Query Recommendation System

Ghaith Kouki [236847]
ghaith.kouki@studenti.unitn.it
University of trento

Jihed Dachraoui [236842]
jihed.dachraoui@studenti.unitn.it
University of trento

1 INTRODUCTION

Over the past few years, and due to the development of communication technology, a massive amount of information has been widely available to users at any time, which has led to information overload. The question of how to help users find information that satisfies their needs is an important issue, and the recommender system is considered a significant way to solve this problem. Query recommendation systems can be very effective in improving the user experience, as such systems can help users find relevant information both quickly and efficiently by suggesting possible queries as they type. These systems are commonly used in search engines, online libraries, and other information-rich websites to help users find the information they need easily.

Besides improving the user experience, query recommendation systems can also be beneficial to the organization or website that hosts the system. By helping users better reach the information they need, these systems can reduce the workload of customer service and support teams, improving the overall efficiency of the search process.

In this report, we introduce a hybrid collaborative filtering algorithm that combines use of k-nearest neighbors (KNN), user-based and query-based methods to deliver personalized and accurate query recommendations. The solution will be further discussed in the Solution section.

However, implementing such query recommendation system presents a number of challenges. One of the challenging parts of this work was to balance the strengths of both user-based and query-based collaborative filtering methods. User-based collaborative filtering is known to provide more accurate recommendations, but it can also suffer from the problem of data sparsity, as users may not have enough interactions with other users to compute similarities. On the other hand, query-based collaborative filtering can overcome the problem of data sparsity, but it may not provide as accurate recommendations as user-based methods. Another challenge was ensuring that the system is accurate and reliable, as users can become frustrated if the suggested queries do not match their requirements.

To evaluate its performance, we tested our algorithm on two different utility matrices, one randomly generated and one pseudo-randomly generated, with varying hyperparameters. The results of our experiments which we will discuss in the experimental section, show that this hybrid approach provides effective query recommendations.

2 PROBLEM STATEMENT

The main problem that a query recommendation system aims to solve is the difficulty that users often face in finding relevant information quickly and efficiently. With the vast amount of information available online, it can be challenging for users to identify the most relevant sources, particularly when searching for complex or specialized information. This can lead to frustration and a poor user experience, as users may have to sift through numerous irrelevant search results before finding what they are looking for.

We suppose that we have a set of users and, for each user, a set of queries that they have sent to the database. When a user asks a question to the database, he receives a set of tuples in answer. If he is satisfied with this answer, he gives them a high score, otherwise he gives them a low score. So given these inputs in a csv format:

(1) A relational table representing our database where each column is an attribute
(2) A set of users that is nothing more than a set of ids.
(3) A set of queries where each query has a unique query id and its definition. The definition is a set of conditions on the dataset attributes.
(4) A User-Query utility matrix that contains the scores from 1 to 100 assigned by users to queries. Each row of the matrix represents a user, and each column represents a query. The entries in the matrix are the scores that users have given to queries. The matrix has missing values knowing that not every user tried every query and rated it.

Thus, given this utility matrix, our task is to develop a method to fill in the missing values in the matrix. The completed matrix should accurately reflect users' preferences and be able to predict their ratings for queries they have not yet rated so that we can recommend the top-k queries that might be of interest to the user u. The method should consider the utility of a new query for all users, ensure personalization, balance accuracy and efficiency, and be scalable and have low complexity while delivering high-quality recommendations.

In summary, the algorithm presented in this work is a method for filling in the missing values in the utility matrix and thus recommending to the user a set of queries whose answer set contains data that might be of interest to the user. The query recommended to the user has not been requested by him in the database before, and it is believed that if he requests it, he will give it a high score.

3 RELATED WORK

3.1 Collaborative filtering

Collaborative Filtering is a technique that is commonly used to make recommendations by exploiting the relationships between users and items[1]. It is based on the idea that users who have similar preferences in the past are likely to have similar preferences in the future. There are two main types of collaborative filtering: user-based and item-based. User-based collaborative filtering uses the similarities between users to make recommendations, while item-based collaborative filtering uses the similarities between items to make recommendations.

3.2 K-nearest neighbors

K-nearest neighbors (KNN) is a popular algorithm used in collaborative filtering to find the k most similar users or items to a given user or item[2]. The k nearest neighbors are then used to make recommendations by aggregating their ratings or results. KNN has been found to be effective in both user-based and item-based collaborative filtering.

3.3 Cosine similarity

Cosine similarity is a measure of similarity between two non-zero vectors of an inner product space. It is commonly used to measure the similarity between users or items in a collaborative filtering system[5]. By using cosine similarity, we are able to measure the similarity between users and queries based on their interactions, resulting in a similarity matrix that serves as the foundation of our recommendation system.

$$\text{cosine similarity}(A, B) = \frac{\sum_{i=1}^{n} A_i \cdot B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \cdot \sqrt{\sum_{i=1}^{n} B_i^2}}$$

where n is the length of the vectors and A_i and B_i are the i^{th} components of vectors A and B respectively.

3.4 Pearson correlation

Pearson correlation coefficient is a widely used similarity measure for collaborative filtering and recommendation systems[4]. It is used to quantify the linear association between two variables. In the context of recommendation systems and collaborative filtering, it is commonly employed to predict missing values in a utility matrix. However, it has some limitations as it assumes a linear relationship between the variables and a normal distribution of data. Despite this, it is still a popular choice in practice due to its ease of calculation and effectiveness in many real-world problems. Other similarity measures such as cosine similarity may be more suitable in certain situations.

$$\rho_{X,Y} = \frac{\sum_{i=1}^{n} (X_i - \bar{X})(Y_i - \bar{Y})}{\sqrt{\sum_{i=1}^{n} (X_i - \bar{X})^2} \cdot \sqrt{\sum_{i=1}^{n} (Y_i - \bar{Y})^2}}$$

where $\bar{X}$ and $\bar{Y}$ are the sample means of X and Y, respectively, and X_i and Y_i are the i^{th} observations of X and Y, respectively.

3.5 Jaccard similarity

Jaccard similarity is a measure of similarity between two sets, commonly used in text and data analysis. It is defined as the size of the intersection divided by the size of the union of the two sets. It ranges from 0, representing complete dissimilarity, to 1, representing complete similarity. Jaccard similarity is useful for comparing the similarity between sets of items or objects.

$$\text{Jaccard similarity}(A, B) = \frac{|A \cap B|}{|A \cup B|}$$

where A and B are the two sets being compared and $|A \cap B|$ represents the size of their intersection and $|A \cup B|$ represents the size of their union.

4 SOLUTION

4.1 Hybrid Collaborative Filtering Algorithm for Query Recommendation

In our proposed solution for building a query recommendation system, we propose a hybrid collaborative filtering algorithm. The algorithm combines the use of k-nearest neighbors (KNN) cosine similarity and Pearson correlation, three well-established methods in collaborative filtering[3], to make recommendations based on both user-user similarity and query-query similarity.
We first compute the cosine similarity between users and Pearson correlation between queries using the standardized utility matrix, which contains the interactions data, as outlined in Algorithm 1 and Algorithm 2 respectively, in this hybrid method. By leveraging cosine similarity as well as Pearson correlation, we are able to measure the similarity between users and queries based on their interactions, resulting in similarity matrices that serve as the foundation of our recommendation system. Next, we use KNN to identify the most similar users and queries to a given user or query. This allows us to make recommendations that are tailored to the individual needs and preferences of each user. By selecting the k most similar users and queries, we are able to make recommendations that are both accurate and relevant to the user.
In addition, we incorporate the use of weights for both users and queries, which allows us to assign different levels of importance to users or queries similarities. This can help to improve the performance of the recommendations by taking into account factors such as the user's recent activity or the popularity of a query.

Algorithm 1 User Similarity

```
function USER_SIMILARITY(dataframe, k, center_matrix, standardize_matrix)
    gh ← dataframe.fillna(0)
    if standardize_matrix then
        gh ← gh.apply(standardize)
    end if
    if center_matrix then
        gh ← gh.apply(center)
    end if
    u_sim_matrix ← cosine_similarity(gh) return np.round(k_nearest_neighbor(u_sim_matrix, k), 4)
end function
```

As you can see in Algorithm 4 The system is designed to fill in missing values in a utility matrix that represents user-query interactions. The utility matrix is first standardized to create user similarity and query similarity matrices. The k-nearest neighbors algorithm is then applied to these matrices, using input parameters for the number of nearest neighbors to consider. A prediction is made for each missing value in the utility matrix by taking a weighted sum of the user similarity score and query similarity score, calculated

Algorithm 2 Query Similarity

function QUERY_SIMILARITY(*dataframe*, *k*, *center_matrix*, *standardize_matrix*)
  $gh \leftarrow dataframe.fillna(0)$
  if *standardize_matrix* **then**
    $gh \leftarrow gh.apply(standardize)$
  end if
  if *center_matrix* **then**
    $gh \leftarrow gh.apply(center)$
  end if
  $q\_sim\_matrix \leftarrow np.array(gh.apply(center).corr(method = 'pearson'))$
  return np.round($k\_nearest\_neighbor(q\_sim\_matrix, k), 4$)
end function

Algorithm 3 k_nearest_neighbor

Require: Utility matrix *arr* and number of nearest neighbors *k*
Ensure: Output matrix with k-nearest neighbor values
$n, m \leftarrow$ shape of *arr*
for *i* in range(n) **do**
  $indexes \leftarrow arr_i$ sorted indexes of top $m - k$ least similar users/queries
  $arr_{i,indexes} \leftarrow 0$
end for
return *arr*

Algorithm 4 Predict

Require: The utility matrix : *utility_matrix*, number of query neighbors : *q_sim_neighbors*, number of user neighbors : *u_sim_neighbors*, The user similarity weight : *u_sim_weight*, and the query similarity weight : *q_sim_weight*
Ensure: Output matrix with predicted values
$utility\_matrix\_copy \leftarrow utility\_matrix$
Fill all NA values of *utility_matrix_copy* with 0
$output \leftarrow utility\_matrix\_copy$
$columns \leftarrow$ columns of *utility_matrix*
Create similarity matrices for users and queries
Apply *k*-nearest neighbors to the similarity matrices using *u_sim_neighbors* and *q_sim_neighbors*
$scores\_to\_predict \leftarrow$ all the locations in *utility_matrix* where there is a missing value
for each pair of indices (i, j) in *scores_to_predict* **do**
  $user\_score \leftarrow utility\_matrix\_copy_j \cdot u\_sim\_matrix_i$ / (u_sim_matrix$_i$ · non-NA values of *utility_matrix* at column *j*)
  $query\_score \leftarrow utility\_matrix\_copy_i \cdot q\_sim\_matrix_j$ / (q_sim_matrix$_j$ · non-NA values of *utility_matrix* at row *i*)
  $prediction \leftarrow user\_score \times u\_sim\_weight + query\_score \times q\_sim\_weight$
  $output_{i,j} \leftarrow$ round($prediction, 2$)
end for
  return *output*

using the user similarity matrix, query similarity matrix, and the initial utility matrix. Computing the collaborative filtering at the user/query level can be summarized by the formula below:

$$\hat{r}_{ui} = \frac{\sum j \in N_u{}^k sim(u, j) \cdot r_{uj}}{\sum_{j \in N_u}^{k} |sim(u, j)|}$$

where:
$\hat{r}_{ui}$ is the predicted rating of user u for query i
N_u is the set of all users who have rated query i
k is the number of nearest neighbors to consider
$sim(u, j)$ is the similarity between user u and user j
r_{uj} is the actual rating given by user j for query i
This formula is similar to the basic collaborative filtering formula, but the number of neighbors considered is limited by the value of k. The final output is a completed utility matrix with predicted values for the missing entries. This approach can be used to make personalized recommendations for queries to users based on their past interactions.

Algorithm 5 Top-k Queries for a User

function TOP_K_QUERIES(*user_id*, *k*, *predictions*)
  $user\_predictions \leftarrow predictions_{user\_id}$
  $user\_predictions \leftarrow$ sort_values($user\_predictions, ascending = False$)
  return $user\_predictions[: k].index.to\_list()$
end function

the function "top_k_queries" shown above in Algorithm 5, is used to generate top k query recommendations for a specific user. The function takes in three inputs: the user id, the number of recommendations desired (k), and the predictions generated by the query recommendation system. The function first extracts the user id from the input user id string and checks if the id is valid by checking if it is within the range of the number of rows in the predictions data frame. If the id is invalid, the function raises a ValueError.

Next, the function selects the row corresponding to the user from the predictions dataframe, and sorts the predictions in descending order. The function then prints out the top k queries and their corresponding predicted scores for the specific user. The function can be useful in recommending the most relevant queries to a user, based on their past behavior and similarity to other users.

4.2 Query Utility Computation Method

In this section of the report, we propose a method to compute the utility of a new query for all users.
The algorithm calculates the scores of a new query for all users by comparing it to previous queries and their associated scores in a utility matrix. It uses Jaccard similarity to find exact matches or the KNN algorithm to find the most similar queries if no exact match is found. The scores for the new query are then computed based on the similarity values and the pre-scored queries.

The pseudo-code shown in Algorithm 6 defines three functions for a query recommendation system. The jaccard_similarity function calculates the Jaccard Similarity between two sets of queries. The KNN function computes the K nearest neighbors of a query

Algorithm 6 Query Utility Computation

```
function COMPUTE_QUERY_SCORE()query, utility_matrix, queries,usersID )
    queriesIDs ← queries['queryID'].values.tolist()
    queries ← queries['query'].values.tolist()
    query_list ← []
    for i ∈ queries do
        temp_query ← {}
        for x ∈ i.split(',') do
            aux ← x.split('=')
            temp_query.update(aux[0] : aux[1])
        end for
        query_list.append(temp_query)
    end for
    values_similarity ← []
    for i ∈ query_list do
        values_similarity.append(jaccard_similarity(query.values(), i.values()))
    end for
    values_similarity ← np.array(values_similarity)
    queriesIDs ← np.array(queriesIDs)
    if values_similarity.max() == 1 then
        return utility[queriesIDs[values_similarity.argmax()]]
    else
        values_similarity, queriesIDs ← KNN(25, values_similarity, queriesIDs)
        scores ← (np.array(utility_matrix[queriesIDs]) · values_similarity)/values_similarity.sum()
        return pd.DataFrame(scores, index = usersID, columns = ['Q2001'])
    end if
end function
```

based on their Jaccard Similarity and returns a sorted list of similarities and their corresponding query IDs.

Finally, the compute_query_score function takes a query, a utility matrix with user ratings for the queries, and lists of queries and user IDs as input. It splits the queries into dictionaries, calculates the Jaccard Similarity between the input query and all other queries, and finds the K nearest neighbors using the KNN function. If the highest similarity score is 1, indicating an exact match, it returns the corresponding rating score from the utility matrix. If not, it computes the query score by taking a weighted average of the K nearest neighbor's rating scores, weighted by their Jaccard Similarity scores. The computed scores are returned as a Pandas DataFrame with user IDs as the index and query ID as the column.

5 EXPERIMENTS

5.1 Data Generation

Dataset generation

To generate our dataset, we first imported the Faker library to randomly generate names, cities, and occupations. We also specified that we wanted the names and cities to be generated in Italian. Next, we used a loop to generate 200 of each of these items and wrote them to separate text files. These text files were then used to randomly assigns an age, name, occupation, and city to each row of the dataset, which were then written to a CSV file.

Query and user set generation

In this part,we will be discussing the query set that we generated for our query recommendation system. A query set is a collection of past queries, each with a unique query id and its definition, which is the set of conditions. The query set is stored in a CSV file, where each line contains the conditions in the format of attribute=value separated by a comma. The first element of each line is the query id. For example, a row in the query set would be: Q1,name=Sandra,city=Trani,age=57.

To generate the query set, we wrote a script that uses a while loop to create a 2000 queries. Within the loop, we first created a unique query id by concatenating the letter "Q" with the current iteration number. We then used an if-else statement and the random module to select a random name, city, age, and job to include as conditions for the query. If a condition was not selected, it would not be included in the final query. After all the conditions were selected, we added the query id and the final query to a list of queries. We then used the pandas library to convert the list to a dataframe and save it as a CSV file. The final output is a CSV file containing the queries set.

Additionally, we also generated a set of users by creating a list of user IDs in a for loop, and then saved it also to a CSV file. This set of user IDs will be used in conjunction with the utility matrix to evaluate the performance of the query recommendation algorithm.

Utility matrix generation

In this part, we will be discussing the utility matrix that we generated for our query recommendation system. We chose to create 2 types of utility matrix one totally random and the other is pseudo random. The random utility matrix have ratings assigned randomly to the queries for each user, while the pseudo-random utility matrix have ratings assigned based on some pre-determined pattern or logic, although not necessarily reflecting the actual preferences of the users. We divided the users into 12 different types, each with their own unique tendency in terms of rating distribution according to the output of each query.

Creating two types of utility matrices can be useful for experimentation and comparison purposes. For instance, we can compare the performance of our recommendation algorithm on the random utility matrix versus the performance on the pseudo-random utility matrix to see the impact of the underlying data structure on the recommendations generated by the algorithm. This can help to evaluate the effectiveness of the algorithm and also help to fine-tune the parameters of the algorithm for better performance.

5.2 Evaluation

For evaluating the performance of our algorithms, we used a process called "masking" or "holdout evaluation". This involves masking or hiding a portion of the ratings in the utility matrix, treating those as unknown and using the algorithm to predict the masked ratings. The predicted ratings can then be compared to the actual masked ratings to evaluate the performance of the algorithm. 20% of the ratings are removed from the matrix and treated as unknown.

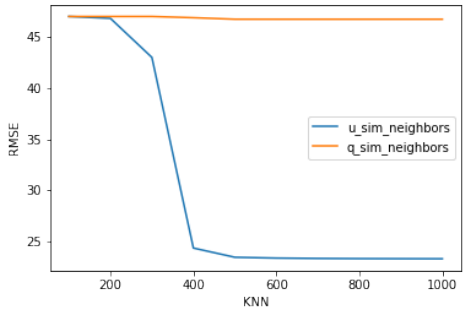

Figure 1: Variation of RMSE in the function of the k nearest neighbor in a randomly generated utility matrix

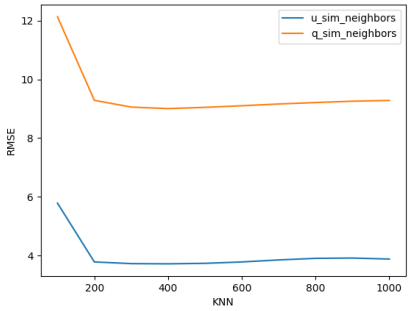

Figure 2: Variation of RMSE in the function of the k nearest neighbor in a pseudo-random generated utility matrix

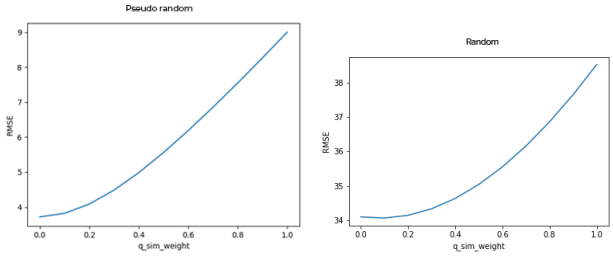

Figure 3: Variation of RMSE in the function of the user/query weight

Evaluation metrics

In evaluating the performance of our query recommendation system , we used the Root Mean Squared Error (RMSE) metric. RMSE is a commonly used evaluation metric in regression problems and measures the average difference between the predicted values and the actual values. In our case, we used RMSE to evaluate the prediction accuracy in the utility matrix. The utility matrix contains users' rating scores for the queries. The RMSE was calculated as the average difference between the predicted rating scores and the actual rating scores for a given user. A lower RMSE indicates a better fit of the model to the data and thus a more accurate prediction. This metric helped us to evaluate the quality of the recommendations generated by our model and to make necessary improvements for better performance. The formula for RMSE is given by:

$$\sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}$$

where N is the number of observations, y_i is the actual rating score and $\hat{y}_i$ is the predicted rating score.

The predicted top-K queries can be represented as a set, and the actual queries (that a user has rated highly) can also be represented as a set. The Jaccard similarity between these two sets can then be calculated and used to evaluate the performance of the recommendation algorithm.
A higher Jaccard similarity score indicates a better match between the predicted items and the actual items, and therefore a better performance of the recommendation algorithm.

Tests

The tests are about a grid search for the optimal hyperparameters of a recommendation system based on the user-query utility matrix.

The first generates the prediction for the hyperparameter q_sim_neighbors (which is the k nearest neighbor to a certain query).
The second generates the prediction for the hyperparameter u_sim_neighbors (which is the k nearest neighbor to a certain user).

In both tests, we calculate the root mean squared error (RMSE) between the predicted and true ratings, and the hyperparameters are searched over the range of 100 to 1000 with a step size of 100.

Figures 1 and 2 present the RMSE variation graphs for the K nearest neighbor for the first two tests conducted on both generated matrices. The results indicate that our algorithm performs better on the pseudo-random utility matrix as compared to the randomly generated utility matrix. This may be due to the structure or pattern present in the pseudo-random matrix that the algorithm can use to make more accurate recommendations. In contrast, the random utility matrix may lack such structure and not provide meaningful information for the algorithm to use. These results highlight the importance of the utility matrix when evaluating the performance of a recommendation algorithm and the crucial need for reel data. Both figures also reveal that the optimal results are achieved with u_sim_neighbors=600 and q_sim_neighbors=600 for the random matrix in Figure 1, and with u_sim_neighbors=400 and q_sim_neighbors=400 for the pseudo-random matrix in Figure 2.

The third test generates the prediction for each hyperparameter combination of q_sim_weight and u_sim_weight (which are the weights for user similarity and query similarity, respectively), calculates the root mean squared error (RMSE) between the predicted and true ratings. The hyperparameters are searched over the range of 0 to 1 with a step size of 0.1.
The results of the tests on both the random and pseudo-random

utility matrices, presented in Figure 3 show that the best performance of the algorithm was achieved when the user similarity weight (u_sim) was set to 1 and the query similarity weight (q_sim) was set to 0. This indicates that the algorithm's recommendations are largely driven by the similarity between users, rather than the similarity between queries.
These findings support the idea that user-based collaborative filtering, where the recommendations are based on the similarity between users, can be an effective approach for generating recommendations. They also highlight the importance of properly setting the similarity weights when using this type of algorithm, as the choice of weights can have a significant impact on the performance of the algorithm.

We also tested this algorithm that adds the mean score of a query or given by a certain user as a bias term to the predicted rating. The modified formula would be:

$$\hat{r}_{ui} = \mu_u + \frac{\sum\limits_{v \in N_i^k(u)} \text{sim}(u,v) \cdot (r_{vi} - \mu_v)}{\sum\limits_{v \in N_i^k(u)} \text{sim}(u,v)}$$

or

$$\hat{r}_{ui} = \mu_i + \frac{\sum\limits_{j \in N_u^k(i)} \text{sim}(i,j) \cdot (r_{uj} - \mu_j)}{\sum\limits_{j \in N_u^k(i)} \text{sim}(i,j)}$$

where:
μ_u is the mean score given by a certain user u
μ_i is the mean score of a certain query i

	RMSE	
Method	Random Scores	Pseudo Random Scores
CF_KNN_User Based	24.34	3.72
CF_KNN_Query Based	46.7	9
CF_Mean_KNN_User Based	34.8	5.10
CF_Mean_KNN_Query Based	38.5	13.45

Table 1: Performance Evaluation of The algorithm on the two generated matrices

As u can see in Table 1 the algorithm mentioned earlier didn't outperform the algorithm that we adopted first for both utility matrices except for the random one while using query-based collaborative filtering.
The computation of Jaccard similarity for the top-k predicted queries, shown in Figure 4, demonstrates that similarity increases as k increases. This result is expected because a higher value of k indicates a wider error tolerance margin in the algorithm, leading to a higher similarity between the predicted and actual queries.
The results of the Query Utility Method can be observed by incorporating a new query into the system with the details of for example name being "Franco", city being "Trento", and job being "Producer". The utility of this query was computed for each user, presenting a predicted rating score. The output of this computation example is displayed in Figure 5.
This method was evaluated by masking each query and predicting its score for each user. The mean root mean squared error

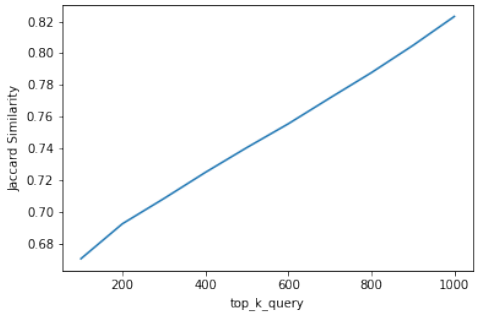

Figure 4: Variation of Jaccard similarity ratio in function of the k nearest neighbor in a pseudo-random generated utility matrix

U1	67.00
U2	88.23
U3	1.00
U4	78.00
U5	21.00
	...
U1996	86.95
U1997	86.91
U1998	87.49
U1999	13.16
U2000	63.21

Figure 5: the scores of the new query for all users

(RMSE) was then calculated between the predicted query scores and the actual scores recorded in the utility matrix filled using the CF_KNN_User Based algorithm which performed the best on both utility matrices. The result of this evaluation was an RMSE of 35.51 for the pseudo-random one and 44.3 for the other. The high RMSE value can be attributed to the fact that the utility matrix is heavily influenced by user preferences rather than the queries themselves.

6 CONCLUSION

In conclusion, this project aimed create to a query recommendation system based on the user-query utility matrix. We built a hybrid collaborative filtering algorithm combining the use of k-nearest neighbors (KNN) cosine similarity and Pearson correlation for computing inter-user/ inter-queries simililarities and we put it under test. Three tests were conducted to search for the optimal hyperparameters of the algorithm, including the k nearest neighbor for both query and user similarity, and the weights for user and query similarity. The results of the tests indicated that the algorithm performed better on the pseudo-random utility matrix than on the random matrix, demonstrating the importance of the utility matrix in evaluating the performance of a recommendation algorithm. The optimal results were achieved with a high number of nearest

neighbors and with user similarity weight set to 1 and query similarity weight set to 0, indicating that the recommendations were largely driven by the similarity between users. The computation of Jaccard similarity for the top-k predicted queries showed that similarity increases as k increases. Finally, the evaluation of the Query Utility Computation Method resulted in a high mean root mean square error, which can be attributed to the influence of user preferences on the utility matrix. These findings support the idea that user-based collaborative filtering can be an effective approach for generating recommendations and highlight the importance of the structure and pattern in the data used to evaluate the performance of a recommendation algorithm.

REFERENCES

[1] Collaborative Filtering Recommender Systems April 2013Research Journal of Applied Sciences, Engineering and Technology 5(16):4168-4182 DOI: 10.19026/rjaset.5.4644.
[2] B. Li, S. Wan, H. Xia and F. Qian, "The Research for Recommendation System Based on Improved KNN Algorithm," 2020 IEEE International Conference on Advances in Electrical Engineering and Computer Applications(AEECA), Dalian, China, 2020, pp. 796-798, doi: 10.1109/AEECA49918.2020.9213566.
[3] Jianrui Chen,Chunxia Zhao,Uliji,Lifang Chen. Collaborative filtering recommendation algorithmbased on user correlation and evolutionary clustering[J]. Complex Intelligent Systems,2020,6
[4] L. Sheugh and S. H. Alizadeh, "A note on pearson correlation coefficient as a metric of similarity in recommender system," 2015 AI Robotics (IRANOPEN), Qazvin, Iran, 2015, pp. 1-6, doi: 10.1109/RIOS.2015.7270736.
[5] H. Khatter, N. Goel, N. Gupta and M. Gulati, "Movie Recommendation System using Cosine Similarity with Sentiment Analysis," 2021 Third International Conference on Inventive Research in Computing Applications (ICIRCA), Coimbatore, India, 2021, pp. 597-603, doi: 10.1109/ICIRCA51532.2021.9544794.