Query Recommendation System

Elsa López Pérez
Eleonora Renz
elsa.lopezperez@studenti.unitn.it
eleonora.renz@studenti.unitn.it

1 INTRODUCTION

With the vast amount of data stored in many systems nowadays, it is important to have a way to retrieve relevant information in an easy, fast and accurate manner. Relational database systems are becoming increasingly popular for this purpose. These types of databases allow users to submit SQL queries and retrieve results. However, even though such database systems allow users to run complex queries over large data sets, the discovery of useful information is still a big challenge [5]. For example, users might not be familiar enough with a specific database or might not have the required SQL knowledge to run complex queries and retrieve the most relevant results. One way of tackling this problem is using recommender systems.

Recommender systems have been an important research area since the publication of the first paper on collaborative filtering in the 1990s [8]. The general idea behind recommender systems is to help users find content, products, or services. Nowadays, recommender systems have a very important role in fields such as business, e-commerce and entertainment, to name a few. However, the problem with current recommender systems is the large data volume that they have to deal with. As John Naisbitt said in 1982, *we are drowning in information but starved for knowledge*. Generally speaking we can distinguish between two types of recommender systems: those based on collaborative filtering and those that are content based [7]. The former mimics user-to-user recommendations and predicts users preferences as a linear, weighted combination of other user preferences. The latter, on the other hand, makes recommendations based on user preferences for product features. Both methods have limitations, and this is why some hybrid methods have been developed, in which both are combined to tackle some of the problems they have.

In this paper we aim to develop a recommender system based on the k-means clustering technique to deal with the scalability problem by dividing it into sub-problems. In addition, in order to avoid computing the similarities among items and/or users, we use a voting scheme in each of the clusters. This way, the system will recommend queries to a user in a specific cluster by using solely the rating statistics of the users in that particular cluster. Not only will we reduce the cost of similarities, but we will also reduce the dataset by focusing on a specific cluster.

1.1 Experimental results

First results were achieved using a small database, proving the feasibility of the proposed approach. In a next step a larger dataset was created to test our methods in a more rigorous manner. We have been capable of filling out the utility matrix in a logic and coherent way, and provide a metric to evaluate the importance of a given random query.

Three experiments were conducted to test the quality of the query recommendation system. First, 30% of values in the original utility matrix were removed. Second, 70% of ratings for a specific user were removed. And, third, 70% of ratings for a specific query were removed. These values were removed from the original dataset to be used as a ground truth to compare the results of our solution against. These experiments will be described more in depth in section 6. The results for all three test cases are satisfactory and prove the overall quality of the algorithm to be adequate. The conclusions that can be drawn from these experiments are that the proposed approach works well both in the case of a new user appearing in the list, as well as a completely new query being asked. In the former scenario a new user will still be recommended relevant queries and for the latter we can conclude that a new query will be considered for all users, once posed.

One caveat of the experimental evaluation, however, is that the data used for the entire project is randomly generated. This should be kept in mind when discussing the results, for example, when clustering the queries as representative results might not be achieved. More detailed information will be found in the following sections.

1.2 Motivation and challenges

The importance of developing a query recommendation system for a database stems from the fact that they differ from well established web search engines and their recommender systems because querying a database returns data, not documents [4]. Additionally, the interaction types with the systems differ.

The challenge in this query recommendation system lies in the indirect rating of results. In our scenario users do not rate single results presented to them, but rather their overall satisfaction of all results returned by a specific query. For this reason, we need to deduce the satisfaction of the returned results using the rating of the queries. This corresponds to the behavior of users in real life, as people do not want to spend time rating every result presented to them and expect only the results that suit them best to be returned. Online information retrieval systems such as Google and Amazon solve this challenge by looking at the click-through-rate of individual users as an alternative feedback mechanism. However, for a database like the one used in this scenario, this is not a possible solution. Results are not directly interacted with within the system, but rather just looked at by the user once they are retrieved. Another challenge is the scalability of a solution based on collaborative filtering. With a growing number of users using the system, there are exponentially more comparisons that need to be made between users to be able to match similar users with each other.

2 PROBLEM STATEMENT

Recommender systems are extensively used to provide personalized suggestions to users regarding information, products, or services. One of the most popular recommendation techniques is **collaborative filtering**. This method is based on the assumption that similar users have similar preferences. Hence, by finding users that are similar to the active user and by examining their preferences, the recommender system is able to predict the active user's preferences for certain items and provide a ranked list of the items that the user will most probably like. The preferences database is a user-item matrix, $R = [r_{i,j}]$, where $r_{i,j}$ represents user $i's$ ranking of element j. There are, however, some downsides regarding this method, with the most important ones being **data sparsity** and **scalability**. The first one makes reference to the number of missing values we have in the matrix R, due to lack of ranking for several items [3]. The second one arises from the fact that with growth of the number of users and the number of items, the computational cost grows exponentially. In order to address these problems, we have decided to use k-means as a cluster technique to cluster the queries. Once queries are divided into clusters, we can compute a similarity measure between queries in a same cluster, hence reducing the scalability problem. We will choose the ranking as the average of the k closest queries. The advantage of this technique is that the recommendation system is applied to each cluster, reducing the scalability and sparsity at the same time. The assignment of queries to clusters will be done by means of a voting scheme, since clusters are created regarding the rows of our database, which are the outcomes of the queries. Since queries might have more than one outcome, in order to divide queries in clusters we have to look at the cluster in which the query has the largest amount of tuples in.

As for the ad hoc methods and their challenges to query recommendation systems, combining the collaborative filtering paradigm with the database context is more challenging than it might seem. On the one hand, the same results of the database can be retrieved by using different conditions, which complicates the evaluation of similarity among users. In the web paradigm, similarity between users can be expressed as the similarity between the products they visit, rate or purchase, but this is no longer possible when working with queries. On the other hand, the recommendations to the users have to be in the form of SQL queries, since those describe what the retrieved data represent. This means that we need to first decompose the user queries into lower level components in order to compute similarities and make predictions, and then re-construct them back to meaningful SQL queries that we can recommend.

3 RELATED WORK

3.1 Clustering in recommender systems

Clustering algorithms belong to the so-called unsupervised classification algorithms, which aim to divide data in groups according to some natural patterns [2]. There are two types of clustering algorithms: hierarchical and partitional. We can find both type of algorithms in the literature regarding recommender systems. Regarding the former, in the paper mentioned in [1], they propose a recommender system based in Agglomerative Hierarchical Clustering for collaborative filtering. They used similarity metrics such as Pearson correlation, Jaccard Mean Square Difference or Euclidean distance. As for partition clustering recommender systems, the authors in [10] propose methods to address the problem of choosing the value of k and the initial centroid points. They use singular value decomposition to solve the k-means initialization problem.

3.2 Voting algorithms

Voting schemes have been used for ages in political science and economics to select compromise choices from several conflicting alternatives. Thus, using voting theory in recommender systems is logical because it holds the promise of recommending items that have the desirable properties required to satisfy user preferences. Summing up, the idea of voting algorithms is to select a choice among different conflicting possibilities based on the popularity of these. The advantage of using a voting scheme in recommendation systems is that we can avoid the costly computation of similarities among items or users. In [6] they resent a new web-based movie recommender system that combines voting based ranking procedure with guaranteed properties that use syntactic features.

4 SOLUTION

4.1 Part A: Utility matrix

Even though we are aiming to develop a recommender system, we wanted to avoid using the classical collaborative filtering approach because of its downsides and lack of originality. Instead, we propose a method based on clustering and voting to both fill in the utility matrix and recommend queries to users.

The first step to develop our solution is to cluster the database, this is, the rows of our artificially generated matrix. To be able to cluster the tuples of the database we need to represent them as vectors in a Euclidean space. For simplicity, we will be using numerical values, so that the vector is simply an array in which each position is the value of the corresponding feature. As we have mentioned in previous sections, we will use the vector after performing PCA dimensionality reduction. To keep track of the clusters, we add a column to the database indicating, for each row, the cluster to which the corresponding tuple is assigned to. As for the used method, we will finally use k-means.

Once the database has been clustered, we will assign each query to a cluster based on the outcomes of that query. This is, for each query, we retrieve the rows of the database that it refers to, we count the number of outcomes in each cluster, and we assign the query to the cluster with a higher number of outcomes. Here is where our voting scheme comes into place. The assignation is simply based on a counting technique. The cluster label will be added to an additional column in the queries database.

Once we have the queries divided in clusters, we define a similarity between queries in the same cluster, so that we can then fill the utility matrix and provide recommendations. In order to do so, we have to use a representation of the queries as vectors or sets. Contrary to what we have done with the rows of the database, we cannot represent the queries as the corresponding values of the database, since a query might have more than one tuple of values and these might not even be in the same cluster. Hence, what we have done is represent queries based on the conditions and used the condition repeated as many times as the numerical value. This is, if

we have a query *Q1: nrooms = 2, windows = 4*, then we will represent this query as *Q1: (nrooms, nrooms, windows, windows, windows)*. We will store the results in a JSON file.

Once we have the vector representation of queries, we can use the Jaccard similarity to find the top k queries which are most similar to a given one. Note that since the queries are not sets as we are considering repeated values, in the intersection we will also take into account the number of times a value appears and not only the value itself. This is, if *nrooms* appears twice in each query, then it will appear twice in the intersection. Since we are interested in ranking the queries that have not been ranked by a user, we proceed as follows: (i) for each user, we divide queries into ranked and not ranked. (ii) We assign the ranked queries to the corresponding cluster. (iii) We calculate the average ranking of all the ranked queries in each cluster. (v) For each of the non-ranked queries, we compute the Jaccard similarity of this query to the ranked queries by the user, and we store the results of the top k. (vi) The rank of the given query will be calculated as the weighted average of the top k queries, with the weights being the corresponding ranking of the query. (vii) If the Jaccard similarity of the query to all the ranked queries in the cluster is zero, we assign the average of all the queries in the given cluster, that have been previously calculated. (viii) We store the results, and we recommend the top k ranked queries that were not initially ranked by the user. We keep track of the ranking and index of the recommended queries. We store the results in descending order, being the *top 1* the highest ranked query, *top 2* the second highest, and so on. (ix) Finally, we have normalized the values of each row so that the minimum value for each user is 0 and the maximum value is 100. The reason behind this is that the highest ranking for a user may be 80, while for a different user it may be 100. This will be crucial when doing part B, since we are calculating the overall importance of a query for all users and not a particular one.

Algorithm 1 Pseudo code for the implemented solution

```
Use k-means to cluster the database

for every query in Queries do
    Represent the query as a vector
    Cluster the queries using a voting scheme
    Calculate similarity using Jaccard Similarity
end for
for Every user in Users do
    Separate not ranked and ranked queries
    for every not ranked query do
        Find k closest ranked queries in the cluster
        if similarity is 0 for all ranked queries then
            Rank = average ranking of queries in the cluster
        else
            Rank = average of the k most similar
        end if
    end for
end for
```

The complete code can be found under the name **CodePartA.ipynb**

4.2 Clustering algorithm

The goal of the clustering algorithm is to partition the queries in the dataset according to their output tuples. Hence, if we denote the set of queries by Q, we partition this set into p partitions $Q_1, Q_2, ..., Q_p$, where $Q_i \cap Q_j =$ for $i \neq j$, and $Q_1 \cup Q_2 \cup ... \cup Q_p = Q$. We first partition the database in clusters, using the reduced vector after performing PCA. Once we have the database divided in clusters, we can assign queries to clusters by retrieving the corresponding outputs of the query and calculating the number of outcomes of that query in each cluster. The one with the highest number will be the selected cluster for that query. We will try two different clustering algorithms: k-means and DBSCAN.

k-means: is a centroid-based or partition-based clustering algorithm. This algorithm partitions all the points in the sample space into K groups of similarity. The similarity is usually measured using Euclidean Distance. Initially, k centroids are randomly placed, one for each cluster. The distance of each point from each centroid is calculated, and each data point is assigned to its closest centroid, forming a cluster. After each assignment, the position of the k centroids is recalculated. Since we have to define the number of clusters beforehand, a common method is to use the k-elbow graph. This means that we compute the clustering algorithm for different values of k, and for each k we calculate the within-cluster sum of squares(WCSS). We then plot the curve of WCSS according to the number of clusters. The location of the bend in the plot is generally considered an indicator of the approximate number of clusters.

DBSCAN:. is a density-based clustering algorithm. The key fact of this algorithm is that the neighborhood of each point in a cluster which is within a given radius must have a minimum number of points. This algorithm has proved extremely efficient in detecting outliers and handling noise. DBSCAN algorithm requires two parameters:

- **eps**: it is used to define the neighborhood around a data point i.e., if the distance between two points is lower or equal to eps, then they are considered neighbors.
- **MinPts**: minimum number of neighbors within the eps radius. The larger the dataset is, the larger is the value of MinPts that we have to choose.

The algorithm starts with a random data point that has not already been visited and looks for the eps neighbors of this point. If the number of points is greater or equal than MinPts, then a cluster is started. Otherwise, the point is considered as noise. Note that the point might be considered as part of a cluster in a later step. If a point is found to be part of a cluster, all the eps neighbors are also part of the cluster, and are then added. The process is continued until all the neighbors are found, and we then restart it with a different unvisited point.

Unlike k-means, which is only suitable for compact and well separated clusters, DBSCAN is capable of detecting clusters of arbitrary shape and deal with noise in data. The problem in our case, and the reason why DBSCAN algorithm might not work, is that since we are working with synthetically generated data, we might not have different density areas and the algorithm may just

detect one cluster. Since for k-means we have to define the number of clusters, we can be sure that we have at least two different groups.

4.3 Voting scheme

Once the database has been partitioned into clusters, we are going to assign queries to each cluster based on a voting scheme. In general, a voting scheme works as follows: given a list of alternatives to choose from $A = \{a_1, a_2, ..., a_n\}$, a set of voters $V = \{v_1, v_2, ..., v_m\}$ and a preference function P that returns the rank ordering of the alternatives given a voter, a voting scheme will produce a ranking of the alternatives. Straffin [9] has listed several criterion to rate the desirability of the outcome. In our case, we will be using a simple voting scheme, in which the query is assigned to the cluster in which the majority of the tuples are. For example, say we have 6 clusters, and for a given query q_i we have the following outcomes regarding the location of the tuples associated to the query (Table 1). Then, we will assign query q_i to cluster 1. In case of a tie, we simply assign it to the cluster with higher number of elements, this is, tuples, in general. So, for query j in Table 1, if cluster 1 has 1000 elements and cluster 2 has 200 elements, we will assign query j to cluster 1.

Query ID	Cluster 1	Cluster 2	Cluster 3	Cluster 4	Cluster 5	Cluster 6
i	672	485	539	378	484	378
j	100	100	5	37	4	89

Table 1: Example of query assignation

4.4 Part B: importance of a random query

For this part of the project, the idea is the following: given a random query, that might not be in the database, provide a metric of the importance of that query for each user. We have chosen the following metric to define the importance:

$$\text{Importance} = mean(\frac{\text{Ranking} \times \text{Similarity} \times \%\ \text{occurences}/100}{100}) \quad (1)$$

where *Similarity* corresponds to the similarity value between the highest ranked query for user i and the most similar query to the random query in the database (note that if the query exists in the database, then the most similar is the query itself), *% occurrences* stands for the number of times the random query (or similar queries in case it is not present) appear in the database, and *Ranking* corresponds to the ranking of the highest ranked query by user i.

Note that if the query is in the database, then he similarity equals 1 and the importance is simply the product of the ranking calculated in part A (i.e. the value of the utility matrix) and the percentage of occurrences. Hence, if a query has ranked the random query with the top value (i.e. 100) and the query has been asked by all users (i.e. % occurrences is 100) then we have that the importance equals 1, being the maximum value of the *Importance*. On the other hand, if the query does not appear in the database, for sure the *Similarity* value will be less than one, ensuring that the overall importance is less than 1. In order to have an importance of 0, we should have a *Ranking* value of 0.

Note that since users might rank queries in different ranges, it is important to normalize the rankings of all users between 0 and 100 before computing the *Importance*

We have divided the problem in two cases, depending on whether the query has already been posed or not:

Case 1: the query already exists in the database: In this case, the *% of occurrences* refers to the number of users that asked the query divided by the total number of users. Note that we can have this information before filling the utility matrix, as we know what queries were actually asked by each user. As for the ranking, we can use the results of part A. Note that in this case the similarity will be 1, as the query is in the database, so it is similar to itself.

Case 2: the query is not in the database: In this case, we have done the following: (i) We gather all the queries with similar conditions and values to the original query. This is, if the random query is *Q1: (nrooms=2, windows=3, floors=1)*, we find all the queries that contain all the possible combinations of this given query. Hence, we will look for queries matching one of the 7 possibilities: (nrooms=2), (windows=3), (floors=1), (nrooms=2,windows=3), (nrooms=2, floor=1), (windows=3, floor=1), (nrooms=2, windows=3, floor=1). We will denote this set of queries as similar queries S, and will denote each of them by s. (iii) We calculate the percentage of occurrences of each similar query in the database. (iv) For each user, we select the top 5 highest ranked queries, both by the user itself and the predicted rank from part A. (v) For each of the top 5 queries, q, we calculate the Jaccard Similarity of q and the similar queries s that are in the cluster of q. (vi) For each of the top 5 queries q, we store the most similar query s in the similar queries S and the corresponding similarity value. (vii) We calculate the Jaccard Similarity between the original random query and the similar queries s. (viii) We define the overall similarity as the product between the similarity of the random query and the similar query s and the similarity of the user query q and the similar query s. (ix) Finally we can calculate the *importance* as the average of the product of the Ranking of query q, the overall similarity of query s and the occurrences of the most similar query s for each query q.

Note that in this case the *Importance* is always going to be less than 1, since the *Similarity* values are below 1 and they appear in the equation as multiplications. In the case that either the ranking is zero or the similarity is zero, the *Importance* will also be zero.

As an example, imagine we have the following random query that is not in the database: Q_r *= [distcity=7, nrooms=1]*. The corresponding query set will be *[nrooms, distcity, distcity, distcity, distcity, distcity, distcity, distcity]*. For a given user, we have that the top 5 ranked queries by the user and their corresponding rankings are: u_i *= [18, 481, 629, 647, 885]*, r_i *= [100.0, 98.55, 97.1, 95.65, 94.2]*, where these values have been normalized so that the maximum ranking is 100 and the minimum is 0. For each of the top 5 queries u_i, we calculate the most similar query in the set of similar queries S, and we obtain *s = [1258, 667, 1237, 441, 1352]*, where the first one, 1258, is the most similar to query 18, the second one, 667, is the most similar to query 481, and so on. The corresponding similarity values are *v = [0.67, 0.46, 0.9, 0.38, 0.43]*. In addition, we calculate the similarity of every query in s with the random query Q_r, and obtain *sq = [0.02, 0.02, 0.04, 0.57, 0.07]*. Finally, we have that the percentage of occurrences of the similar queries is as follows: *% occurrences = [20,13,7,9,13]* We therefore calculate the overall similarity as the product of v[i] with sq[i], and obtain *w = [0.0134, 0.0092, 0.036, 0,2166, 0.0301]*. Finally, we can define the importance using

equation 1 and averaging the results:

$$\text{Importance} = mean((([100.0, 98.55, 97.1, 95.65, 94.2] \times [0.01, 0.01, 0.04, 0.22, 0.03] \times [20, 13, 7, 9, 13]/100)/100) = 0.0572 \quad (2)$$

From this result, since the *Importance* value is between 0 and 1, we can state that for the given user, the query is not very important.

Algorithm 2 Psuedo code for part B

```
Qr = random query
if existing query then
    for every user in Users do
        Importance = Ranking
    end for
else
    S = [queries with matching conditions in the database]
    O = [% occurrences of query s in S]
    for every user in Users do
        top5 = [top 5 ranked queries by user]
        similarity = [0,0,0,0,0]
        similarityRandomQuery = [0,0,0,0,0]
        for i,query in top5 do
            mostSimilar = most similar query of S in cluster(query)
            mostSimilarOcurrence = % of mostSimilar
            similarity[i] = JaccardSimilarity(query, mostSimilar)
            similarityRandomQuery[i] = JaccardSimilarity(Qr, mostSimilar)
        end for
        Importance = mean((top5Ranking × similarity × similarityRandomQuery × mostSimilarOcurrence)/100)
    end for
end if
```

The complete code for part b can be found in the code folder under the name **CodePartB.ipynb**

5 DATABASE GENERATION

We have created an artificial database for this project. The data is based on house features, and the used attributes are: 'number of rooms', 'number of bedrooms', 'number of baths', 'square meters', 'garden square meters', 'number of floors','garage square meters', 'price', 'year of construction', 'number of windows', 'distance to the city', 'number of doors'. The values for each of the attributes will be a random integer in a given range, which is dependent on the attribute itself. We have also created a list of users with their unique user ID, and a list of queries. For the latter, we have selected random columns of the database and a random row, so that we have queries with different number of conditions, but we are sure that we have outcomes for the used queries.

As for the ranges in the numerical values, we have based the choices in the fact that for each query we have to construct a representation vector containing as many repetitions of a given attribute as the corresponding value. Hence, for example, for the year, instead of choosing a year between, say, 1990 and 2020, we have simply defined the range as the amount of years in between. If this were not the case, we should have used a mapping function to avoid repeating an attribute 2000 times. This mapping would have identified the first year, say 1990, with 1, the second year with 2, and so on.

6 EXPERIMENTAL EVALUATION

The goal of this project is to develop a recommender system based on the k-means clustering technique to deal with the scalability problem by dividing it into sub-problems. In addition, in order to avoid computing the similarities among items and/or users, we use a voting scheme in each of the clusters. This way, the system will recommend queries to a user in a specific cluster by using solely the rating statistics of the users in that particular cluster. Not only will we reduce the cost of similarities, but we will also reduce the dataset by focusing on a specific cluster.

We have run our model in different random data sets that we have generated, varying as well the number of tuples, queries and users. The main downside when trying to explain the results is that we are working with random data, and thus, we cannot expect our model to provide accurate recommendations.

On the one hand, once we have clustered the tuples of our data set, when we assign the queries to the clusters only 30 % of the clusters are used. Indeed, in general we have $k = 10$ for k means, and only 3 clusters are used to cluster the queries. For the last experiment that was done, we achieved 5 used clusters, which has been the maximum. As stated before, k-means might not be the most accurate clustering method, but other methods such as DBSCAN that are based on density regions are not valid for our randomly generated data either. This is because DBSCAN is capable of separating clusters of high density from clusters of low density. Hence, it does a great job of seeking areas in the data that have a high density of observations, versus areas of the data that are not very dense with observations. In the way that we have generated the random data, it is likely that we will only have one dense region, and therefore, DBSCAN will fail to find different clusters.

On the other hand, even though we have clustered queries as to reduce the scalability problem, computing the corresponding representation of each query and then the Jaccard similarity requires a large computation time.

As explained in the solution, we have ranked queries using the most similar queries in the cluster. If the similarity among all were zero, then the ranking was defined as the average of the rankings of all the queries in the cluster. Table 2 shows the average number of queries for each scenario, being three possible scenarios: (i) non-similar queries, (ii) at least three similar queries, (iii) less than three similar queries.

Case 1	Case 2	Case 3	Proportion 1	Proportion 2	Proportion 3
3.02	878.14	500.08	0.0027	0.6351	0.3625

Table 2: Average values of each type of scenario for the ranking calculation

Case 1 corresponds to non-similar queries and hence, the rank is the average of the ranking of all queries of the cluster. Case 2

corresponds to the case where there are at least k queries with a similarity value different than zero, and hence the ranking is the weighted average of their rankings. Finally, case 3 corresponds to having less than k non-zero similarity values. Thus, the ranking is the weighted average of the non-zero values. The last 3 columns are simply the proportional values, meaning that we divide the total amount between the number of not ranked queries. Note that we have calculated the proportion for each user and then the average of the proportions.

In order to test the model, we have performed three experiments. All three experiments are based on the same idea and approach that a certain percentage of values in the original utility matrix are removed. By doing this, we are essentially creating a test set based on our ground truth. From here, we can compare the results of the algorithm based on the experiment dataset with the original ones and provide a measurement of error. We are going to define the error as the absolute value of the difference, once the utility matrix has been normalized. The reason is that a 10 units difference has not the same impact in a user that ranges from 0 to 100 than in a user whose length interval of ranking is 40.

1. Remove 30% percent of ratings across the entire utility matrix.
2. Remove 70% of ratings for a specific user.
3. Remove 70% of ratings for a specific query.

The obtained results are summarized in Table 3

	Experiment 1	Experiment 2	Experiment 3
Error	30.73	16.51	14.19

Table 3: Average error for each of the experiments. Values between 0 and 100.

Taking into account that values are randomly generated, and that the error values are in the interval $[0, 100]$, we can say that the model is quite capable of providing good values for the ranking and recommendations. It seems logic that the error for the first experiment is higher than for the other two, since we are working with a similarity algorithm and hence, altering one user or one query is expected to have a lower impact on the overall performance than modifying various queries and users at the same time.

7 CONCLUSION

As we have seen in the state of the art, there are many different methods that can be used for recommender systems, and, among them, for query recommender systems. One of the most well known methods is collaborative filtering. However, the main downsides are the data sparsity and scalability problem. To tackle these problems, we have used a clustering technique, k-means, to cluster queries and compute the similarity only among queries of the same cluster. This helps to reduce the computation time. In addition, a voting scheme has been used to assign queries to clusters, since the clusters are created according to the tuples of our database, that are the outcomes of the queries. Note that one query may include several tuples, each of them in different clusters. Hence, the cluster with higher representation will be chosen for that particular query.

Once the queries have been assigned to clusters, we have to define a similarity measure between them. We have used the conditions of the queries repeated as many times as the numerical number. Since we have queries represented as sets, but accounting for repeated elements, we can use the Jaccard Similarity to select the top k most similar queries to a given one in a certain cluster. This way, we can fill in the values of the utility matrix by assigning the weighted average of the k most similar queries to a non-ranked query.

As for the second part, we have to provide a measurement of the importance of a random query (Equation 1), which has values between 0 and 1. Note that it is key to normalize the ranking results, since users might have different maximum or minimum scores when ranking queries. We have divided the problem in to two parts: (i) the query exists in the database. (ii) The query is new.

The model has been tested by performing three different experiments, described in Section 6. The results are shown in Table 3. Overall we can say that the results achieved are satisfactory for this study, as an average error across the different experiments is 20%. It is, however, at the same time difficult to interpret the results and to measure the efficacy of the experiments since we are working with randomly generated data. Thus, it might be hard to obtain significant clusters, for example, since data is completely random. It would be useful to retrieve real data of housings to test the results more accurately.

REFERENCES

[1] César Inga Chalco, Rodolfo Bojorque Chasi, and Remigio Hurtado Ortiz. 2019. Hierarchical Clustering for Collaborative Filtering Recommender Systems. In *Advances in Artificial Intelligence, Software and Systems Engineering*, Tareq Z. Ahram (Ed.). Springer International Publishing, Cham, 346–356.

[2] Mustansar ali Ghazanfar. 2012. *Robust, scalable, and practical algorithms for recommender systems.* Ph. D. Dissertation.

[3] Miha Grčar, Dunja Mladenić, Blaž Fortuna, and Marko Grobelnik. 2006. Data Sparsity Issues in the Collaborative Filtering Framework. 58–76. https://doi.org/10.1007/11891321_4

[4] Emilia Kacprzak, Laura M. Koesten, Luis-Daniel Ibáñez, Elena Simperl, and Jeni Tennison. 2017. A Query Log Analysis of Dataset Search. In *Web Engineering*, Jordi Cabot, Roberto De Virgilio, and Riccardo Torlone (Eds.). Springer International Publishing, Cham, 429–436.

[5] Sarika Mittal, Jothi Varman, Gloria Chatzopoulou, Magdalini Eirinaki, and Neoklis Polyzotis. 2010. QueRIE: A Query Recommender System Supporting Interactive Database Exploration. 1411–1414. https://doi.org/10.1109/ICDMW.2010.43

[6] Rajatish Mukherjee, Partha Dutta, and Ip Sen. 2001. MOVIES2GO - A new approach to online movie recommendation. (09 2001).

[7] Deuk Hee Park, Hyea Kyeong Kim, Il Young Choi, and Jae Kyeong Kim. 2012. A literature review and classification of recommender systems research. *Expert Systems with Applications* 39, 11 (2012), 10059–10072. https://doi.org/10.1016/j.eswa.2012.02.038

[8] Paul Resnick, Neophytos Iacovou, Mitesh Suchak, Peter Bergstrom, and John Riedl. 1994. GroupLens: an open architecture for collaborative filtering of netnews. In *Conference on Computer Supported Cooperative Work*.

[9] Fred W. Roush. 1982. Topics in the theory of voting, The UMAP expository monograph series : Philip D. Straffin, Jr. Boston, Birkhauser, 1980, US $5.00. *Mathematical Social Sciences* 2 (1982), 111–112.

[10] Murchhana Tripathy, Santilata Champati, and Srikanta Patnaik. 2022. SVD-initialised K-means clustering for collaborative filtering recommender systems. *International Journal of Management and Decision Making* 21 (2022), 71–91. https://doi.org/10.1504/IJMDM.2022.119580 Publisher Copyright: © 2022 Inderscience Enterprises Ltd.. All rights reserved..