Movie Recommender System Report

1 Introduction

Recommender systems are a common and useful method for providing new personalized content to users on digital platform such as Netflix, Instagram, TikTok, Youtube, etc. There are many implementations for recommender system, but in this report, 3 approaches are explored: user-user collaborative filtering, item-item collaborative filtering, and a Pixie-inspired random walk algorithm.

2 Dataset Description

The MovieLens 100K dataset was used. It contains 943 user ratings for 1682 movies. The user ratings are between 1 and 5 and each user has rated a minimum of 20 movies. Each movie has an associated movie ID, title, and release date. The dataset was preprocessed by merging tables, pivoting into a user-movie rating matrix, and filling missing values with users' mean ratings. This matrix served as the basis for both collaborative filtering and graph construction.

3 Methodology

Method 1: User-User Collaborative Filtering

The aim is to calculate a weighted average of ratings from users that are similar to the user of interest. The weight for each similar user is given by the user's similarity to the user of interest. If a user is more similar to the user of interest, their ratings will be more impactful in the weighted sum. This weighted sum provides an estimate of the rating the user of interest would give to a movie(s) they have yet to rate. Therefore, if the estimate for a movie is high, it is likely the user of interest would like enjoy the movie and is therefore given as a recommendation.

First the user-movies matrix is created as follows,

```
udata = pd.read_csv('udata.csv')

user_movie_matrix = udata.pivot(index="user_id", columns="movie_id", values="rating")
user_movie_matrix.columns = user_movie_matrix.columns.astype(int)
user_movie_matrix.to_csv('user_movie_matrix.csv', index=True)
```

where `udata` looks like,

```
   user_id  movie_id  rating  timestamp
0      196       242       3  881250949
```

```
1      186       302       3  891717742
2       22       377       1  878887116
```

Ally `NaN` values are filled with the mean value of a user's ratings and each user row is z-scale normalized as shown below,

```
UM = UM.apply(lambda row: row.fillna(row.mean()), axis=1)
UM_norm = UM.apply(lambda row: (row - row.mean())  / row.std(), axis=1)
```

Z-scale normalized removes the user bias and normalizes for rating spread.

First, a user-user similarity matrix is created by taking the cosine similarity of every user with every other user in the z-scale normalized user-movie ratings matrix. This step is shown below,

```
user_sim = pd.DataFrame(cosine_similarity(UM_norm), index=UM_norm.index,
columns=UM_norm.index)
```

The result is a DataFrame where each row represents 1 user's similarity score with every other user.

Once a user of interest is given, the `user_sim` matrix can be indexed using the user-of-interest's user ID. That row of the DataFrame is then sorted in descending order, so that the most similar users are at the start of the row, and only the most similar users are kept. It is also important to drop the user-of-interest from this result as the cosine similarity with itself will always be 1. This is show below,

```
sims = user_sim[user_id].drop(user_id).sort_values(ascending=False).head(50)
```

In this case, the top 50 most similar users are kept. The user-movie ratings matrix can then be indexed using the most similar users to retrieve their movie ratings as follows,

```
sim_ratings = UM_norm.loc[sims.index]
```

To get the estimated ratings for all movies for the user-of-interest, we take the weight sum discussed earlier,

```
pred_ratings = sim_ratings.T.dot(sims.T) / sims.sum()
pred_ratings = pred_ratings.loc[unrated.loc[user_id]]
```

Movies that the user-of-interest has already rated are dropped from `pred_ratings` in the second line so that the movie recommendations do not contain previously rated movies.

`pred_ratings` is then sorted in descending order (most similar movies at the start of the array) and the first `num` movies are stored as shown below,

```
recommend_idx = pred_ratings.sort_values(ascending=False).head(num).index
```

Movie titles can then be retrieved from the data set as follows,

```
movie_titles = movies.loc[recommend_idx, 'title'].to_list()
```

The algorithm all-together is,

```
def recommend_movies_for_user(user_id, num=5):
    sims = user_sim[user_id].drop(user_id).sort_values(ascending=False).head(50)
    sim_ratings = UM_norm.loc[sims.index]

    pred_ratings = sim_ratings.T.dot(sims.T) / sims.sum()
    pred_ratings = pred_ratings.loc[unrated.loc[user_id]]
    recommend_idx = pred_ratings.sort_values(ascending=False).head(num).index

    movie_titles = movies.loc[recommend_idx, 'title'].to_list()

    result_df = pd.DataFrame({
    'Ranking': range(1, num+1),
    'Movie Name': movie_titles
    })
    result_df.set_index('Ranking', inplace=True)

    return result_df
```

Method 2: Item-Item Collaborative Filtering

The item-item algorithm is used to find movies that are similar to a movie of interest. This algorithm is essentially the same as the user-user algorithm but using the transpose of the user-movie rating matrix and normalizing across the movies ratings row rather than the user ratings row. The data processing is shown below,

```
UM = UM.apply(lambda row: row.fillna(row.mean()), axis=1)
MU = UM.T
MU_norm = MU.apply(lambda row: (row - row.mean())  / row.std(), axis=1)
```

Movie-movie similarities are then calculated as follows,

```
movie_sim = pd.DataFrame(cosine_similarity(MU_norm), index=MU_norm.index,
columns=MU_norm.index)
```

The algorithm all-together is,

```
def recommend_movies(movie_name, num=5):
    movie_id = int(movies[movies["title"] == movie_name].index[0])
    sims = movie_sim.loc[movie_id].drop(movie_id)
    sim_idx = sims.sort_values(ascending=False).head(num).index.astype(int)

    movie_titles = movies.loc[sim_idx, 'title']

    result_df = pd.DataFrame({
    'Ranking': range(1, num+1),
    'Movie Name': movie_titles
    })
    result_df.set_index('Ranking', inplace=True)

    return result_df
```

Method 3: Pixie-Inspired Random Walk

The goal with this algorithm is to create a bipartite graph where every user is connected to each of its rated movies. 2 approaches will be implemented. One with an unweighted graph and one with a weighted graph.

In the unweighted approach, user recommendations will be made by starting at the node of a user-of-interest. One of its neighboring nodes (i.e. previously rated movies), will be chosen randomly and will be set as the new current node. One of the movie's neighboring nodes (i.e. users that have rated the movie) will be chosen randomly and be set as the new current node. This process will repeat for a desired walk length, `walk_length`. During this process, the algorithm will keep track of the number of times each movie is visited. The idea is that movies visited align more with movies the user-of-interest would enjoy.

For the weighted approach, the graph will be built the same way as in the unweighted approach, but with the ratings being assigned as edge weights. The algorithm, too, will work the same way with the exception that the next node will not be chosen uniformly randomly, but probabilistically based on edge weight. The idea is that the edge weights should guide the random walk to movies more similar to the user's interest thereby making the recommendations more accurate.

The data processing for the Pixie inspired algorithm is initially similar to the collaborative filtering approach for generating the user-movie ratings matrix,

```
movies = pd.read_csv('uitem.csv', index_col='movie_id').:wreset_index()
UM = pd.read_csv('user_movie_matrix.csv', index_col='user_id')
UM = UM.apply(lambda row: (row - row.mean())  / row.std(), axis=1)
```

However, a ratings DataFrame is then calculated where each row contains a `user_id` - `movie_id` pair with its associated rating as shown below,

```
ratings = UM.stack().reset_index()
ratings.columns = ['user_id', 'movie_id', 'rating']
ratings['movie_id'] = ratings['movie_id'].astype(int)
ratings = ratings.dropna()
ratings = ratings.merge(movies[['movie_id', 'title']], on='movie_id')
print(ratings)
```

with the first 3 lines of the output being,

```
   user_id  movie_id    rating                         title
0        1         1  1.099812             Toy Story (1995)
1        1         2 -0.482986             GoldenEye (1995)
2        1         3  0.308413            Four Rooms (1995)
```

The unweighted graph is built using `ratings` as follows,

```
graph = defaultdict(set)
for _, row in ratings.iterrows():
    user, movie = row['user_id'], row['title']
    graph[user].add(movie)
    graph[movie].add(user)
```

The random walk algorithm is then designed as follows,

```
def random_walk(graph, start_node, walk_length=100):
    current = start_node
    visit_counts = defaultdict(int)

    for _ in range(walk_length):
        neighbors = list(graph[current])
        if not neighbors:
            break

        current = random.choice(neighbors)
        if isinstance(current, str):
            visit_counts[current] += 1

    return visit_counts
```

with the following 2 algorithms implemented for user recommendation, and finding similar movies respectively,

```
def recommend_for_user(graph, user_id, walk_length=100, top_n=5):
    if user_id not in graph:
        return []

    visits = random_walk(graph, user_id, walk_length)
```

```
    rated_movies = graph[user_id]
    sorted_movies = sorted(
        [(movie, count) for movie, count in visits.items() if movie not in
rated_movies],
        key=lambda x: x[1], reverse=True
    )

    top_movies = [movie for movie, _ in sorted_movies[:top_n]]
    result_df = pd.DataFrame({
        'Ranking': range(1, len(top_movies) + 1),
        'Movie Name': top_movies
    })
    result_df.set_index('Ranking', inplace=True)

    return result_df
```

```
def recommend_similar_movies(graph, movie_title, walk_length=100, top_n=5):
    if movie_title not in graph:
        return []

    visits = random_walk(graph, movie_title, walk_length)
    sorted_movies = sorted(visits.items(), key=lambda x: x[1], reverse=True)
    top_movies = [movie for movie, _ in sorted_movies if movie != movie_title][:top_n]

    result_df = pd.DataFrame({
        'Ranking': range(1, len(top_movies) + 1),
        'Movie Name': top_movies
    })
    result_df.set_index('Ranking', inplace=True)

    return result_df
```

The weighted graph is constructed as follows,

```
graph_weight = defaultdict(dict)

for _, row in ratings.iterrows():
    user = row['user_id']
    movie = row['title']
    rating = row['rating']

    graph_weight[user][movie] = rating
    graph_weight[movie][user] = rating
```

The random walk algorithm for the weighted graph approach is implemented below,

```
def biased_coin_random_walk(graph, start_node, walk_length=100):
    current = start_node
    visit_counts = defaultdict(int)
```

```
    for i in range(walk_length):
        neighbors = list(graph.get(current, {}).keys())
        if not neighbors:
            break

        accepted = False
        while not accepted:
            neighbor = random.choice(neighbors)
            rating = graph[current][neighbor]
            acceptance_prob = rating

            if random.random() <= acceptance_prob:
                accepted = True
                current = neighbor

        if isinstance(current, str):
            visit_counts[current] += 1

    return visit_counts
```

The user recommendation and movie similarity algorithms are shown below,

```
def recommend_for_user_biased_coin(graph, user_id, walk_length=100, top_n=5):
    if user_id not in graph:
        return []

    visits = biased_coin_random_walk(graph, user_id, walk_length)
    sorted_movies = sorted(visits.items(), key=lambda x: x[1], reverse=True)
    top_movies = [movie for movie, _ in sorted_movies[:top_n]]

    result_df = pd.DataFrame({
        'Ranking': range(1, len(top_movies) + 1),
        'Movie Name': top_movies
    })
    result_df.set_index('Ranking', inplace=True)

    return result_df
```

```
def recommend_similar_movies_biased_coin(graph_weight, movie_title, walk_length=100,
top_n=5):
    if movie_title not in graph_weight:
        return []

    visits = biased_coin_random_walk(graph_weight, movie_title, walk_length)
    sorted_movies = sorted(visits.items(), key=lambda x: x[1], reverse=True)
    top_movies = [movie for movie, _ in sorted_movies if movie != movie_title][:top_n]

    result_df = pd.DataFrame({
        'ranking': range(1, len(top_movies) + 1),
        'movie_name': top_movies
    })
```

```
    result_df.set_index('ranking', inplace=True)

    return result_df
```

5 Results

The collaborative filtering approach yielded the following 10 recommendations for user 1,

Ranking	Movie Name
1	Schindler's List (1993)
2	Casablanca (1942)
3	Glory (1989)
4	Close Shave, A (1995)
5	Lawrence of Arabia (1962)
6	Dr. Strangelove or: How I Learned to Stop Worr...
7	One Flew Over the Cuckoo's Nest (1975)
8	Butch Cassidy and the Sundance Kid (1969)
9	Rear Window (1954)
10	North by Northwest (1959)

The collaborative filtering approach yielded the following 10 similar movies for the movie "Star Wars (1977)",

Ranking	Movie Name
1	Return of the Jedi (1983)
2	Empire Strikes Back, The (1980)
3	Raiders of the Lost Ark (1981)
4	Indiana Jones and the Last Crusade (1989)
5	Butterfly Kiss (1995)
6	Man of the Year (1995)
7	New York Cop (1996)
8	Safe Passage (1994)
9	Destiny Turns on the Radio (1995)
10	Foreign Correspondent (1940)

It is harder to speculate about the recommendations for user 1 since there would have to be a comparison to the movies user 1 has previously rated. However, I think there could be a theme of older classic like "Schindler's List", "Casablanca", and "Lawrence of Arabia". I would say these recommendations are plausible since they share some themes.

As for the similar movies, the first 2 results certainly make sense. Star Wars is similar to other Star Wars movies. Raiders of the Lost Ark and Indiana Jones and the Last Crusade are stories written by George Lucs, the creator of Star Wars, and therefore share a lot of similarity. However, the latter 6 movies are

more of a mystery. I'm unsure of how these movies are connected or related to Star Wars.

The results for user recommendation for user 1 from the unweighted graph approach with a walk length of 100000 yielded,

1	Liar Liar (1997)
2	Scream (1996)
3	English Patient, The (1996)
4	Air Force One (1997)
5	Titanic (1997)
6	Mission: Impossible (1996)
7	Schindler's List (1993)
8	Ransom (1996)
9	Sense and Sensibility (1995)
10	Saint, The (1997)

These results are quite different from those from the collaborative filtering approach. The only similar movie is Schindler's List. These movies generally seem to share themes of thriller/action like "Air Force One", "Misson: Impossible", and "Ransom".

The results for similar movies to "Star Wars (1977)" from the unweighted graph approach with a walk length of 100000 yielded,

Ranking	Movie Name
1	Return of the Jedi (1983)
2	Contact (1997)
3	Liar Liar (1997)
4	English Patient, The (1996)
5	Fargo (1996)
6	Air Force One (1997)
7	Pulp Fiction (1994)
8	Raiders of the Lost Ark (1981)
9	Silence of the Lambs, The (1991)
10	Scream (1996)

Other than the science fiction movies, I don't see a connection to the other movies other than that all of the movies have mainstream appeal.

The following is the results for the user recommendation for user 1 and the similar movie algorithm with the same settings but using the weighted graph approach,

Ranking	Movie Name
1	Star Wars (1977)

```
2                         Contact (1997)
3                     Forrest Gump (1994)
4                          Titanic (1997)
5               Return of the Jedi (1983)
6                     Pulp Fiction (1994)
7                        Toy Story (1995)
8        Silence of the Lambs, The (1991)
9                       Braveheart (1995)
10                   Air Force One (1997)
```

```
ranking                        movie_name
1                             Fargo (1996)
2                           Contact (1997)
3                 Return of the Jedi (1983)
4           Raiders of the Lost Ark (1981)
5                           Toy Story (1995)
6                       Jerry Maguire (1996)
7                Princess Bride, The (1987)
8              English Patient, The (1996)
9         Silence of the Lambs, The (1991)
10         Empire Strikes Back, The (1980)
```

My comments for these results are the same as the other recommendations.

6 Conclusion

I think the results for from these algorithms are mixed. There are some indications that the movie similarity algorithms are working correctly for all algorithms since they all give either Star Wars movies or movies created by George Lucas. However, they still contain suggestions I don't understand.

One theme I notice is that almost all of the movies are movies that were received generally well and had mainstream appeal. This, to me, indicates that the algorithms are biased towards generally good and big movies. This also seems to happen in the graph based approach regardless of the walk being biased or not. The higher the walk lengths, the more the movies seem to converge on more generally well received movies.

With more time I would investigate the effects of manipulating the bias of the random walks and different walk lengths on the recommendations. I also think that z-scale normalization across movie's ratings, rather than across user's, could help avoid a convergence on popular movies. Z-scale normalization on movie's ratings would remove the mean which removes the influence of how well received movies are from the recommendations. I am unsure of how scaling by the

inverse of the standard deviation of a movie's ratings (normalizing for the polarization of a movie) would affect results.