Evaluating Dataset portions based on query logs

Stefano Genetti
stefano.genetti@studenti.unitn.it
University of Trento
Trento, Italy

Pietro Fronza
pietro.fronza@studenti.unitn.it
University of Trento
Trento, Italy

KEYWORDS

datasets, data mining, recommendation system, database

1 INTRODUCTION

The purpose of a *recommendation system* is to provide suggestions for items that are most pertinent to a particular user. There is an extensive class of applications that involve predicting user responses to options. For example, online stores provides suggestions about what the customer might like to buy, media services providers arrange personalized music playlists based on a prediction of listener interests.

The goal of this work is to design, develop, implement a sophisticated query recommendation system which recommends to the user a set of queries, the answer set of which contains data that are of interest to the user. We take into account a database consisting of one single relational table populated with data about people. Though we consider interrogations about persons, our recommendation system can be easily extended to applications regarding any kind of topic.

Research on this field can be useful in several contexts. Indeed, nowadays search engines are characterized with the opportunity to search easily and quickly: entering some keywords for the things user is looking for and list of suggested queries will be shown. For instance with the rapid growth of users in social networks, recommender systems are already integrated in every search query of social network's users.

The problem is challenging and non-trivial to solve. First of all, it is necessary to deal with the common difficulties of recommendation systems. It is intrinsically complicated to understand the taste of a complex organism like the human being.
Another typical challenge of recommender systems is the *cold start* problem: a new user enters the system or a new query is presented, and therefore, it will be difficult for the algorithm to predict the taste or preferences of the new user, or the rating of the new query, leading to less accurate recommendations.
Moreover, while features of movies, or features of news articles are quite straightforward to identify, it is harder to understand what a user likes or does not like about a query.
Another relevant complex aspect of our problem setting is about engineering a solution which is feasible to be scaled for contexts characterized by large input datasets.

In this work, we propose a hybrid recommendation approach to face the problem at hand. Our solution overcomes some of the common problems in recommender systems such as cold start and the sparsity problem in collaborative approach. Moreover, adopting a hybrid methodology we aim to offset the drawbacks of *content-based* and *collaborative* recommendation systems by combining both the methods. The proposed algorithm decides whether to rely on a collaborative approach or on a content-based approach, according to the number of preferences expressed by the target user and the number of votes received by the query taken into consideration. For instance, if the given user reviewed a lot of queries, we can probably outline their tastes quite accurately, hence a content based approach is applied in this case.
In order to face the challenges related to the design of a query recommendation system, we implement sophisticated data mining solutions like clustering to search for suitable query features and similarity metrics to compare users and items profiles.

In addition to this, we evaluated the performance of our solution in terms of *accuracy* and *responsiveness*. As expected, we noticed that the hybrid approach typically achieves more accurate recommendations than the fully content-based and fully collaborative filtering counterparts.
Finally, we compared our implementation with a baseline algorithm proposed by another group. On small datasets characterized by sophisticated user preferences, the accuracy of our implementation outperforms the baseline. However, our solution proposal turned out to not be able to scale well in the context of larger datasets. As a result, we end up getting worse results than the baseline algorithm when dealing with larger amount of data.

This report is organised as follows:

- at the beginning we provide a formal definition of the problem that we try to solve;
- then, we briefly overview the main techniques which have been employed;
- the purpose of the following section is to describe in detail our solution proposal;
- we provide an experimental evaluation to understand how well our implementation works. We compare our solution with a baseline algorithm to figure out which are the performance of our technique;
- finally, we conclude with a summary of the main highlights of the presented work and some final considerations.

Woodstock '18, June 03–05, 2018, Woodstock, NY
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

2 PROBLEM STATEMENT

In a recommendation-system application there are two classes of entities, which we shall refer to as *users* and *items*. In our setting, users are individuals who query a database containing data about people; items are sets of tuples in response to a given query. Users have preferences for certain items, and these preferences must be tested out of the data. The data itself is represented as a *utility matrix*, giving for each user-item pair, a value that represents what is known about the degree of preference of that user for that item. Values are expressed by integers from 1 to 100 indicating how happy the user is with the answer set of a given query. We assume that the matrix can be sparse, meaning that most entries are unknown. An unknown rating implies that we have no explicit information about the user's preference for the item. The goal of a recommendation system is to predict the blanks in the utility matrix.

More formally, the **input** of our problem consists of:

(1) A relational table *DB* populated with tuples with the following schema:

Person(id, name, address, age, occupation)

For the sake of simplicity, we assume that there are no *NULL* values: all the fields of all the tuples have a value.

(2) A user set U. (The user set might be empty).

(3) A query set Q which contains queries which have been posed in the past. Each query $q \in Q$ is characterized by:
- A unique query identifier q_{id};
- A set of conditions q_{def}. Each condition $c \in q_{def}$ is a collection of (attribute,value) pairs.

An example of a well defined query is a statement q such that q_{id} = *Q1821* and $q_{def} = \{(\text{name}, \text{John}), (\text{age}, 55)\}$. In SQL, this interrogation corresponds to:

```
SELECT * FROM Person WHERE name="John" AND age=55;
```

(4) A utility matrix A, whose entries $a[u, q]$ are integer values belonging to the interval $[1, 100]$, which indicate the rating that the user u assigned to query q.

On the other hand the **output** of our solution is an algorithm which implements an *utility function* $f_1 : U \times Q \longrightarrow R$ which associates to a given user $u \in U$ and query $q \in Q$ the expected rating that the user u would give to query q. The utility function is used to fill the missing values of the utility matrix A received as input. What is more, given the complete utility matrix filled with all the missing values, we propose a way to compute the utility of a query in general for all the users. Formally, we output a function $f_2 : Q \longrightarrow R$ such that $f(q)$ quantifies how much we recommend query $q \in Q$ to all the users in general.

3 RELATED WORK

In this section we provide a brief overview of the main topics and technologies which have been adopted to prepare our work.

3.0.1 Recommendation system. In a recommendation-system application there are two classes of entities: *users* and *items*. Users have preferences for certain items, and these preferences must be teased out of the data. The data itself is represented as a *utility matrix*, giving for each user-item pair, a value that represents what is known about the degree of preference of that user for that item. Essentially, the goal of a recommendation system is to predict the rating of user u on an un-rated item i or recommend some items for user u based on the existing ratings. In order to accomplish this task, recommender systems use a number of different technologies. We can classify these systems into two broad groups:

- *Content-based systems.* The idea is to suggest to a user items that have been previously rated highly by them.
- *Collaborative filtering systems.* The idea is to exploit the similarities among users and items in order to suggest items which are similar to the ones that similar users like.

In their work *Toward the next generation of recommender systems: a survey of the state-of-the-art and possible extensions*, Gediminas Adomavicius et al., provide a complete review of recommender systems. According to the literature, most of the systems now use a hybrid approach, combining collaborative filtering, content-based filtering, and other approaches. As an example, in *The Netflix Recommender System: Algorithms, Business Value, and Innovation*, Carlos A. Gomez-Uribe and Neil Hunt, explain that Netflix implements a hybrid recommendation solution. The popular website makes recommendations by comparing the watching and searching habits of similar users (i.e., collaborative filtering) as well as by offering movies that share characteristics with films that a user has rated highly (i.e., content-based filtering).

3.0.2 Clustering. Clustering is the process of examining a collection of "points", and grouping them into "clusters" according to some distance/similarity measure. The goal is that points in the same cluster have a small distance from another (i.e., they are similar), while points in different clusters are at a large distance from one another (i.e., they are not similar). Examples of widely used similarity metrics are *Jaccard similarity*, *cosine similarity*, *Pearson similarity*, *edit distance*. Overall, there are two major approaches to clustering: hierarchical and point-assignment. The former can be further divided into *agglomerative* and *divisive* approach. In agglomerative clustering, initially each point is a cluster, then the algorithm repeatedly combines the two "nearest" clusters into one with a bottom-up fashion. In divisive clustering, the procedure starts with one cluster which is recursively split into smaller clusters with a top-down approach. On the other hand, one representative point-assignment clustering technique is k-means clustering. Parameter k refers to the number of centroids we aim to identify in our data. A centroid is the imaginary or real location representing the center of the cluster. k-means algorithm identifies k number of centroids, and then allocates every data point to the nearest cluster. The objective is to minimize the sum of distances between the data points and the cluster centroid, to identify the correct group each data point should belong to. The main challenges related to clustering are: the *curse of dimensionality problem*, dealing with non-Euclidean spaces, choosing the proper number of clusters. In "*The Review on determining number of Cluster in K-Means Clustering*", Trupti M. Kodinariya and Prashant R. Makwana, explain how to face this latter problem in the case of k-means clustering.

3.0.3 Bucketization and hash table. A hash table is an extensively used data structure that implements an associative array or dictionary. A hash table uses a hash function to compute an index into an array of buckets. During insertion the hash function is used to put the value in the proper bucket. During lookup, the key is hashed and the resulting hash code indicates where the corresponding value is stored. The main challenge in this field is about handling *collisions*. Ideally, the hash function will assign each key to a unique bucket. However, due to memory constraints, hash functions are not perfect and might cause collisions where the hash function generates the same index for more than one key. The two main techniques to accommodate *collision resolution* are usually referred to by the literature as *separate chaining* and *open addressing*. The principal advantage of hash table is that the average time complexity for each lookup is independent of the number of elements stored in the data structure. Furthermore, the average complexity to search, insert, and delete data in a hash table is $O(1)$ (i.e., constant time). For this reason, they are widely used in many kinds of computer software applications.

4 SOLUTION

According to the literature, recommendation algorithms can be divided into two main categories: i) *content-based* recommendation systems and ii) *collaborative filtering* recommendation systems. In this work we propose an hybrid solution to solve the problem of query recommendation. Our aim is to conveniently combine a content-based approach with a collaborative filtering approach in order to benefit from their respective advantages and limit their inherent drawbacks. Our algorithm decides the proper method to use according to the number of queries that each user voted in the past.

Motivated by the above, at the beginning the input users are partitioned into three sets depending on the number of preferences in their row of the utility matrix.

(1) Users who voted a lot of queries are well characterized by their row in the utility matrix. As a rule of thumb we consider frequent voter each user who voted at least half of the queries in the query log. According to their preferences we can outline their tastes and provide them recommendations according to their predicted *user profile*. In order to succeed in achieving this objective we designed a content-based recommendation algorithm as described in Subsection 4.1.
(2) At this point, the algorithm suggested recommendations to fill the gaps in the utility matrix rows corresponding to the users who voted a lot of queries. On the other hand, we do not provide recommendations in the same way for the users who voted few queries of the query log. Indeed, due to the sparsity of their corresponding rows of the utility matrix it is not reasonable to predict their votes merely relying on their prior expressed preferences. Instead, we iteratively look for users who seem to have similar flavours and for queries which are similar to the ones already evaluated by the target user. This step of our solution is described in Subsection 4.2.
(3) Finally, in Subsection 4.3 we present a way to overcome the cold start problem related to users who did not vote any query of the query log. Perhaps, this is the case of new users that has not provided any interaction yet, therefore it is not possible to build a user profile following a content-based solution nor to provide them recommendations according to similar users or queries.
Note that, on the other hand, the case of a query without votes is already took into account by both the content-based and the collaborative filtering systems as described in Subsection 4.1 and Subsection 4.2 respectively.

The three points are conveniently summarized in Algorithm 1. Together with this report we provide a Python implementation of the presented algorithms.

Algorithm 1 Hybrid query recommendation system

```
input:
  relational table := $DB$
  set of users := $U$
  set of queries := $Q$
  utility matrix := $A$

▷ partition users in three sets according to the number of voted queries

frequentVoters = set()   ▷ set of users who voted a lot of queries
rareVoters = set()       ▷ set of users who voted few queries
coldStartUsers = set()   ▷ set of users who voted nothing

for user in $U$ do
    votedQueries ← 0
    for query in $Q$ do
        if $U$[user][query] is not empty do
            votedQueries ← votedQueries + 1
        end if
    end for
    if votedQueries > $\frac{|Q|}{2}$ do
        frequentVoters.add(user)
    else if votedQueries == 0 do
        coldStartUsers.add(user)
    else
        rareVoters.add(user)
    end if
end for

▷ compute recommendations for frequent voters with content based recommendation
content_based(...)                        ▷ Algorithm 2

▷ compute recommendations for voters who voted few queries with collaborative filtering recommendation
collaborative_filtering(...)              ▷ Algorithm 5

▷ compute recommendations for voters who voted nothing
cold_start_users_recommendation(...)      ▷ Algorithm 7

output:
user_recommendation   ▷ the expected vote for each missing entry $(u, q) \in A$
```

id	name	age	occupation	address
0	Stefano	20	student	via Rossi
1	Pietro	20	student	via Rossi
2	Luca	30	professor	via Gialli
3	Sara	40	farmer	via Gialli
4	Maria	50	farmer	via Gialli
5	Sofia	60	retaired	via Verdi

Table 1: Toy example: small relational table Person.

query id	query definition
q_1	occupation="student"
q_2	address="via Gialli"
q_3	name="Stefano" and age=20
q_4	name="Stefano"
q_5	address="via Rossi"

Table 2: Toy example: query log.

4.1 Content-based recommendation

As explained above, in a recommendation-system application there are two classes of entities: *users* and *items*. In our application these lasts are the query in the log. Our ultimate goal for content-based recommendation is to create both an item profile $p(q)$ consisting of feature-value pairs which encode its most important characteristics, and a user profile $p(u)$ summarizing the preferences of the user, based on their row of the utility matrix. We represent both item profile and user profile with a fixed length multidimensional vector. The decision to represent entity profiles with a fixed length vector results in considerable advantages in terms of time and space computational complexity. Moreover, as described in the following of this section, the distance between two vectors of limited size is more meaningful. More in depth, each entity profile is encoded by a ten dimensional vector as depicted in Figure 1. Each position of the vector corresponds to a specific feature. For each user, the first half of its own vector profile represents the most important features of that user and it is constant regardless of the query we are evaluating. On the other hand, the last five positions of the user profile represent the most valuable features of the query at hand. Symmetrically, suppose we are computing the recommendation for a user u about a query q. The initial five positions of the query profile of q represents the most important features of user u. Instead, the second half of the profile of query q encodes the major features of q and it is constant regardless of the user. In order to clarify the notion of *important feature* for users and queries and to explain how the gaps of their corresponding profiles are filled, we detail the steps of the algorithm by means of a toy example. The overall procedure is summarized in Algorithm 2. In our toy example, we consider a small relational table (Table 1), a set of three users $\{u_1, u_2, u_3\}$, a set of five queries $\{q_1, q_2, q_3, q_4, q_5\}$ (Table 2) and a utility matrix A such that there is only a missing value for the entry $A[u_1, q_5]$ (Table 3). The purpose of our content-based recommendation system is to estimate the vote of this empty entry of the utility matrix.

Algorithm 2 Content Based

```
input:
  set of users := $U$
  set of queries := $Q$
  utility matrix := $A$
  relational table := $DB$

globalTupleImportance = expected importance of each tuple
                                                  ▹ subsection 4.1.1
globalAttributeImportance = expected importance of each search
attribute                                         ▹ subsection 4.1.1

num_cluster = 5
k_means_clustering($DB$, num_cluster)             ▹ subsection 4.1.2

for each user row $u$ in $A$ do:
    user_obj = new User()                         ▹ init user object
    for each column query $q$ in $A$ do:
        if $A[u, q] \neq \emptyset$ do:
            user_obj.addVotedQuery(q)
        else do:
            user_obj.addUnVotedQuery(q)
        end if
    end for
    user_obj.avgVote ← compute user avg vote

                                                  ▹ subsection 4.1.3
    user_profile ← compute user profile of the current user $u$

                        ▹ compute query profile of each unvoted query
    for query $q \in$ user_obj.getUnVotedQueries() do
                                                  ▹ subsection 4.1.4
        query_profile[$q$] ← profile of $q$ w.r.t. user $u$
    end for

                        ▹ predict recommendation for each unvoted query
    for query $q \in$ user_obj.getUnVotedQueries() do
▹ complete $u$ profile according to $q$'s features (subsection 4.1.5)
        user_profile ← update profile of $u$ w.r.t. query $q$

                        ▹ predict recommendation $A[u, q]$ (subsection 4.1.5)
        $A[u, q]$ ← recommendation(user_profile, query_profile[$q$])
    end for
end for

output:
user_recommendation        ▹ the expected vote for each missing
entry $(u, q) \in A$
```

	q_1	q_2	q_3	q_4	q_5
u_1	96	5	95	90	?
u_2	100	50	100	50	100
u_3	5	60	5	60	5

Table 3: Toy example: utility matrix.

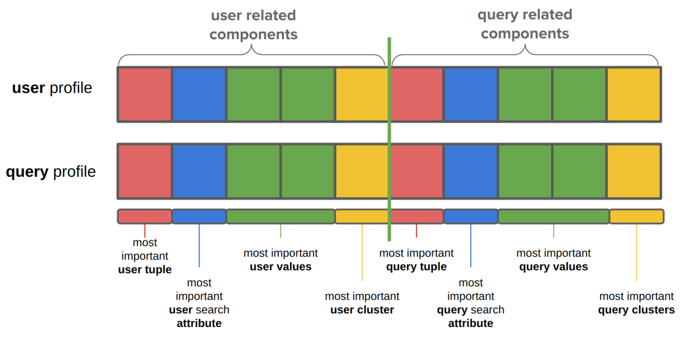

user related components
query related components
user profile
query profile
most important user tuple
most important user search attribute
most important user values
most important user cluster
most important query tuple
most important query search attribute
most important query values
most important query clusters

Figure 1: User profile and query profile vector.

4.1.1 Global important tuples and search attributes. Intuitively, the general importance of a tuple and of a search attribute, regardless of the specific user and query, depends on the number of times they appear respectively in the result sets and in the definitions of the queries in the log. For instance, in our toy example, the search attribute name="Stefano" is probably important since it appears in two of the five queries in the log. Similarly, tuple 0 is probably a relevant person, indeed it appears in the result sets of queries q_1, q_3, q_4, q_5. In such a small dataset it is reasonable to provide the exact number of occurrences of each tuple and search attribute. However, it is not feasible to compute these values exactly for larger datasets. Indeed, it would require an excessive amount of space to store the number of times that each tuple and each search attribute appear. In order to face this limitation, we implement a hash table with a fixed amount of buckets. We opt for 1223 buckets. Actually, the number is prime and reasonably large. Each bucket is associated with a counter. Every time we execute the hash function on an item we increment the counter of the corresponding bucket. In order to estimate the number of occurrences of a given item, we compute its hash and read the counter of the output basket. To do this we adopt the Python built-in hash function hash(). Following this strategy we are able to approximately estimate the number of occurrences of each tuple and search attribute in constant space (1223 buckets with their counters) as described in Algorithm 3 and Algorithm 4. Due to the small number of tuples and attributes which characterize our toy example, it is unlikely that there would be collisions. As a consequence, the expected number of occurrences of each tuple and search attribute is equal to the actual number of times they appear. The situation in memory is schematized in Figure 2 and Figure 3. It is important to observe that we memorize only the numerical counters, the tuples and the attributes are not stored in the buckets. For very large datasets it is possible to reduce the number of iterations of Algorithm 3 considering a random sample of queries and not the entire log.

4.1.2 Clustering of table Person. Our aim at this point is to partition table Person into categories of people. Intuitively, as an example, a user who interrogates the database could manifest interest about young professors or about students who live in via Rossi. It is not trivial to identify user categories. To accomplish this task, we execute k-means clustering on table Person. In particular we rely on the implementation provided by scikit-learn Python library. This latter

Algorithm 3 Expected global importance of tuples and search attributes

```
input:
  set of queries := $Q$

expectedTupleFrequency = HashTable()
expectedAttributeFrequency = HashTable()

for query $q \in Q$ do
                                        ▷ retrive query search attributes
    searchAttributes = retrieveSearchAttributes($q$)
                                        ▷ compute query result
    queryResult = computeQueryResult($q$)
    for attribute $a \in$ searchAttributes do
        expectedAttributeFrequency.insertValue($a$)
    end for
    for tuple $t \in$ queryResult do
        expectedTupleFrequency.insertValue($t$)
    end for
end for

output:
  expectedTupleFrequency
  expectedAttributeFrequency

▷ it is possible to get the expected importance of a tuple ($x$) with:
expectedTupleFrequency.getValue($x$)
```

Algorithm 4 HashTable

```
attributes:
  number of buckets = 1223
  bucket = dictionary()        ▷ bucket[$i$] is the counter of bucket $i$

methods:
                                   ▷ hash function of the input data
hash_function(data){
    hash_code = hash(data)             ▷ built-in Python function
    return hash_code mod |bucket|
}

                    ▷ update the expected importance of the input data
insertValue(data){
    key = hash_function(data)
    bucket[key] = bucket[key] + 1
}

                        ▷ get expected importance of the input data
getValue(data){
    key = hash_function(data)
    return bucket[key]
}
```

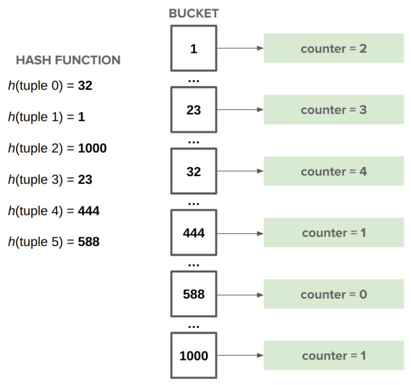


Figure 2: Toy example: important tuples computation. Tuple 0 is the most important of the database since it appears in the result sets of four queries.

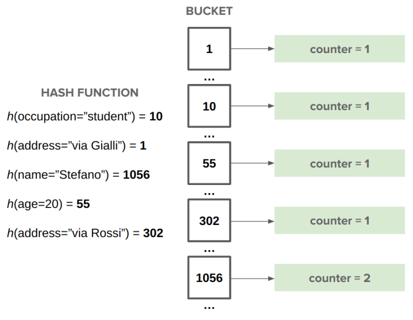


Figure 3: Toy example: important search attribute computation. Search attribute name="Stefano" is the most important since it appears in the definition of two queries.

partitions the relational table into k clusters. The optimal value of k depends on the particular dataset at hand. In order to choose the most proper value we use the *elbow method.* All in all, selecting a perfect value of k is not important for our purposes, by default it is set to $k = 5$ which works reasonably well in almost every dataset. Indeed, partitioning the tuples into five categories is typically enough.

For the purpose of executing k-means algorithm it is required to encode each categorical variable of table Person with one hot encoding. One hot encoding involves creating a new column for each attribute value in the relational table. Then a 1 or 0 is assigned depending on if that categorical value is in the tuple or not. We one hot encode the categorical values in our dataset using the pandas "get_dummies" function. In table Person there is also a continuous attribute: column age. At the beginning we tried to normalize the values in this column and to consider their normalized values during k-means clustering execution. However, we noticed that in this way the values of attribute age tended to be interpreted as being less important than the other. As a result we decided to make age column categorical as follows:

- people younger than 12 years old are classified as "baby"

id	**v.Gialli**	**v.Rossi**	**farmer**	**student**	**medium**	**young**
0	0	1	0	1	0	1
1	0	1	0	1	0	1
2	1	0	0	0	0	1
3	1	0	1	0	0	1
4	0	0	1	0	1	0
5	0	0	0	0	1	0

Table 4: Toy example: one hot encoding of table Person.

id	name	age	occupation	address	**cluster**
0	Stefano	20	student	via Rossi	**0**
1	Pietro	20	student	via Rossi	**0**
2	Luca	30	professor	via Gialli	**3**
3	Sara	40	farmer	via Gialli	**1**
4	Maria	50	farmer	via Gialli	**4**
5	Sofia	60	retaired	via Verdi	**2**

Table 5: Toy example: small relational table Person with clusters. People who live in via Rossi have been assigned to the same group.

- people older than 12 and younger than 40 years old are classified as "young"
- people older than 40 and younger than 60 years old are classified as "medium"
- people older than 60 years old are classified as "old"

At this point we have all the features required to execute k-means clustering. However, the number of features identified with the procedure just explained are too many. Indeed we end up having a column for each possible value for each attribute of table Person. This amount of data would result in a time and memory demanding clustering computation. Moreover, considering too many features leads to the *curse of dimensionality* problem. Therefore, we count the number of 1s in each column and consider only the six features which appear more frequently in the relational table. In the context of our toy example the obtained table could be similar to Table 4. Now we can execute k-means clustering on Table 4 to partition table Person in cluster-categories. A possible outcome is represented in Table 5.

4.1.3 Compute the first half of the user profile. Aiming at providing a recommendation to user u_1 about query q_5 we have to compute the user profile of u_1 according to their row in the utility matrix. First of all, it makes sense to normalize the utilities by subtracting the average value for the user. That way, we get negative weights for queries with a below-average rating, and positive weights for items with above-average ratings. In our example, user u_1 gives an average rating of 71.5. As a result, after the normalization process we get an utility matrix like Table 6. Hereafter, we refer to the normalized utility matrix A with the notation $\bar{A}$. In order to limit both time and space computational complexity related to the elaboration of the user profile we do not consider all the queries for which the user expressed a preference in the past. On the contrary, we consider only the most remarkable queries which are the ones voted with a value which is significantly different with respect to the user average. To do this we sort in descending order the array of queries voted by u_1 in the past according to the absolute value of their normalized received mark, and we take up to the first twenty queries. In our example, if we sort the queries voted by user u_1 with this criterion we get:

$$[q_2, q_1, q_3, q_4]$$

In this case we consider all the available queries since $4 < 20$. Let $\mathcal{S}$ be the set of the queries taken into account for the computation of the user profile. We estimate the importance of a search attribute a for a user u with the following formula:

$$\textbf{importance}(a) = \sum_{q \in \mathcal{S}} TF(a, |querydefinition|) * |\bar{A}[u, q]| \quad (1)$$

In the formula *TF* is the *time frequency measure* of attribute a with respect to the length of the query definition. For example, in the query (name="Stefano" and age=20) attribute name="Stefano" has $TF = \frac{1}{2}$. The idea is that an attribute which appears alone in the query definition could be more important and so we weight more its contribution. $|\bar{A}[u, q]|$ is the absolute value of the normalized vote assigned by user u to query q according to the normalized utility matrix $\bar{A}$. In this case the idea is that, if the vote assigned by u to q is far from the vote average of u, probably it is reasonable to consider as important the attributes which constitute the definition of q. For the sake of clarity, let us consider as an example how to calculate the importance of attribute name="Stefano" for user u_1.

$$\textbf{importance}(\text{name="Stefano"}) = \frac{1}{2} * 23.5 + 1 * 18.5 = 30.25$$

Similarly we can compute for a specific user u the importance of a tuple t, of a value v, of a cluster c. Let $rs(q)$ be the result set of a query q. For instance, considering our toy example, the $rs(q_2)$ is reported in Table 7. The importance for a user u of a tuple t, of a value v and of a cluster c are calculated respectively as:

$$\textbf{importance}(t) = \sum_{q \in \mathcal{S}} TF(t, |rs(q)|) * |\bar{A}[u, q]| \quad (2)$$

$$\textbf{importance}(v) = \sum_{q \in \mathcal{S}} TF(v, |rs(q)|) * |\bar{A}[u, q]| \quad (3)$$

$$\textbf{importance}(c) = \sum_{q \in \mathcal{S}} TF(c, |rs(q)|) * |\bar{A}[u, q]| \quad (4)$$

In these last formulas, TF represents the frequency of a cluster, attribute or value in the long result set of a query. For example the TF of value "farmer" in query q_2 is $\frac{2}{3}$ since two of the three people who live in via Gialli are farmers. Intuitively, a value with high TF is more relevant than another which appears few times in the result of a query. In our first implementation we weighted the occurrences with $TF * IDF$ (*term frequency inverse document frequency*) instead of just TF. However, we noticed that the IDF component favours excessively terms which appear in few query results, which is not reasonable for our application. With the above formulas we compute the importance for a specific user u of each attribute, tuple, value and cluster which appear in the queries in $\mathcal{S}$. At this point we consider the user most important tuple $u[t]$ which is the feature associated with the first position of the user profile; the user most important attribute $u[a]$ which is the feature associated with the second position of the user profile; the two user most importatant values $u[v_1], u[v_2]$ which are the features associated with the third and fourth position of the user profile; the user most important cluster $u[c]$ which is the feature associated with the fifth position of the user profile. The first entry of the user profile vector $p(u)[0]$ is filled with the average normalized vote assigned by user u to tuple $u[t]$ when it appears in the result set of the queries in S.

$$p(u)[0] = \frac{\sum_{q \in \mathcal{S}: u[t] \in r(q)} \bar{A}[u, q]}{|\{q \in \mathcal{S} : u[t] \in r(q)\}|} \quad (5)$$

The second entry of the user profile vector $p(u)[1]$ is filled with the average normalized vote assigned by user u to search attribute $u[a]$ when it appears in the definition of the queries in S.

$$p(u)[1] = \frac{\sum_{q \in \mathcal{S}: u[a] \in qdefinition} \bar{A}[u, q]}{|\{q \in \mathcal{S} : u[a] \in qdefinition\}|} \quad (6)$$

The third entry of the user profile vector $p(u)[2]$ is filled with the average normalized vote assigned by user u to value $u[v_1]$ when it appears in the result set of the queries in S.

$$p(u)[2] = \frac{\sum_{q \in \mathcal{S}: u[v_1] \in r(q)} \bar{A}[u, q]}{|\{q \in \mathcal{S} : u[v_1] \in r(q)\}|} \quad (7)$$

The fourth entry of the user profile vector $p(u)[3]$ is filled with the average normalized vote assigned by user u to value $u[v_2]$ when it appears in the result set of the queries in S.

$$p(u)[3] = \frac{\sum_{q \in \mathcal{S}: u[v_2] \in r(q)} \bar{A}[u, q]}{|\{q \in \mathcal{S} : u[v_2] \in r(q)\}|} \quad (8)$$

The fifth entry of the user profile vector $p(u)[4]$ is filled with the average normalized vote assigned by user u to cluster $u[c]$ when it appears in the result set of the queries in S.

$$p(u)[4] = \frac{\sum_{q \in \mathcal{S}: u[c] \in r(q)} \bar{A}[u, q]}{|\{q \in \mathcal{S} : u[c] \in r(q)\}|} \quad (9)$$

If we do all the above calculations for the first user u_1 of our toy example we get:

- the most important tuple for u_1 is tuple 2 which appears only in the result set of query q_2 but with a vote really far from the user average vote. Query q_2 was voted by u_1 with mark 5. We fill the first entry of $p(u_1)$ with:
$$\frac{(5 - 71.5)}{1} = -66.5$$
- the most important search attribute for u_1 is address="via Gialli" which appears only in the definition of query q_2 but with a vote really far from the user average vote. We fill the second entry of $p(u_1)$ with -66.5.
- the most important values for u_1 are "via Gialli" and "Luca" which appear only in the result set of query q_2 but with a vote really far from the user average vote. We fill the third and fourth entries of $p(u_1)$ with -66.5.
- the most important cluster for u_1 is cluster 3 which appears only in the result set of query q_2 but with a vote really far from the user average vote. We fill the fifth entry of $p(u_1)$ with -66.5.

Due to the littleness of the dataset these values are not very enjoyable, but the purpose of the toy example is to make the solution procedure more clear.

	q_1	q_2	q_3	q_4	q_5
u_1	24.5	-66.5	23.5	18.5	?

Table 6: Toy example: normalized utility matrix.

id	name	age	occupation	address	cluster
0	Stefano	20	student	via Rossi	0
1	Pietro	20	student	via Rossi	0

Table 7: Toy example: q_5 result set.

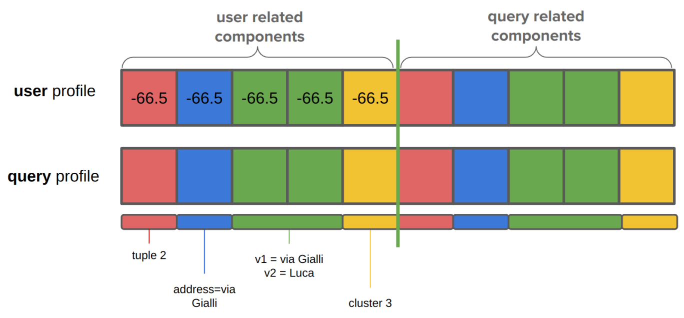


Figure 4: Toy example: first half of the user profile.

It is important to underline that for the computation of the first entries of the user profile we do not use the information described in Section 4.1.1.

So far we completed the first half of $p(u_1)$ as illustrated in Figure 4. The second half depends on the query under evaluation as explained in Section 4.1.5.

4.1.4 Compute query profile. The first half of the query profile vector $p(q)$ of a query q depends on the particular target user u. The feature associated with the first entry of the query profile represents the user most important tuple $u[t]$. The feature associated with the second entry of the query profile represents the user most important attribute $u[a]$. The features associated with the third and fourth entries of the query profile represent the user most important values $u[v_1], u[v_2]$. The feature associated with the fifth entry of the query profile represents the user most important cluster $u[c]$. The values to fill these entries are computed as follows:

- in $p(q)[0]$ we write how many times tuple $u[t]$ appears in the result set of query q with respect to the length of the result set.
$$p(q)[0] = \frac{|\{t \in r(q) : u[t] \in t\}|}{|r(q)|} \tag{10}$$
In the toy example, tuple 2 which is the most important tuple of user u_1 do not appear in the result set of query q_5. As a consequence:
$$p(q_5)[0] = 0$$
- in $p(q)[1]$ we store how many times attribute $u[a]$ appears in the definition of query q with respect to the length of its definition.
$$p(q)[1] = \frac{|\{a \in qdefinition : u[a] = a\}|}{|qdefinition|} \tag{11}$$
In the toy example, attribute address="via Gialli" which is the most important attribute of user u_1 do not appear in the definition of query q_5. As a consequence:
$$p(q_5)[1] = 0$$
- in $p(q)[2]$ we write how many times value $u[v_1]$ appears in the result set of query q with respect to the length of the result set.
$$p(q)[2] = \frac{|\{t \in r(q) : u[v_1] \in t\}|}{|r(q)|} \tag{12}$$
In the toy example, value "via Gialli" which is the first important value of user u_1 do not appear in the result set of query q_5. As a consequence:
$$p(q_5)[2] = 0$$
- in $p(q)[3]$ we write how many times value $u[v_2]$ appears in the result set of query q with respect to the length of the result set.
$$p(q)[3] = \frac{|\{t \in r(q) : u[v_2] \in t\}|}{|r(q)|} \tag{13}$$
In the toy example, value "Luca" which is the second important value of user u_1 do not appear in the result set of query q_5. As a consequence:
$$p(q_5)[3] = 0$$
- in $p(q)[4]$ we write how many times cluster $u[c]$ appears in the result set of query q with respect to the length of the result set.
$$p(q)[4] = \frac{|\{t \in r(q) : u[c] \in t\}|}{|r(q)|} \tag{14}$$
In the toy example, cluster 3 which is the most important cluster of user u_1 do not appear in the result set of query q_5. As a consequence:
$$p(q_5)[4] = 0$$

At this point the arrays are populates as illustrated in Figure 5.

Now, we can fulfill the second half of the query profile which does not depend on the user at hand. The result set of the query is a collection of tuples. We measure the relevance of each tuple as its global importance calculated as described in Section 4.1.1. The sixth entry of the query profile represents the most important tuple $q[t]$ according to this metric in the result set of the query. Analogously, we measure the relevance of each attribute in the query definition as its global importance calculated as described in Section 4.1.1. The seventh entry of the query profile represents the most important attribute $q[a]$ in the definition of the query. The relevance of each value v of the result $r(q)$ is measured by the number of occurrences of v in $r(q)$. Analogously, the relevance of each cluster c of the result $r(q)$ is measured by the number of occurrences of c in $r(q)$. Indeed, estimating the general importance of values and clusters with the procedure described in 4.1.1 would be significantly more computationally demanding than the global importance estimation of tuples and attributes. The eighth, ninth and tenth entries of the query profile represent respectively: the first most important value $q[v_1]$, the second most important value $q[v_2]$ and the most important cluster $q[c]$ in the result set of the query. Taking into account the result set of query q_5 (Table 7) of our toy example, we obtain:

- in $r(q_5)$ there are tuples 0 and 1. According to Section 4.1.1, tuple 0 has an expected global importance of 4 while tuple 1 has an expected global importance of 2. As a result, entry $p(q_5)[5]$ represents tuple 0.
- in the definition of q_5 there is only one search attribute: address="via Rossi". As a result, entry $p(q_5)[6]$ represents attribute address="via Rossi".
- in $r(q_5)$ value "Stefano" appears one time, age 20 which corresponds to value "young" appears two times, value "student" appears two times, value "via Rossi" appears two times and value "Pietro" appears one time. As a result, entry $p(q_5)[7]$ represents value "vai Rossi" and entry $p(q_5)[8]$ represents value "student".
- in $r(q_5)$ there is only one cluster which is cluster 0. As a result, entry $p(q_5)[9]$ represents cluster 0.

We fill the entries of the second half of the query vector in the same way as described for the first half at the beginning of this section. In $p(q)[5]$ we write the fraction of occurrences of tuple $q[t]$ with respect to the length of $r(q)$ (always $\frac{1}{|r(q)|}$ since a give tuple cannot appear more than one time in the result of a query). In $p(q)[6]$ we write the fraction of occurences of attribute $q[a]$ with respect to the length of the definition of q (always $\frac{1}{|qdefinition|}$ since a given search attribute cannot appear more than one time in the definition of a query). In $p(q)[7]$ we write the fraction of occurrences of value $q[v_1]$ with respect to the length of $r(q)$. In $p(q)[8]$ we write the fraction of occurrences of value $q[v_2]$ with respect to the length of $r(q)$. Finally, in $p(q)[9]$ we write the fraction of occurrences of cluster $q[c]$ with respect to the length of $r(q)$. The number of tuples in the result set of query q_5 of our toy example is 2. As a consequence, we get:

- $p(q_5)[5] = \frac{1}{2}$,
- $p(q_5)[6] = \frac{2}{2} = 1$,
- $p(q_5)[7] = \frac{2}{2} = 1$,
- $p(q_5)[8] = \frac{2}{2} = 1$,
- $p(q_5)[9] = \frac{2}{2} = 1$.

The updated situation is illustrated in Figure 6.

4.1.5 Compute the recommendation. Ultimately, we can complete the second half of the user profile $p(u)$ which depends on the target query. Indeed, entry $p(u)[5]$ is associated with feature $q[t]$; entry $p(u)[6]$ is associated with feature $q[a]$; entry $p(u)[7]$ is associated with feature $q[v_1]$; entry $p(u)[8]$ is associated with feature $q[v_2]$; entry $p(u)[9]$ is associated with feature $q[c]$. In each of these positions we store the average normalized vote assigned by the user to the corresponding feature. The idea is better explained with our toy example as follow:

- entry $p(u_1)[5]$ is associated with feature "tuple 0" as explained in Section 4.1.4. Tuple 0 appears in the result set of queries q_1, q_3, q_4 which have been rated by user u_1 with vote

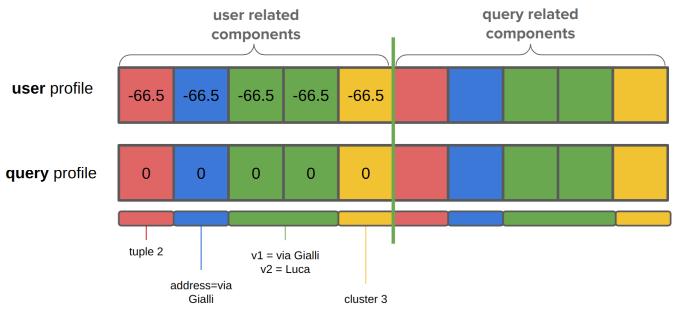


Figure 5: Toy example: first half of the query profile.

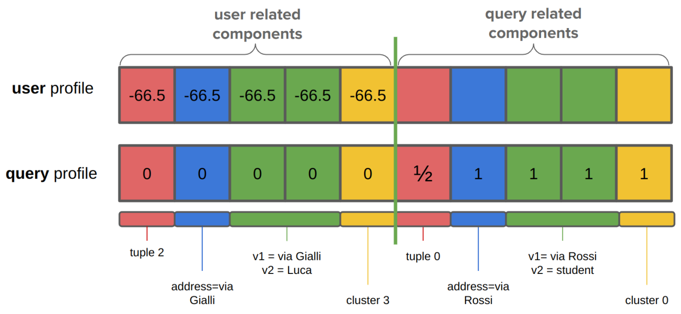


Figure 6: Toy example: second half of the query profile.

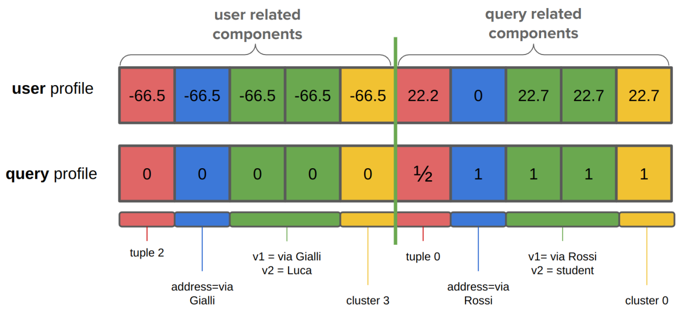


Figure 7: Toy example: second half of the user profile.

96, 95 and 90 respectively. As a result, in $p(u_1)[5]$ we write:

$$p(u_1)[5] = \frac{(96 - 71.5) + (95 - 71.5) + (90 - 71.5)}{3} = 22.2$$

- entry $p(u_1)[6]$ is associated with feature address="via Rossi" as explained in Section 4.1.4. Search attribute address="via Rossi" does not appear in any query posted by u_1 in the past. As a result, in $p(u_1)[6]$ we write value 0.
- entry $p(u_1)[7]$ is associated with value "via Rossi" as explained in Section 4.1.4. Value "via Rossi" appears in the result set of queries q_1, q_3, q_4 which have been rated by user u_1 with vote 96, 95 and 90 respectively. In q_1 value "via Rossi" appears two times; in q_3 it appears one time; in q_4 it appears one time. As a result, in $p(u_1)[7]$ we write:

$$p(u_1)[7] = \frac{2 * (96 - 71.5) + (95 - 71.5) + (90 - 71.5)}{4} = 22.7$$

- entry $p(u_1)[8]$ is associated with value "student" as explained in Section 4.1.4. Value "student" appears in the result set of queries q_1, q_3, q_4 which have been rated by user u_1 with vote 96, 95 and 90 respectively. In q_1 value "student" appears two times; in q_3 it appears one time; in q_4 it appears one time. As a result, in $p(u_1)[8]$ we write:

$$p(u_1)[8] = \frac{2*(96-71.5)+(95-71.5)+(90-71.5)}{4} = 22.7$$

- entry $p(u_1)[9]$ is associated with cluster 0 as explained in Section 4.1.4. Cluster 0 appears in the result set of queries q_1, q_3, q_4 which have been rated by user u_1 with vote 96, 95 and 90 respectively. In q_1 cluster 0 appears two times; in q_3 it appears one time; in q_4 it appears one time. As a result, in $p(u_1)[9]$ we write:

$$p(u_1)[9] = \frac{2*(96-71.5)+(95-71.5)+(90-71.5)}{4} = 22.7$$

At this stage we completed both the user profile and the query profile as illustrated in Figure 7.

With profile vectors for both users and queries, we can estimate the degree to which a user would prefer a query by computing the *cosine distance* between the user's and query's vectors.

$$cosine\_distance(u,q) = \frac{p(u)\cdot p(q)}{\|p(u)\|\|p(q)\|} \tag{15}$$

We observed that the vector for a user will have positive numbers for features that tend to appear in queries the user likes and negative numbers for features that tend to appear in queries the user does not like. Consider a query with many features the users likes, and only a few or none that the user does not like. The cosine of the angle between the user's and query's vectors will be a large positive fraction. That implies an angle close to 0, and therefore a small cosine distance between the vectors. Next, consider a query with about as many features that the user likes as those the user does not like. In this situation, the cosine of the angle between the user and query is around 0, and therefore the angle between the two vectors is around 90 degrees. Finally, consider a query with mostly features the user does not like. In that case, the cosine will be a large negative fraction, and the angle between the two vectors will be close to 180 degrees - the maximum possible cosine distance. Note that, since we consider reasonably small vectors of fixed size, the distance is less computationally demanding to be computed and more meaningful. Indeed, the two vectors belong to a dimensional space with a reasonable fixed number of dimensions and their entries correspond to the most important features. Splitting the vector profiles into two parts: features important for the user and features important with respect to the query, allows to outline better the characteristics of the two entities and avoids the situation such that one of the two vectors is the null vector. What is more, notice that having normalized the votes in the utility matrix, the features which have not been seen by a user are not interpreted as something they does not like. This is the case of user u_1 who has never voted a query with search attribute address="via Rossi" in its definition. Looking at its profile we understand that u_1 is indifferent with respect to search attribute address="via Rossi", since the corresponding feature has value 0.

Computing the cosine distance between u_1 and q_5 in our toy example we get:

$$cosine\_distance(u_1, q_5) = 0.2476$$

The cosine distance measure returns a value in the interval $[-1, 1]$. From this value we have to retrieve the expected user vote. To do this, firstly we scale the distance value in range [0,1] with the following formula:

$$x = \frac{\text{val} - \text{min}}{\text{max} - \text{min}} = \frac{cosine\_distance(u,q)+1}{2} \tag{16}$$

Then we scale the obtained value in range [1,100]:

$$y = 99x + 1 \tag{17}$$

The obtained vote y does not take into account the user vote average. Indeed, some people tend to assign lower or higher marks than others. Hence, we need to measure the offset between y and 50 and add this value to the user average vote $\text{avg}(u)$.

$$z = (y - 50) + \text{avg}(u) \tag{18}$$

Finally, we ensure that the predicted vote is an integer value between 1 and 100 as required by the problem specifications:

$$\text{predicted vote}(u,q) = \text{round}(\min(z, 100))$$

$$\text{predicted vote}(u,q) = \text{round}(\max(z, 1))$$

At the end of the day, the recommendation of query q_5 for user u_1 is:

$$\text{predicted vote}(u,q) = [(99*\frac{0.2476+1}{2}+1)-50]+71.5 = 84.5 \approx 85$$

The result is reasonable since from the utility matrix we can presume that u_1 likes query results related to people who live in via Rossi.

4.2 Collaborative filtering recommendation

Another technique suggested by the literature to develop recommender systems is *collaborative filtering*. This group of algorithms can be further split into two main categories: i) user-based collaborative filtering; ii) item-based collaborative filtering. Indeed, one way of predicting the value of the utility matrix entry for user u on query q is to find the n users (for some predetermined n) most similar to u and average their ratings for query q, counting only those among the n similar users who have rated q. Dually, we can use item similarity to estimate the entry for user u and query q. Find the m queries most similar to q, for some m, and take the average rating, among the m queries, of the ratings that u has given. In general, it can be hard to detect similarity among either queries or users, because we could have little information about user-query pairs in the sparse utility matrix. For instance, even if two queries are considered similar, there could be very few users who rated both. Likewise, even if two users have expressed similar preferences, they may not have rated any query in common. In order to deal with this limitation, in this work we present a collaborative filtering algorithm which faces this problem clustering both queries and users. More in depth, there is little reason to try to cluster into a small number of clusters immediately. Rather, in the following of this section we propose a bottom-up hierarchical clustering approach.

For convenience, the overall procedure is summarized in Algorithm 5 and Algorithm 6.

4.2.1 Cluster initialization. At the beginning we create a cluster object for each user and for each query in the dataset. Then, we cluster together all the queries which have the same definition. Indeed, two queries q and q' with the same definition are identical and so perfect candidates to be placed in the same cluster. Moreover, we can reasonably assume that if a user u voted query q with vote v, they would assign a similar mark to query q'. With the proper bookkeeping this procedure can be executed in $O(|Q|)$, where Q is the set of queries in the log.

4.2.2 Generating recommendations. At this point we can revise the utility matrix A. For each missing entry $A[u,q]$ we find the cluster to which u and q belong, say cluster C and D, respectively. Now, let us define $C' \subseteq C$ as the subset of the users in C who voted at least one component of D.

- If $C' = \emptyset$ we temporarily cannot compute entry $A[u,q]$. Further merge operations are required.
- If $C' \neq \emptyset$ we compute $A[u,q]$ as:

$$A[u,q] = \overline{v_u} + \frac{\sum_{c\in C'}\sum_{d\in D} min(1, v_{c,d})(v_{c,d} - \overline{v_c})}{\sum_{c\in C'}\sum_{d\in D} min(1, v_{c,d})} \tag{19}$$

where $\overline{v_u}$ is the average rating of user u and $v_{u,q}$ is a function such that:

$$v_{u,q} = \begin{cases} A[u,q] & \text{if } A[u,q] \text{ is not empty} \\ 0 & \text{otherwise} \end{cases}$$

Now, we mark as completed all the users whose row in the utility matrix has no missing entries. Hereafter, we define as uncompleted all the users who are not completed.

4.2.3 Condense. At this point of the execution, we iteratively merge user clusters until we get half as many clusters as there are users. To do this, at each iteration we compute the similarity between every possible couple of user clusters and we merge the most similar pair. We define the similarity between two clusters C_1, C_2 as the average similarity among their components.

$$\text{similarity}(C_1, C_2) = \frac{\sum_{x\in C_1}\sum_{y\in C_2} \text{sim}(x,y)}{|C_1 \times C_2|} \tag{20}$$

Similarity computation between cluster components is a crucial step of a collaborative filtering algorithm. The basic idea of similarity computation is co-rating. The similarity between user u and user v is computed using the items which have been rated by both users. Among many methods to compute similarity, we decide to use *Pearson correlation*, which is one of the most popular. Pearson correlation between user u and user v is computed as follows:

$$w_{u,v} = \frac{\sum_{i\in I}(r_{u,i} - \overline{r_u})(r_{v,i} - \overline{r_v})}{\sqrt{\sum_{i\in I}(r_{u,i} - \overline{r_u})^2}\sqrt{\sum_{i\in I}(r_{v,i} - \overline{r_v})^2}} \tag{21}$$

where I is the set of items rated by both user i and user j, $r_{u,i}$ is the rating of user u on item i and $\overline{r_j}$ is the average rating of user j.

We name the procedure for which the pairwise distance between each pair of clusters is computed and the most similar clusters are merged, "*condense*". The cost of the procedure is $O(N^3)$ such that N is the number of cluster components. Such computational complexity is too expensive for big datasets. In the case of large amount of data we apply some adjustments:

- in procedure condense we do not iterate over all the cluster pairs. Instead, we sample at random a subset Σ from the set of clusters and compute the similarity only for couples in $\{(\alpha,\beta) \in \Sigma \times \Sigma : \alpha \neq \beta\}$.
- similarly, in order to compute the similarity between two clusters, we do not consider all the components but only a random sample of limited size.
- moreover, we adjust the Pearson metric a little bit. Given two users u_1 and u_2, the Pearson similarity is calculated not on the entire collection of the queries voted by u_1 and u_2 in the past, but on a random sample of their voted queries.

Obviously, in these cases, the quality of the results increases considering larger samples. We have to trade-off among quality of the recommendations, time constraints and available computational resources. Fortunately, in most applications, it is not required to compute very accurate recommendations: it is sufficient to understand what the target user might like.

Once we obtain $\frac{|U|}{2}$ user clusters, such that U is the user set, we repeat exactly the same procedure in order to get half as many clusters as there are queries. This time, the similarity between query i and query j is computed by working on the users who have rated both of these queries. Pearson correlation between query i and query j is computed as follows:

$$w_{i,j} = \frac{\sum_{u\in U}(r_{u,i} - \overline{r_i})(r_{u,j} - \overline{r_j})}{\sqrt{\sum_{u\in U}(r_{u,i} - \overline{r_i})^2}\sqrt{\sum_{u\in U}(r_{u,j} - \overline{r_j})^2}} \tag{22}$$

At this stage, we can update the utility matrix so the columns represent clusters of queries, the rows represent clusters of users. We can upgrade the utility matrix entries with the procedure discussed in Section 4.2.2.

4.2.4 Complete the utility matrix. Finally, we keep merging user clusters and query clusters until all the users are not completed. The procedure is iterative. At each iteration:

(1) we compute average cluster quality of both user clusters and query clusters. The quality of a cluster is the average similarity among its components. A cluster has good quality if it contains components reciprocally similar. If user clusters are characterized by a higher quality than query clusters, we merge two user clusters as described in the second point of this list, otherwise we merge two query clusters as described in the third point of this list. To improve the outcomes of the algorithm we introduce a little heuristic in this step. With a probability of 0.33 the algorithm does not follow the cluster quality criterion. In other words, with 33% probability, even if the user clusters quality is better than the query clusters quality, the algorithm decides to merge two query clusters. Symmetrically, even if the query clusters quality is better than the user clusters quality, with 33% probability, the algorithm decides to merge two user clusters.
(2) if the algorithm decides to merge two user clusters we proceed as follows. We do not compare each pair of user clusters

since, as explained above, it is often too expensive. On the contrary, we randomly sample a user u which is not completed yet. Let Φ be the cluster which contains u. The algorithm merges cluster Φ with the most similar user cluster.

(3) if the algorithm decides to merge two query clusters we proceed as follows. We do not compare each pair of query clusters since, as explained above, it is often too expensive. On the contrary, we randomly sample a query q which is not completed yet. Let Π be the cluster which contains q. The algorithm merges cluster Π with the most similar query cluster.

(4) the utility matrix is revised as described in Section 4.2.2. At the end of each iteration we mark as completed all the users whose corresponding row in the utility matrix has no missing entries.

4.3 Dealing with new users

Up to now we explained how we compute recommendations for users who voted a lot of queries of the log (Section 4.1) and for users who voted few queries (Section 4.2). In this Section we describe how we overcome to the well-known cold start issue, which occurs in recommendation problems. The cold start problem occurs when the system is unable to predict a recommendation because it has insufficient data about users or items or both. This is the case of a query who has not been voted by any user yet, or of a new user, whose corresponding row in the utility matrix is empty. Following our solution proposal, we can already face the situation of a query without votes. Indeed, in content based recommendation, even if the given query has not been voted yet, we can look at its important features and compute its query profile. Additionally, in collaborative filtering recommendation, sooner or later the algorithm will agglomerate the query without votes in a cluster with other queries. Note that, since in our hybrid solution we execute content based recommendation before collaborative filtering recommendation, it is unlikely to deal with queries without votes in the collaborative filtering step.

On the other hand, we have not accounted yet the situation for which a user has not voted any query. For users whose row in the utility matrix is empty we cannot compute the profile and, since they did not provide any preference, we can neither adopt a collaborative filtering approach. In order to face this problem we proceed as follows. The procedure is summarized in the pseudocode of Algorithm 7.

(1) First of all, we sample some users at random. Hereafter we refer to this subset of users as $\mathcal{G}$.
(2) Then, we cluster together queries with the same definition, as described in subsection 4.2.1.
(3) We assign a vote to each query cluster C. To do this we sample a query q from the cluster randomly. Than we assign to the cluster C the average vote given by each user $u \in \mathcal{G}$ to query q.

$$\text{vote}(C) = \frac{\sum_{u \in \mathcal{G}} v_{u,q}}{|\mathcal{G}|} \quad (23)$$

Algorithm 5 Collaborative Filtering

```
input:
  set of users := $U$
  set of queries := $Q$
  utility matrix := $A$

for user $u \in U$ do
    u_completed[u] = False
    u.cluster ⟵ new Cluster()                ▷ cluster for user $u$
end for
        ▷ u_completed[u]==True IFF its row in $A$ is completed
for query $q \in Q$ do
    q_completed[q] = False
    q.cluster ⟵ new Cluster()                ▷ cluster for query $q$
end for
        ▷ q_completed[q]==True IFF its column in $A$ is completed

cluster queries such that queries with the same definition are in
the same cluster;
update utility matrix votes()                ▷ Algorithm 6

while $numUserClusters > \frac{|U|}{2}$
    cluster the two most similar user clusters according to the
average Pearson similarity among their components
end while
while $numQueryClusters > \frac{|Q|}{2}$
    cluster the two most similar query clusters according to
the average Pearson similarity among their components
end while

update utility matrix votes()                ▷ Algorithm 6

while $\exists u \in U$ : u_completed[u] == False do
    queryClusterQuality ⟵ avg quality of query clusters
    userClusterQuality ⟵ avg quality of user clusters
    if queryClusterQuality>userClusterQuality do
        let $q'$ be a query such that q_completed[q'] == False
        $D'$ ⟵ cluster to which $q'$ belongs
        $F$ ⟵ query cluster most similar to $D'$
        merge($D', F$)
    else if userClusterQuality>queryClusterQuality do
        let $u'$ be a user such that u_completed[u'] == False
        $C'$ ⟵ cluster to which $u'$ belongs
        $G$ ⟵ user cluster most similar to $C'$
        merge($C', G$)
    end if
    update utility matrix votes()            ▷ Algorithm 6
end while

output:
user_recommendation      ▷ the expected vote for each missing
entry $(u, q) \in A$
```

Algorithm 6 Update utility matrix votes

```
for user $u \in U$ : u_completed[u] == False:
    $C \leftarrow$ cluster to which $u$ belongs
    for query $q \in u$.unvotedQueries do
        $D \leftarrow$ cluster to which $q$ belongs
        $normAvgVote \leftarrow$ avg normalized vote assigned by users in $C$ to queries in $D$
        if $normAvgVote$ is calculable do
            $A[u,q] \leftarrow normAvgVote + u.avgVote$
        end if
    end for
    if $u$ is now completed do
        u_completed[u] $\leftarrow True$
    end if
end for
```

where q is a random query sampled from C and $v_{u,q}$ is the rank given by user u to query q.

(4) Finally, we fill each entry $[u, q]$ of the utility matrix with the vote of cluster C calculated with Formula 23. Actually, cluster C is the cluster which contains query q.

Algorithm 7 Cold start users

```
input:
  set of users := $U$
  set of queries := $Q$
  utility matrix := $A$
  cold start users := $\Gamma$

$\mathcal{G} \leftarrow U.randomSample(min(20, |U|))$
cluster queries such that queries with the same definition are in the same cluster;
for each query cluster $C$ do
    $c \leftarrow C.randomSample(1)$
    voteSum $\leftarrow 0$
    for user $u \in \mathcal{G}$ do:
        voteSum $\leftarrow$ voteSum $+ A[u,c]$
    end for
    vote$[C] \leftarrow \frac{\text{voteSum}}{|\mathcal{G}|}$
end for

for user $\gamma \in \Gamma$ do
    for each query cluster $C$ do
        for each query component $q \in C$ do
            $A[\gamma][q] \leftarrow$ vote$[C]$
        end for
    end for
end for
```

4.4 Evaluate algorithm performance

The purpose of this section is to explain how we evaluate the goodness of the utility matrix produced by our recommendation system algorithm. We measure the quality of our solution in terms of **accuracy** (subsection 4.4.1) and in terms of **responsiveness** (subsection 4.4.2).

4.4.1 Measure accuracy. Each dataset used in our experiments has an input utility matrix A and a ground truth utility matrix A'. This last is used to test the quality in terms of accuracy of our solution. In particular we rely on two main metrics: i) *mean error* (*ME*) and ii) *root mean squared error* (*RMSE*). Let R be the collection of user-query couples (u, q) for which the recommendation system predicted a vote. Actually, the set of couples in R correspond to the entries in A with missing values.

$$ME = \frac{\sum_{(u,q)\in R} |A'[u,q] - R[u,q]|}{|R|} \tag{24}$$

$$RMSE = \sqrt{\frac{\sum_{(u,q)\in R} (A'[u,q] - R[u,q])^2}{|R|}} \tag{25}$$

4.4.2 Measure responsiveness. In order to understand the performance of our solution in terms of execution time, we simply measure how much time it takes to fill all the missing entries of the input utility matrix. To do this we use function time() of the homonymous Python module. In particular, as presented in Section 5, we test algorithm responsiveness on datasets of increasingly large dimensions in order to understand how well our solution scales to larger amount of data.

4.5 Computing the utility of a query in general for all the users

Given the complete utility matrix A filled with all the missing values, in this section we propose a way to compute the utility of a query q_e in general for all the users. For the sake of clarity, the idea is described step by step.

First of all, we revise the columns of the utility matrix so that there are not two queries with exactly the same definition. The definition of a query is a set of attributes. For example, the definition of Q124 name="Stefano" AND age=10, is {name="Stefano",age=10}. Intuitively, it is not important to consider all the duplicates of a given query. Indeed, two queries characterized by the same set of attributes are a compact representation of the same set of tuples. As a result, their columns in the utility matrix will be almost identical.

At this point, we partition the set of queries into a reasonable amount of clusters. There are no reason to try to cluster into a small number of clusters immediately. Rather, a bottom-up hierarchical approach, where we leave many clusters unmerged is a good option. As an indication, we might leave half as many clusters as there are queries. During this operation, the similarity S between a couple of clusters C_1, C_2 is calculated from the average similarity s among their components:

$$S(C_1, C_2) = \frac{1}{|C_1 \times C_2|} \sum_{i\in C_1} \sum_{j\in C_2} s(i, j) \tag{26}$$

With this formulation we can iteratively merge the couple of clusters characterized by the higher similarity.
In order to compare the similarity between a couple of queries

q_1, q_2 we profitably encode each of them as a set of integers $T = \{i_1, \ldots, i_n\}$ such that each $i \in T$ is the unique identifier of a tuple which appear in the result set of the query. Once, the two queries have been summarized, we can estimate how close they are using *Jaccard similarity.* Let T_1, T_2 be the two numerical sets which encode query q_1 and q_2 respectively. The similarity between q_1 and q_2 is calculated as:

$$s(q_1, q_2) = \frac{|T_1 \cap T_2|}{|T_1 \cup T_2|} \tag{27}$$

In Figure 8 we propose a self-explanatory illustration of this latter idea.

Once the iterative procedure has built the proper number of query clusters, we identify a representative for each group. We preset a possible solution for this step with an example. Consider a cluster C which contains four queries $\{q_1, q_2, q_3, q_4\}$. The definition of each query is reported in Table 8. From these definitions, we build a *triangular matrix* M as the one illustrated in Figure 9. Each entry $M[q_i, q_j]$ of the matrix is filled with the result of the intersection over union operation between the two set of attributes of q_i and q_j. For example, entry $M[q_1, q_2] = \frac{1}{3}$ since $|\{attr1 = A, attr2 = B\} \cap \{attr1 = A, attr3 = C\}| = 1$ and $|\{attr1 = A, attr2 = B\} \cup \{attr1 = A, attr3 = C\}| = 3$. This is a proper similarity measure since, as we mentioned above, the definition of a query concisely describes its result set. At this point, we can estimate the importance of a cluster's query $\boldsymbol{imp}_{\textbf{cluster}}(q_i)$ as the summation of the similarities between q_i and the other components of the cluster. For example the importance of q_4 is calculated as follows:

$$\boldsymbol{imp}_{\textbf{cluster}}(q_4) = \frac{2}{3} + \frac{1}{4} + \frac{1}{2} = 1.42$$

The general formula is:

$$\boldsymbol{imp}_{\textbf{cluster}}(q_i) = \sum_{j=1}^{i-1} M[j][i] + \sum_{z=i+1}^{n} M[i][z] \tag{28}$$

The query with the highest importance according to this metric, is chosen as the representative of the cluster. In our example q_1 is the designated representative.

$$\boldsymbol{imp}_{\textbf{cluster}}(q_1) = \frac{1}{3} + \frac{2}{3} + \frac{2}{3} = 1.67$$

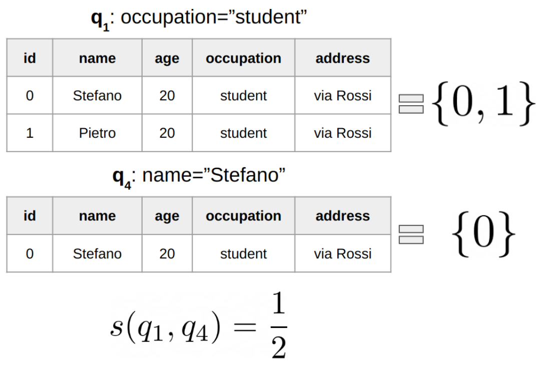
If we need to compute the utility of a query q_e we can proceed as follows. Let assume that currently, our system is composed by m query clusters $\{C_1, \ldots, C_m\}$. Each of them has a representative query $\{r_1, \ldots, r_m\}$ such that r_i is the designated representative of cluster C_i. We calculate the similarity between q_e and each cluster representative r_i with the expression suggested in Formula 27. At this stage there are two possible situations:

(1) There exists a representative r_{max} such that $s(q_e, r_{max}) > 0$ and $\nexists r_i \in \{r_1, \ldots, r_m\} : r_i \neq r_{max} \wedge s(q_e, r_i) > s(q_e, r_{max})$. Let us suppose that the cluster C_{max} of which r_{max} is representative is composed by k members and that our utility matrix A has n rows. We compute the general importance of q_e for all the users as:

$$\boldsymbol{imp}(q_e) = \frac{1}{k * n} \sum_{i=1}^{n} \sum_{j=1}^{k} (v_{i,j} - \overline{v_i}) \tag{29}$$

Where $v_{i,j}$ is the vote assigned by user i to query j and $\overline{v_i}$ is the average vote of user i according to the utilities. Generally, it is reasonable to subtract the average vote of a user since some individuals tend to assign higher or lower ranks than others. Essentially, according to the above definition, the importance of q_e is the normalized average vote assigned by the utility matrix users to the queries of cluster C_{max} which is the group more similar to the input query.

(2) It is possible that q_e is completely different with respect to all the other queries, i.e. $s(q_e, r_i) = 0 \; \forall r_i \in \{r_1, \ldots, r_m\}$. In this case we can claim that according to the given utility matrix the importance of q_e is 0. As an alternative, we can sample some users from the utility matrix and predict their vote to query q_e using our recommendation algorithm. Then, the importance of q_e is estimated as its average normalized rank assigned by the sampled set of user.



q_1: occupation="student"

id	name	age	occupation	address
0	Stefano	20	student	via Rossi
1	Pietro	20	student	via Rossi

$\{0, 1\}$

q_4: name="Stefano"

id	name	age	occupation	address
0	Stefano	20	student	via Rossi

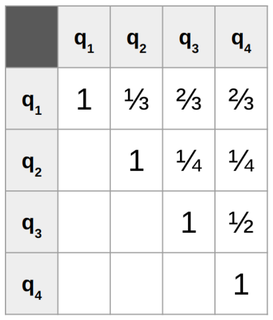
$\{0\}$

$$s(q_1, q_4) = \frac{1}{2}$$

Figure 8: PART B: similarity between two queries. Query results are obtained from the relational table Person in Table 1.

query id	query definition
q_1	attr1=A and attr2=B
q_2	attr1=A and attr3=C
q_3	attr1=A and attr2=B and attr3=D
q_4	attr1=A and attr2=B and attr4=E

Table 8: PART B: example of query definitions.



	q_1	q_2	q_3	q_4
q_1	1	⅓	⅔	⅔
q_2		1	¼	¼
q_3			1	½
q_4				1

Figure 9: PART B: Example of cluster triangular matrix M.

5 EXPERIMENTAL EVALUATION

In order to evaluate the performance of our solution we built three datasets:

- *UNIVERSITY*: this dataset is about a University.
  - In relational table Person (DB) four occupations are taken into consideration: "student", "professor", "researcher", "other". It is not very likely to find two people who live in the same place or with the same name. On the other hand it is highly probable to find two people with the same occupation. Most of the people in the relational table are students. Most of the students are young (age between 16 and 40). Most of the professors are medium-age (age between 41 and 60). Most of the researchers are young or medium-age. Most of the rest of the personnel is young or medium-age.
  - There are 16 users who did queries on the database in the past. Each of the user has their own preferences, but in general everyone do not like professor Debralee except user u_5 and user u_6.
  - Arisa and Royce are a famous researcher and a famous professor respectively.
  - User u_0 has never interacted with the database. Their row in the utility matrix is empty.
  - In Figure 11 we illustrate the amount of queries voted by each user.
  - In Figure 10 we concisely describe the preferences of each user.

  This dataset is small but user tastes are very outlined. Indeed, the data collection is intended to measure the *accuracy* of our recommendation system.
  - $|DB| = 100$
  - $|U| = 16$
  - $|Q| = 88$
- *VILLAGE*: This dataset is about a small village.
  - In relational table Person (DB) four occupations are taken into consideration: "student", "retiree", "farmer", "other". It is not very likely to find two people with the same name. On the other hand it is highly probable to find two people that live in the same place (since the village is small) or with the same occupation. Most of the people in the relation table are retiree. Most of the retiree are old (age between 61 and 100). Most of the farmer are medium-age (age between 41 and 60). Most of the students are young or medium-age. Most of the rest of the people who lives in the village is medium-age.
  - There are 11 users who did queries on the database in the past. They can be divided into two main categories:
    - ∗ one that likes the retirees, but hates the farmers (users $u_0, \dots u_5$)
    - ∗ one that likes the farmers, but hates the retirees (users $u_6, \dots u_{10}$)

    These two categories contains some differences, like some users likes also students, or they hate them, or are interested in some people in particular. The quirk of these users is that only one person has voted a lot. All the others users have voted only for few queries.
  - User u_5 has never interacted with the database. Their row in the utility matrix is empty.
  - Miliam and Kerby are famous students. Leshay is a famous retiree. Graceanne is a farmer who is hated by more or less half the population. Rosalva and Tara are farmers.
  - In Figure 13 we illustrate the amount of queries voted by each user.
  - In Figure 12 we concisely describe the preferences of each user.

  This dataset is small but user tastes are very outlined. Indeed, the data collection is intended to measure the *accuracy* of our recommendation system.
  - $|DB| = 100$
  - $|U| = 11$
  - $|Q| = 67$
- *BIG*: The purpose of this dataset is to test how well our recommendation system scales considering larger amount of data. The dataset can be arbitrarily large. In the following we briefly explain how we generated this dataset. Once the relational database is generated randomly, we read through the tuples which populate the table Person in order to find the most frequent values which appear in the data collection. Then, we generate queries about these frequent values. These lasts ideally correspond to interesting topics of the data collection. Once all the queries are created, we partition the user set into clusters:
  - different users are interested in different topics
  - different users do not like different topics
  - some users voted a lot of queries
  - some users voted a medium-size amount of queries
  - some users voted few queries
  - some users tend to assign high votes to query result sets
  - some users tend to assign medium-high votes to query result sets
  - some users tend to assign low votes to query result sets

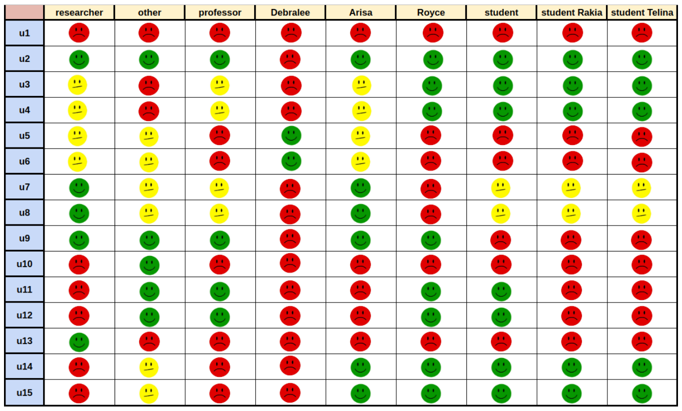
  Given this information about the peculiarities of each user, we fill the utility matrix assigning a vote for each user-query pair.

We organize the rest of this section as follows:

- in Subsection 5.1 we present our experiments on dataset UNIVERSITY to evaluate the performance in terms of *accuracy* of our solution.
- In Subsection 5.2 we present the results of the experiments which have been performed to study the *accuracy* of our algorithm on dataset VILLAGE.
- In Subsection 5.3 and 5.4 we report the results in terms of *accuracy* and *responsiveness* achieved on dataset BIG by our solution.
- Finally in Subsection 5.5 we compare the performance of our solution with a baseline provided by another group.

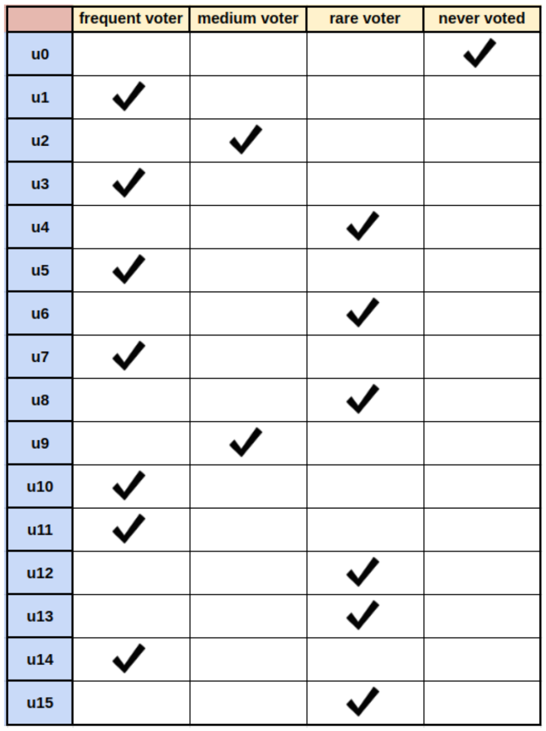
5.1 Dataset UNIVERSITY accuracy test

5.1.1 Person clustering. Observing the plot of *elbow evaluation* in Figure 14 we notice that it is reasonable to partition tuples in table Person into five categories. In Figure 15 and Figure 16 we understand that k-means clustering outlines two very distinct categories:



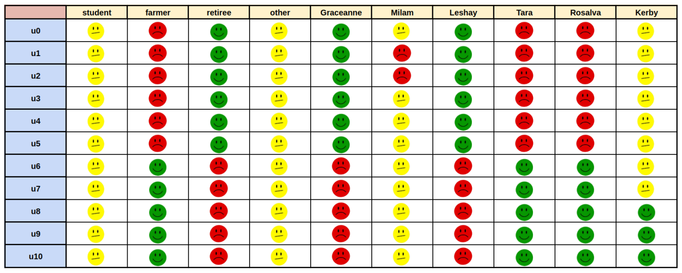
	researcher	other	professor	Debralee	Arisa	Royce	student	student Rakia	student Telina
u1	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴
u2	🟢	🟢	🟢	🔴	🟢	🟢	🟢	🟢	🟢
u3	🟡	🔴	🟡	🔴	🟡	🟢	🟢	🟢	🟢
u4	🟡	🔴	🟡	🔴	🟡	🟢	🟢	🟢	🟢
u5	🟡	🟡	🔴	🟢	🟡	🔴	🔴	🔴	🔴
u6	🟡	🟡	🔴	🟢	🟡	🔴	🔴	🔴	🔴
u7	🟢	🟡	🟡	🔴	🟢	🔴	🟡	🟡	🟡
u8	🟢	🟡	🟡	🔴	🟢	🔴	🟡	🟡	🟡
u9	🟢	🟢	🟢	🔴	🟢	🟢	🔴	🔴	🔴
u10	🔴	🟢	🔴	🔴	🔴	🔴	🔴	🔴	🔴
u11	🔴	🟢	🟢	🔴	🔴	🟢	🟢	🔴	🔴
u12	🔴	🟢	🟢	🔴	🔴	🟢	🟢	🔴	🔴
u13	🟢	🔴	🔴	🔴	🔴	🔴	🔴	🔴	🔴
u14	🔴	🟡	🔴	🔴	🟢	🟢	🟢	🟢	🟢
u15	🔴	🟡	🔴	🔴	🟢	🟢	🟢	🟢	🟢

Figure 10: Dataset UNIVERSITY: user's preferences.



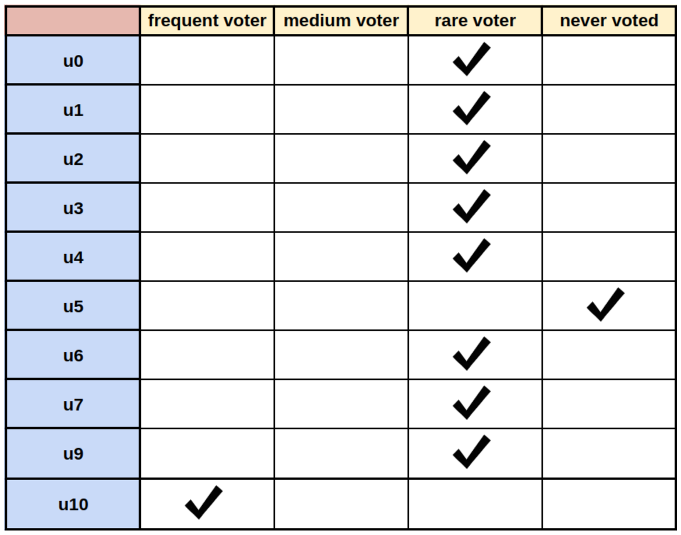
	frequent voter	medium voter	rare voter	never voted
u0				✓
u1	✓			
u2		✓		
u3	✓			
u4			✓	
u5	✓			
u6			✓	
u7	✓			
u8			✓	
u9		✓		
u10	✓			
u11	✓			
u12			✓	
u13			✓	
u14	✓			
u15			✓	

Figure 11: Dataset UNIVERSITY: vote frequency of users.



	student	farmer	retiree	other	Graceanne	Milam	Leshay	Tara	Rosalva	Kerby
u0	🟡	🔴	🟢	🟡	🟢	🟡	🟢	🔴	🔴	🟡
u1	🟡	🔴	🟢	🟡	🟢	🔴	🟢	🔴	🔴	🟡
u2	🟡	🔴	🟢	🟡	🟢	🔴	🟢	🔴	🔴	🟡
u3	🟡	🔴	🟢	🟡	🟢	🟡	🟢	🔴	🔴	🟡
u4	🟡	🔴	🟢	🟡	🟢	🟡	🟢	🔴	🔴	🟡
u5	🟡	🔴	🟢	🟡	🟢	🟡	🟢	🔴	🔴	🟡
u6	🟡	🟢	🔴	🟡	🔴	🟡	🔴	🟢	🟢	🟡
u7	🟡	🟢	🔴	🟡	🔴	🟡	🔴	🟢	🟢	🟡
u8	🟡	🟢	🔴	🟡	🔴	🟡	🔴	🟢	🟢	🟢
u9	🟡	🟢	🔴	🟡	🔴	🟡	🔴	🟢	🟢	🟢
u10	🟡	🟢	🔴	🟡	🔴	🟡	🔴	🟢	🟢	🟢

Figure 12: Dataset VILLAGE: user's preferences.



	frequent voter	medium voter	rare voter	never voted
u0			✓	
u1			✓	
u2			✓	
u3			✓	
u4			✓	
u5				✓
u6			✓	
u7			✓	
u9			✓	
u10	✓			

Figure 13: Dataset VILLAGE: vote frequency of users.

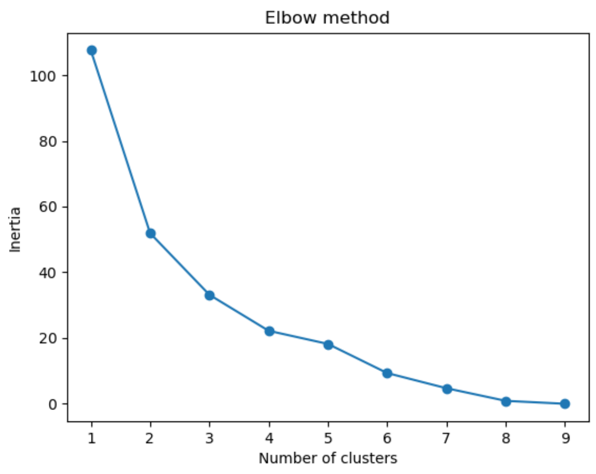


Figure 14: Dataset UNIVERSITY: elbow evaluation.

young students (orange cluster) and researchers (red cluster). The other categories are more heterogeneous.

5.1.2 *Fully content-based recommendation.*

If we execute our content-based recommendation algorithm on all the users, we notice that the system tends to achieve high accuracy on frequent users and poor accuracy on users who voted few queries. In particular, user u_0 is characterized by very bad results. This observations are illustrated in Figure 17 where we plot the average vote estimation error for each user. Actually on frequent voters we achieve lower values of mean error (*ME*) and root mean squared error (*RMSE*) (Table 9). Indeed, if we plot for each user the system prediction together with the ground truth, it is evident that the predicted curve of frequent voters (Figures 18, 19, 20, 21) follows the ground truth behaviour significantly more accurately than the predicted curve of

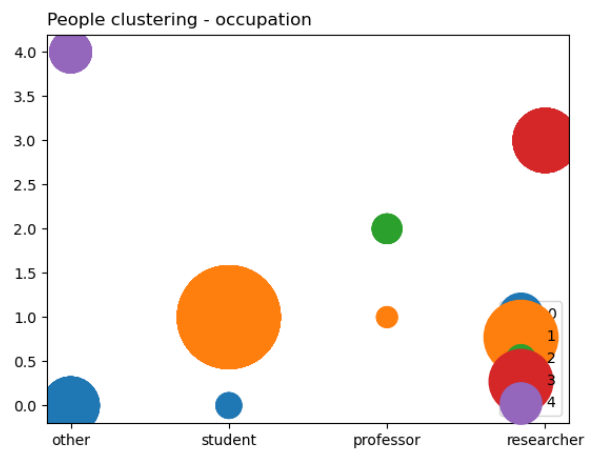


Figure 15: Dataset UNIVERSITY: k-means clustering.

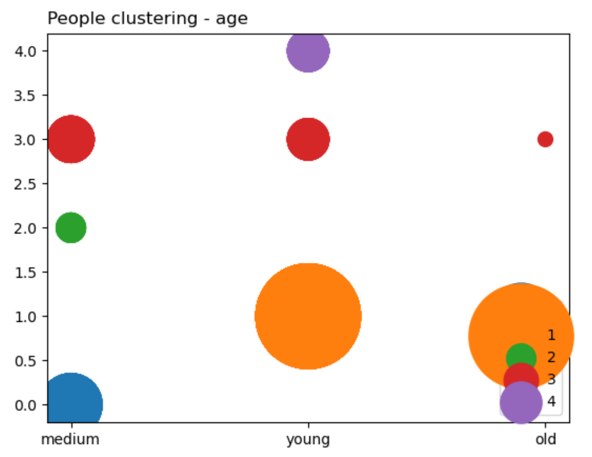


Figure 16: Dataset UNIVERSITY: k-means clustering.

rare voters (Figures 22, 23). The reason behind these outcomes is associated with the fact that our content-based recommender system has troubles in outlining the profiles of users with few votes in the utility matrix. Nevertheless, there are also users like u_{13} who voted few interrogations but, since they are characterized by very simple tastes, the system is still able to perform good recommendations for them. Looking carefully at the plot in Figure 17 we understand that the most tricky recommendations are the ones with conflicting aspects. For instance, this is the case of user u_{11} and u_{12} who like all the students except Telina and Rakia and are interested in all the professors except Debralee.

5.1.3 Hybrid recommendation system. From Figure 24 we notice that, if we combine our content-based recommendation approach with the collaborative filtering method we improve the accuracy

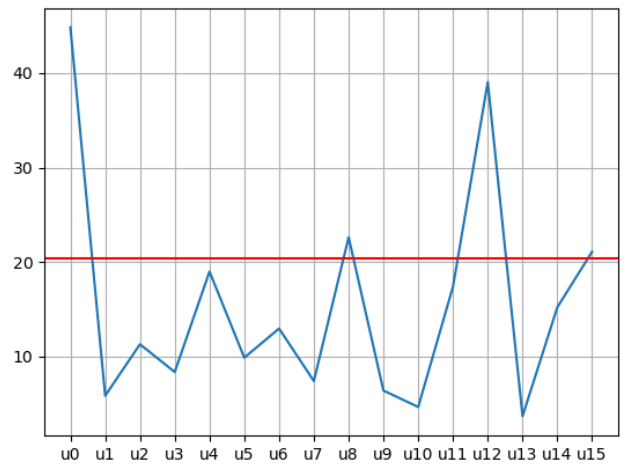


Figure 17: Dataset UNIVERSITY: user average vote estimation error. The red line is the average vote estimation error among all the users.

Dataset UNIVERSITY accuracy		
	frequent voters only	**entire user set**
ME	**10.31**	20.36
RMSE	**13.33**	27.58

Table 9: On frequent voters we achieve better accuracy performance than on rare voters.

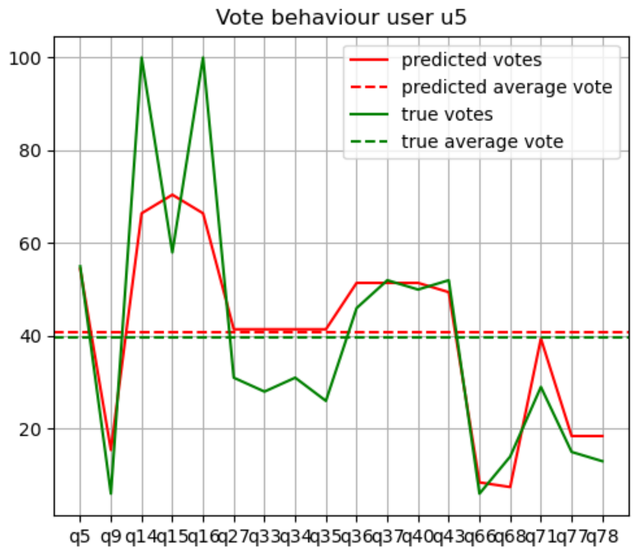


Figure 18: Dataset UNIVERSITY: content-based recommendation accuracy on user u_5.

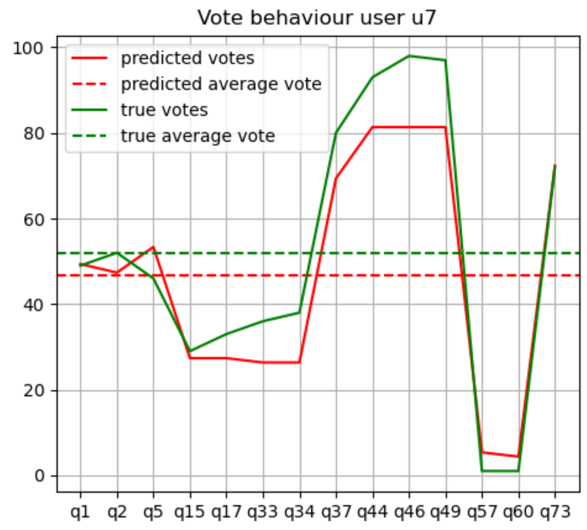


Figure 19: Dataset UNIVERSITY: content-based recommendation accuracy on user u_7.

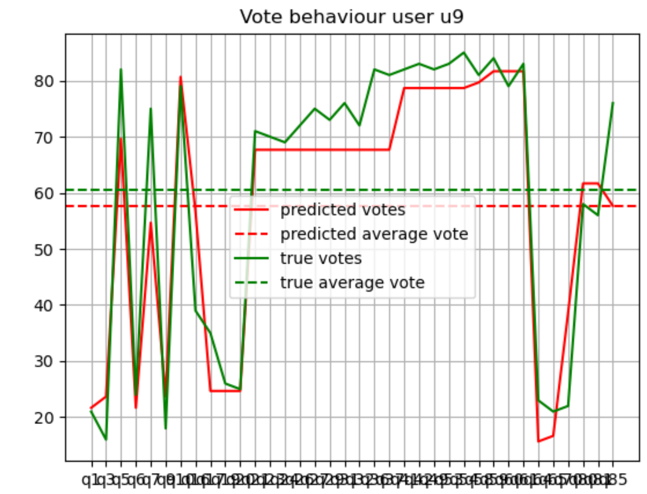


Figure 20: Dataset UNIVERSITY: content-based recommendation accuracy on user u_9.

on some users and worsen a bit the accuracy on some others. Indeed, the collaborative filtering algorithm does not always cluster users perfectly (Table 10). For instance, users u_{11} and u_{12} are correctly placed in the same cluster. Indeed, according to Table 10 they manifest similar preferences. However, their recommendations are worsen because also u_{10} has been included in the same cluster. In addition, we achieve bad performance on user u_4 who has not been placed in the cluster of user u_3 who is their most similar colleague according to Table 10. On the other hand, most of the other users

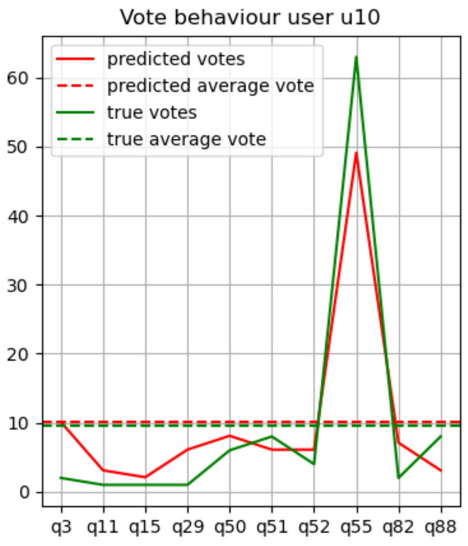


Figure 21: Dataset UNIVERSITY: content-based recommendation accuracy on user u_{10}.

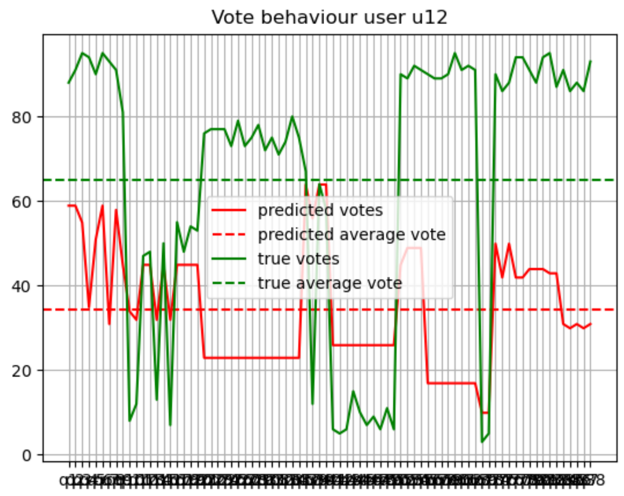


Figure 22: Dataset UNIVERSITY: content-based recommendation accuracy on user u_{12}.

have been partitioned properly. For instance, we can compare the quality of the recommendations for user u_{15} in Figure 25, with respect to the results achieved with the fully content-based recommendation approach of Figure 23. Evidently, now the curve of predicted recommendations follows better the ground truth curve. Overall, both the *RE* and the *RMSE* are improved as described in Table 11.

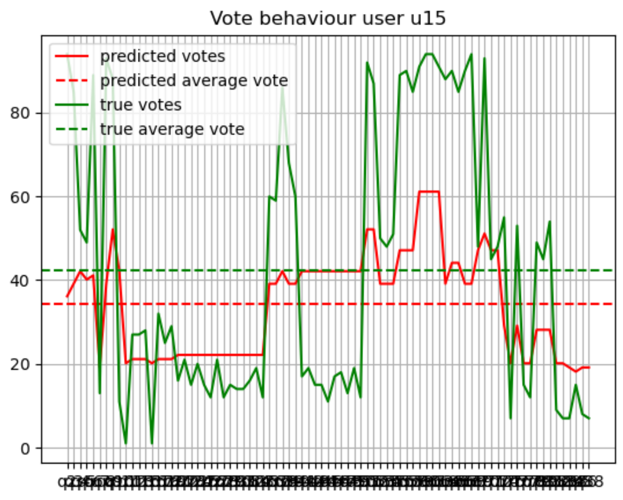


Figure 23: Dataset UNIVERSITY: content-based recommendation accuracy on user u_{15}.

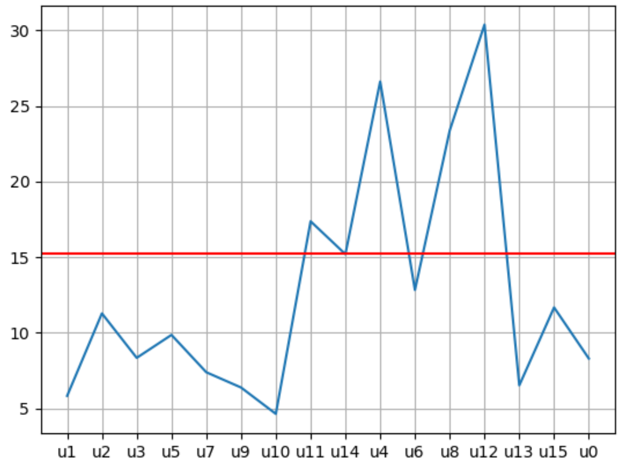


Figure 24: Dataset UNIVERSITY: user average vote estimation error obtained executing our hybrid recommendation system. The red line is the average vote estimation error among all the users.

5.2 Dataset VILLAGE accuracy test

5.2.1 Person clustering. This time, looking at Figure 26 we notice that k-means clustering with $k = 5$ outlines four very distinct categories: students (orange cluster), retirees (green cluster), farmers (red cluster), people whose occupation is other (purple and blue cluster). In particular, as we can see in Figure 27 these lasts are further partitioned into two groups: medium-age (blue cluster) and young+old others (purple cluster).

Dataset UNIVERSITY collaborative filtering clusters	
cluster ID	**componets**
1	u_1
2	u_9
3	u_3, u_{14}, u_{15}
4	u_2, u_8, u_7, u_{13}
5	u_{10}, u_{11}, u_{12}
6	u_4, u_5, u_6

Table 10: The user clusters built by our collaborative filtering algorithm executed on dataset UNIVERSITY.

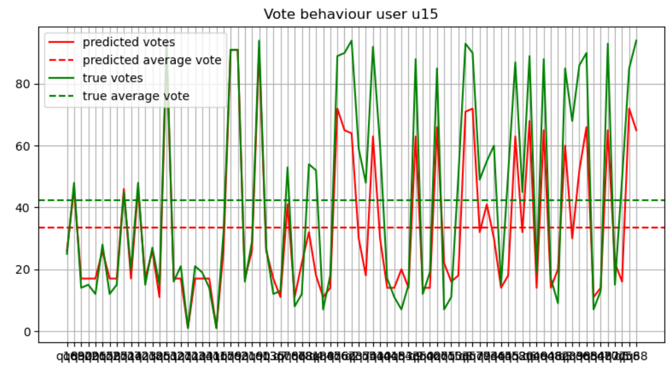


Figure 25: Dataset UNIVERSITY: hybrid recommendation accuracy on user u_{15}.

Dataset UNIVERSITY *ME* and *RMSE*		
	ME	**RMSE**
fully content-based	20.36	27.58
fully collaborative filtering	16.18	23.19
hybrid	**15.26**	**21.65**

Table 11: *ME* and *RMSE* achieved by our fully content-based, fully collaborative filtering and hybrid solutions on dataset UNIVERSITY.

5.2.2 Fully collaborative filtering recommendation. In the context of this dataset, the fully collaborative filtering approach achieves better accuracy than the fully content-based approach. Indeed, if we compare the plot in Figure 28 with the plot in Figure 29, we observe that the content-based recommender is characterized by a higher average user prediction error. Indeed, most of the users of dataset VILLAGE voted few queries. As a consequence, it is challenging for the content-based algorithm to outline user profiles. Actually, content-based algorithm performs well on user u_{10} who is the only frequent voter of the data collection. We can further appreciate how much better the fully collaborative filtering approach behave with respect to its content-based counterpart, looking at Figures 30 and 31. It is evident that, the red predicted curve follows considerably better the green ground truth behaviour. Furthermore, in Table 12 we annotate the clusters of users at each iteration of our

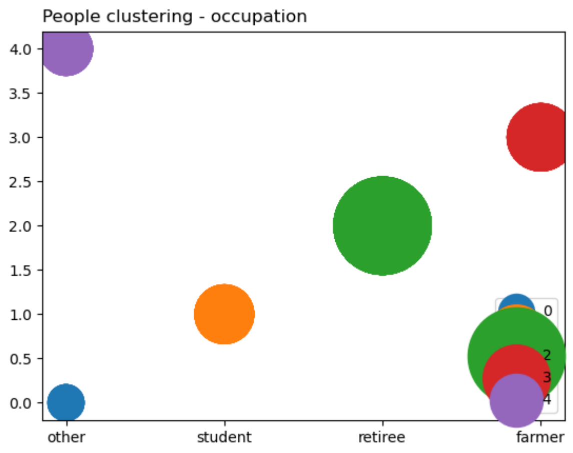


Figure 26: Dataset VILLAGE: k-means clustering.

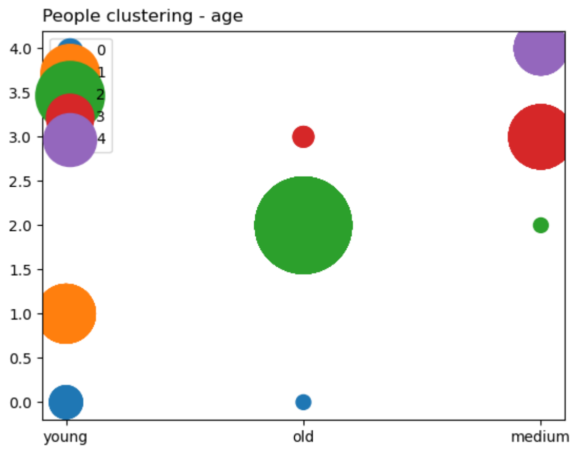


Figure 27: Dataset VILLAGE: k-means clustering.

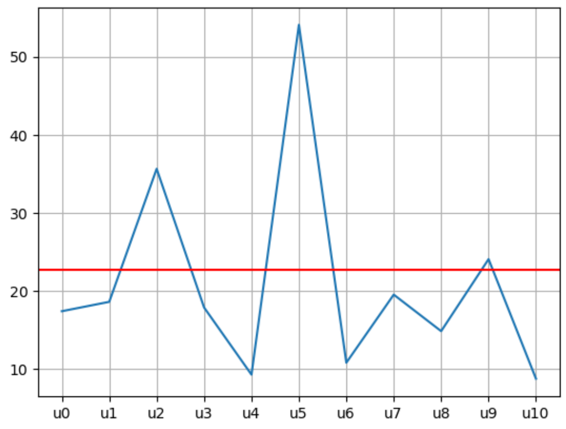


Figure 28: Dataset VILLAGE: user average vote estimation error obtained executing our content-based recommendation system. The red line is the average vote estimation error among all the users.

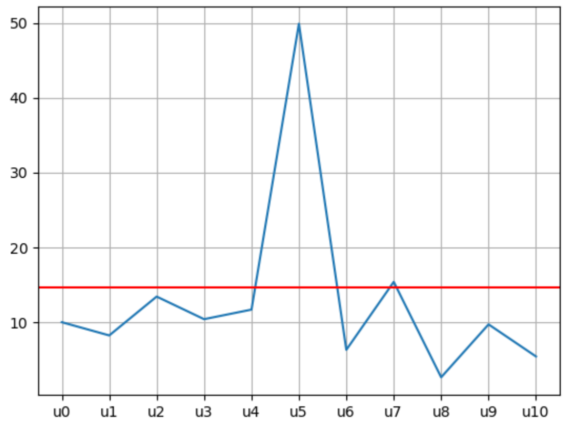


Figure 29: Dataset VILLAGE: user average vote estimation error obtained executing our collaborative filtering recommendation system. The red line is the average vote estimation error among all the users.

collaborative filtering execution. Successfully, the voters have been clustered as expected. In fact, as explained at the beginning of this section, VILLLAGE dataset is characterized by two categories of users: $\{u_0, u_1, u_2, u_3, u_4\}$ and $\{u_6, u_7, u_8, u_9, u_{10}\}$.

5.2.3 Hybrid recommendation system. Finally, if we execute our hybrid solution on dataset VILLAGE, we succeed in combining the advantages of both the methods. Indeed, as reported in Table 13, we notice that the hybrid algorithm outperforms its competitors in terms of mean error and root mean squared error.

5.3 Dataset BIG accuracy test

As reported in Table 14, our solution achieves good accuracy performance also on significantly larger datasets. In particular, the *ME* and *RMSE* values in Table 14 have been obtained executing our algorithms on an instance of dataset BIG with:

- 1000 tuples in relational table Person
- 100 users
- 500 queries

On this dataset, the fully content-based execution gets more accurate outcomes. We have identified the following motivations behind this result. Even if rare voters ranked a small percentage of the queries in the log, the amount of voted queries is still sufficiently large to outline reasonably accurate user profiles since the total size of the query log is large. Moreover, the preferences of each

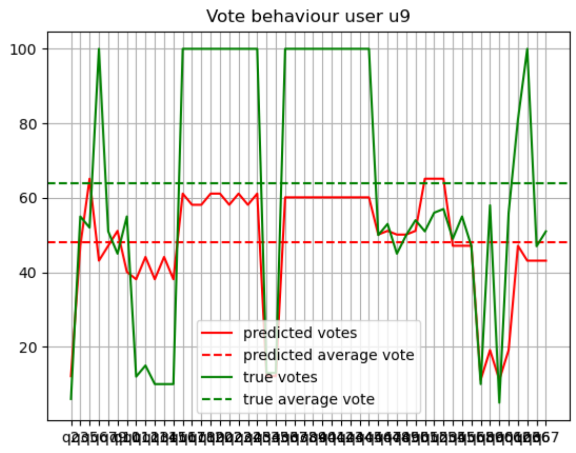


Figure 30: Dataset VILLAGE: content-based recommendation accuracy on user u_9.

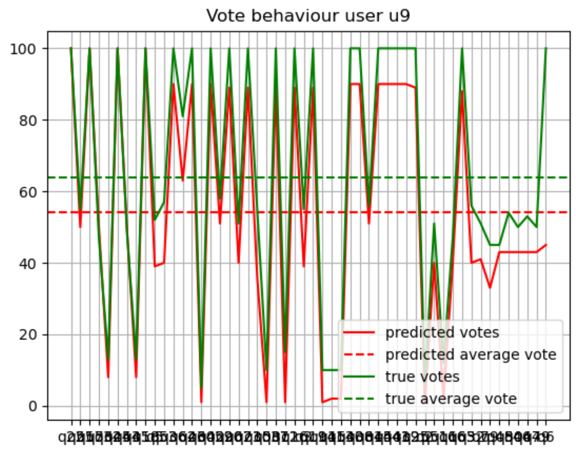


Figure 31: Dataset VILLAGE: collaborative filtering recommendation accuracy on user u_9.

user in this dataset are not sophisticated as the ones introduced in UNIVERSITY and VILLAGE. As a consequence, our content-based algorithm is able to identify users' peculiarities pretty easily.

5.4 Dataset BIG responsiveness test

For the purpose of understanding how well our solution behaves with larger datasets, we measured the execution time of our algorithm on increasingly bigger data collections. To accomplish this, we generated several instances of dataset BIG. In Figure 32 we plot the execution time with respect to the size of the dataset. We ran the tests on a HP Pavilion laptop, with 16 GB of RAM memory and Intel i7-1165G7 2.80GHz CPU. Unfortunately, our solution

Dataset VILLAGE collaborative filtering clusters	
FIRST iteration	
cluster ID	**componets**
1	u_0, u_1, u_2
2	u_6, u_8, u_9, u_{10}
3	u_3, u_4
4	u_5, u_7
SECOND iteration	
5	u_0, u_1, u_2
6	u_3, u_4
7	$u_5, u_6, u_7, u_8, u_9, u_{10}$
THIRD iteration	
8	$u_5, u_6, u_7, u_8, u_9, u_{10}$
9	u_0, u_1, u_2, u_3, u_4

Table 12: The user clusters built by our collaborative filtering algorithm executed on dataset VILLAGE.

Dataset VILLAGE *ME* and *RMSE*		
	ME	**RMSE**
fully content-based	22.81	33.01
fully collaborative filtering	14.71	25.44
hybrid	**12.19**	**18.60**

Table 13: ***ME*** **and** ***RMSE*** **achieved by our fully content-based, fully collaborative filtering and hybrid solutions on dataset VILLAGE.**

Dataset BIG *ME* and *RMSE*		
	ME	**RMSE**
fully content-based	**6.71**	**9.09**
fully collaborative filtering	19.00	25.45
hybrid	14.80	21.05

Table 14: ***ME*** **and** ***RMSE*** **achieved by our fully content-based, fully collaborative filtering and hybrid solutions on dataset BIG ($|DB| = 1000, |U| = 100, |Q| = 500$).**

does not scale very well. Indeed, the curve exhibits an exponential behaviour. We are confident that, our results can be improved considering smaller random samples of data in the most expensive steps of our implementation. For example, we noticed considerable responsiveness improvements considering smaller samples of clusters in procedure "*condense*" (Section 4.2.3). Clearly, if we reduce the size of the samples we achieve better responsiveness, but we inevitably worse the accuracy. Fortunately, in most applications, it is not required to compute very accurate recommendations: it is sufficient to understand what the target user might like.

5.5 Baseline comparison

The purpose of this section is to compare the performance of our recommendation system (we will call it **OS** in this section) with respect to another baseline algorithm written by another group (for simplicity we will call it **OM**). This comparison is divided into four different points:

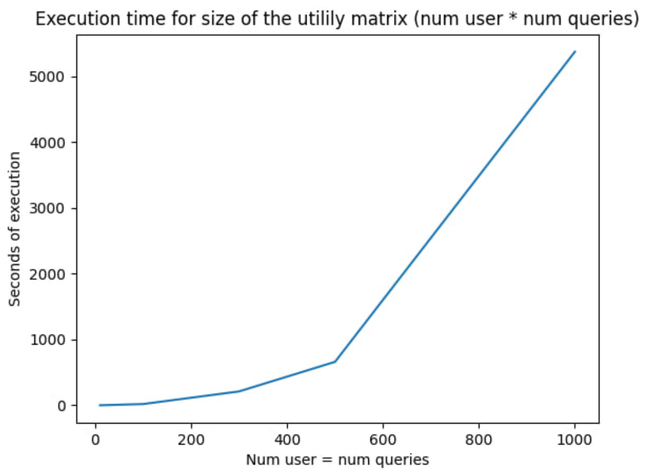


Figure 32: Dataset BIG: responsiveness test on datasets of increasingly larger dimensions. Hardware specifications: HP Pavilion laptop, with 16 GB of RAM memory and Intel i7-1165G7 2.80GHz CPU.

	Our solution (OS)	Baseline algorithm (OM)
Execution Time	101.9 s	**2.3 s**

Table 15: Execution time comparison.

- Execution time on the same dataset
- Accuracy of the solution
- *ME* and *RMSE* comparison
- How the two algorithms deal with the cold start problem

5.5.1 Time performance on the same dataset. Firstly, we have compared the execution time performance of OS and OM considering the same dataset. In particular, we took into account our dataset BIG in order to measure time performance on a large amount of data. This instance of the dataset is composed by 100 users and 500 queries. In Table 15 we report the execution time of the main core of the two algorithms. We notice that, on a large amount of data, our solution performs in a much worsen way w.r.t. the baseline method OM. In fact, we can see how OS is 50 times slower than OM in predicting the votes of the users.

5.5.2 Accuracy of the solution. In this subsection we compare the accuracy of the two methods analysing the results obtained executing the two algorithms on dataset UNIVERSITY. As we can observe in Figure 33, the query average vote estimation error that we achieve with OS is lower than the one predicted by OM. Furthermore, from the graphical representation in Figure 34, we notice that our user average vote estimation error outperforms the one obtained by OM.

5.5.3 ME and RMSE Comparison. In Figure 35 and 36, we report the mean error (*ME*) and the root mean squared error *RMSE* obtained by the two algorithms on our datasets. We observe that on dataset UNIVERSITY and VILLAGE our solution is characterized by smaller

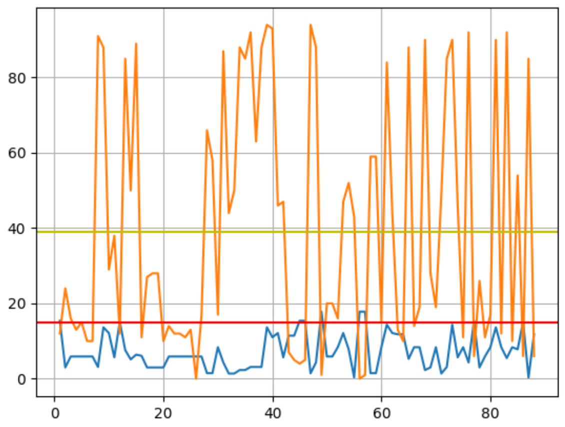


Figure 33: Dataset UNIVERSITY: query average vote estimation error for OS (blue) and OM (orange). The red line is the average vote estimation error among all the queries while the yellow one is the average vote estimation error among all the queries for OM. Values on the x axis correspond to the query identifiers.

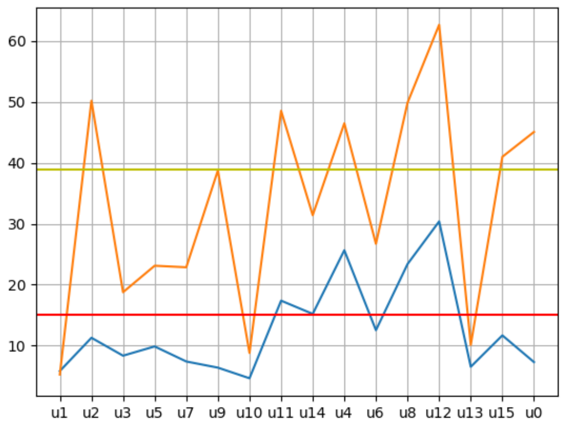


Figure 34: Dataset UNIVERSITY: user average vote estimation error for OS (blue) and OM (orange). The red line is the average vote estimation error among all the users while the yellow one is the average vote estimation error among all the users for OM. Values on the x axis correspond to the user identifiers.

ME and *RMSE* values. On the contrary, algorithm OM achieves better outcomes in the case of dataset BIG. As an exception, as we can see in Figure 36, our fully content-based approach obtains better performance on this dataset than OM. In general, we can conclude that on large datasets, the algorithm of the other group

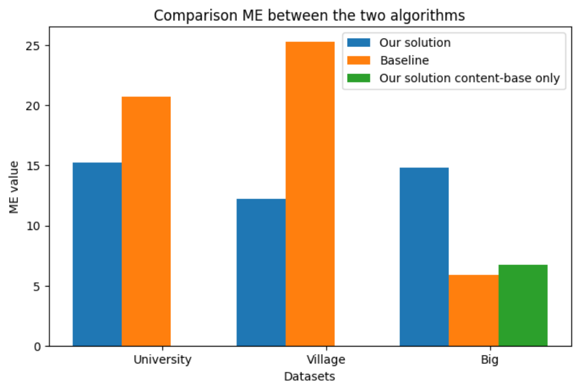


Figure 35: ***ME*** **of our algorithm (OS) and** ***ME*** **obtained by the baseline solution. We also add the** ***ME*** **obtained by our fully content-based algorithm on dataset BIG since it achieves good performance.**

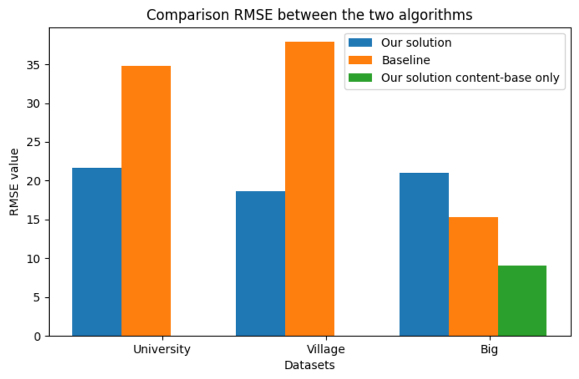


Figure 36: ***RMSE*** **of our solution (OS) and** ***RMSE*** **of the baseline (OM). We also add the** ***RMSE*** **obtained by our fully content-based algorithm on dataset BIG since it achieves good performance.**

(OM) is more accurate, but we perform much better on smaller data collections.

5.5.4 How the two algorithms deal with the cold start problem. Finally, we noticed that OM, when dealing with users that are new in the system, i.e. they did not vote anything yet, does not work. In fact it will not be able to compute the profile of the user, throwing an exception. On the other hand our solution works completely fine with new users, being able to suggest some preferences.

6 CONCLUSION

In this report, we have presented a memory-based hybrid solution to the problem of query recommendation. Our aim has been to combine the widely used content-based and collaborative filtering methods in order to take advantage of their strengths and limit their cons. Moreover, we developed an ad-hoc solution to handle the cold start problem related to users whose row in the utility matrix is empty. Overall, the execution is divided into three steps. First, content-based recommendations for users who voted a lot of queries in the past are computed. To do this, we identify the most important features of users and queries in order to outline their vector profiles. With profile vectors for both users and items, we can estimate the degree to which a user would prefer a query by computing the cosine distance between the user’s and item’s vectors.
Then, we execute the collaborative filtering recommender. The key idea of this step is to cluster both users and queries according to Pearson similarity among their rows and columns in the utility matrix. Clustering is performed following a bottom-up hierarchical approach. The utility matrix is revised so that the rows represent cluster of users and the columns represent cluster of queries. Given a user u in cluster U and query q in cluster Q, the entry $[u, q]$ is the average rating that users in U gave to the members of Q (Formula 19). This technique is particularly helpful in the case of sparse utility matrix.
Finally, we estimate utility matrix values for those users who have never interacted with the database. To accomplish this, we group together queries with the same definition; we randomly sample a set of users S for which the system has already completed the corresponding utilities; we fill each missing entry of the matrix with the average marks assigned by users in S to the target queries. In addition, given the output utility matrix we proposed a way to compute the utility of a query in general for all the users

We evaluate the performance of our solution on three datasets. Two of them are small but really refined. Due to their restricted dimensions, these data collections are characterized by high explainability. On the other hand, the third dataset exhibits simpler correlations but it can be generated with arbitrarily large relational table, set of users and set of queries. On all the datasets we achieve very good *accuracy*, measured by means of *RMSE* and *ME*. On the contrary, measuring the execution time on progressively larger data collections, we experienced bad *responsiveness* performance. Actually, the proposed algorithm does not scale well. Finally, in Section 5.5 we compare our performance with a baseline solution developed by another group.

We believe that, our proposed solution can open new interesting research directions. In particular, it is crucial to study how to refine our solution in order to provide recommendations in reasonable amount of time in the context of large-scale data. In this regard, a possible approach is to try exploiting parallelism by converting our algorithm into a distributed application, for example in the perspective of a *MapReduce* framework.