Query Recommendation System

Seyed Mohammad Mousavi
seyed.mousavi@studenti.unitn.it
University of Trento
Trento, Italy

Omar Facchini
omar.facchini@studenti.unitn.it
University of Trento
Trento, Italy

KEYWORDS

data mining, recommendation system, query recommendation system, query score prediction

ACM Reference Format:
Seyed Mohammad Mousavi and Omar Facchini. 2023. Query Recommendation System. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

1.1 Introduction to the problem and its importance

Nowadays, there is a massive amount of data generated and used. This might overwhelm users that are interested in finding only certain specific information which is most likely a tiny percentage of the whole existing data. This is one of the reasons why recommendation systems have been developed. They are a subclass of information filtering systems that aim to predict the rating or the interest of a user on some data and are mostly used in commercial applications (e.g. Netflix). This is done in order to find and suggest data that is likely to be interesting for the user.

In this paper, we introduce a recommendation system that suggests a defined amount **K** of queries that if a certain user poses to a database, said user will be satisfied with the results received.

1.2 Highlights of the solution presented

This is achieved through collaborative filtering(CF) which is a technique based on collecting information from different sources that can be items or users based on the chosen approach. In this case both methods have been applied using the same metric of evaluation for the similarity, which in this case is a modified version of the Jaccard similarity to keep track of both the likes and dislikes of the user to act on the entire taste rather than only on the likes. This allows the system to have a more precise evaluation when comparing the users, hence will be able to retrieve much similar users and given the fact that the prediction is based on those users, that will be more accurate too.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1.3 Results obtained

This process allowed us to notice that the Item-Item CF is much more accurate than the user-user approach, which is expected as the items are normally straightforward compared to the users that each have their own taste. Another notable thing is that based on the growth of users compared to the queries we can see that once the users reach a certain point the user-user approach gets better compared to the counterpart while the more queries are present the better the item-item approach is.

1.4 Motivation for this work

As the modern amount of data is so massive, a single user is only able to view a tiny portion of it and we want to make sure that he will get something that he's interested in or he will most likely be interested in rather than showing him something he dislikes. Being able to do that accurately and consistently is very important as the user will be happy with the product we are offering and he will keep using it rather than trying different ones and this is a core concept for the majority of the companies. This also helps us understand the taste of different people, the trends of the moment or how different items impact different people.

1.5 Challenges

This is not an easy process, understanding and comparing different approaches to decide which would be the best fit for the problem is a tough process and the correctness of the choice is assured only after the evaluation. Finding a good similarity also is not as straightforward as some formulas account for some problems that might occur while other similarities consider and solve other issues, hence why there is not always a correct choice.

2 PROBLEM STATEMENT

The given input is the following:

(1) A set of users, represented only by an id, meaning no information about them is known (can be empty).
(2) A relational table that contains all the data in the form of a CSV file where the first row contains the parameters and all the other rows the actual data, important to note that no parameter can be **NULL**.
(3) A set of queries that have been posed in the past. Each query has a unique ID and its query description. The query description is a set of conditions.
(4) A utility matrix that contains all the possible relations between queries and users with a rating represented by a value from 1-100. This score indicates how satisfied the user was with the results obtained by posing a specific query. Since not every user posed all the queries, there are some empty points in the matrix.

The first task consists in filling the empty points in the utility matrix. Meaning that a prediction is required on the score of queries that a certain user has not requested and rated. **The output of this task** is the completed utility matrix. The top k queries that is predicted the user is going to give a high score are retrievable from this matrix. The second task consists in predicting the utility of a certain query generally for all users. **The output of this task** is simply the general score predicted for that query.

3 RELATED WORK

3.1 Content-based filtering

This type of recommendation system is based on suggesting items to customers that are similar to items the user has already highly rated. This is done creating an **item profile** for each item which is a set of features that describe the item.

3.1.1 *Pros.*

(1) Only need data for the single user as the system only accounts for the items that specific user rated rather than comparing different users.
(2) The system is able to identify users with unique taste as only the item profiles are taken into account hence there is no bias on the taste to recommend an item but only the similarity of the items.
(3) Can freely insert new items without a cold-start problem as there is no need to have ratings for that item but only the similarity with different items has to be calculated to start suggesting it.
(4) Easy understanding of the recommendation as the item profile of the suggestion gives all the information needed about the item.

3.1.2 *Cons.*

(1) Hard to insert new users as the recommendation is based on previous ratings of that user which in this case would not exist.
(2) Overspecialization as the system would most likely fixate on a single kind of items highly rated in the beginning while the user might develop different interests over time and these new interests would never be recommended.
(3) Selection of the features for the items profile is a core process but finding the correct ones to generate the profile is difficult, not only due to understanding what would best describe the item but also how many features would be required.

3.2 Collaborative Filtering

This type of recommendation system is based on suggesting items to customers using data from similar users, this works on the assumption that people who liked the same items will most likely like the same items in the future. This approach is called user-user collaborative filtering, a similar approach is the item-item collaborative filtering which consists in finding for each item the similar items and estimate the rating based on what the similar items have been rated.

3.2.1 *Pros.*

(1) Works for any item as there is no need for features.
(2) This system is able to let a user discover new interests as similar users might also like different kinds of items that the user did not rate, therefore recommending them.

3.2.2 *Cons.*

(1) Suffering from **cold-start** as inserting a new user or a new item gives no information about previous interactions as there is none hence making it impossible to calculate the similarity and consequently recommending it. A way to solve this problem requires the ability to receive feedback by the user by recommending items that are the most popular or highest rated to populate the rating allowing the system to calculate a similarity and start working as intended
(2) **Sparsity** as it is hard to find users that have rated the same items, let alone agreeing on the rating making it hard to find similar users
(3) Cannot identify unique tastes as it is based on interactions with different users or items. This system tends to suffer from a popularity bias.
(4) These systems tend to have high complexity as are based on the **Top k** nearest neighbors approach which consist in taking in consideration only the ratings of the K most similar users.

4 SOLUTION

After an initial stage of planning, in which existing algorithms that could solve the problem have been revised in order to better understand which would fit best and why, a possible solution to solve the task at hand was formulated. The solution presented in this section is based mainly on the following steps:

(1) **Dataset collection:** creation of users,queries and history of queries posed and rated by different users.
(2) **Proper load of the data in memory:** load of the data in memory while casting and storing different information in more useful data types such as dictionaries. During the load the starting average rating for each user is also calculated.
(3) **Acquisition of query likes and dislikes for each user:** creation of 4 dictionaries to have easily at hand for each user what queries he liked and disliked and for each query what users liked and disliked it.
(4) **Calculation of similarity for both user-user and item-item:** application of the Jaccard similarity to see how similar each user is to one another, same for the items.
(5) **Data pre-processing:** procedure of getting only the data we need to predict rather than the whole, finalized to ease the prediction computation. This step also saves some ratings and removes them from the original data to use as evaluation later on.
(6) **Collaborative filtering:** application of the approach described in section 3.2 on the clean data in order to predict the rating of unrated items for each user.
(7) **Filling the values:** insertion of the predicted ratings into the original data to fill the matrix.
(8) **Evaluation of the results:** procedure of comparing the data stored in **step 5** with the predicted ones to determine the accuracy of the system.

(9) **Evaluation of the utility of a query:** procedure of understanding how a query would impact all the users and giving a value.

4.1 Dataset collection

For clarity purposes the dataset was generated ad-hoc for this specific task, this was chosen as it would give a better understanding of the data and would allow the manipulation of the data based on what was required to be tested. This also eliminates the need to search, understand and verify usability of an already existing dataset on the internet.

4.1.1 What is in the dataset. The dataset consists of four CSV(comma separated values) files that contain all the data required by the system to work and are given as an input.

One file, named **userSet** contains only IDs of the users that have posed queries and require the prediction of the evaluation they will most likely give to queries they missed.

The **querySet** file contains all the queries that have been posed in the past by some users. Each line contains a query which is expressed as a set of conditional statements of syntax attribute=value except for the first parameter which is a single value representing the ID of the query in the line.

The **userData** file contains all the information about the users and acts as an actual database on which the queries in the **querySet** file have been run on. The first row of the file is the header and is made up of the parameters that represent our data which in this case are **ID, name, surname, height, age and city**. From the second row onward, each row represents a user and are filled with only values, there cannot be empty attributes.

The **utilityMatrix** file contains all the possible combinations of query-user and for all the combinations that already happened in the past the value is filled with the rating, from 1 to 100, the user gave to that specific query. The first row of the file consists of all the **query IDs** and from the second row onward the syntax consists of **user ID** followed by a rating for each query or an empty space if the user has not posed it yet. The blank spaces are combinations we will fill with expected rating in order to decide if the system should recommend that or not.

4.1.2 Different datasets. The need to use different datasets arises in order to compare how the system would perform based on the shape of the data. Three different sizes of datasets have been used, a **small one** which was used as a baseline, a **medium one** which was divided in two different sets where one contains twice the amount of users posing queries to the system compared to the small dataset and the other one has twice the queries. Same goes for the **large dataset** with the only difference being the doubling was made on the medium dataset rather than the small one. All the tests were run with a baseline dataset of 400 queries, 100 users posing queries and 200 users worth of data to run the queries on.

4.1.3 Generation of the data. The **userSet** was generated using a simple loop, this generates a contiguous list of numbers. The **querySet** was generated by giving a random percentage for each attribute to be added into the query with a random value sampled from possible values.To avoid having empty queries in that situation all the attributes get added, this also allows to cover more specific tastes for the users.

Algorithm 1 generate query set

```
while i < number of queries required do
    for all parameter ∈ userattributes do
        choice ← randomvalue0 − 100
        if choice ≤ probabilityoftheattribute then
            Query.append("parameter" = random.sample())
            emptyQuery ← False
    if emptyQuery = True then
        for all parameter ∈ userattributes do
            Query.append("parameter" = random.sample())
```

The **userData** file was generated by sampling a random value for each attribute the user would have, this to make sure there would be no **NULL** values.

The **utilityMatrix** file was generated by looping for the amount of users in the userSet on the querySet. Users were grouped in order to generate different biases, this would allow to have an **N** amount of users like a specific attribute while another group of **M** users would like the same attribute and dislike another one or vice versa with also the possibility to pose random queries outside their bias scope and rate them. This was done to simulate different tastes in users that could also overlap, while the randomness factor has been added to simulate the curiosity of human nature.

Algorithm 2 generate utilityMatrix

```
for all user ∈ userSet do
    for all query ∈ querySet do
        poseQuery ← random.choice(True, False)
        if userID is in chosen range of values then
            if poseQuery=true ∧ query has biased attribute then
                utilityMatrix[user][query] ← rating
            else
                utilityMatrix[user][query] ← emptyRating
```

4.2 Data loadout

The data will be loaded from each CSV file in the following format.

(1) **utilityMatrix:** it is a dictionary with the user IDs as a key and a dictionary containing ratings as a value. The dictionary containing ratings has the query IDs as a key and the rating to that query as the value.
(2) **querySet:** it is a dictionary with the query IDs as a key and the a dictionary containing conditions as a value. The condition dictionary has the attribute as a key and the value of that attribute as a value.
(3) **userIDs:** a list of all user IDs.
(4) **queryIDs:** a list of all query IDs.

4.3 Data pre-processing

The system is based on a collaborative filtering approach using a weighted variation of the Jaccard similarity. This requires some pre-processing in order to ease the computation at run-time, this is done by preparing 4 dictionaries to be able to instantly get the likes and dislikes of the users and the queries. The similarity between user-user and item-item is pre-calculated allowing to know the value whenever needed instead of having to compute for every item. Another information that is being preprocessed concerns what queries require rating as this is only a portion of the whole data, being able to go once through a simple dictionary containing all the information needed eases the task. The dictionary has as keys the IDs of the users that have some queries to rate and as values for each user all the query IDs they have not rated yet and require prediction. This is done by looping once on the utilityMatrix and checking if the value is blank, if so we insert the query for the user.

Algorithm 3 getQueriesToPredict

```
queriesToPredict ← {}
for all userID, ratings ∈ utilityMatrix do
    for all queryID, rating ∈ ratings do
        if query was not rated by the user then
            queriesToPredict[user] ← queryID
```

4.3.1 Likes and dislikes dictionaries. The system makes use of 4 different dictionaries to keep track of likes and dislikes: **userQueryLikedDict** has as keys the IDs of the users in the userSet and for each user the value is a list of the ids of the queries he rated above his average rating value. **userQueryDislikedDict** is the counterpart to the dictionary just described as they have the same structure but here the values stored are the IDs of the queries the user rated below his average rating value.

These two dictionaries are used to understand what query each user has liked and are used in the **user-user collaborative filtering** approach.

queryUserLikedDict has as keys the IDs of the queries in the querySet and for each query the value is a list of the ids of the users that rated the query above their average rating value. **queryUserDislikedDict** is the counterpart to the dictionary just described as they have the same structure but here the values stored are the IDs of the user that rated the query below their average rating value.

These two dictionaries are used to understand what query each user has liked and are used in the **item-item collaborative filtering** approach.

4.3.2 Jaccard similarity. To better compare the users, a simple Jaccard similarity would have sufficed to compare only the queries that the users liked as metric. A better idea was to take in consideration also the disliked queries as disliking the same items, similarity-wise, is equivalent to liking the same items hence why the formula used to calculate the similarity between two users is given by:

$$\text{numerator1} = \text{liked(id1)} \cap \text{liked(id2)} \tag{1}$$

Algorithm 4 GenerateLikeDislikeDictionaries

```
UserLikes ← {keys ∈ userSet}
userDislike ← {keys ∈ userSet}
queryLike ← {keys ∈ querySet}
queryDislike ← {keys ∈ querySet}
for all userID, ratings ∈ utilityMatrix do
    for all queryID, rating ∈ ratings do
        if query was rated by the user then
            if rating > average of the user then
                userLikes[userID] ← queryID
                queryLike[queryID] ← userID
            else
                userDislike[userID] ← queryID
                queryDislike[queryID] ← userID
```

$$\text{numerator2} = \text{disliked(id1)} \cap \text{disliked(id2)} \tag{2}$$

$$\text{denominator1} = \text{liked(id1)} \cup \text{liked(id2)} \tag{3}$$

$$\text{denominator2} = \text{disliked(id1)} \cup \text{disliked(id2)} \tag{4}$$

$$\text{similarity} = \frac{numerator1 \cup numerator2}{denominator1 \cup denominator2} \tag{5}$$

Which in words would be explained as the intersection of likes united to the intersection of dislikes, all over the union of all the likes and dislikes of both users. Similarly to the section 4.3.1, the system makes use of 2 different dictionaries to store the similarities: **userSimilarity** makes use of the two dictionaries stated in section 4.3.1 that have the user IDs as key, these are used in order to calculate the user-user similarity, which explains how similar a user is to all the other users. **querySimilarity** is the counterpart of the userSimilarity but for the queries, therefore it calculates the item-item similarity, which explains how similar an item is to all the other items.

These values will be used as a weight during the prediction process as the higher it is, the more identical the two objects or users being compared are therefore the rating given should be treated differently based on that.

4.3.3 Data setup for evaluation. The system is not interactive, therefore the only possible approach to evaluate the correctness of the results achieved is to keep track of some values that are pre-existing and are confirmed to be correct. These values have to be removed from the data that the system will be working on, this will allow the system to predict the rating of those values too. This is simply done by looping on the utilityMatrix and randomly removing some existing values. Once the system is done we can compare the results achieved with the original values deleted before-hand.

4.4 Collaborative filtering application

There are two main approaches for the collaborative filtering, user-user and item-item. Ideally only one would be used for a system but for research purposes both have been implemented in order to compare performances.

Algorithm 5 calculateVariantOfJaccardSimilarity(likedQueries, dislikedQueries, IDs)

```
similarities ← {}
for all id1 ∈ IDs do
    similarities[id1] ← {}
    for all id2 ∈ IDs do
        likedDict1 ← set(likedQueries[id1])
        likedDict2 ← set(likedQueries[id2])
        dislikeDict1 ← set(dislikedQueries[id1])
        dislikeDict2 ← set(dislikedQueries[id2])
        likedIntersection ← likedDict1 ∩ likedDict2
        dislikedIntersection ← dislikedDict1 ∩ dislikedDict2
        likedUnion ← likedDict1 ∪ likedDict2
        dislikedUnion ← dislikedDict1 ∪ dislikedDict2
        unionOfIntersection ← likedIntersection ∩ dislikedIntersection
        unionOfUnion ← likedUnion ∪ dislikedUnion
        if len(unionOfIntersection)/len(unionOfUnion) >= 0 then
            similarities[id1][id2] ← len(unionOfIntersection)/len(unionOfUnion)
        else
            similarities[id1][id2] ← 0
```

4.4.1 User-user collaborative filtering. This approach consists in comparing the user-user similarity in order to calculate the rating. Comparing a user with all the other users would be a waste of time, even considering the system has access to a list with the similarity values, and it would also be inadequate as some users completely differ from others hence making it pointless to take them into account. For these reasons, for each user, the system sorts the similarity dictionary in a descending order with the purpose of taking into consideration only the **top N** most similar users where the number is chosen beforehand when structuring the approach. The choice of N is fairly relevant as a higher value allows to have more accurate and precise predictions but generates the risk of taking into consideration bad similarities as there might be only few users that are very similar to the one being compared, hence worsening the prediction. A high value of N might also drastically increase the amount of runtime as in the worst case scenario can lead to comparing every other user. On the other hand a small value of N can lead to inaccuracies as not enough samples are being taken into consideration. Therefore the choice of the value N depends on the tradeoff between runtime, accuracy and dataset size.

The predicted rating is therefore calculated, only utilizing the most similar users that posed the query that is being predicted, using a weighted average where the weights are the similarities of the users with the one the prediction is being calculated for. The formula used is:

$$\text{prediction} = \frac{\sum \text{similarity} * \text{rating}}{\sum \text{rating}} \tag{6}$$

This allows more influence to ratings done by users based on how similar they are to the main user based on the idea that the more similar two users are the smaller the variance their ratings will have.

In a situation where no similar user is found, which occurs mostly on new users that have never posed any query, their ratings are filled with their own average rating value as it is not possible to make a prediction and the average rating is equivalent to a neutral rating. A better solution would require an interactive system that suggests queries to the user, either randomly or based on popularity, and retrieves a rating. This would be done on a different number of queries until the system would be able to find similar users and start running autonomously. Since the system at hand could not be interactive the average approach has been used.

4.4.2 Item-item collaborative filtering. The concept is almost identical to the one explained in section User-user collaborative filtering with the only difference being that the comparison is not done on the users but on the items instead.

Algorithm 6 collaborativeFiltering

```
predictedRatings ← {}
for all userID, queries ∈ queriesToPredict do
    for all queryID ∈ queries do
        similarQueries ← ordered queries bases on similarity
        for all similarQuery, similarity ∈ similarQueries do
            if similarQuery was rated by user then
                totalRating += rating * similarity
                totalSimilarity += similarity
                similarFound++
            if reached desired amount comparison then
                break
        if similarFound > 0 then
            prediction = totalRating/totalSimilarity
            predictedRating[userID][queryID] ← prediction
        else
            predictedRating[userID][queryID] ← averageRating[usedID]
```

4.5 Data finalization

Once the collaborative filtering is applied to the data, the utilityMatrix gets filled and a prediction is now existing for each possible user-query interaction. From the utilityMatrix is now also possible to retrieve the top-K queries that may be of interest to the user. This can be done on runtime as the queries that need to be predicted are known, allowing to know what are the top-k queries the user might like that he has not posed yet. This is done by going once through the dictionary of queries that need to be predicted for each user, for each query the utilityMatrix gets accessed and the predicted rating is retrieved and stored in a sorted list of size K which then gets returned to the user indicating the queries. In the situation where the user has a smaller number of interactions than K to be predicted the system will return only that amount of queries as are the only ones he has not posed yet.

4.6 Evaluation of the utility of a query

For calculating a general utility of a query, first, it is measured how interesting each condition was for the users and based on that a utility will be assigned to each condition and then, the general

utility of the query itself will be calculated by averaging the utility of all of its conditions. Calculating the utility of a single condition is done based on the **similar queries** to that single condition in the query set. Since before calculating the general utility of a query, the utilityMatrix is filled, it is guaranteed that for each query in the query set, there are ratings by all users available in the utilityMatrix. A similar query to a single condition is all of the queries in the query set containing the exact same single condition. The $SimilarQuery_{condition}$ is a set of all of the similar queries to $condition$.

$$SimilarQuery_{condition} = \{query \mid condition \in query\} \quad (7)$$

The utility of each condition, is a weighted average of the average ratings that users gave to the similar queries.

$$weight_{query} = \frac{min\{|ValueCount(condition)| \mid condition \in query\}}{|query|} \quad (8)$$

Where:

$ValueCount(condition)$ = count of the value of the condition in dataset.

The minimum count of the values in the conditions of a query will determine the maximum possible number of rows that can be returned by that query. It will show how close the result of running the similar query is compared to the result of the single condition. This measurement is optimistic and the number of returned rows might get much lower as the number of conditions increases. For this reason, the weight will be divided by the number of conditions to make the weight smaller in the similar queries with a higher number of conditions. If there is a similar query containing only one condition, since by definition a similar query should contain the single condition that the utility of it is wanted, the weight of this similar query will be larger than all of the other similar queries. The reason behind this is that the numerator of the weight for all of the other similar queries at maximum would be the same, but the denominator would be more than one, resulting in a smaller weight. When a similar query is exactly the single condition itself, it is the best representative for measuring the utility of that condition, hence, the proposed method is giving it a higher weight. The utility of a condition is measured by the weighted average of all of its similar queries.

$$Utility(condition) = \frac{\sum_{q \in SimilarQuery_{condition}} Avg(q) \times weight_q}{\sum_{q \in SimilarQuery_{condition}} weight_q} \quad (9)$$

Where:

$Avg(q)$ = average of all of the users' ratings to q.

Ultimately, the **general utility** of a query can be calculated by averaging the utility of all of its conditions.

$$GeneralUtility(query) = \frac{\sum_{condtion \in query} Utility(condition)}{|query|} \quad (10)$$

By running both the single condition and the similar query on the dataset and comparing the returned result, it is possible to more accurately understand how much of a good representative the similar query is for the single condition. However, it requires running each similar query on the dataset and the problem will escalate if the dataset is too large and cannot be kept in memory. The provided method of estimation requires reading the dataset only once to save the count of unique values and there is no need to access the dataset after that. If one of the single conditions in the query is not available in any of the queries in the query set, it will be ignored in the calculation of the general query. However if none of the conditions exist, the same solution can be provided simply by replacing the condition, which contains an attribute and a value, by only the attribute. In that case the asked question is not how interesting the condition was to the users, the question is how interesting the attribute of the condition was for the users.

5 EXPERIMENTAL EVALUATION

Two different types of tests were ran in order to compare the approaches used. In one the number of users posing the queries was augmented while, in the other, the number of queries posed was augmented. The augmentation was a simple doubling of the previous value as explained in section 4.1.2 that generates 3 different types of datasets. The tests were ran using both user-user and item-item collaborative filtering on all the datasets allowing to compare the scalability of the approach based on different factors. The baseline used to overview the performance of the system with different approaches and setups is the small database described in section 4.1.2.

5.1 Evaluation measurement

The main function used to verify the performance of the system is the **root mean square error function(RMSE)**, which is one of the most commonly used metric to evaluate the predictions as it indicates how wrong the result is by calculating the euclidean distance. This also punishes based on how wrong the prediction was, hence allowing a margin of error to empathize with human nature while not allowing completely wrong results.

5.2 Results achieved

Given the fact that the RSME is an error function, the ideal result would be a function that lowers or remains the same as the value being augmented gets higher.

5.2.1 Number of users augmentation. These tests were ran on the user-augmented datasets. The expectation was to notice the error for the user-user approach starting to get lower compared to the item-item approach the higher the amount of users gets. What was also expected was for the user-user approach to improve with a higher amount of data to work on. As shown in figure 1, the first expectation was met at around 350 users where the user-user approach achieves a lower error than the item-item. On the other hand it's surprising to notice how the RMSE for the user-user lowers only marginally until it reaches a point, at around 200 users, where it stabilizes rather than lowering. This indicates that while it gets better in the long run, the approach does not scale well

and does not allow the system to very accurately predict ratings.

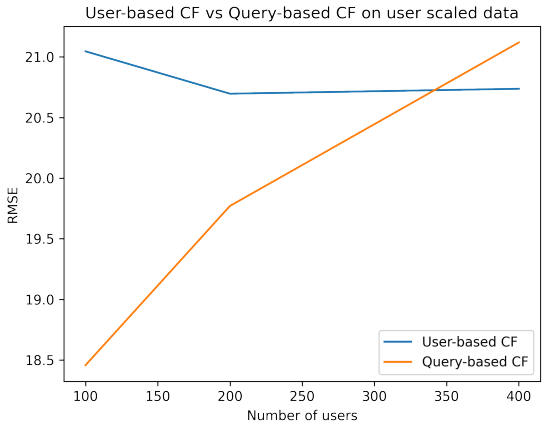

Figure 1: scalability with augmentation of the users

5.2.2 number of queries augmentation. These tests were ran on the query-augmented datasets. As opposed to the 5.2.1 section, the expectation was to notice the item-item taking over the higher the amount of queries would get. As shown in figure 2, the expectation was met by a large margin as from the baseline the item-item was producing a much lower error. The error keeps lowering extremely rapidly while augmenting the number of items up until around 800 queries where the error almost stabilizes with a small tendency to lower. This indicates that, while not exceedingly well, the approach scales while allowing fairly accurate predictions as in the worst case scenario, which is this case is the baseline as it contains the least amount of queries, the error is around 18.5% and the augmentation only lowers that value.

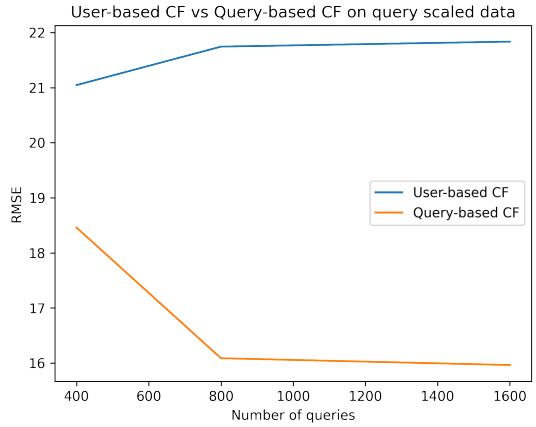

Figure 2: scalability with augmentation of the queries

6 CONCLUSION

To recap, in this report a recommendation system that given a utilityMatrix, consisting of rated user-query interactions, has to fill the missing values in said matrix allowing the users to retrieve the top-k most interesting queries for him. This was achieved through the application of collaborative filtering approach, for research purposes both user-user and item-item collaborative filtering have been tested. These approaches allowed the system to generate the predicted rating exploiting the similarity between each user or item, based on the method chosen. The similarity was calculated using a variant of the Jaccard similarity that makes use of the different ratings to take into account the taste for both liking and disliking to achieve better results. Results-wise it's notable that the item-item approach tends to scale better and achieve better results, which was expected as this method is known to perform better due to the nature of the items. The user-user approach turns out to be more stable but returning less accurate results compared to the counterpart until a certain amount of of users is reached.

REFERENCES

[1] Dardan Shabani Lule Ahmedi. [n. d.]. *Search Engine Query Recommendation Using SNA over Query Logs with User Profiles.* Ph. D. Dissertation. University of Prishtina, Prishtina.

[2] Qingwen Liu Yingcai Ma Wenwu Ou Junxiong Zhu Beidou Wang Ziyu Guan Deng Cai Yu Zhu, Yu Gong. 2019. *Query-based Interactive Recommendation by Meta-Path and Adapted Attention-GRU.* Ph. D. Dissertation.