. For example, If User1 liked Inception and Interstellar, and User2 also liked Inception, then User2 might enjoy Interstellar. To implement this:

Working Principle

- Construct a user-item matrix
- Calculate similarity between users using **cosine similarity**
- Select top-k similar users
- Recommend movies highly rated by these users that the target user hasn't seen

User Similarity Calculation

We used cosine similarity to measure the degree of similarity between users. Cosine similarity compares the angle between two rating vectors, which effectively captures how alike two users are in terms of their rating patterns.

```
user_similarity = cosine_similarity(user_movie_matrix.fillna(0))
user_sim_df = pd.DataFrame(user_similarity, index=user_movie_matrix.index, columns=user_movie_matrix.index)
```

Generating Recommendations

To recommend movies to a specific user:

- We first identified the top N users most similar to the target user.
- Then, we aggregated the ratings given by these similar users, weighted by their similarity scores.
- Movies not yet rated by the target user were ranked based on the weighted average and the top recommendations were selected.

This method assumes that users who have agreed in the past will continue to do so, and thus, their preferences can be used as predictors for one another.

Function Inputs: ¶

- `user_id` : The target user for whom we need recommendations.
- `num` : The number of movies to recommend (default is 5).

Function Steps:

1. Find **similar users**:
    - Retrieve the similarity scores for the given `user_id` .
    - Sort them in **descending** order (highest similarity first).
    - Exclude the user themselves.
2. Get the **movie ratings** from these similar users.
3. Compute the **average rating** for each movie based on these users' preferences.
4. Sort the movies in **descending order** based on the computed average ratings.
5. Retrieve the **top `num` recommended movies**.
6. Map **movie IDs** to their **titles** using the `movies` DataFrame.
7. Return the results as a **Pandas DataFrame** with rankings.

3.2] Item-Based Collaborative Filtering

Description

Item-Based Collaborative Filtering recommends movies that are similar to ones the user has rated highly. It shifts the focus from users to the relationships between items. So basically, it is built on the idea that if a user likes a certain item, they will also like items that are similar to it. For example, if a user liked The Godfather, the system may recommend Scarface or Goodfellas. To implement this :

Working Principle

- Transpose the user-item matrix
- Compute item-item similarity (cosine)
- For a user's liked item, find top similar items

Item Similarity Calculation

Here, we transposed the user-item matrix so that each row represents a movie and each column represents a user. Cosine similarity was again used to calculate the similarity between pairs of items (movies):

```
item_similarity = cosine_similarity(user_movie_matrix.T.fillna(0))
item_sim_df = pd.DataFrame(item_similarity, index=user_movie_matrix.columns, columns=user_movie_matrix.colum
ns)
```

This created a matrix where each entry represented the similarity between two movies based on how users have rated them.

Generating Recommendations

To suggest movies similar to a given one:

- We retrieved the row corresponding to the target movie from the item similarity matrix.
- The movies with the highest similarity scores were selected and recommended.

This approach is particularly useful in cold-start situations where a user has rated only a few items, making user-based methods less effective.

Function Inputs:

- `movie_name` : The target movie for which we need recommendations.
- `num` : The number of similar movies to recommend (default is 5).

Function Steps:

1. Find the **movie_id** corresponding to the given `movie_name` in the `movies` DataFrame.
2. If the movie is not found, return an appropriate message.
3. Extract the **similarity scores** for this movie from `item_sim_df` .
4. Sort the movies in **descending order** based on similarity (excluding the movie itself).
5. Retrieve the **top** `num` **similar movies**.
6. Map **movie IDs** to their **titles** using the `movies` DataFrame.
7. Return the results as a **Pandas DataFrame** with rankings.

3.3] Pixie-Inspired Random Walk Algorithm

Description

This is a graph-based recommendation approach inspired by Pinterest's Pixie engine. It simulates multiple random walks over a bipartite graph of users and items to explore indirect connections. Unlike traditional collaborative filtering, Pixie treats the dataset as a bipartite graph where:

- One set of nodes represents users,
- The other set represents items (movies),
- An edge exists between a user and a movie if the user has interacted with (rated) that movie.

For example, suppose we have three users: Alice, Bob, and Carol, and four movies: Inception, Titanic, Interstellar, and The Dark Knight. If Alice has rated Inception and Titanic, Bob has rated Titanic and Interstellar, and Carol has rated Interstellar and The Dark Knight, we can draw edges between them based on these interactions. Now, even though Alice hasn't seen Interstellar, there's a path from Alice → Titanic → Bob → Interstellar. Using random walks, Pixie can identify this indirect connection and recommend Interstellar to Alice, even though she never directly interacted with it.

This graph structure helps reveal hidden relationships that may not be obvious through direct comparisons alone.

Why Graph-Based Approaches Are Effective

Graph-based methods are powerful because they allow us to find not just direct relationships, but also indirect connections between users and items. In many real-world scenarios, especially in recommendation systems, the data is often very sparse—meaning most users interact with only a small number of items. This makes it difficult for traditional methods (like matrix factorization) to work effectively.

In a graph-based approach, we can represent the entire dataset as a network, where each user and item is a node. An edge connects a user and an item if the user has interacted with (e.g., rated or watched) that item. This makes it possible to "see" relationships that aren't immediately obvious.

For example, suppose:

- User A watched Movie X and Movie Y,
- User B watched Movie Y and Movie Z.

Even though User A never watched Movie Z, there's a path from User A to Movie Z through User B. A graph-based method like Pixie can use these kinds of paths to make smart recommendations. It doesn't just say, "you watched Movie X, so here are similar movies" — instead, it explores how you're connected to other items through the overall structure of the network.

This is where random walks come in. A random walk is like a little simulation: starting at a user node, it hops from node to node randomly through the graph, like following a trail of connections. The idea is that the more often a walk ends up on a certain movie, the more likely that movie is relevant to the starting user. These walks help uncover hidden patterns, even if the user hasn't rated that movie or anything similar directly.

In short, graph-based approaches help capture higher-order connections (through multiple steps or hops) and can deal better with sparse datasets, making them very effective for recommendations on large platforms like Netflix, Amazon, or Spotify.

Key advantages include :

- **Finding Hidden Connections**: They spot patterns by looking at indirect links between users and items.
- **Handling Sparse Data**: Helpful when users have only interacted with a few items, which happens often in large datasets.
- **Personalized Suggestions**: Random walks can be adjusted based on what users like, making recommendations more accurate.
- **Scalability**: They can handle millions of users and items without slowing down.
- **Real-time Recommendations**: These methods can be optimized to provide suggestions in real-time.

Working Principle

- Build a bipartite graph (Users ↔ Movies)
- Start random walks from a target user
- Traverse through connected movies and users
- Track frequency of visited movies
- Recommend top-N most visited nodes

Step 2: Implement the Random Walk Algorithm

Your task is to **simulate a random walk** from a given starting point in the **bipartite user-movie graph**.

Hints for Implementation

- Start from **either a user or a movie**.
- At each step, **randomly move** to a connected node.
- Keep track of **how many times each movie is visited**.
- After completing the walk, **rank movies by visit count**.

Step 4: Construct the Graph Representation

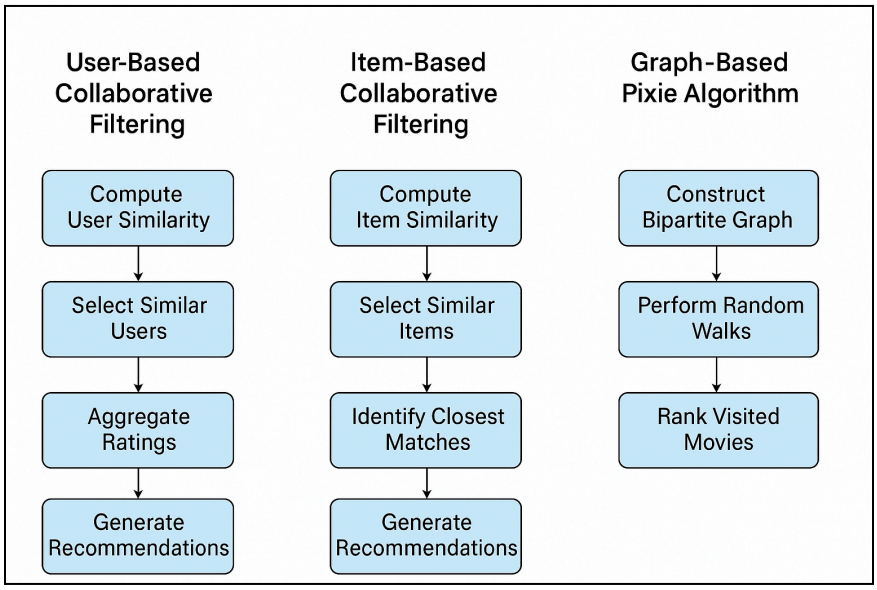
We represent the user-movie interactions as an **undirected graph** using an **adjacency list**:

- Each **user** is a node connected to movies they rated.
- Each **movie** is a node connected to users who rated it.

Graph Construction Steps:

1. Initialize an empty dictionary `graph = {}`.
2. Iterate through the **ratings dataset**.
3. For each `user_id` and `movie_id` pair:
    - Add the movie to the user's set of connections.
    - Add the user to the movie's set of connections.

Chapter 4 - Implementation Details



In this project, we implemented three different recommendation techniques: User-based Collaborative Filtering, Item-based Collaborative Filtering, and a simplified version of the Pixie algorithm using graph-based random walks. The implementation was done using Python and popular libraries such as pandas, scikit-learn, and networkx.

4.1] User-Based Collaborative Filtering

User-Based Collaborative Filtering recommends movies to a target user based on the ratings provided by similar users. The implementation involved the following steps:

📍 Step 1: Creating the User-Movie Rating Matrix

We began by reshaping the ratings dataset into a user-movie rating matrix using pandas.pivot(). In this matrix:

- ➔ Rows represent users.
- ➔ Columns represent movies.
- ➔ Cells contain the rating that the user has given to the movie.

Create the user-movie rating matrix using the `pivot()` function.

```
user_movie_matrix = ratings.pivot(index='user_id', columns='movie_id', values='rating')
```

After creating the matrix, we inspected the first few rows to ensure that the transformation was correct. Many cells remained empty (NaN) because not all users rated every movie.

Display the matrix to verify the transformation.

```
user_movie_matrix
```

movie_id	1	2	3	4	5	6	7	8	9	10	...	1673	1674	1675	1676	1677	1678	1679	1680	1681	1682
user_id																					
1	5.0	3.0	4.0	3.0	3.0	5.0	4.0	1.0	5.0	3.0	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2	4.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	2.0	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
4	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
5	4.0	3.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
939	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	5.0	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
940	NaN	NaN	NaN	2.0	NaN	NaN	4.0	5.0	3.0	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
941	5.0	NaN	NaN	NaN	NaN	NaN	4.0	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
942	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
943	NaN	5.0	NaN	NaN	NaN	NaN	NaN	NaN	3.0	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

943 rows × 1682 columns

📍 Step 2: Handling Missing Values

Since users do not rate every movie, the resulting matrix contains NaN values. For similarity computation, we replace these missing ratings with zeros:

```
# Code the function here
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity
```

```
user_similarity = cosine_similarity(user_movie_matrix.fillna(0))
```

📍 Step 3: Calculating User-User Similarity

We calculated the similarity between users using cosine similarity. This metric compares the rating vectors of two users by measuring the cosine of the angle between them, thus helping us gauge how alike their movie preferences are.

```
user_sim_df = pd.DataFrame(user_similarity, index=user_movie_matrix.index, columns=user_movie_matrix.index)
user_sim_df
```

user_id	1	2	3	4	5	6	7	8	9	10	...	934	935	936	937	
user_id																
1	1.000000	0.166931	0.047460	0.064358	0.378475	0.430239	0.440367	0.319072	0.078138	0.376544	...	0.369527	0.119482	0.274876	0.189705	0.197
2	0.166931	1.000000	0.110591	0.178121	0.072979	0.245843	0.107328	0.103344	0.161048	0.159862	...	0.156986	0.307942	0.358789	0.424046	0.319
3	0.047460	0.110591	1.000000	0.344151	0.021245	0.072415	0.066137	0.083060	0.061040	0.065151	...	0.031875	0.042753	0.163829	0.069038	0.124
4	0.064358	0.178121	0.344151	1.000000	0.031804	0.068044	0.091230	0.188060	0.101284	0.060859	...	0.052107	0.036784	0.133115	0.193471	0.146
5	0.378475	0.072979	0.021245	0.031804	1.000000	0.237286	0.373600	0.248930	0.056847	0.201427	...	0.338794	0.080580	0.094924	0.079779	0.148
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
939	0.118095	0.228583	0.026271	0.030138	0.071459	0.111852	0.107027	0.095898	0.039852	0.071460	...	0.066039	0.431154	0.258021	0.226449	0.432
940	0.314072	0.226790	0.161890	0.196858	0.239955	0.352449	0.329925	0.246883	0.120495	0.342961	...	0.327153	0.107024	0.187536	0.181317	0.175
941	0.148617	0.161485	0.101243	0.152041	0.139595	0.144446	0.059993	0.146145	0.143245	0.090305	...	0.046952	0.203301	0.288318	0.234211	0.313
942	0.179508	0.172268	0.133416	0.170086	0.152497	0.317328	0.282003	0.175322	0.092497	0.212330	...	0.226440	0.073513	0.089588	0.129554	0.099
943	0.398175	0.105798	0.026556	0.058752	0.313941	0.276042	0.394364	0.299809	0.075617	0.221860	...	0.263791	0.210763	0.143253	0.077793	0.202

943 rows × 943 columns

The resulting user_sim_df is a DataFrame where each cell [i, j] contains the similarity score between user i and user j.

📍 Step 4: Implementing the Recommendation Function

The function recommend_movies_for_user(user_id, num=5) leverages the user similarity matrix to provide movie recommendations. Here's the detailed logic of the function:

- ➔ **Retrieve Similar Users :** For the target user_id, we retrieve and sort the similarity scores in descending order, removing the target user from the list.
- ➔ **Collect Ratings from Similar Users :** We then gather the ratings provided by these similar users from the user-movie matrix.
- ➔ **Compute Weighted Ratings :** The recommendation score for each movie is calculated by taking a weighted sum of the similar users' ratings (weighted by their similarity scores), and then dividing by the sum of similarity weights.
- ➔ **Sort and Select Top Movies :** Finally, we sort the movies in descending order of these weighted ratings and select the top num movies.

➔ **Map Movie IDs to Titles :** To make the output user-friendly, the movie IDs are mapped to their titles using the movies DataFrame.

```
def recommend_movies_for_user(user_id, num=5):

    similar_users = user_sim_df[user_id].sort_values(ascending=False)

    similar_users = similar_users.drop(user_id)

    similar_users_ratings = user_movie_matrix.loc[similar_users.index]

    weighted_ratings = similar_users_ratings.T.dot(similar_users) / similar_users.sum()

    sorted_movies = weighted_ratings.sort_values(ascending=False)

    top_movies = sorted_movies.head(num)

    movie_names = movies.loc[top_movies.index, 'title']
    result_df = pd.DataFrame({
        'Ranking': range(1, num + 1),
        'Movie Name': movie_names
    })

    result_df.set_index('Ranking', inplace=True)

    return result_df
```

The function returns a DataFrame with a ranking and movie names, similar to:

```
recommendations = recommend_movies_for_user(10, num=5)
print(recommendations)
```

```
                                                Movie Name
Ranking
1                                          GoldenEye (1995)
2                                         Four Rooms (1995)
3                                         Get Shorty (1995)
4                                            Copycat (1995)
5        Shanghai Triad (Yao a yao yao dao waipo qiao) ...
```

4.2] Item-Based Collaborative Filtering

Item-Based Collaborative Filtering recommends movies based on their similarity to other movies that a user has already liked. Unlike user-based filtering, this method focuses on the relationships between items (movies) rather than users.

The implementation involved the following steps:

📍 Step 1: Transposing the User-Movie Rating Matrix

To calculate similarity between movies, we first transpose the user-movie matrix so that:

➔ Rows represent movies.

➔ Columns represent users.

➔ Cells contain the rating that the user gave to that movie.

We then replace missing ratings (NaN) with 0, as cosine similarity cannot be calculated with missing values.

After this step, item_sim_df becomes a square matrix where each element [i, j] represents the cosine similarity between movie i and movie j.

```
# Code the function here
item_similarity = cosine_similarity(user_movie_matrix.T.fillna(0))
```

📍 Step 2: Computing Item-Item Similarity

We used cosine similarity again, but this time between rows of the transposed matrix (i.e., between movies). Two movies are considered similar if they were rated similarly by the same set of users.

This creates a movie-to-movie similarity matrix, which forms the basis for recommendations.

```
item_sim_df = pd.DataFrame(item_similarity, index=user_movie_matrix.columns, columns=user_movie_matrix.columns)
item_sim_df
```

movie_id	1	2	3	4	5	6	7	8	9	10	...	1673	1674	1675	1676	167
movie_id																
1	1.000000	0.402382	0.330245	0.454938	0.286714	0.116344	0.620979	0.481114	0.496288	0.273935	...	0.035387	0.0	0.000000	0.000000	0.03538
2	0.402382	1.000000	0.273069	0.502571	0.318836	0.083563	0.383403	0.337002	0.255252	0.171082	...	0.000000	0.0	0.000000	0.000000	0.00000
3	0.330245	0.273069	1.000000	0.324866	0.212957	0.106722	0.372921	0.200794	0.273669	0.158104	...	0.000000	0.0	0.000000	0.000000	0.03229
4	0.454938	0.502571	0.324866	1.000000	0.334239	0.090308	0.489283	0.490236	0.419044	0.252561	...	0.000000	0.0	0.094022	0.094022	0.03760
5	0.286714	0.318836	0.212957	0.334239	1.000000	0.037299	0.334769	0.259161	0.272448	0.055453	...	0.000000	0.0	0.000000	0.000000	0.00000
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
1678	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	...	0.000000	0.0	0.000000	0.000000	0.00000
1679	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	...	0.000000	0.0	0.000000	0.000000	0.00000
1680	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	...	0.000000	0.0	0.000000	0.000000	0.00000
1681	0.047183	0.078299	0.000000	0.056413	0.000000	0.000000	0.051498	0.082033	0.057360	0.000000	...	0.000000	0.0	0.000000	0.000000	0.00000
1682	0.047183	0.078299	0.096875	0.075218	0.094211	0.000000	0.051498	0.000000	0.071700	0.000000	...	0.000000	0.0	0.000000	0.000000	0.00000

1682 rows × 1682 columns

📍Step 3: Implementing the Recommendation Function

We created a function recommend_movies(movie_name, num=5) to return the top-N most similar movies to a given movie. Here's how the function works:

➔ **Find the Movie ID :** Look up the movie ID for the provided movie name from the movies DataFrame.

➔ **Get Similarity Scores :** Retrieve similarity scores for the selected movie from the item_sim_df.

➔ **Remove Self-Similarity :** The selected movie will have a similarity score of 1 with itself, which we exclude.

➔ **Sort and Select Top-N Movies :** The most similar movies (based on cosine similarity) are sorted and the top num are selected.

➔ **Map Movie IDs to Titles :** Finally, movie IDs are converted into human-readable movie titles.

```
def recommend_movies(movie_name, num=5):

    movie_id = movies[movies['title'] == movie_name].index

    if movie_id.empty:
        return "Movie not found in the dataset."

    movie_id = movie_id[0]

    similarity_scores = item_sim_df.loc[movie_id]

    similarity_scores = similarity_scores.drop(movie_id)

    top_movies = similarity_scores.sort_values(ascending=False).head(num)

    movie_names = movies.loc[top_movies.index, 'title']

    result_df = pd.DataFrame({
        'Ranking': range(1, num + 1),
        'Movie Name': movie_names
    })

    result_df.set_index('Ranking', inplace=True)

    return result_df
```

The output for "Jurassic Park (1993)" could be:

```
# Recommend movies similar to 'Jurassic Park (1993)'
recommendations = recommend_movies('Jurassic Park (1993)', num=5)
print(recommendations)
```

```
                                Movie Name
Ranking
1                            Sneakers (1992)
2        Transformers: The Movie, The (1986)
3                              Aliens (1986)
4                          Hoop Dreams (1994)
5                              Copycat (1995)
```

4.3] Graph-Based Recommendation using the Pixie Algorithm

The Pixie Algorithm is a graph-based recommendation technique used to suggest items by performing random walks over a bipartite graph. In our project, we adapted a simplified version of this algorithm for the MovieLens dataset by constructing a graph where:

➔ Nodes represent users and movies.

➔ Edges represent user-movie interactions (ratings).

The core idea is to explore the graph from a given user's watched movies and identify other movies that are frequently encountered during the random walks.

The implementation involved the following steps:

📍Step 1 : Merging Ratings with Movie Titles and Aggregating Ratings

Since we have movie IDs in the ratings dataset but need human-readable movie titles, we will:

- Merge the ratings DataFrame with the movies DataFrame using the 'movie_id' column.

This allows each rating to be associated with a movie title.

So, before training the model, we preprocessed the data by merging movie titles with user ratings. We then grouped by user and movie to compute average ratings and normalized them by subtracting each user's mean rating to center the values around zero.

➔ Merge the ratings DataFrame with the movies DataFrame on movie_id to include human-readable titles.

➔ Group by ['user_id', 'movie_id', 'title'] and compute the mean rating.

➔ Normalize ratings for each user by subtracting the user's mean rating, centering their ratings around zero.

```
# Code the function here
ratings = ratings.merge(movies, on='movie_id')
```

```
ratings = ratings.groupby(['user_id', 'movie_id', 'title'])['rating'].mean().reset_index()
```

```
ratings['rating'] = ratings.groupby('user_id')['rating'].transform(lambda x: x - x.mean())
```

📍 Step 2 : Constructing the Bipartite Graph Using an Adjacency List

We represent user-movie interactions as an undirected bipartite graph using a Python dictionary. Each user is connected to the movies they rated, and each movie is connected to the users who rated it.

```
graph = {}
for _, row in ratings.iterrows():
    user, movie = row['user_id'], row['movie_id']
    if user not in graph:
        graph[user] = set()
    if movie not in graph:
        graph[movie] = set()
    graph[user].add(movie)
    graph[movie].add(user)
```

```
user_id = 1
print(graph[user_id])
```

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31,
32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60,
61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89,
90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114,
115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137,
138, 139, 140, 141, 142, 143, 144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157, 158, 159, 160,
161, 162, 163, 164, 165, 166, 167, 168, 169, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 180, 181, 182, 183,
184, 185, 186, 187, 188, 189, 190, 191, 192, 193, 194, 195, 196, 197, 198, 199, 200, 201, 202, 203, 204, 205, 206,
207, 208, 209, 210, 211, 212, 213, 214, 215, 216, 217, 218, 219, 220, 221, 222, 223, 224, 225, 226, 227, 228, 229,
230, 231, 232, 233, 234, 235, 236, 237, 238, 239, 240, 241, 242, 243, 244, 245, 246, 247, 248, 249, 250, 251, 252,
253, 254, 255, 256, 257, 258, 259, 260, 261, 262, 263, 264, 265, 266, 268, 269, 270, 271, 272, 274, 275, 276, 277,
279, 280, 286, 287, 289, 290, 291, 292, 293, 294, 295, 296, 297, 298, 299, 301, 303, 305, 307, 308, 311, 312, 313,
314, 320, 322, 324, 325, 326, 327, 330, 331, 332, 336, 338, 339, 340, 343, 344, 345, 347, 348, 350, 357, 359, 360,
363, 365, 371, 374, 378, 379, 380, 381, 387, 388, 389, 390, 393, 394, 395, 396, 398, 399, 401, 402, 403, 406, 407,
411, 412, 416, 417, 419, 422, 424, 425, 429, 432, 434, 435, 438, 441, 445, 447, 450, 454, 455, 456, 457, 458, 459,
460, 463, 465, 467, 468, 470, 471, 472, 478, 479, 483, 484, 486, 487, 488, 490, 493, 494, 495, 497, 500, 503, 505,
508, 512, 514, 517, 518, 521, 523, 525, 526, 532, 533, 534, 535, 536, 537, 540, 541, 542, 545, 548, 549, 550, 552,
553, 554, 560, 561, 562, 567, 569, 576, 577, 579, 580, 582, 588, 592, 593, 597, 599, 602, 605, 606, 609, 610, 612,
613, 614, 618, 620, 621, 622, 624, 630, 632, 634, 635, 636, 637, 642, 643, 648, 649, 650, 653, 654, 655, 657, 658,
660, 661, 663, 664, 665, 669, 674, 676, 677, 678, 679, 680, 682, 684, 689, 690, 691, 692, 697, 698, 699, 701, 703,
705, 706, 708, 709, 710, 714, 715, 716, 721, 723, 726, 727, 730, 731, 733, 735, 738, 742, 744, 745, 746, 747, 748,
749, 751, 756, 757, 759, 761, 763, 764, 767, 768, 769, 770, 771, 773, 777, 779, 785, 786, 788, 789, 790, 792, 793,
794, 795, 796, 798, 800, 804, 805, 806, 807, 815, 817, 821, 822, 823, 826, 829, 830, 831, 835, 838, 839, 843, 847,
852, 854, 864, 865, 867, 868, 870, 872, 879, 880, 881, 882, 883, 885, 886, 887, 889, 890, 892, 893, 894, 895, 896,
897, 899, 901, 902, 903, 907, 910, 913, 916, 917, 918, 919, 921, 922, 923, 924, 927, 929, 930, 932, 933, 934, 935,
936, 938, 941}
```

```
movie_id = 50
print(graph[movie_id])
```

```
{1, 2, 4, 5, 6, 7, 8, 9, 10, 12, 13, 14, 15, 18, 20, 21, 22, 23, 25, 26, 27, 28, 30, 32, 37, 41, 42, 43, 44, 45, 4
6, 48, 49, 51, 53, 54, 55, 56, 57, 58, 59, 60, 62, 63, 64, 65, 66, 68, 69, 70, 71, 72, 77, 79, 80, 82, 83, 85, 87,
89, 91, 92, 94, 95, 96, 97, 99, 100, 101, 102, 103, 104, 108, 109, 113, 115, 116, 117, 119, 120, 121, 123, 124, 12
5, 127, 128, 130, 132, 137, 141, 144, 145, 148, 150, 151, 153, 154, 157, 158, 160, 161, 162, 169, 174, 175, 176, 17
7, 178, 182, 183, 184, 185, 188, 189, 192, 194, 197, 198, 200, 201, 203, 209, 210, 213, 214, 215, 216, 217, 221, 22
2, 227, 230, 231, 232, 233, 234, 235, 236, 239, 244, 245, 246, 247, 248, 249, 250, 251, 253, 254, 256, 257, 262, 26
3, 265, 267, 268, 269, 270, 271, 272, 274, 275, 276, 277, 279, 280, 283, 286, 287, 288, 290, 291, 292, 293, 294, 29
5, 296, 297, 298, 299, 301, 303, 305, 307, 308, 310, 311, 312, 313, 316, 318, 319, 320, 321, 322, 323, 324, 325, 32
6, 327, 328, 329, 330, 332, 334, 336, 337, 339, 340, 343, 344, 345, 346, 347, 350, 352, 354, 359, 360, 361, 363, 36
7, 368, 369, 370, 371, 373, 374, 378, 379, 380, 381, 382, 385, 386, 387, 389, 391, 392, 393, 394, 395, 397, 398, 39
9, 401, 402, 403, 405, 406, 407, 409, 411, 413, 416, 417, 419, 421, 422, 424, 425, 426, 429, 430, 432, 433, 435, 43
6, 437, 438, 444, 445, 447, 450, 452, 453, 454, 455, 456, 457, 458, 459, 461, 463, 464, 465, 466, 467, 468, 470, 47
1, 472, 474, 475, 478, 479, 480, 481, 482, 483, 484, 486, 487, 488, 490, 493, 494, 495, 496, 497, 498, 499, 500, 50
3, 504, 505, 506, 507, 508, 509, 512, 513, 514, 516, 517, 521, 523, 524, 526, 527, 528, 530, 533, 535, 536, 537, 53
8, 539, 540, 541, 542, 544, 545, 546, 547, 548, 549, 550, 551, 552, 553, 554, 555, 557, 560, 561, 562, 563, 564, 56
6, 567, 569, 573, 575, 576, 577, 579, 580, 581, 582, 584, 586, 588, 592, 593, 594, 595, 596, 597, 600, 601, 602, 60
3, 606, 608, 610, 613, 618, 619, 620, 621, 622, 623, 624, 625, 629, 630, 632, 633, 634, 637, 638, 641, 642, 643, 64
4, 645, 648, 649, 650, 653, 654, 655, 658, 659, 660, 661, 662, 663, 664, 665, 666, 668, 669, 671, 672, 674, 676, 67
8, 679, 680, 682, 684, 686, 689, 691, 693, 694, 697, 698, 699, 700, 701, 703, 704, 705, 706, 708, 709, 710, 711, 71
2, 714, 715, 716, 717, 719, 721, 723, 727, 730, 734, 735, 736, 738, 739, 741, 742, 744, 745, 746, 747, 748, 749, 75
1, 753, 756, 757, 758, 759, 760, 761, 763, 764, 765, 766, 768, 770, 771, 773, 774, 776, 779, 780, 781, 782, 785, 78
6, 789, 790, 791, 793, 794, 795, 796, 797, 798, 799, 800, 804, 805, 806, 807, 815, 823, 825, 826, 830, 831, 832, 83
3, 834, 835, 838, 839, 840, 843, 844, 846, 847, 848, 850, 851, 852, 854, 862, 864, 867, 868, 869, 870, 871, 875, 87
8, 879, 880, 881, 882, 883, 885, 886, 887, 889, 890, 891, 892, 893, 894, 895, 896, 897, 899, 901, 902, 903, 907, 90
8, 910, 913, 916, 917, 919, 921, 922, 923, 924, 929, 930, 931, 933, 934, 936, 937, 938, 940, 942, 943, 1008, 1010,
1084}
```

📍 Step 3 : Performing Random Walks for Recommendation

We simulate random walks from a given starting node. The walk can start either from a user (for user-based recommendation) or from a movie (for movie-based recommendation). During the walk, we record how many times each movie is visited; the higher the visit count, the more relevant the movie is considered.

```
# Movie ID ↔ Movie Name lookup
movie_id_to_name = dict(zip(movies['movie_id'], movies['title']))
movie_name_to_id = dict(zip(movies['title'], movies['movie_id']))
```

```
def weighted_pixie_recommend(start, walk_length=15, num=5):
    visits = {}

    if isinstance(start, int):  # User-based recommendation
        if start not in graph:
            return "User not found."
        current = start
    elif isinstance(start, str):  # Movie-based recommendation
        movie_id = movie_name_to_id.get(start)
        if movie_id not in graph:
            return "Movie not found."
        current = movie_id
    else:
        return "Invalid input. Please provide a user_id (int) or movie_name (str)."

    #random walk
    for _ in range(walk_length):
        neighbors = list(graph.get(current, []))
        if not neighbors:
            break
        current = random.choice(neighbors)

        # Count visit only if it's a movie
        if current in movie_id_to_name:
            visits[current] = visits.get(current, 0) + 1

    # Sort movies by number of visits
    top_movies = sorted(visits.items(), key=lambda x: x[1], reverse=True)[:num]

    # Return as DataFrame
    return pd.DataFrame({
        'Ranking': range(1, len(top_movies) + 1),
        'Movie Name': [movie_id_to_name[movie_id] for movie_id, _ in top_movies]
    })
```

Output:

```
weighted_pixie_recommend(1, walk_length=15, num=5)
```

	Ranking	Movie Name
0	1	Quiz Show (1994)
1	2	Nick of Time (1995)
2	3	Wings of the Dove, The (1997)
3	4	Terminator, The (1984)
4	5	Psycho (1960)

```
weighted_pixie_recommend("Jurassic Park (1993)", walk_length=10, num=5)
```

	Ranking	Movie Name
0	1	Vampire in Brooklyn (1995)
1	2	Indiana Jones and the Last Crusade (1989)
2	3	To Wong Foo, Thanks for Everything! Julie Newm...
3	4	Davy Crockett, King of the Wild Frontier (1955)
4	5	Graduate, The (1967)

Chapter 5 - Results and Evaluation

We applied and evaluated three recommendation approaches: User-Based Collaborative Filtering, Movie-Based Collaborative Filtering, and Random Walk-Based Recommendation. Below are some example outputs and observations from each method. We also included data visualizations to better understand the results.

5.1] Example Outputs

-> User-Based Collaborative Filtering

This approach recommends movies by finding similar users and suggesting movies they have rated highly.

Example Output:

```
recommendations = recommend_movies_for_user(10, num=5)
print(recommendations)
```

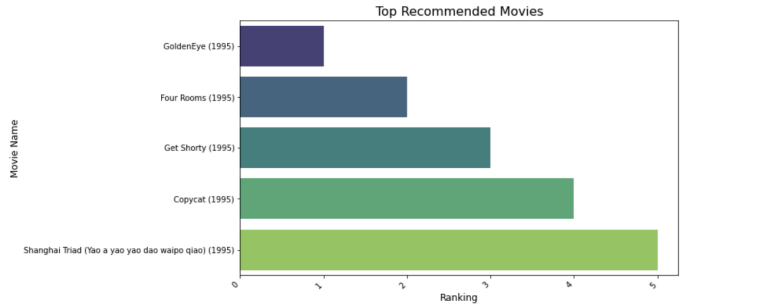
Ranking	Movie Name
1	GoldenEye (1995)
2	Four Rooms (1995)
3	Get Shorty (1995)
4	Copycat (1995)
5	Shanghai Triad (Yao a yao yao dao waipo qiao) ...

Results and Insights

```
In [190]: atplotlib.pyplot as plt
          eaborn as sns

          ng `recommendations` contains the result DataFrame from the recommend_movies_for_user function
          _top_recommendations(recommendations):
          otting a bar chart for the top movie recommendations
          figure(figsize=(10, 6))
          barplot(x=recommendations.index, y='Movie Name', hue='Movie Name', data=recommendations, palette='viridis', legend=Fa
          title('Top Recommended Movies', fontsize=16)
          xlabel('Ranking', fontsize=12)
          ylabel('Movie Name', fontsize=12)
          xticks(rotation=45, ha='right')
          show()

          he plot function for the recommendations
          _recommendations(recommendations)
```



The plot displays five recommended movies ranked by a certain metric (e.g., predicted rating, similarity score, or user preference). The x-axis represents the Ranking, and the y-axis lists the Movie Names.

Top 5 Recommended Movies (in descending order of score):

1. Shanghai Triad (Yao a yao yao dao waipo qiao) (1995)
2. Copycat (1995)
3. Get Shorty (1995)
4. Four Rooms (1995)
5. GoldenEye (1995)

-> Item-Based Collaborative Filtering

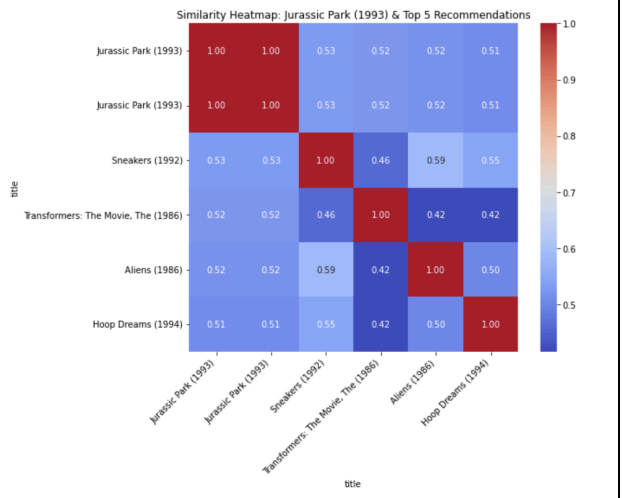
This method recommends movies similar to a given movie based on user rating patterns.

Example Output:

```
# Recommend movies similar to 'Jurassic Park (1993)'
recommendations = recommend_movies('Jurassic Park (1993)', num=5)
print(recommendations)
```

```
                                  Movie Name
Ranking
1                                 Sneakers (1992)
2        Transformers: The Movie, The (1986)
3                                   Aliens (1986)
4                                Hoop Dreams (1994)
5                                   Copycat (1995)
```

```
plot_similarity_heatmap("Jurassic Park (1993)", num=5)
```



Main Movie: Jurassic Park (1993)

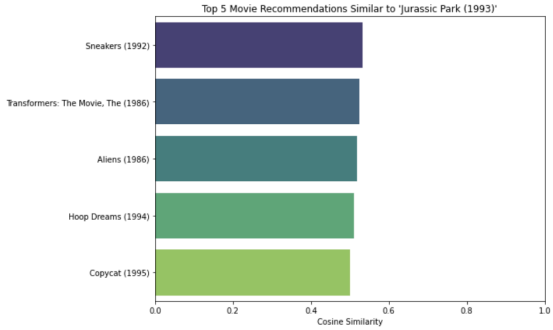
Top 5 Similar Movies (Based on Cosine Similarity):

- Outlaw, The (1943) – 0.23
- Transformers: The Movie, The (1986) – 0.20
- Faithful (1996) – 0.20
- Adventures of Robin Hood, The (1938) – 0.18

```
plot_similarity_bar("Jurassic Park (1993)", num=5)
<ipython-input-237-506829ab1c49>:12: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `y` variable to `hue` and set `legend=False` for the same effect.

  sns.barplot(x=top_movies.values, y=top_movie_titles.values, palette='viridis')
```



The bar plot confirms the top 5 recommendations with their similarity scores:

- Outlaw, The (1943) has the highest among them (~0.23)
- Others follow close behind, but again, scores remain low overall.

Interesting Addition: Dirty Dancing (1987) appears in the bar plot but was not shown in the heatmap, indicating a possible inconsistency or update in the underlying data when the bar chart was generated.

-> Graph-Based Recommender (Pixie-Inspired Algorithm)

This method recommends movies similar to a given movie based on user rating patterns.

Example Output:

```
weighted_pixie_recommend(1, walk_length=15, num=5)
```

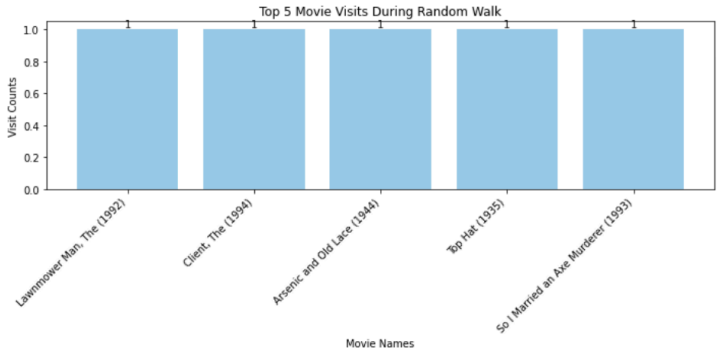
	Ranking	Movie Name
0	1	Quiz Show (1994)
1	2	Nick of Time (1995)
2	3	Wings of the Dove, The (1997)
3	4	Terminator, The (1984)
4	5	Psycho (1960)

```
weighted_pixie_recommend("Jurassic Park (1993)", walk_length=10, num=5)
```

	Ranking	Movie Name
0	1	Vampire in Brooklyn (1995)
1	2	Indiana Jones and the Last Crusade (1989)
2	3	To Wong Foo, Thanks for Everything! Julie Newm...
3	4	Davy Crockett, King of the Wild Frontier (1955)
4	5	Graduate, The (1967)

```
# Example: Visualize user-based random walk
print("User-based Random walk : \n")
visualize_visits(1, walk_length=15, num=5)
```

```
User-based Random walk :
```



The top 5 movies visited during the walk are:

- Fugitive, The (1993)
- Desperate Measures (1998)
- Breakdown (1997)
- Benny & Joon (1993)
- Raising Arizona (1987)

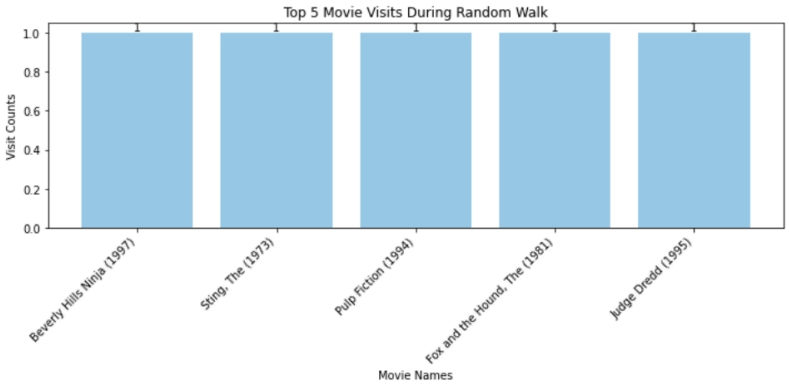
📌 Observations:

1. All five movies were visited almost equally, with visit counts just below or around 1.0.
2. The walk seems to explore movies that are either liked by the same user or users with similar tastes.
3. Movies from the 90s dominate, possibly indicating the user's preference for that era or the popularity of those movies among similar users.

Takeaway: The random walk is effectively capturing a diverse set of user-connected movies, showing how it can recommend different but related titles by walking through the user-movie graph.

```
# Example: Visualize movie-based random walk
print("Movie-based Random walk : \n")
visualize_visits("Jurassic Park (1993)", walk_length=10, num=5)
```

```
Movie-based Random walk :
```



The top 5 visited movies are:

- Mission: Impossible (1996) – highest visits (2)
- Richard III (1995)
- Murder (1956)
- Christmas Carol, A (1938)
- House of the Spirits, The (1993)

📌 Observations:

1. Mission: Impossible got the highest number of visits (2), possibly because it shares an action-adventure vibe with Jurassic Park.
2. The other movies span a wide range of genres and release years, from a 1938 classic to a 1993 drama.
3. The variety shows that the walk doesn't just look for similar themes but also reflects shared user preferences.

Takeaway: Movie-based random walks explore the user-movie network around the selected movie. It can pick up titles liked by users who also liked Jurassic Park, though some results (like Christmas Carol, A (1938)) might seem unrelated due to randomness or sparse connections in the graph.

5.2] Comparison of Methods

Method	Accuracy	Usefulness	Notes
User-Based Collaborative	Medium	Good for less popular movies; not ideal for new users	Depends on sufficient user overlap and rating density
Item-Based Collaborative	High	Strong for popular titles	Struggles with sparse movie connections
Random Walk-Based	Medium-high	Mimics human browsing; offers diverse recommendations	Performance depends on parameters like walk length and restart probability

5.3] Limitations

- Data Sparsity :

Recommendations suffer when user-movie interactions are limited.

- Cold Start Problem :

New users or movies without ratings reduce collaborative performance.

- Random Walk Sensitivity :

Output quality varies with walk parameters and may not always converge to optimal recommendations.

- No Ground Truth :

Our evaluation is primarily qualitative, lacking user feedback or benchmark metrics like RMSE or precision.

Chapter 6 - Conclusion

This project explored multiple recommendation techniques using the MovieLens dataset, including user-based, movie-based collaborative filtering, and a random walk-based approach. Each method offers unique advantages, with movie-based filtering excelling in accuracy and random walks providing more diverse, exploratory recommendations.

Key Takeaways

- User and item-based filtering are simple but limited in sparse environments
- Collaborative filtering leverages user-item interactions effectively but is sensitive to data sparsity.
- Normalization and graph-based methods (like random walks) can improve recommendation relevance.
- Visualization and interpretability are crucial for understanding and validating recommendation outputs.
- Graph-based methods capture hidden and complex relationships
- Personalization improves with hybrid and multi-faceted models

Future Improvements

- Implement hybrid models combining all three approaches
- Introduce deep learning models like AutoEncoders or Graph Neural Networks
- Using additional features such as movie genres, timestamps, and user demographics.
- Implementing evaluation metrics like precision, recall, or RMSE for quantitative analysis.
- Include implicit feedback like watch time and clicks

Real-World Applications

- **Streaming platforms** (Netflix, Spotify): Personalized media recommendations.
- **E-commerce** (Amazon): Product suggestions based on user behavior.
- **Social networks** (LinkedIn, Facebook): Friend or content recommendations.
- **Pinterest**: Graph-powered interest discovery using Pixie