School of Computing

CA400 Year

Machine learning Recommendation Model & Evaluation

Genesis Uwumangbe 20459666 & Gideon Amaechi 20364806

At the bottom of this document is a list of references which helped us finalise this model

Our recommendation model is known as a Hybrid Recommender System it combines content based filtering and collaborative based filtering which in hand provides a more personal recommendation.[6]

Collaborative Filtering:
User-Based: This system primarily uses user-based collaborative filtering, where recommendations are generated based on the preferences of similar users. The system calculates similarity scores between users using a cosine similarity matrix derived from user preferences for different movie genres. Users with higher similarity scores are considered similar.[2]

Genre Preferences: It also incorporates elements of content-based filtering by considering the genres of the movies. The system collects genre data for each movie and uses this information to create genre profiles for users, which then influence recommendations based on user similarity.

We created our recommendation model on google collab we used libraries such as:

Pandas - Pandas is a library used for data manipulation and analysis. It provides a high-level interface for working with tabular data, such as CSV files or SQL database tables.
Pymongo - is a python library used to load your databases from mongo DB

Sklearn library-Is used for machine learning in Python. It includes a wide range of functions for supervised and unsupervised learning, including classification, regression, clustering, and dimensionality reduction

Sklearn.feature_extraction.text -This provides utilities to convert raw text data into feature vectors that can be used as input to machine learning algorithms

Sklearn.metrics.pairwise - This provides various pairwise distance and similarity metrics for evaluating the similarity between pairs of samples.

The vision for our recommendation model was to initially collect the users preferences and store these preferences inside our database.

In the above image it shows how the users preferences are stored. Once the preferences have been collected. It will then be used in our initial content based recommendation of projects. Inside our mongo DB database it includes a collection with all the projects from last year's expo (2022/23) . These projects have a section that includes program, technology and area. Essentially we will be pattern matching these preferences to the subheadings and we will be giving weighed scores based on what type of match it is, for example if its a program that matches their preference they will get plus two added to the score we will then list the projects in decreasing order this will then leave the most suited project to be listed at the top

```python
def get_default_recommendations(user_id):
    # Retrieve the user's preferences from the database
    preferences = PreferencesCollection.find_one({'user_id': user_id})

    # If theres no preferences found for the user then an empty list is returned
    if not preferences:
        return []

    # Extracts and turns the user's preferred programs, technologies, and areas to lowercase
    preferred_programs = [prog.lower() for prog in preferences['preferences'].get('programs', [])]
    preferred_technologies = [tech.lower() for tech in preferences['preferences'].get('technologies', [])]
    preferred_areas = [area.lower() for area in preferences['preferences'].get('Areas', [])]

    # Retrieve all projects from the database
    all_projects = list(AllProjects.find())

    # Initialize a list to store recommended projects
    recommended_projects = []

    # Used to calculate scores and filter through the projects
    for project in all_projects:
        score = 0
        project_programs = project['student_programme'].lower()
        project_areas = [area.lower().strip() for area in project['project_area'].split(',')]
        project_technologies = [tech.lower().strip() for tech in project['project_technology'].split(',')]

        # Check if the project's program matches any of the user's preferred programs
        if project_programs in preferred_programs:
            score += 2

        # Check if any of the project's areas matches any of the user's preferred areas
        if any(area in project_areas for area in preferred_areas):
            score += 1

        # Same as aboev but for technologies
        if any(tech in project_technologies for tech in preferred_technologies):
            score += 1

        # Only take 2 or higher scores as recommended
        if score >= 2:
            recommended_projects.append((score, project['project_id']))

    # Sort recommended projects based on scores in descending order
    recommended_projects.sort(reverse=True, key=lambda x: x[0])

    # Extract only the project IDs from the sorted list
    return [project_id for _, project_id in recommended_projects]
```

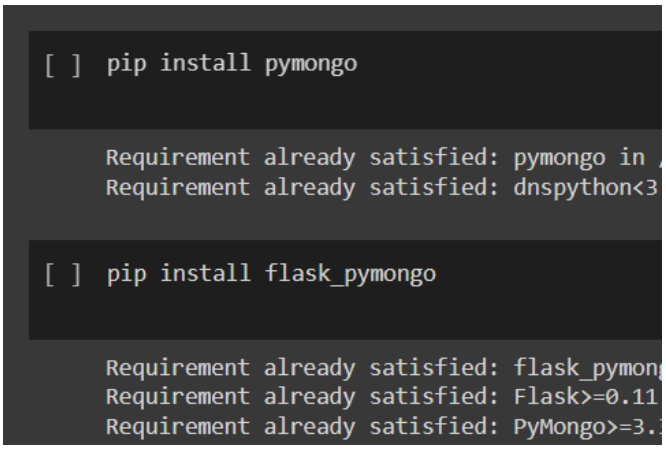
In the above image is how we computed this functionality along with comments

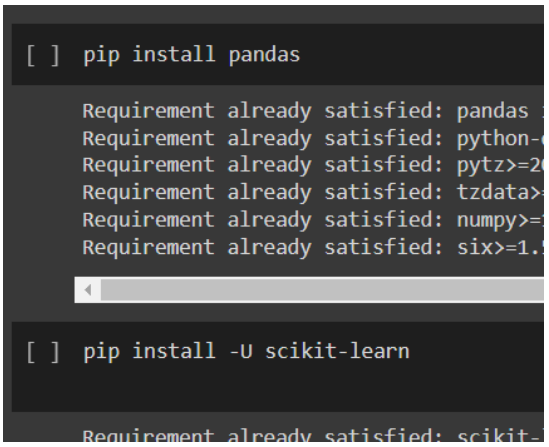
Content based filtering to Collaborative filtering model.

This model would be introduced to the user once they have rated 5 projects or more. The implementation started off by us retrieving and cleaning the data. There were a lot of hiccups due to our data being in json

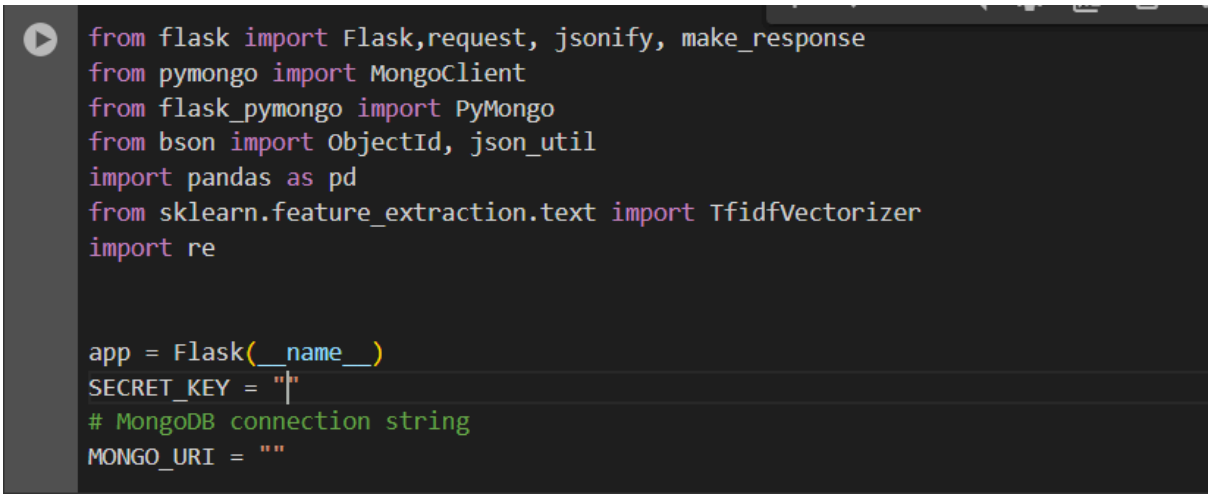
format. We had to convert it to a dataframe that can be used for training and testing.

we started off by downloading all the dependencies needed on google collab

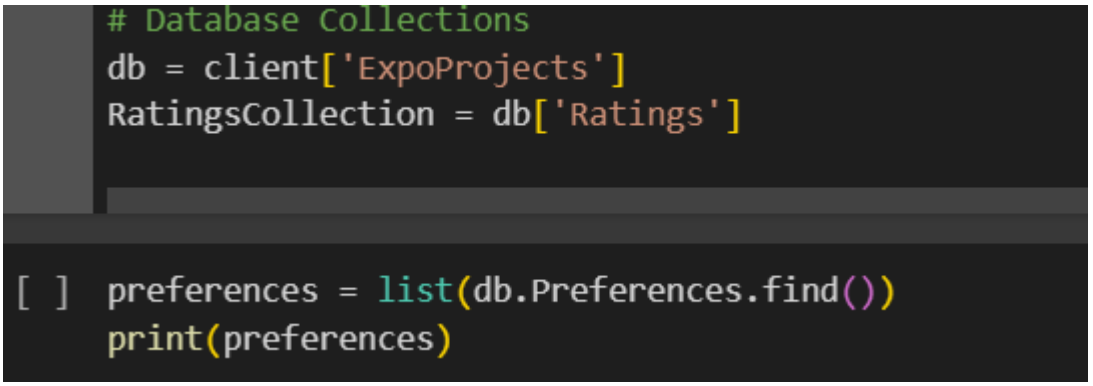




Then we imported our mongo db database.



From our database we called the collections that we wished to use which were ratings and preferences.



As mentioned earlier our data was in json format and some of the the preference terms had spaces in them and special characters so we made a function that would pre process the data essentially it would replace the spaces and parentheses with underscore and also to keep words that have a period together e.g Next.js will be kept as one word.

```python
def preprocess_preference(preference):
    # Replace spaces and parentheses with underscores to keep terms together
    # No need to escape periods here
    return re.sub(r'[\s\(\)]+', '_', preference)
```

So in order to create the data frame that showcases users and their preferences we loop through each user's preference and process each preference category to combine all the preferences together. We then created a data frame with the user ids and preferences being listed.
We wanted to get a binary matrix representation so we added a CountVectorizer to the all preferences column below is how we conducted this.

```python
# Process the preferences data
user_ids = []
all_preferences = []

# Loop through each user's preferences
for user_pref in preferences:
    user_ids.append(user_pref['user_id'])
    # Preprocess each preference category and combine them
    programs = ' '.join(preprocess_preference(p) for p in user_pref['preferences']['programs'])
    technologies = ' '.join(preprocess_preference(t) for t in user_pref['preferences']['technologies'])
    areas = ' '.join(preprocess_preference(a) for a in user_pref['preferences']['Areas'])
    combined_prefs = ' '.join([programs, technologies, areas])
    all_preferences.append(combined_prefs)

# Create a DataFrame
preferences_df = pd.DataFrame({
    'user_id': user_ids,
    'all_preferences': all_preferences
})

# Initialize CountVectorizer with binary=True to get a binary matrix representation
vectorizer = CountVectorizer(binary=True, token_pattern=r'(?u)\b[\w_]+\b')

# Apply the vectorizer to the 'all_preferences' column to create binary feature vectors
X_binary = vectorizer.fit_transform(preferences_df['all_preferences'])
binary_df = pd.DataFrame(X_binary.toarray(), columns=vectorizer.get_feature_names_out(), index=preferences_df['user_id'])
```

Below is an image of our binary_df as you can see the user ids are placed as rows and the preferences are columns. The 0 represents a user has not chosen the item as their preference and a 1 indicates that they have .

user_id	android	artificial_intelligence	biomedical_engineering	c	cloud_computing	computer_applications	css	data_analytics	data_science	digital_signal_processing
10	0	0	0	0	1	1	0	0	1	0
11	0	0	0	1	0	0	0	1	1	0
12	1	0	0	0	1	1	0	0	0	0
13	1	0	0	0	1	1	0	0	0	0
14	1	0	0	0	0	1	0	0	1	0
7	0	1	0	0	0	0	0	0	0	0
16	0	0	0	0	0	0	0	0	0	0
17	0	1	0	0	1	0	1	0	1	0
20	0	0	1	0	0	0	0	0	0	1

In order to find the most similar users we decided to use the cosine similarity[2] which was imported from sklearn.metrics.pairwise.
This graph showcases which users are more similar as we can see for user 10 and its corresponding 10 the value is 1 so they are identical

user_id	10	11	12	13	14	7	16	17	20
user_id									
10	1.000000	0.300000	0.527046	0.700000	0.559017	0.000000	0.000000	0.632456	0.000000
11	0.300000	1.000000	0.210819	0.200000	0.447214	0.182574	0.000000	0.421637	0.000000
12	0.527046	0.210819	1.000000	0.632456	0.353553	0.000000	0.000000	0.222222	0.000000
13	0.700000	0.200000	0.632456	1.000000	0.447214	0.000000	0.000000	0.421637	0.158114
14	0.559017	0.447214	0.353553	0.447214	1.000000	0.204124	0.000000	0.471405	0.000000
7	0.000000	0.182574	0.000000	0.000000	0.204124	1.000000	0.000000	0.384900	0.000000
16	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	1.000000	0.000000	0.223607
17	0.632456	0.421637	0.222222	0.421637	0.471405	0.384900	0.000000	1.000000	0.000000
20	0.000000	0.000000	0.000000	0.158114	0.000000	0.000000	0.223607	0.000000	1.000000

We then created a test to see which user is more similar to user 10 and it says user 13 is more similar. Let's prove this by analysing their preferences in the collection

```
# Replace 'target_user_id' with the actual user ID you w
target_user_id = user_ids[0]  # Example: the first user
user_similarity = cosine_sim_df.loc[target_user_id]

# Display the similarity scores
print(user_similarity)
```

```
user_id
10    1.000000
11    0.300000
12    0.527046
13    0.700000
14    0.559017
7     0.000000
16    0.000000
17    0.632456
20    0.000000
Name: 10, dtype: float64
```

Here is user 10's preferences

and here's user 13's preferences

Collaborative filtering section

Active User Identification:

The code starts by setting the active_user_id, which represents the user for whom we want to generate recommendations. This is just for testing and wont be used when implemented to our code

Similar User Retrieval:

It retrieves a list of similar users to the active user based on cosine similarity scores stored in the cosine_sim_df DataFrame. Similar users are those with a similarity score greater than 0.[2][4][5]

Weighted Ratings Calculation:

For each similar user, the code fetches their ratings from a MongoDB database stored in a collection named RatingsCollection. We only take ratings that are 3 and above as we believe they are positive ratings. It then calculates a weighted rating for each project the similar user has rated. The weighting is done by multiplying the user's rating by the similarity score between the similar user and the active user.

Top Recommendations Generation:

The projects are sorted based on their aggregated weighted scores in descending order to determine the top recommendations for the active user.

Filtering Already Rated Projects:

Projects that the active user has already rated are filtered out from the list of recommendations.

Top N Recommendations:

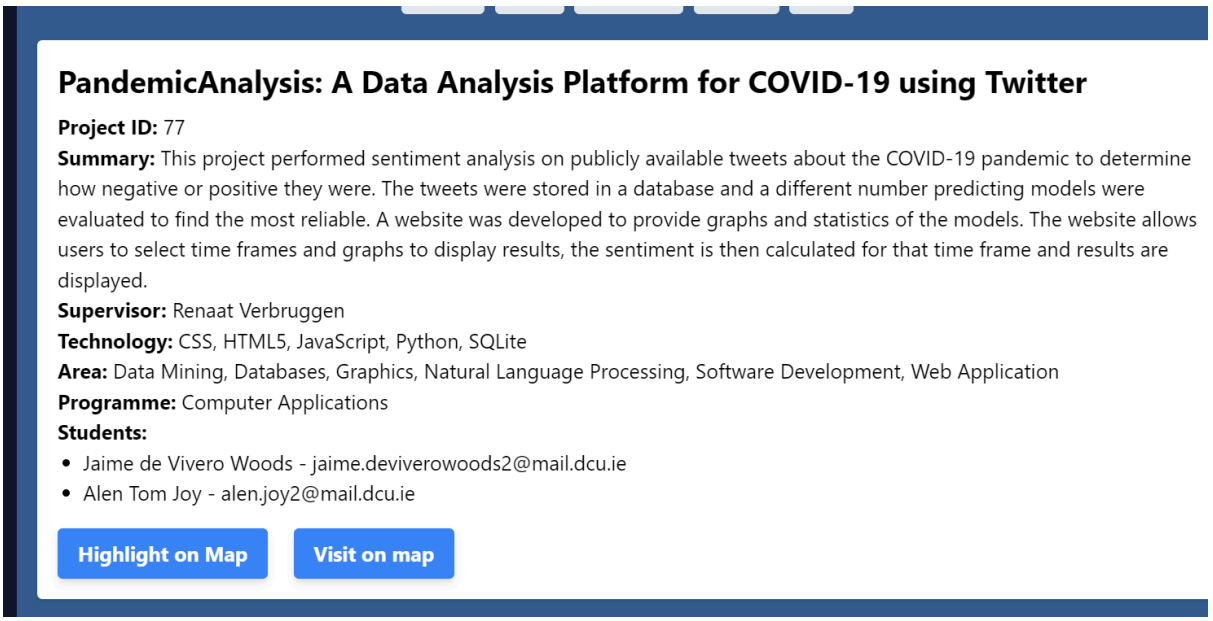
Finally, the code selects the top N recommendations (specified by the variable N) to display to the active user.

```python
# Active user ID for whom we want to generate recommendations
active_user_id = 10

# Let's assume 'cosine_sim_df' is your cosine similarity DataFrame
# For the active user, get the list of similar users and their similarity scores
similar_users_series = cosine_sim_df.loc[active_user_id]
similar_users = similar_users_series.index[similar_users_series > 0].tolist()  # Exclude zero similarity scores

# Fetch ratings from similar users and weight them by similarity score
weighted_ratings = {}
for similar_user in similar_users:
    # Retrieve the ratings from MongoDB
    user_ratings = RatingsCollection.find_one({'user_id': similar_user})
    if user_ratings and 'ratings' in user_ratings:
        for rating in user_ratings['ratings']:
            project_id = rating['project_id']
            project_rating = rating['rating']
            # Filter out ratings less than 3
            if project_rating >= 3:
                if project_id not in weighted_ratings:
                    weighted_ratings[project_id] = 0
                # Multiply the rating by the similarity score and accumulate it
                weighted_ratings[project_id] += project_rating * similar_users_series[similar_user]

# Sort projects by the aggregated score in descending order
recommended_projects = sorted(weighted_ratings.items(), key=lambda item: item[1], reverse=True)

# Retrieve projects that the active user has already rated
active_user_ratings = RatingsCollection.find_one({'user_id': active_user_id}, {'_id': 0, 'ratings': 1})
active_user_rated_projects = set()
if active_user_ratings:
    active_user_rated_projects = {rating['project_id'] for rating in active_user_ratings['ratings']}

# Filter out projects the active user has already rated
final_recommendations = [project for project in recommended_projects if project[0] not in active_user_rated_projects]

# Top N recommended projects based on similar user ratings
N = 5  # Number of recommendations you want to show
top_n_recommendations = final_recommendations[:N]

# Display the top N recommended projects
print(f"Top {N} recommendations for user {active_user_id}: {top_n_recommendations}")
```

Below are the results from the whole model.

```
7] final_recommendations

[('77', 8.64425308427871),
 ('38', 8.612282366598492),
 ('43', 8.21720680758398),
 ('57', 6.82490710000944),
 ('31', 6.462277660168379),
 ('44', 6.29834563766817),
 ('178', 5.829822128134703),
 ('49', 5.430316355348387),
```

Project 77 is known to be the most recommended project for user 10 based on what the most similar users have rated highly let's have a look at project 77.



This project matches with the users preferences and also the most similar user to them have rated it highly.

Training With Movie Dataset and Evaluation

We did not have enough data to test the model so we had to find another approach just for validation purposes.

To ensure our model is usable we decided to find a dataset online the dataset we chose to validate our model on was the MovieLens Dataset It contains 20000263 ratings and 465564 tag applications across 27278 movies. These data were created by 138493 users between January 09, 1995 and March 31, 2015. This dataset was generated on October 17, 2016.[1] we obtained the movies dataset and the ratings dataset and we loaded it using pandas. We chose this data set due to its robustness and having the similar qualities to our original dataset.

Once we have validated that it has produced reasonable results for movies We had confidence in it and went ahead to integrate it with our approach for project recommendations

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt t
```

```
import pandas as pd
moviesdata = '/content/drive/My Drive/movie.csv'
ratingsdata = '/content/drive/My Drive/rating.csv'
```

```
ratings = pd.read_csv(ratingsdata)
movies = pd.read_csv(moviesdata)
```

The data set did not have errors, it was clean but it was too large for us to run on google collab. So we had to find ways to minimise it. We loaded the first 600 users to train the model and only looked at movies rated 4.5 and above from them.

We used Genres as user preferences this is due to there not being a preferences section. So we assumed that the movies rated 4.5 - 5 by the user means that the genre for those movies are their preferences. We also only looked at 30 ratings per user

```
# Filter ratings to include only ratings greater than or equal to 4
high_ratings = ratings[ratings['rating'] >= 4.5]

# Filter the ratings dataframe to include only the first 2000 users
ratings_first_2000_users = high_ratings[high_ratings['userId'] <= 2000]

# Sample 30 ratings per user
sampled_ratings = ratings_first_2000_users.groupby('userId').apply(lambda x: x.sample(min(len(x), 30), random_state=1)).reset_index(drop=True)

# Merge datasets to get genre information
sampled_data = pd.merge(sampled_ratings, movies, on='movieId')

# Split and explode genres into separate rows
sampled_data['genres'] = sampled_data['genres'].str.split('|')
sampled_data = sampled_data.explode('genres')
sampled_data = sampled_data[sampled_data['genres'] != '(no genres listed)']
```

We then created a binary data frame as mentioned earlier

userId	Action	Adventure	Animation	Children	Comedy	Crime	Documentary
1	1	1	0	0	0	1	0
2	1	1	0	0	1	1	0
3	1	1	0	1	1	1	0
4	1	1	0	0	0	0	0
5	1	1	1	1	1	1	0
...	...	...	...	...	...	...	...
1996	0	0	0	0	1	0	1
1997	1	1	1	1	1	1	0
1998	1	1	1	1	1	1	0
1999	1	1	1	1	1	0	0
2000	1	1	0	0	1	1	1

userId	Drama	Fantasy	Film-Noir	Horror	IMAX	Musical	Mystery	Romance
1	1	1	0	1	1	0	0	0
2	1	0	1	1	0	0	1	1
3	1	1	0	1	0	0	1	1
4	1	0	0	0	0	0	0	0
5	1	1	0	0	1	1	0	1
...	...	...	...	...	...	...	...	...
1996	1	0	0	0	0	1	1	1
1997	1	1	1	1	1	1	1	1
1998	1	1	0	0	1	1	0	1
1999	1	1	0	1	0	1	0	1
2000	1	1	1	1	1	1	1	1

We calculated the cosine similarity

userId	1	2	3	4	5	6	7
userId							
1	1.000000	0.492366	0.615457	0.288675	0.589256	0.288675	0.566139
2	0.492366	1.000000	0.818182	0.426401	0.609272	0.533002	0.752618
3	0.615457	0.818182	1.000000	0.426401	0.696311	0.746203	0.919866
4	0.288675	0.426401	0.426401	1.000000	0.408248	0.500000	0.392232
5	0.589256	0.609272	0.696311	0.408248	1.000000	0.714435	0.800641
...	...	...	...	...	...	...	...
1996	0.154303	0.569803	0.569803	0.267261	0.545545	0.534522	0.628971
1997	0.632456	0.778499	0.856349	0.365148	0.819892	0.730297	0.930949
1998	0.566139	0.668994	0.752618	0.392232	0.960769	0.686406	0.846154
1999	0.492366	0.636364	0.818182	0.426401	0.783349	0.746203	0.919866
2000	0.566139	0.752618	0.752618	0.392232	0.640513	0.588348	0.769231

Similar User Identification:

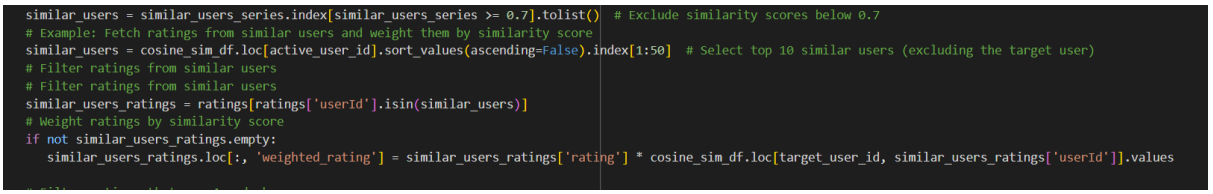
We retrieve the similarity scores between the active user and all other users from the cosine similarity DataFrame (cosine_sim_df).

We filter out users with similarity scores below 0.7 and extract their user IDs as a list of similar users to that active user.[3]

```
active_user_id = 70

# Let's assume 'cosine_sim_df' is your cosine similarity DataFrame
# For the active user, get the list of similar users and their similarity scores
similar_users_series = cosine_sim_df.loc[active_user_id]
similar_users = similar_users_series.index[similar_users_series >= 0.7].tolist()  #
# Example: Fetch ratings from similar users and weight them by similarity score
```

Top Similar Users Selection:

We sort the similar users by their similarity scores in descending order. We select the top 50 similar users (excluding the active user) to consider for recommendations We filter the ratings dataframe (ratings) to include only ratings from the selected similar users. We then calculated a weighted rating for each movie that the similar users have rated. The weighting is done by multiplying each rating by the corresponding similarity score between the similar user and the active user.



We filter out ratings that are below a threshold of 4 after weighting. Then we aggregate the weighted ratings for each movie by summing them up.

Recommendation Generation:

Next we sort the movies based on their aggregated scores in descending order to generate a list of recommended movies for the active user.

Filtering Already Rated Movies:

We retrieve the movies that the active user has already rated.

We filter out these movies from the recommended list to avoid recommending duplicates.

Displaying Recommendations:

Finally, we print the recommended movies for the active user.

```python
# Filter ratings that are 4 and above
similar_users_high_ratings = similar_users_ratings[similar_users_ratings['weighted_rating'] >= 4]
# Aggregate scores for each movie
aggregated_scores = similar_users_high_ratings.groupby('movieId')['weighted_rating'].sum()
# Sort movie IDs by their aggregated score in descending order
recommended_movies = aggregated_scores.sort_values(ascending=False).index

# Retrieve movies that the active user has rated already
active_user_rated_movies = ratings[ratings['userId'] == active_user_id]['movieId'].tolist()

# Filter out movies that the active user has rated already from the recommendations
recommended_movies_filtered = [movie_id for movie_id in recommended_movies if movie_id not in active_user_rated_movies]

# Display the recommended movies
print(recommended_movies_filtered)
```

```
[50, 356, 2324, 1050, 345, 1747, 1694, 232, 1041, 1734, 58, 21, 300, 36, 223, 2959, 457, 1299, 1273, 1271, 1953, 2009, 12
```

We chose to evaluate our recommender system by checking the users preferred genres and we will look at the top 5 recommended movies genres for that user and check if there's similarities

```python
# Example user ID

preferred_genres = binary_df.loc[active_user_id][binary_df.loc[active_user_id] == 1].index.tolist()
print(preferred_genres)

top_5_recommended_movies = recommended_movies_filtered[:5]
recommended_genres = set()
# Step 2: Extract genres for each movie separately
for movie_id in top_5_recommended_movies:
    genres = movies[movies['movieId'] == movie_id]['genres'].iloc[0]
    recommended_genres.update(genres.split('|'))



# Step 3: Display the genres for each movie
print(recommended_genres)
```

```
['Comedy', 'Crime', 'Documentary', 'Romance', 'Thriller']
{'Documentary', 'Thriller', 'Crime', 'War', 'Romance', 'Drama', 'Mystery', 'Comedy'}
```

We then calculated the precision at k, mean reciprocal rank, Recall and the F1 score to see how accurate our model is. We done this for individual users and then we applied it to the whole dataset to find the average scores

```python
def calculate_precision_at_k(preferred_genres, recommended_genres, k):
    intersection = len(set(actual_genres).intersection(recommended_genres[:k]))
    precision = intersection / k if k > 0 else 0
    return precision

def calculate_mrr(preferred_genres, recommended_genres):
    for i, genre in enumerate(recommended_genres, start=1):
        if genre in preferred_genres:
            return 1 / i
    return 0

# Initialize variables to store metrics
```

```python
# Calculate recall for the active user
recall = covered_genres_count / num_preferred_genres if num_preferred_genres > 0 else 0
recall = round(recall, 2)

f1_score = 2 * (precision_at_k * recall) / (precision_at_k + recall) if (precision_at_k + recall) > 0 else 0
f1_score = round(f1_score, 2)
```

Results

For user 70

```
Recall for the active user: 1.0
Precision at k: 0.6
MRR: 1.0
F1 score for the active user: 0.75
```

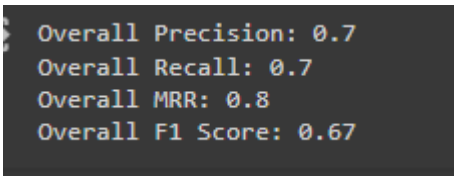
Here is how we captured the models performance for the overall dataset

```python
def get_recommended_movies(user_id, n=5):
    # Fetch top similar users based on cosine similarity
    similar_users = cosine_sim_df.loc[user_id].sort_values(ascending=False).index[1:11]
    # Collect movies rated highly by similar users
    similar_users_ratings = ratings[(ratings['userId'].isin(similar_users)) & (ratings['rating'] >= 4)]
    # Get top N movies based on the frequency of high ratings
    top_movies = similar_users_ratings.groupby('movieId').size().sort_values(ascending=False).head(n).index
    return top_movies.tolist()

def calculate_metrics_for_user(user_id, recommended_movie_ids):
    # Fetch genres for recommended movies
    recommended_genres = set(movies[movies['movieId'].isin(recommended_movie_ids)]['genres'].str.split('|').explode().unique())
    # Fetch user's preferred genres
    preferred_genres = set(binary_df.columns[binary_df.loc[user_id] == 1])


    # Calculate precision at k
    intersection = len(preferred_genres.intersection(recommended_genres))
    precision = intersection / len(recommended_genres) if recommended_genres else 0

    # Calculate recall
    recall = intersection / len(preferred_genres) if preferred_genres else 0

    # Calculate MRR
    mrr = 0
    for i, genre in enumerate(recommended_genres, start=1):
        if genre in preferred_genres:
            mrr = 1 / i
            break

    # Calculate F1 score
    f1_score = 2 * precision * recall / (precision + recall) if (precision + recall) else 0

    return precision, recall, mrr, f1_score
```

```python
# Initialize accumulators
total_precision = 0
total_recall = 0
total_mrr = 0
total_f1 = 0
total_users = 0

# Calculate metrics for each user
for user_id in binary_df.index:
    recommended_movie_ids = get_recommended_movies(user_id)
    precision, recall, mrr, f1_score = calculate_metrics_for_user(user_id, recommended_movie_ids)

    total_precision += precision
    total_recall += recall
    total_mrr += mrr
    total_f1 += f1_score
    total_users += 1

# Average the metrics
average_precision = round(total_precision / total_users, 2)
average_recall = round(total_recall / total_users, 2)
average_mrr = round(total_mrr / total_users, 2)
average_f1 = round(total_f1 / total_users, 2)

# Print the overall metrics
print("Overall Precision:", average_precision)
print("Overall Recall:", average_recall)
print("Overall MRR:", average_mrr)
print("Overall F1 Score:", average_f1)
```

Here are the results based on 600 users from the movielens dataset. We can see that these numbers

Precision (0.7): This suggests that 70% of the recommendations made by the model are relevant to the users. It's a strong score, indicating that the model does a good job at filtering out irrelevant items.
Recall (0.7): Also at 70%, this score indicates that the model is able to identify 70% of all relevant items that should have been recommended. This is a good balance with precision, suggesting that the model isn't too conservative in its recommendations.
MRR (0.8): A Mean Reciprocal Rank of 0.8 is excellent. It implies that the top recommended items are highly relevant, often hitting the mark with the first or second recommendations given.
F1 Score (0.67): This is the harmonic mean of precision and recall and is slightly lower than each individual metric but still quite good. An F1 Score in this range confirms that the model has a balanced performance between precision and recall, without overly favouring one over the other.



References

[1]MovieLens 20M Dataset

[2]Build a Recommendation Engine With Collaborative Filtering – Real Python

[3] Recommending Movies to Users using Cosine Similarity | by Kwalanj | Web Mining [IS688, Spring 2021] | Medium

[4] Recommender System : User based Collaborative filtering | by Deepa Pandit | Medium

[5] Recommender System : User based Collaborative filtering | by Deepa Pandit | Medium

[6]Hybrid Recommender Systems: Survey and Experiments -
Burke, R. (2002). Hybrid recommender systems: Survey and experiments. User Modeling and User-Adapted Interaction, 12(4), 331-370.
Hybrid Recommender Systems: Survey and Experiments | User Modeling and User-Adapted Interaction