CAI LAB 8 - CF RECOMMENDERS

Implement the two versions of the recommendation function, one that does user-to-user CF, and another that does item-to-item CF. Explain the main function.

First, we will explain the main function. We have to choose between item-to-item or user-to-user recommendations. Also, we can choose the *k* top similar users or items from which we will compare the introduced user or item and how many recommendations will appear on the output. Finally, we have done a function to read the introduced item or user depending on the type of recommendation we want.

After that, we developed the algorithm over user-to-user and item-to-item. They are very similar one to the other, but we have decided to make different functions to be able to use internal class functions. We will now explain the algorithm user-to-user. The other one is similar but uses the idea of recommending users that are expected to like a film (and choosing *k* top similar movies) instead of recommending movies to a user (and choosing *k* top similar users).

The algorithm first looks for the introduced user and the similarity between its ratings and the ratings in the database. We will do this sequentially over all available users, and compute the Pearson coefficient over the ratings of the same movie. So we have done the intersection of common films and computed the coefficient over these ratings. When we have all the ratings, we sort the resulting tuple and take the top *k* similar users. Later, we create a dictionary of *movie_id* from all users based on the set difference with the introduced user. So the resulting dictionary will be the not seen movies. After that, we compute the prediction of the user liking the film (not seen) *s*, based on the formula seen in the theory lectures. Afterwards, we select the top-rated movies and delete the predictions if they are under 2. We finally print this result.

Test your code with various lists of recommendations.

First of all, we want to test the recommender using our group of friends as a test. For that, we will put ourselves in the mind of:

- Our friend who likes sci-fi movies.
- Our exquisite friend who likes only sci-fi movies.
- Our friend who likes animation movies.
- Our exquisite friend who likes only animation movies.
- Our friend who likes romantic comedies.
- Our exquisite friend who likes only romantic comedies.
- Our friend who likes golden-age Hollywood musicals.
- Our exquisite friend who likes only golden-age Hollywood musicals
- Our friend who likes horror movies.
- Our exquisite friend who likes only horror movies.

To do so, we have coded a function that randomly selects a list of movies and rates them according to the criteria of each friend. This is done by extracting the *genre* feature of each film and giving a rating to the movie depending on the user preferences. For example, in the case of our friend who likes horror movies, we will give ratings above 4 to movies containing *Horror* in their *genre* list, ratings between 4 and 2 to movies with related genres in the *genre* list, such as Mystery, *Drama* or *Thriller*, and ratings below 2 to movies with no genres related. Furthermore, this classification will be extreme for the exquisite friends, where all films that do not contain the desired genre will have a rating of 1.

Now, it is time to experiment with the obtained ratings. Results for the ten users are attached with this report, but we are going to comment on some of them next.

For instance, in the case of our friend who likes sci-fi in a normal way (it is not an exquisite fan), we have generated the following input:

[('1129', '4.5'), ('6433', '0.5'), ('1952', '1.5'), ('57536', '4.0'), ('6387', '3.0'), ('2363', '4.5'), ('798', '1.0'), ('45517', '3.0'), ('129397', '4.5'), ('151769', '0.5'), ('26003', '0.5'), ('26838', '4.0'), ('1121', '2.0'), ('3501', '4.0'), ('2941', '1.0'), ('26554', '4.5'), ('150254', '4.0'), ('26176', '2.0'), ('5678', '2.5'), ('3594', '2.0'), ('29', '4.5'), ('71484', '4.5'), ('1959', '0.5'), ('44931', '2.0'), ('1499', '0.5'), ('126', '2.0'), ('8371', '4.5'), ('78174', '3.0'), ('2249', '3.0'), ('940', '0.5'), ('2052', '4.0'), ('3087', '3.0'), ('2748', '2.5'), ('391', '1.0'), ('7118', '4.5'), ('69406', '4.0'), ('164280', '1.5'), ('92509', '0.5'), ('6350', '4.5'), ('89580', '2.0'), ('757', '1.5'), ('7720', '3.0'), ('27255', '1.5'), ('115667', '4.0'), ('95207', '4.0'), ('4613', '2.5'), ('119145', '3.0'), ('155892', '3.5'), ('8368', '1.0'), ('240', '0.5'), ('54121', '1.0'), ('110330', '1.5'), ('70994', '3.0'), ('131578', '4.0'), ('96417', '1.5'), ('114818', '2.5'), ('62792', '0.5'), ('43908', '2.0'), ('108932', '3.5'), ('610', '4.5'), ('3986', '4.5'), ('1841', '0.5'), ('96608', '4.5'), ('6269', '2.0'), ('3729', '1.0'), ('26322', '0.5'), ('26528', '0.5'), ('142997', '3.0'), ('50999', '2.5'), ('1599', '2.0'), ('82461', '4.5'), ('49286', '3.5'), ('163985', '4.5'), ('26429', '3.0'), ('308', '3.5'), ('4158', '3.0'), ('176389', '4.0'), ('1968', '3.0'), ('34450', '3.5'), ('247', '1.0'), ('31193', '2.0'), ('5238', '1.0'), ('56782', '1.5'), ('126088', '3.5'), ('103539', '3.0'), ('1280', '1.0'), ('7032', '0.5'), ('100556', '2.0'), ('5361', '2.0'), ('135567', '4.5'), ('95510', '4.5'), ('3900', '3.5'), ('3372', '1.5'), ('129657', '2.0'), ('74916', '3.5'), ('45361', '3.0'), ('85397', '3.5'), ('944', '2.0'), ('198', '4.5'), ('2935', '3.0')]

Our recommender has compared his likes with all users and selected the 5 more similar ones. Then, as we wanted to recommend him up to 10 movies, we have got the following:

Here you have a ranking with the results:
1 . Silence of the Lambs, The (1991) with predicted rate: 3.6
2 . Pulp Fiction (1994) with predicted rate: 3.5
3 . Forrest Gump (1994) with predicted rate: 3.5
4 . Jurassic Park (1993) with predicted rate: 3.4
5 . Batman (1989) with predicted rate: 3.4
6 . Memento (2000) with predicted rate: 3.4
7 . Donnie Darko (2001) with predicted rate: 3.4
8 . Beauty and the Beast (1991) with predicted rate: 3.4
9 . WALL·E (2008) with predicted rate: 3.4
10 . Braveheart (1995) with predicted rate: 3.3

In case our friend is really exquisite with not-sci-fi movies, we have generated the following input:

[('1129', '4.5'), ('6433', '1.0'), ('1952', '1.0'), ('57536', '1.0'), ('6387', '1.0'), ('2363', '4.5'), ('798', '1.0'), ('45517', '1.0'), ('129397', '4.5'), ('151769', '1.0'), ('26003', '1.0'), ('26838', '1.0'), ('1121', '1.0'), ('3501', '1.0'), ('2941', '1.0'), ('26554', '4.5'), ('150254', '1.0'), ('26176', '1.0'), ('5678', '1.0'), ('3594', '1.0'), ('29', '4.5'), ('71484', '4.5'), ('1959', '1.0'), ('44931', '1.0'), ('1499', '1.0'), ('126', '1.0'), ('8371', '4.5'), ('78174', '1.0'), ('2249', '1.0'), ('940', '1.0'), ('2052', '1.0'), ('3087', '1.0'), ('2748', '1.0'), ('391', '1.0'), ('7118', '4.5'), ('69406', '1.0'), ('164280', '1.0'), ('92509', '1.0'), ('6350', '4.5'), ('89580', '1.0'), ('757', '1.0'), ('7720', '1.0'), ('27255', '1.0'), ('115667', '1.0'), ('95207', '1.0'), ('4613', '1.0'), ('119145', '1.0'), ('155892', '1.0'), ('8368', '1.0'), ('240', '1.0'), ('54121', '1.0'), ('110330', '1.0'), ('70994', '1.0'), ('131578', '1.0'), ('96417', '1.0'), ('114818', '1.0'), ('62792', '1.0'), ('43908', '1.0'), ('108932', '1.0'), ('610', '4.5'), ('3986', '4.5'), ('1841', '1.0'), ('96608', '4.5'), ('6269', '1.0'), ('3729', '1.0'), ('26322', '1.0'), ('26528', '1.0'), ('142997', '1.0'), ('50999', '1.0'), ('1599', '1.0'), ('82461', '4.5'), ('49286', '1.0'), ('163985', '4.5'), ('26429', '1.0'), ('308', '1.0'), ('4158', '1.0'), ('176389', '1.0'), ('1968', '1.0'), ('34450', '1.0'), ('247', '1.0'), ('31193', '1.0'), ('5238', '1.0'), ('56782', '1.0'), ('126088', '1.0'), ('103539', '1.0'), ('1280', '1.0'), ('7032', '1.0'), ('100556', '1.0'), ('5361', '1.0'), ('135567', '4.5'), ('95510', '4.5'), ('3900', '1.0'), ('3372', '1.0'), ('129657', '1.0'), ('74916', '1.0'), ('45361', '1.0'), ('85397', '1.0'), ('944', '1.0'), ('198', '4.5'), ('2935', '1.0')]

Our recommender has compared his likes with all users and selected the 5 more similar ones. Then, as we wanted to recommend him up to 10 movies, we have got the following:

Here you have a ranking with the results:
1 . Shawshank Redemption, The (1994) with predicted rate: 2.4
2 . Pulp Fiction (1994) with predicted rate: 2.2
3 . Forrest Gump (1994) with predicted rate: 2.2
4 . Matrix, The (1999) with predicted rate: 2.1
5 . Dark Knight, The (2008) with predicted rate: 2.1
6 . Léon: The Professional (a.k.a. The Professional) (Léon) (1994) with predicted rate: 2.1
7 . Jurassic Park (1993) with predicted rate: 2.1
8 . Godfather, The (1972) with predicted rate: 2.1
9 . Godfather: Part II, The (1974) with predicted rate: 2.1
10 . Star Wars: Episode V - The Empire Strikes Back (1980) with predicted rate: 2.1

From these tests, we can observe that the predicted ratings for exquisite users are significantly lower than the ones for the other users, which is not surprising due to the high requirement level of the exquisite users. Moreover, if we look at the predicted movies, we can see that exquisite users are more likely to get recommendations of their genre of interest, which makes sense as they are not as tolerant as other users with other genres.

See if the recommendations make sense to you, and if you see any difference in quality between user-to-user CF and item-to-item CF.

User-based CF works by finding similar users to the target user and using their ratings to predict the target user's preferences. It is generally more effective when the user has rated a relatively small number of items, as it can take advantage of the ratings of other similar users to make more accurate predictions. However, it can be less effective when the user has rated a large number of items, as the similarity between users may not be as strong when there are more ratings to consider. Item-based CF works by finding similar items to the items that the target user has rated and using those similarities to predict the target user's preferences for other items. It is generally more effective when the user has rated a relatively large number of items, as it can take advantage of the similarities between the items to make more accurate predictions. However, it can be less effective when the user has rated a small number of items, as there may not be enough data to accurately compute the similarities between items. In general, user-based CF and item-based CF can both be useful approaches for generating recommendations, and the choice of which method to use may depend on the specific characteristics of the dataset and the preferences of the users.

Apart from that, if we analyze all recommendations we will see that, despite some movies corresponding to the favorite genre of the user we are recommending, users that like a specific genre get recommended movies from other ones. This can be tricky and lead us to think that our recommendation algorithm is not working properly, but there are some other factors that we must consider. For instance, even if users like movies from a specific genre, they can also like some random film from another one (it is possible that a horror film fan loves “Toy Story”). Also, some movies are not considered to be part of a specific genre but have elements related to that. Furthermore, if we take into consideration all elements of a movie (acting, soundtrack, art, etc.) we can see that genre is not the only criterion to categorize them. All this, added to the fact that tastes and preferences are not exact sciences, can make the behavior of our predictions differ from what we expected.

Analyze (big-Oh style) the time and memory costs of your recommender functions.

The time complexity of the function *recommend_user_to_user* is $O(n \cdot k \cdot m)$, where n is the number of similar users in the dataset, k is the number of similar users we want to select, and m is the number of films that the new user has not rated. The inner loop that iterates over *top_similar_users* has a time complexity of $O(k)$, since it will run for a maximum of k iterations. The loop that iterates over *film_rating_pred.keys()* has a time complexity of $O(m)$, since it will run for a maximum of m iterations. These two loops are nested, so the overall time complexity of the inner loop is $O(k \cdot m)$. The outer loop that iterates over *self._user_ratings* has a time complexity of $O(n)$, since it will run for a maximum of n iterations. This outer loop contains the inner loop, so the overall time complexity of the function is $O(n \cdot k \cdot m)$. The memory complexity of this function is also $O(n \cdot k \cdot m)$, since it stores the ratings for all n users, the similarities between the new user and the k most similar users, and the predicted ratings for the m films that the new user has not rated.

The same happens for the function *recommend_item_to_item*. Its time complexity is also $O(n \cdot k \cdot m)$, where n is the number of movies in the dataset, k is the number of similar movies we want to select, and m is the number of users who have not rated the new movie. The inner loop that iterates over *top_similar_films* has a time complexity of $O(k)$, since it will run for a maximum of k iterations. The loop that iterates over *user_rating_pred.keys()* has a time complexity of $O(m)$, since it will run for a maximum of m iterations. These two loops are nested, so the overall time complexity of the inner loop is $O(k \cdot m)$. The outer loop that iterates

over *self.movieid_list()* has a time complexity of *O(n)*, since it will run for a maximum of *n* iterations. This outer loop contains the inner loop, so the overall time complexity of the function is *O(n·k·m)*. The memory complexity of this function is also *O(n·k·m)*, since it stores the ratings for all *n* movies, the similarities between the new movie and the *k* most similar movies, and the predicted ratings for the *m* users who have not rated the new movie.
Finally, the time complexity of the function *compute_cosine_similarity_user* (same for *compute_cosine_similarity_item*) is *O(n·n)*, where *n* is the number of common films that both users have rated (or the number of people who have rated both movies, in the other case). The loop that iterates over *common_films* has a time complexity of *O(n)*, since it will run for a maximum of *n* iterations. The two loops that iterate over *user1_ratings* and *user2_ratings* each have a time complexity of *O(n)*, since they will each run for a maximum of *n* iterations. So the overall time complexity of the function is *O(n·n+n·n) = O(n·2n) = O(n·n)* (analogous conclusions for the loops in *compute_cosine_similarity_item*). The memory complexity of this function is *O(n)* (same for *compute_cosine_similarity_item*), since it stores the ratings for the *n* common films in the lists *user1_match* and *user2_match*, and the product of their ratings in the list product.

What would be the major changes if you did not have ratings in 0..5, but just boolean “liked/disliked” ratings? Or just positive “likes”?

If the ratings are binary "liked/disliked" or just positive "likes," this would affect the way that collaborative filtering algorithms are implemented. In user-based collaborative filtering, the similarity between users is typically computed using a distance measure such as cosine similarity, which takes into account the magnitudes and directions of the ratings vectors. With binary ratings, the magnitudes of the ratings vectors would always be 1, so the cosine similarity would not be able to capture the differences in the ratings between users. Instead, other similarity measures such as Jaccard similarity could be used, which only consider the presence or absence of ratings rather than their magnitudes. In item-based collaborative filtering, the similarities between items are typically computed using a distance measure such as cosine similarity or Pearson correlation, which take into account the ratings that users have given to the items. With binary ratings, these measures would not be able to capture the differences in the ratings between items. Instead, other similarity measures such as Jaccard similarity or Dice coefficient could be used, which only consider the presence or absence of ratings rather than their magnitudes.
For instance, probably something like this would work:

```
def compute_jaccard_similarity_user(user1_ratings, user2_ratings):
    # get the common films that both users have rated
    common_films = set([id for (id, rate) in user1_ratings]).intersection(set([id for (id, rate) in user2_ratings]))
    # if the users have not rated any common films, return 0
    if len(common_films) == 0:
        return 0
    # compute the Jaccard similarity between the ratings of the two users
    num_common_likes = 0
    for film in common_films:
        # get the rating for each film in common
        for (id, rate) in user1_ratings:
            if film == id and rate == 1:
                num_common_likes += 1
                break
        for (id, rate) in user2_ratings:
            if film == id and rate == 1:
                num_common_likes += 1
                break
    return num_common_likes / len(common_films)
```
