Query recommendation system

Data Mining Project 2022/2023

Erik Nielsen
mat.238755
University of Trento
38123 Povo TN, Italy
erik.nielsen@studenti.unitn.it

Fabrizio Sandri
mat.238825
University of Trento
38123 Povo TN, Italy
fabrizio.sandri@studenti.unitn.it

ABSTRACT

Since the birth of computing, there's always been discussion on how an information system can be used to recommend a product to a customer based on his interests. From the first practical application for a system called Grundy [11], a computer-based library that recommended novels to the users based on their interests, recommendation systems have gained wider attention thanks to the widespread diffusion of internet. Nowadays, these systems are fundamental to the user experience inside web services and social networks. In many cases, recommendation systems are the reason why one service is preferred over another.

This paper explores different strategies for integrating several data mining techniques to build a query-based recommendation system, starting with a baseline solution. After introduction to the techniques, we give a formal description of the problem and discuss the possible strategies to solve it. In the final part we examine the quality of the results with particular emphasis on the differences between the baseline solution and the final one. Our results demonstrate that our technique can produce useful query recommendations, demonstrating that it has the potential to be applied in real-world scenarios.

1 INTRODUCTION

Data became omnipresent in the last decade, and with it came the necessity to store massive volumes of data in an effective manner. Databases became the standard method for storing logically modeled collections of information in an efficient manner. Since the 1970s, most database management systems have been designed around the relational model[3] where the most fundamental elements that characterize it are relations, commonly referred to as tables. A relation is a collection of tuples, or table rows, each of which shares a set of characteristics, or table columns. The most typical method of retrieving data from a DBMS is to send a query, or a structured request for a set of data.

Let's suppose that we have access to a large number of queries that each user submits to a database management system (DBMS), each of which is associated with a rating that indicates how satisfied the user is with the query's outcome. In particular, let's assume that our DBMS is made up of a single relation that allows users to submit multiple queries to retrieve data. The question that remains is whether we can exploit all of this data to suggest queries to users based on previous interests.

Making recommendations is essential in many fields, starting with e-commerce websites where these systems attempt to suggest the best products that match the user's interest in order to improve sales. This methodology is used in e-commerce as well as libraries, as in the case of the aforementioned Grundy system. Another example is the case of streaming services that attempt to match the user with the movie that best suits their preferences.

These systems are usually based on the so-called *Utility Matrix*, which represents all the data about how each user preferences a specific item of the service. The matrix itself has some blank spaces, as usually the users don't have recorded data about each item of the matrix. The goal of the previously mentioned Recommendation Systems is to provide some algorithms able to predict the values of the blank spaces inside the matrix.

There is an infinite amount of ways to create a recommendation system, starting with the two fundamental approaches: content based or collaborative filtering. These are not the only ones; in fact, clustering is another option that can be taken into account while looking for groups of commonalities that can suggest one thing to another member of the same group. Each of these approaches must cope with the problem of dealing with huge amount of data and finding the best recommendation in the shortest amount of time; in most of the cases, it shouldn't take more than a few seconds to find a good recommendation, as is the case with e-commerce websites where the customer can see recommendations right away. In general, making recommendations based on massive volumes of data is still a challenging research problem.

As recommendation systems are mainly based on already existing data, the presence of some sort of database is necessary for the workflow of the algorithm. These databases are usually tables organized in rows and columns, built with the purpose to have an easy architecture and more understandable data. In this paper, we show how locality sensitive hashing can be exploited to improve similar items search for finding queries in a quick way to provide recommendation to a user that is consulting a needed database.

2 RELATED WORK

The work that has been conducted so far in the field of recommendation systems has been focused on finding ways to exploit similarities in existing data to make recommendations. This prior knowledge can take on a variety of shapes; one well-known form is provided in terms of a utility matrix. It can also be obtained from other sources of information that draw on patterns that exploits similarities, such as user behaviours [5].

Depending on the kind of prior knowledge we have regarding the problem that is being attempted to solve, there are primarily two categories in which recommendation systems can be categorized; these methods are *content-based filtering* and *collaborative-filtering*. In addition to these two methods in the last few years emerged the necessity to combine the benefits of the two aforementioned methods into *hybrid recommendation systems*.

Content-based methods use a combination of the features associated with each product and the ratings given by each user to provide suggestions. This method requires the construction of user profiles that outline each user's preferences as well as item profiles that highlight a product's key features.

The collaborative-filtering method pushes the system to only consider the relationships between users and items, ignoring either the features of users or the characteristics of items: with this approach, the utility matrix's relationships are the sole thing being considered. It is possible to create collaborative-filtering recommendation systems by either locating similar items that may be of interest based on the user's past interests, this is called *item-item collaborative filtering*, or by utilizing user similarities to recommend products that another user has rated highly, this is *user-user collaborative filtering*; in both cases the similarity of items and users is determined by the similarity of the ratings given by one users to an item.

As was already anticipated, the fundamental component of a recommendation system is the kind of prior knowledge provided for a particular problem that enables us to make reasoning about the given information. In the case of collaborative-filtering the prior knowledge is commonly embedded in a so called *utility matrix*. Given a set of users U and a set of items I, the utility matrix can be formally represented by an *utility function* f that associates users $u \in U$ and items $i \in I$ to a rating $r \in R$, where R is a set of valid ratings, i.e. $f : U \times I \longrightarrow R$.

2.1 Similarity measures

The concept of similarity is crucial to the system regardless of the technique we choose to provide recommendations. In recent years, data mining research has taken into account the issue of selecting appropriate similarity measurement methods as another crucial and effective aspect in the quality of results. According to the literature [6], similarity measures depend on the problem at hand, which means that although one measure may perform well for one data structure, it might be worse with another structure.

2.1.1 Jaccard similarity. The *Jaccard similarity*, often referred to as the Jaccard index, is a well established method to measure the similarity between sets. The similarity of two given sets A and B, can be measured as the intersection of the sets divided by the union of the sets.

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|} \tag{1}$$

This similarity measure can be adapted to work also for binary vectors, in fact a binary vector is a common and convenient way to represent a set. Given a universe set containing all the possible elements $U = \{x_1, x_2, ..., x_n\}$, any subset $S \subseteq U$ can be represented as a n-dimensional vector $\vec{v}$ where each component of the vector is 1 if the ith element from the universal set is present in S, 0 otherwise. More formally $\forall 1 \leq i \leq n$: $x_i \in S \implies \vec{v}[i] = 1$. From this definition we can derive that the Jaccard similarity for a binary vector is the number of times in which both vectors has 1 in the the same component, divided by the total amount of times at least one vector has 1 in the ith component. It's clear that the Jaccard similarity for vectors makes sense only in the case of vectors made only of 0s and 1s.

2.1.2 Cosine similarity. The *cosine similarity* is a similarity measure between two n-dimensional vectors $\vec{a}$ and $\vec{b}$. Let's denote a_i the ith component of the vector $\vec{a}$. The Cosine similarity of $\vec{a}$ and $\vec{b}$ corresponds to the angle between the two vectors.

$$S(\vec{a}, \vec{b}) = \frac{\vec{a} \cdot \vec{b}}{\|\vec{a}\| \|\vec{b}\|} = \frac{\sum_{i=1}^{n} a_i b_i}{\sqrt{\sum_{i=1}^{n} a_i^2} \sqrt{\sum_{i=1}^{n} b_i^2}} \tag{2}$$

This similarity measure is defined for vectors of any form, unlike the Jaccard similarity, which is defined on sets and consequently on binary vectors.

2.2 Fast similarities search

Finding similarities between objects is a fundamental problem in many fields; we may need to use these similarities for clustering, for instance, to locate plagiarism[13] and to identify almost identical web pages [8], as well as recommendation systems, to recommend items based on previous interests. The majority of current research in this area relies on approximate algorithms and in particular when dealing with enormous amounts of data, as in the case of data mining, these algorithms are essential. This paper explores at first an approach improve similarity search in the context of recommendation systems, focusing on the use of the *locality-sensitive hashing*(LSH) technique in conjunction with two locality-sensitive functions suitable for the Jaccard similarity and the Cosine similarity, respectively: *minHash*[2] and *simHash*[10]. In a further stage, this paper investigates how to create a hybrid recommendation system by combining content-based and collaborative filtering methodologies.

3 PROBLEM STATEMENT

In this section, the problem is formally defined after a few fundamental concepts are formally defined.

Definition 3.1 (Relational table). Given a set of n domains $D = \{D_1, D_2, ..., D_n\}$, a *relational table* $R(A : d_1, B : d_2, ...)$ is defined on these n domains as a subset of the Cartesian product of D, where a domain $d_i \in D$ is the set of possible values that a data element may contain [4].

$$R(A : d_1, B : d_2, ...) \subseteq \times\{D_i : i = 1, 2, ..., n\} \tag{3}$$

The notion $R(A : d_1, B : d_2)$ denotes a relational table made of two *attributes* A and B taking values respectively from domain d_1 and d_2, i.e. $A \in d_1$ and $B \in d_2$.

Definition 3.2 (Query). A user can retrieve information from a relational table by sending a *query* which is nothing more than a set of conditions over the possible attributes. A query q is formally defined as a function taking as input a set of conditions C over the attributes of the relational table R, returning a subset of tuples of R that satisfy the conditions in C; the queries are assumed to be defined as a set of conjunctions, i.e.

$$q(C) := \bigwedge_{c_i \in C} c_i \tag{4}$$

Throughout the entire paper, the set of all the queries will be referred to as Q.

Definition 3.3 (Rating function)**.** Before defining the problem it's important to define the concept of rating function. A *rating function* r, which is a specialized version of the utility function described in the related work section, associates users taken from a set of users S and queries taken from a set of queries Q with ratings in the range $1 - 100$. A rating is the means by which a user expresses his opinion regarding the outcome of a query: a rating of 1 denotes that the result of the query is unsatisfactory, whereas a rating of 100 denotes that the result satisfies the user.

$$r\colon S \times Q \to \mathbb{N} \in [1, 100] \tag{5}$$

A rating function facilitates the definition of a *utility matrix* U, which is a $|S| \times |Q|$ matrix containing for each cell U_{ij} a rating in range $1 - 100$ if the user $i \in S$ has rated query $j \in Q$, otherwise U_{ij} will be empty.

Definition 3.4 (Problem statement)**.** Given the following inputs

- a relational table R defined on a set of possible domains
- a set of users S
- a set of queries Q
- an utility matrix M, defined according to the aforementioned rating function r

fill in all the blanks in the utility matrix M in such a way that if a user $i \in S$ would likely appreciate the result of query j, the corresponding rating U_{ij} in the utility matrix should be filled with a positive rating. In the same way, the rating U_{ij} associated with user i should be negative if it is likely that the user will not appreciate the result of query j. The objective is to use the utility matrix information to provide query recommendations to users based on their preferences. In fact this method can be used for finding the most promising queries that the user hasn't rated but that may be of interest to him.

Definition 3.5 (Problem statement - Part B)**.** Given as input

- the utility matrix M, filled with all the missing ratings
- a generic query q, defined as a conjunction of conditions as shown in Definition 3.2

compute the importance of the query q according to how important the query is to the individual users, by taking into account the ratings each user has already given to other queries, as specified by the utility matrix M.

The general idea is to build a sophisticated query recommendation system to be integrated in a DBMS in order to take advantage of similarities in user ratings to recommend queries to other users based on the aforementioned inputs. Figure 1 accurately depicts the workflow of the recommendation system, which involves a user sending a query to a database through a DBMS, which then retrieves some data from it. Once the user receives the query result, he chooses to provide a rating. Based on those ratings, the recommendation system gives back to the user some recommendations.

4 SOLUTIONS

Multiple solutions are introduced in this part, starting with a preliminary naive approach and then refining it step by step to arrive at a

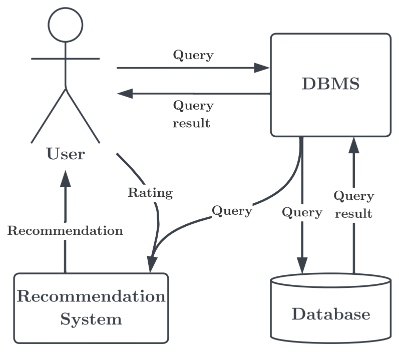

Figure 1: The workflow of the query recommendation system

definitive solution. The idea is to start with one of the traditional methods for handling recommendation systems and then improve it by combining it with many data mining techniques designed for handling massive amounts of data.

4.1 Naive solution

Building a recommendation system based on the traditional methodologies is the easiest and most straightforward way to address the aforementioned problem; as was covered in the section on related work, the two primary approaches for these kinds of tasks are content-based and collaborative filtering. Considering that the input of the problem is an utility matrix partially filled with ratings and the remaining cells left blank, the choice fell in a first stage on collaborative filtering rather then content based, avoiding in this way to define item and user profiles. The choice of collaborative filtering gives the possibility to build either an Item-Item or a User-User collaborative filtering recommendation system, where the missing values — the cells U_{ij} in the utility matrix that are blank — are filled according to one of the following schema:

(1) Item-Item collaborative filtering: this technique looks for the top K queries that are most similar to query j in the utility matrix. The average of the top-K items as rated by user i is then used to fill the value for U_{ij}.
(2) User-User Collaborative Filtering: This method is similar to the previous one in concept, but differs slightly in that instead of looking for the top-K items that are similar to item j, this method looks for the top-K users that are similar to user i, and the value U_{ij} is then filled with the average of the K most similar users to user j that have rated item j.

Both these strategies requires to define a similarity measure to locate a neighborhood of K similar items or users. Due to a number of factors, which can be summed up as follows, the Item-Item approach was chosen over the User-user approach:

- Item's neighbourhood tends to change much slower in comparison to user's neighborhood. This is due to the fact that the user-user approach is good at making recommendations to users with unique tastes;
- Explainability of the model. In comparison to describing why a person has similar preferences to another, it is considerably simpler to explain the reasoning behind why an item has

been recommended to a certain user based on previously rated goods;
- Finding items of the same type is easier than finding users who only like items of a particular type, so item-item similarity frequently provides more reliable information;
- The systems considered in the aforementioned problem definition typically have more users than items, and the more users there are, the more expensive it will be to find the K closest neighbors.

4.1.1 Choosing a similarity measure. Finding a good similarity measure is crucial for making a solid neighborhood choice, but there are a few factors to take into account before choosing the best one. Two similarity measures — the Jaccard similarity and the Cosine similarity — were introduced in the section 2.1 on similarity measures, one for sets (including binary vectors) and the other for vectors in euclidean space. The optimal similarity measure for this problem is the cosine similarity, considering that neighborhood selection is based on item ratings corresponding to a column of the utility matrix i.e. a vector. Even though the Jaccard similarity is not appropriate for handling vectors, a slightly modified version of this similarity measure is provided in the section for the next solution in order to handle vectors of user ratings.

4.1.2 Pseudocode. The idea of Item-Item collaborative filtering is to iterate over all the utility matrix's cells. If a cell U_{ij} is empty, it's rating is predicted by averaging the ratings of user i for the K queries that are most similar to query j. The similarity between two queries is given by the *cos_sim* procedure which takes as input two vectors to compare; the vector of the ratings of all the users for a single query j is denoted as U_{*j}. In the pseudocode the neighborhood size K is assumed to be 1, i.e. the rating for U_{ij} is predicted with the rating of user i in the query that is most similar to query j.

Algorithm 1 Naive version of Item-Item collaborative filtering

```
1: for $i \leftarrow 1$ to $|S|$ do
2:     for $j \leftarrow 1$ to $|Q|$ do
3:         if $U_{ij} = \emptyset$ then                ▷ Find empty cells $U_{ij}$
4:             for $q \leftarrow 1$ to $|Q|$ do
5:                 sim $\leftarrow$ sim $\cup$ cos_sim($U_{*j}, U_{*q}$)
6:             end for
7:             $top \leftarrow$ argmax $sim$        ▷ Find most similar query
8:             $U_{ij} \leftarrow top_i$
9:         end if
10:    end for
11: end for
```

This method involves computing the cosine similarity for each potential pair of queries. The cost of computing the cosine similarity is equal to the cost of computing the dot product of two queries, each of which corresponds to a column vector of $|S|$ components and has a cost of $O(|S|)$. The final cost to predict one single missing value of the utility matrix is given by $O(|S| \cdot |Q|^2)$. It's clear that predicting all the missing values of the utility matrix is costly, in particular when the dealing with a massive amount of users and queries.

4.2 MinHash for LSH

The primary issue with the previous solution is the algorithm's time complexity. In particular, the cost of finding the top K queries that are similar to a given query is largely driven by the need for the algorithm to compute the similarity between all potential combinations of queries in the utility matrix. In order to address this issue, a more effective technique is presented in this section that employs minHash in conjunction with LSH to increase the similarity search's speed at the expense of accuracy. The purpose of LSH is to avoid checking the similarity between items that are definitely dissimilar and to concentrate solely on pairs of items that are likely to be similar; the latter are referred as *candidate pairs*. Finding those candidate pairs in the original utility matrix is expensive; for this reason the idea is to apply LSH on a signature matrix rather than on the original utility matrix.

4.2.1 MinHash. LSH typically operates on small signatures derived from a *characteristic matrix*, which is a common representation of the data in order to describe the characteristic of the items. In the specific case of documents, the characteristic may be seen as a matrix C with each row representing a potential word from a given vocabulary and each column representing a document: the word i is present in the document j if and only if $C_{ij} \neq 0$. There is a clear connection between the concept of characteristics matrix and the concept of utility matrix of recommendation systems, with the latter being a characteristics matrix in which each column j is an item(a query) and each row i corresponds to a user: the user i has rated item i if and only if $C_{ij} \neq 0$. This correlation between the twos clearly allows to exploit the MinHash technique to obtain a compressed representation of the utility matrix(from now on the utility matrix is considered to be the characteristic matrix).

Dealing with the utility matrix directly is impractical in some cases due to its massive size; for this reason the idea is to replace the characteristics matrix with a compressed version called *signature matrix* in such a way that the similarity of the items in the signature matrix is as close as possible to the similarity of items in the original characteristics matrix. No method can guarantee that the similarity will be exactly the same because some information from the characteristics matrix may be lost despite the fact that it was crucial in determining the similarity. The MinHash technique[2] has been devised for the Jaccard similarity, thus as more signatures are added to the signatures matrix, the estimate of the Jaccard similarity between two items will get more accurate and similar to that for the original utility matrix.

The idea of MinHash is to generate a signature matrix δ by taking a number of H of permutations π of the rows of the utility matrix U(corresponding to the characteristics matrix) and finding for each of the items the first row in the permutation π_t of U_{*j} that has a value of one, i.e.

$$\delta_{tj} \leftarrow \min_{\pi}(U_{\pi_t j} = 1) \tag{6}$$

Example. Given a utility matrix U of three items and a set of two permutations π, the signature matrix δ is produced as follows.

$$\pi = \begin{array}{|c|c|}\hline \pi_1 & \pi_2 \\ \hline 4 & 6 \\ \hline 3 & 3 \\ \hline 6 & 4 \\ \hline 2 & 1 \\ \hline 1 & 5 \\ \hline 5 & 2 \\ \hline \end{array} \quad U = \begin{array}{|c|c|c|}\hline i_1 & i_2 & i_3 \\ \hline 1 & 1 & 1 \\ \hline 0 & 1 & 1 \\ \hline 1 & 0 & 0 \\ \hline 0 & 0 & 0 \\ \hline 0 & 1 & 1 \\ \hline 1 & 0 & 1 \\ \hline \end{array} \quad \delta = \begin{array}{|c|c|c|}\hline i_1 & i_2 & i_3 \\ \hline 4 & 1 & 1 \\ \hline 1 & 3 & 2 \\ \hline \end{array}$$

4.2.2 Permutations generation. Generating the permutations and storing them in memory might be infeasible when dealing with utility matrices having a massive amount of users. For this reason a possibility is to simulate the effect of random perturbations by picking H hash functions, one for each permutation, that return the index of a given row in the permutation.

$$\pi(x) = (ax + b) \bmod c \tag{7}$$

where a, b and c are random value less than the number or users in the utility matrix, i.e $a, b, c \leq |S|$.

4.2.3 Adapted MinHash for ratings. Given that the Jaccard similarity is defined for sets and consequently binary vector as exposed in Section 2.1.1, there is a problem in the above formulation of MinHash because it assumes that the characteristics matrix is made of 0s and 1s. However in the problem formulation, described in Section 3, the utility matrix U is made of ratings in the range $[1, 100]$. A solution for this problem is to convert the utility matrix U containing the ratings into a matrix $\widehat{U}$ of zeros and ones, where the value $\widehat{U}_{ij} = 1$ indicates that the user i has liked query j. Given that the threshold at which a query might be considered liked or not might vary depending on the situation, a threshold parameter named T is introduced, according to which a query is considered liked if its rating is greater than T, i.e.

$$\widehat{U}_{ij} = 1 \iff U_{ij} \geq T \tag{8}$$

Example. Given a utility matrix U of ratings in the range $[1, 100]$, missing ratings denoted by cells with a value of 0, and a threshold level of $T = 50$ above which a rating is deemed to be positive, the corresponding utility matrix $\widehat{U}$ of positive ratings is obtained according to Eq. 8 as shown below.

$$U = \begin{array}{|c|c|c|}\hline q_1 & q_2 & q_3 \\ \hline 80 & 5 & 12 \\ \hline 0 & 67 & 0 \\ \hline 46 & 85 & 0 \\ \hline 0 & 55 & 65 \\ \hline 35 & 10 & 90 \\ \hline 95 & 0 & 45 \\ \hline \end{array} \qquad \widehat{U} = \begin{array}{|c|c|c|}\hline q_1 & q_2 & q_3 \\ \hline 1 & 0 & 0 \\ \hline 0 & 1 & 0 \\ \hline 0 & 1 & 0 \\ \hline 0 & 1 & 1 \\ \hline 0 & 0 & 1 \\ \hline 1 & 0 & 0 \\ \hline \end{array}$$

The computation of the modified version of the utility matrix in practice is computationally expensive, in particular given $|S|$ and $|Q|$ respectively the number of users and the number of queries in the utility matrix, the cost of making such conversion is $O(|S| \cdot |Q|)$, which is quadratic in the size of the input. To overcome this issue a possibility is to use a slightly modified version of MinHash, denoted as *MinHash with threshold*, that assigns to the signature matrix entry δ_{tj} the index of the first row in the permutation with a rating above the threshold level T.

$$\delta_{tj} \leftarrow \min_{\pi}(U_{\pi_t j} \geq T) \tag{9}$$

The resulting signature matrix is a good approximation for a modified version of the Jaccard similarity measure in order to take into account positive ratings as 1 if they are greater than a threshold T and 0 the elements that are smaller than T.

An implementation of this idea in pseudocode is shown in Algorithm 2. The idea is to iterate over all queries and H permutations of the utility matrix rows to discover the first row in the permuted order of the utility matrix that has a rating higher than the threshold T. In most cases, the loop of line 4 requires less iterations then the toal number of users $|S|$ in the utility matrix. The index i is used as an input to create the permuted index u, increasing by one with each iteration until it reaches a maximum value of $|S|$.

Algorithm 2 MinHash algorithm: signature matrix generation

```
 1: for $j \leftarrow 1$ to $|Q|$ do
 2:     for $h \leftarrow 1$ to $H$ do
 3:         $i \leftarrow 1$                          ▷ $i$ is the original index of the rows
 4:         while $\delta_{hj} = \varnothing \cap i \leq |S|$ do
 5:             $u \leftarrow \pi_h(i)$                ▷ $u$ is the permutation of $i$
 6:             if $U_{uj} \geq T$ then
 7:                 $\delta_{hj} \leftarrow i$
 8:             end if
 9:             $i \leftarrow i + 1$
10:         end while
11:     end for
12: end for
```

4.2.4 Locality Sensitive Hashing. Once the signature matrix has been generated according to the modified version of MinHash with threshold T, the following step is to use the Locality Sensitive Hashing(LSH) technique to hash the queries in the signature matrix several times, in such a way that similar queries, i.e. queries with similar ratings, are more likely to be hashed to the same bucket than dissimilar queries are. The signature matrix is divided into b bands, each made up of r rows. Each band is then column-wise hashed into a set of buckets that is specific to that band, and queries that hashed to the same bucket are then considered to be candidate pairs. The information included in the candidate pairs can be used to reduce the time required to detect similar queries, so that each query i is tested for similarity with each query j that lie in the same bucket as query i for at least one band, i.e. query i and query j are candidate pairs.

The number of bands b and the number of rows for each band r play a crucial role in the amount of candidate pairs found for each query. In fact these two parameter are directly related one with the other i.e. recalling that H is the number of permutation, namely the number of rows of the signature matrix δ then it holds that $b \cdot r = H$. Given this connection, it follows that the number of candidate pairs that LSH will find will increase if there are many bands and, as a result, few rows per band. On the other hand, if there are few bands and a large number of rows per band, LSH will locate fewer items that are similar to each other. A good trade-off must be chosen for the parameter b defining the number of bands or equivalently for the parameter r defining the number of rows for each band: the size of the signature matrix and the distribution of the data in the original utility matrix are two important criteria that must be taken into consideration while choosing these parameters; in the latter scenario, it is possible that all of the queries are extremely dissimilar from each other, which means that even a small value of r is able to capture a significant number of candidate pairs. In the opposite scenario, it's possible that every query is incredibly similar to each other query, indicating that a low value of r is insufficient and that the number of rows per band should be increased to prevent an excessive number of similar items from going against the goal of finding a good compromise on the number of similar items.

4.2.5 Making recommendations. The final step is to provide recommendations using the item-item collaborative filtering method, similar to the one used in the Naive solution, with the exception that in this case, the iterative process that identifies the queries with the highest degree of similarity only iterates on the candidate pairs of a given query rather than on all possible queries. Considering time complexity, in the worst scenario the number of candidate pairs identified for each query is equal to the set of all the queries, leading the algorithm to fall back on the naive method. However this is just a limit case, in fact the number of candidate pairs can be reduced by appropriately adjusting the number of bands b and the number of permutations H.

4.3 SimHash for LSH

The Jaccard similarity, used with MinHash, is not the proper similarity measure for the problem taken into consideration; in fact, as discussed in section 4.1.1, the Jaccard similarity is not the optimal similarity measure for vectors. In some situations, the modified version of the Jaccard similarity that takes the threshold parameter T into account fails to measure the similarity between two queries. This situation occurs, for instance, when two queries have a rating that is remarkably similar, such as 49 and 51, but the threshold parameter $T = 50$ makes them different. This problem is brought on by the indirect conversion of the utility matrix into a matrix of zeros and ones indicating whether a user liked or didn't like a query. A more suitable similarity metric for the issue under consideration in this study is the Cosine similarity, which can be computed on vectors of any type rather than only binary vectors.

This algorithm's general structure is quite similar to the one discussed for the solution with MinHash for LSH. The goal of the algorithm is to create a signature matrix δ that is representative of the original utility matrix, which means that given two general queries, their similarity should be preserved as much as possible in the signature matrix. The computation of the signature matrix should be quick and affordable, allowing the algorithm to benefit from using the signature matrix rather than the original utility matrix. After computing the signature matrix, the following step is to apply LSH to it in order to identify local similarities used to generated a set of candidate similar queries.

4.3.1 SimHash intuition. The intuition behind of SimHash is to generate a signature matrix of H rows, dividing the space containing the vectors corresponding to queries into H hyperplanes passing through the origin. For any given hyperplane, each query ends up in one region of the hyperplane, either above or below the hyperplane. The signature matrix is filled for each query with its position with respect to the hyperplane H_i corresponding to one row of the signature matrix δ. The idea is that the closer two queries are, the more probable it is that they will be on the same region for a random hyperplane. For instance, on the majority of the planes that can be randomly generated, two queries that are relatively close to each other will end up in the same region. The example in Figure 2 accurately illustrates the intuition by demonstrating that for any arbitrary number of random hyperplanes and two similar queries $\vec{q}_1$ and $\vec{q}_2$, the probability of the queries falling on different sides of the hyperplanes is relatively small compared to the total number of hyperplanes: in that specific case, only the red plane divides the two queries into two different sides of the plane. SimHash has been demonstrated to be a locality-sensitive hash function that roughly approximates the cosine similarity[12] [7].

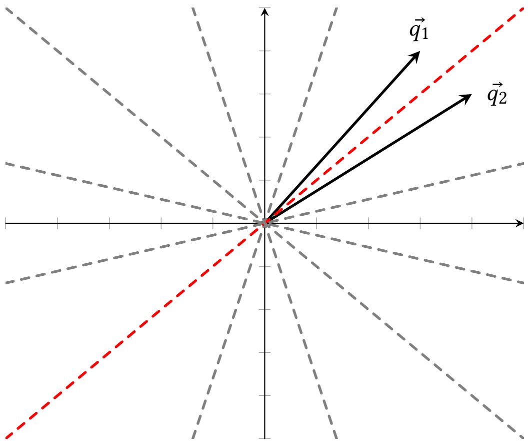

Figure 2: Two similar queries $\vec{q}_1$ and $\vec{q}_2$ are more likely to fall into the same region of any random hyperplane than when they are dissimilar.

4.3.2 Signature matrix computation. The computation of the signature matrix is straightforward based on the prior intuition: generate H random hyperplanes and fill the ith row of the signature matrix with the query's position relative to the hyperplane. From a more formal perspective an *hyperplane* is represented in a n-dimensional space with an n-dimensional vector that is orthogonal to the hyperplane. The position of a query vector $\vec{q_j}$ with respect to a general hyperplane H_t, represented by its orthogonal vector $\vec{h_t}$, is given by the sign of the projection of $\vec{q_j}$ on $\vec{h_t}$. Given that the projection of a vector on another one is computed using the dot product, the signature matrix cell corresponding to the ith hyperplane represented by its orthogonal vector $\vec{h_i}$ and the jth query $\vec{q_j}$ is computed as follows

$$\delta_{tj} \leftarrow \text{sign}(\vec{h_t} \cdot \vec{q_j}) \tag{10}$$

where the sign function returns a value of 1 if the argument that it takes as input is positive, otherwise it returns -1. The values of 1 and −1, respectively, indicate that the query vector is located either on the positive or negative side of the hyperplane.

$$\text{sign}(\vec{h_t} \cdot \vec{q_j}) = \begin{cases} 1 & \text{if } \vec{h_t} \cdot \vec{q_j} \geq 0. \\ 0 & \text{otherwise.} \end{cases} \tag{11}$$

Example. Consider the example depicted in Figure 3 of a generic hyperplane H_1 represented by its orthogonal vector $\vec{h_1} \in \mathbb{R}^2$ and two queries $\vec{q_1}, \vec{q_2} \in \mathbb{R}^2$, characterized by the ratings given by two users. The position of the queries with respect to the hyperplane is determined by projecting $\vec{q_1}$ and $\vec{q_2}$ onto $\vec{h_1}$ and checking the sign, i.e. computing the sign of the dot product between the two vectors. In this specific instance, both queries are found to be on the positive side of the hyperplane H_1, therefore the corresponding cell in the signature matrix is filled with a value of 1. Following the notation of signature matrix cell δ_{ij}, defining the ith row(the ith hyperplane) and the jth column(the jth query) of the signature matrix, the signature for the first plane H_1 and the aforementioned queries is filled with $\delta_{11} \leftarrow 1$ and $\delta_{12} \leftarrow 1$.

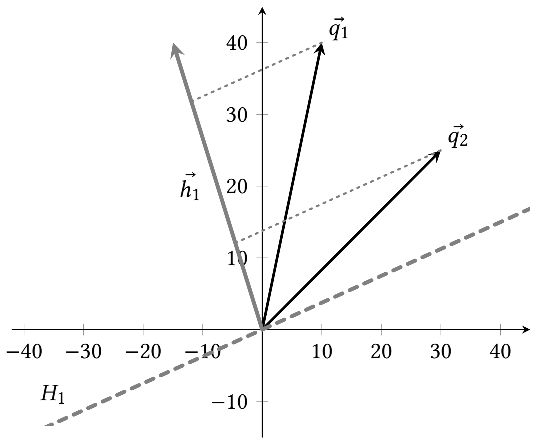

Figure 3: Projection of queries $\vec{q_1}$ and $\vec{q_2}$ onto $\vec{h_1}$ to find their position with respect to the hyperplane H_1.

Preprocessing. Given that a query in the utility matrix is represented by a vector of positive ratings with values between 1 and 100, all query vectors will always be located in the positive quadrant of the space. In the 2-dimensional example depicted in Figure 3, all the queries will always be located in the upper rightmost quadrant. This behavior is problematic because, regardless of the query, the projection on any hyperplane defined by an orthogonal vector falling within the aforementioned quadrant will always have a positive sign. To overcome this issue the solution is to normalize the utility matrix's ratings into the range $[-50, 50]$ by subtracting the average value of 50 from all the ratings in the utility matrix that are not 0(missing rating), i.e. the new utility matrix $\widehat{U}$ is computed as follows:

$$\widehat{U}_{ij} \leftarrow U_{ij} - 50 \quad \forall i, j \text{ s.t. } U_{ij} \neq 0 \tag{12}$$

The initial utility matrix U is shown on the left in the example of Figure 4, and the new utility matrix $\widehat{U}$, which is centered in 0, is shown on the right.

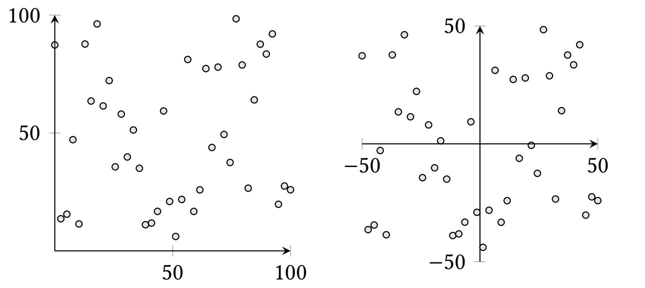

Figure 4: The rating of the queries that originally are in the range $[1, 100]$ are centered in the space to obtain ratings in $[-50, 50]$.

Pseudocode. Compared to the MinHash-based approach for producing the signature matrix, the SimHash algorithm is significantly simpler. Assuming that H is a set of $|H|$ randomly generated hyperplanes, $\widehat{U}$ is the pre-processed utility matrix and sign is the function defined by Equation 11, the signature matrix is computed as follows

Algorithm 3 SimHash algorithm: signature matrix generation

```
1: for $j \leftarrow 1$ to $|Q|$ do
2:     for $t \leftarrow 1$ to $|H|$ do
3:         $\delta_{tj} \leftarrow \text{sign}(\widehat{U}_{*j} \cdot \vec{h_t})$
4:     end for
5: end for
```

4.3.3 Making recommendations. Similar to the approach with MinHash, the next step is to leverage the signature matrix's information to find local similarities between queries using the Locality Sensitive Hashing technique outlined in Section 4.2.4. Given that not all candidate pairs correspond to similar queries, the candidate pairs need to be manually checked for similarity. According to the item-item collaborative filtering approach discussed in Section 4.2.5, similar queries can be used to generate recommendations. The idea is to predict the rating of users i to query j, which corresponds to U_{ij} in the utility matrix, as the average of the ratings given by user i for the K queries that are most similar to query j. The complete architecture is shown in the diagram of Figure 5.

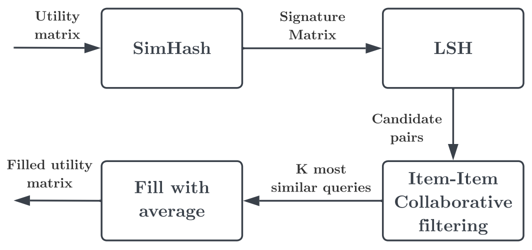

Figure 5: Architecture of the collaborative filtering recommendation system using LSH to quickly identify related queries.

4.4 Content based recommendations

Given that the similarity of queries is related to both their properties as well as their ratings given by users, it is possible for users to give the same ratings to two queries even when they are completely different. In light of this, the previous method — which combines collaborative filtering and LSH — is ineffective for providing relevant suggestions since it only took into account the ratings of queries rather than their structure. Consequently, the idea is to combine the algorithm considered in the previous sections(LSH in combination with collaborative filtering) with a content-based strategy, to build an hybrid recommendation system. This new approach takes advantage of both the properties of the queries as well as the ratings given by the users to provide pertinent query recommendations based on previous user activity. The properties of a query can be interpreted from a conceptual or from a results-oriented point of view, respectively a query can be seen as a conjunction of conditions or as a collection of tuples.

4.4.1 Item and user profiles. Content-based recommendation systems look through item properties to find items that a specific user may be particularly interested in. This kind of recommendation system works on the principle of creating user and item profiles, one for each system user and one for each item, respectively. The principle is that after the profiles are created, it should be easy to calculate how much a user could appreciate an item: compare the user profile to that of the item, and if a high similarity is found, it is probable that the user would appreciate that item.

Query profiles. An item profile should describe the important features of an item, for example in the case of movies, it might be the actors that are part of the movie. According to the same logic, the characteristics of a query, as previously noted, can be represented by its conditions or by the set of tuples it returns. In the instance examined in this work, the decision fell to take into account item profiles (referred ti *query profiles* throughout this paper) as the tuples returned by each query. The query profile for a generic query j, denoted as IP_j, is represented as vector with the same number of components as the total number of rows of the relational table. The ith component of IP_j is set 1 if the query j returned the ith row of the relational table, otherwise 0. For example the query profile of a generic query j that returns only the first and the third row of a relational table with 5 rows can be represented as $IP_j = [1, 0, 1, 0, 0]$.

User profiles. User profiles describe the user preference to the features of items profiles, for this reason to compare user profiles with query profiles, user profiles must have the same number of components of item profiles. Since each component of the user profile correspond to a row of the relational table, the issue becomes evident: In the utility matrix, the user's preferences are expressed on queries rather than the rows that the queries return, so it's not clear how to express the user preference of a single row. To solve this problem the solution is to find the rows returned by each query(simulating a DBMS) and then compute the preference of user t to the ith row of the relational table as the average rating given by user t to all the queries returning the ith row of the relational table. Throughout the article, the user profile for user t is referred to as UP_t.

4.4.2 Hybrid recommendation system. Once user and query profiles have been appropriately generated, they can be used to match users with potentially interesting items. The approach is to calculate the cosine similarity between all the the vectorized user and item profiles in order to predict how much the user would likely appreciate an item. As already discussed in Section 2.1.2 the cosine similarity measures the angle between two vector, so if the cosine similarity between an item profile IP_j and a user profile UP_t is high, it means that the angle between the two vectors is small, thus the user t will probably like the query j. The idea, which is accurately depicted in Figure 6, is to build an *Hybrid recommendation system* that combines the content-based approach introduced in this section with the collaborative filtering technique mentioned in the earlier solutions: the algorithm uses at first collaborative filtering with LSH to find the K most similar queries of each query based on the user ratings in the utility matrix, then on top of that, the algorithm exploits content-based information derived from the queries to generate more accurate recommendations. As shown in figure 6 the content-based phase attempts to match each user of the system with the optimal query among the K most similar queries found by collaborative filtering: each user's profile is compared to the profile of each query among the K that are most similar to a particular one.

Similarly to the previous method, this one begins by running LSH with collaborative filtering. However, instead of predicting the missing ratings using the average of the K most similar queries, it then runs content based to determine which query will be the best match for the user.

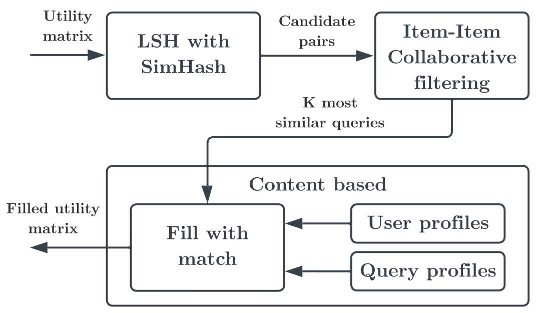

Figure 6: Architecture of the hybrid recommendation system

4.5 Measuring the importance of a query(Part B)

The aforementioned algorithm has been making recommendations based on queries that were already part of the query set, meaning that at least one user had submitted them. It's clear that in a more advanced scenario, queries that have not yet been posed to the DBMS might be of interest to the users of the system. Based on this, the purpose of this section is to provide a method for determining the importance of a query in general for all the users, i.e. how much the users of the system would like a query that may or may not have already been asked to the DBMS.

Measuring the importance of a query must be done carefully and taking into account all the information that is available in the current system. For this reason there are several resources that can be helpful to build a system that computes the importance of a query; First of all there are two fundamental inputs:

- the utility matrix filled with the missing ratings (U)
- the query itself, defined as a set of conditions (q)

The two aforementioned are the main inputs, but there are also some indirect inputs as a result of the utility matrix being calculated utilizing some other resources, namely the relational table (R) and the query set (Q).

Definition 4.1 (Importance)**.** For the purposes of this research, the *importance* of a query has been defined in a way to assigns an integer value that is high if users in general will find that specific query interesting or a low value if users may think it is not significant at all. It was decided to use the convention of quantifying the importance of a query on a percentage scale that ranges from 0% (not relevant) to 100% (important).

There are numerous approaches to finding a simple importance metric that could guarantee a precise enough measurement, but most of the time these approaches only use information from the relational table without considering the user's ratings, which is not enough when the objective is to deliver a sophisticated method that accurately measures the importance. The solution proposed in this section takes into account both the preferences of the users, expressed in terms of the utility matrix, and the factual utility of a query obtained by the rows that it returns from the relational table. Consequently, the suggested algorithm is divided into a preprocessing phase followed by two important steps: the first step computes the importance of a query based on the user ratings provided in the utility matrix, whereas the second one leverages the rows returned by the query, i.e. the result of a query.

4.5.1 Preprocessing. This early phase runs the query through the DBMS and produces results that allow some queries that could initially be considered as unimportant to be excluded from the subsequent phases. The idea is that queries returning all of the rows from the relational table R or queries returning nothing at all should be treated as unimportant, and the algorithm should therefore assign them a low relevance rating, let's say 0%. This task usually takes very little time and is trivial because a query's outputs serve as its primary measure. This step is useful as generally a user would judge a query as a good one if it returns at least some outcomes, but not all the results. If the query is not labelled as unimportant in this preprocessing phase, the algorithm moves on with the next step.

Algorithm 4 Preprocessing

```
1: $rows \leftarrow executeQuery(q)$
2: if $rows = \varnothing$ or $rows = R$ then
3:     $importance \leftarrow 0\%$
4: else
5:     $Step1()$
6: end if
```

4.5.2 Step 1. This step takes into consideration users' preferences provided by the utility matrix. The idea in this phase is to first obtain the average rating of each query from the utility matrix, by taking the average of each single column, corresponding to a query, and then compute the average rating given to each feature f_i of that query as the average rating given by users to queries containing that feature f_i. In detail, as soon as each query q_i is splitted into its conditions, the next step is to gather all of the features that shape each condition, without considering the value of that condition(recall that a feature is a relational table column, for example considering the condition $f_1 = Trento$ then f_1 is the feature and $Trento$ is the value). As each feature f_i is gathered, the algorithm gives to each feature f_i a value of importance that is proportional to the average rating given by users to queries containing feature f_i. If a feature has no recorded grade, it will be counted as non important, so it will be assigned a 1% grade.

Example. Consider a feature *city* which appears in only two queries of the query set, namely q_1 and q_2; suppose that according to the utility matrix the average ratings of query q_1 and q_2 are respectively 70 and 90. The average grade to the *city* feature is computed as the average of 70 and 90, that is 80.

Once all the feature ratings are retrieved, the partial query importance(result of this first part) is computed as the average of its features' importance. To get this outcome the idea is to use the Geometric Mean[9] as the average measure of importance. The choice to use the Geometric Mean is driven by the necessity to give more emphasis to low rated features rather than highly rated features: given that a query is defined as a conjunction of condition, all the conditions must be satisfied in order for the query to return a specific row, thus even a negative grade on a single feature probably means that the entire query is not useful at all. The Algorithm 5 illustrates the pseudocode defining the process of computing the first partial importance, where the partial calculation of a query's importance computed in this first step is indicated as I_a.

4.5.3 Step 2. In the previous step the importance of a query is calculated by the algorithm by solely taking into account the features in the query's conditions, without considering the values assigned to those features. This step instead is meant to take into consideration the values assigned to the features in order to obtain a more accurate measure of importance.

This step starts by computing the importance of the query as the importance of each of its valued-conditions. Differently from the previous phase, this step considers the value of the feature in the conditions, and computes the importance of each condition as the number of rows that satisfy that single condition. To give a reasonable grade to each condition, if the condition returns more than 20% of the rows of the relational table, it will have a 100% score, if 19% it will have a score of 95%, and so for steps of 5% each time. The importance of the query is then given as the geometric mean of the importance of each condition that forms the query. The reason to do so is because if an item appears many times in the relational table, it means it is linked to many other objects inside the dataset, meaning it is connected to a lot of information. Then, the temporary score of the query will be an average of the score of each item.

Algorithm 5 Step 1

```
queryScore ← []
for $j$ ← 1 to $|Q|$ do                      ▷ Avg rating of each query in Q
    $queryScore_i \leftarrow avg(U_{*j})$
end for

$featureRating \leftarrow []$
for $f_i$ in $R.features$ do                 ▷ Avg rating of each feature in R
    $featureScore \leftarrow []$
    for $j \leftarrow 1$ to $|Q|$ do
        if $f_i$ in $q_j$ then
            $featureScore \leftarrow featureScore \cup queryScore_j$
        end if
    end for
    $featureRating_{f_i} \leftarrow avg(featureScore)$
end for

$queryFeatures \leftarrow []$
for $f_i$ in $R.features$ do                 ▷ Importance of the query q
    if $f_i$ in $q$ then
        $queryFeatures \leftarrow queryFeatures \cup featureRating_{f_i}$
    end if
end for
$I_a \leftarrow GeoMean(queryFeatures)$
```

Example. Consider a query made only of two conditions, such as *city = Trento* and *age = 30*. The importance of the two conditions corresponds respectively to a number that is proportional to the number rows of the relational table having the value *Trento* in the *city* column and having the value *30* in the age column. The final importance is then given as the geometric mean of these two values.

From this first score comes a major issue, which is the fact that if a query presents only one condition, its score will rely only on that condition's score, biasing its importance. Due to this, it is significant to find a way to penalize such a type of query, as in most cases they would be considered useless for all the stack of users. There are many ways to weight an outcome based on the amount of conditions and probably the easiest one would be to multiply the importance value obtained so far by a weight factor given by the number of conditions of the query q, denoted as $|q|$ divided by a parameter ω corresponding to the average number of conditions of the queries in the query set. This simple solution is formalized in Equation 13, however this method ends up penalizing too much all the middle range values.

$$I_b = \frac{|q|}{\omega} \cdot I_b \tag{13}$$

For the aforementioned reason, the most proper way is instead to use a weight that correspond to the cumulative distribution function (CDF) of the exponential distribution, evaluated on the number of conditions of the query q. The CDF of the exponential distribution is characterized by a rate parameter α which describes the increase rate of the CDF function, i.e. a small value of α implies a slow asymptotic convergence of the function to the value of 1.

$$I_b = \left(1 - e^{-\lambda \cdot |q|}\right) \cdot I_b \tag{14}$$

The Algorithm 6 illustrates the pseudocode defining the process of computing the second partial importance, where the partial calculation of a query's importance computed in this second step is indicated as I_b.

Algorithm 6 Step 2

```
$itemScore \leftarrow []$
for all $c_i$ in $q$ do              ▷ Importance of each condition $c_i$ of q
    $rows \leftarrow exequteQuery(c_i)$
    $frac \leftarrow \frac{|rows|}{|R|}$
    if $frac \geq 20$ then
        $itemScore \leftarrow 100$
    else
        $itemScore \leftarrow 5 \cdot frac$
    end if
end for
$I_b \leftarrow GeoMean(itemScore)$
$I_b \leftarrow CDF(|q|) \cdot I_b$
```

4.5.4 Final Step. The final value of importance for the query is given by the combination of two aforementioned steps. The idea is to give the rightful weight α to trade off the importance found by each step, emphasizing one over the other depending on the considered scenario. As data is different depending on the scenario, ideally it is better to have a variable weighted sum of the values of importance found in the two steps, rather than using always a constant value of $\alpha = 0.5$. The final formulation to measure the query importance is defined as follows.

$$importance = \alpha \cdot I_a + (1 - \alpha) \cdot I_b \tag{15}$$

The parameter α, as well as λ are two hyper-parameters of the formulation which should be tuned for each scenario, in particular it possible to find the optimal values relying on some kind of optimization technique.

5 EXPERIMENTAL EVALUATION

This part assess the solutions put forth in the preceding sections and demonstrate their efficacy in a simulated real-world scenario using data that was generated in a way that reflect reality. The performance of the algorithms explored in this paper have been measured on a machine with the following specifications: Intel Core i7-6700HQ CPU @ 3.5GHz with 16 GB of RAM.

5.1 Datasets

The algorithms that implement the solutions proposed in this research have been tested on several datasets that were artificially constructed to as closely resemble a real-world scenario. In particular, the synthetic datasets were produced starting with a relational table that was filled with the Scikit-Learn library's *make_blobs* function, which makes it possible to generate correlated data. The choice of generating correlated data is motivated by the observation that in a real world scenario, in a relational table representing for example people, there exist groups of persons who share characteristics like eye color, height, etc.

The query set is also built in a realistic manner; precisely, the conditions of each query are constructed by selecting a random number of features (even zero), giving each one a value in the feature's column with a chance of 99 percent, and using the remaining probability to give a generic random value (even not in the admitted values of that feature). By doing this, the queries are created in a way that causes a great number of them to return some few rows, others to return all the rows, and eventually no rows at all. It's important to note that when a query has no conditions, all of the rows in the relational database are returned.

The core of the dataset is the utility matrix and to generate this the idea is to create categories of users that are users tend to have the same taste of others. In addition users tends to act following patters. For this reason to fill the utility matrix ratings for the users the idea is to first split the users in three categories:

(1) 60 percent of the users rates queries that returns similar rows in the same way. If two queries q_1 and q_2 returns the majority of the rows in common, the user who rates query q_1 with rating r will rate query q_2 with a rating that differs by r by a tiny factor γ, for instance $\gamma = 5$.
(2) 30 percent of users grade the queries proportionately with the amount of rows returned by the query. A user may be satisfied when the query returns some rows, but in another sense unsatisfied if the query does not return any row.
(3) the remaining 10 percent of the users assign random ratings to the queries.

Even though users can evaluate queries on a scale from 1 to 100, it is possible that two users will score a query differently in the real world. For example, one user may rate a query positively with a rating of 50, while another may rate the query positively with a rating of 100. Because of this, the user ratings produced in the previous phase are randomly divided into the following categories: users who rate on a scale of 1 to 50, users who rate on a range of 50 to 100, 1 to 100, etc. Scaling the ratings is a straightforward process that may be carried out using equation 16, where *low* and *high* represent the user's lowest and highest ratings, respectively.

$$U_{ij} = \frac{(\text{high} - \text{low})}{100} U_{ij} + \text{low} \tag{16}$$

5.1.1 Synthetic dataset characteristics. The synthetic dataset is created as previously described using a relational table with 100 features over a total of 10000 rows, and in addition, the values of the relational table were filled with integer numbers; this latter choice is wise because, in general, relational tables take values from a specific domain, making representing cities by their names or by a representative integer number equivalent. The dataset is designed to simulate a system with 500 users who have submitted the DBMS a total of 2000 queries. Consequently, the utility matrix is composed of 500 rows and 2000 columns (queries). Last but not least, a script has been developed to calculate some crucial information from the relational database and the queries, revealing that, out of a total of 2000 search queries, 721 produced at least one row, and the average amount of rows returned by these queries is 3311.

Even if the numbers appear to be small, they are sufficient to evaluate the effectiveness of the proposed algorithms. They are particularly sufficient to demonstrate how the algorithm's performance improves in various scenarios: this dataset in particular was produced at that scale to enable its division into smaller portions for even more specific experiments.

5.1.2 Real dataset. Relational tables typically contain experimental observations of real data; as a result, the solutions proposed in this work are also evaluated on a dataset in which a real relational table replaces the synthetic one. In this instance, the relational table is taken from the relational database of the 1994 Census Bureau [1]. This dataset is made up of 14 columns that reflect individual attributes including age, sex, marital status, and income. The original dataset had over 50000 individuals, however it was scaled down to only include 10000 individuals in order to measure performance. The other components of the entire dataset are created using the same methodology as the synthetic one, yielding a total of 2000 queries and 500 users. In this case the dataset revealed that, out of a total of 2000 search queries, 908 produced at least one row, and the average amount of rows returned by these queries is 3649.

5.2 Fast query similarity search with LSH

This section analyzes the algorithm's initial building block to demonstrate how it performs in comparison to the baseline method for determining query similarity for collaborative filtering. This section specifically aims to demonstrate how the LSH approach, as explained in sections 4.2 and 4.3, may significantly improve the naïve solution without LSH of section 4.1. First, it is shown that using LSH with MinHash reduces the time required to find similar queries for collaborative filtering using the Jaccard similarity as a similarity measure for the queries, and second, it is shown that SimHash with LSH outperforms its corresponding baseline solution, which tries all possible combinations of queries to find the similar ones. Considering the original solutions, the goal of this section is to identify each query's most related queries based on the utility matrix ratings given by the users: without losing generality, the goal is slightly change to locate the most similar query rather than all the K most similar ones.

5.2.1 LSH with MinHash. In this section the LSH MinHash algorithm is compared with its naïve version that computes the Jaccard similarity to find the most similar query. The time efficiency of these two algorithms is compared in two different scenarios: the first involves a changing number of queries with a constant number of users, and the second involves a changing number of users having a constant number of queries. The approach is to divide the dataset used as input into smaller fractions and test the algorithms on those fractions, simulating smaller datasets in the process. In this case of MinHash and Jaccard similarity a dataset with at most 1000 queries and 500 users is sufficient to observe the time complexity growth. The plot of Figure 7 reports the execution time of the two methods in a scenario with a constant amount of 500 users and a variable number of queries. The other scenario is plotted in Figure 8 where the number of queries is kept constant at 1000 queries and the number of users is variable. In both instances, MinHash is set up to generate a signature matrix with 200 rows, corresponding to a total of 200 random perturbations; additionally, the LSH bands are configured to have 6 rows per band, which divides the signature matrix into 33 bands.

The two plots in Figures 7 and Figure 8 clearly show the time improvements brought by the use of LSH as opposed to computing the similarity between every pair of queries. It is interesting to see that the two approaches perform similarly in the early stages with small datasets, but that one algorithm clearly outperforms the other in scenarios with larger datasets.

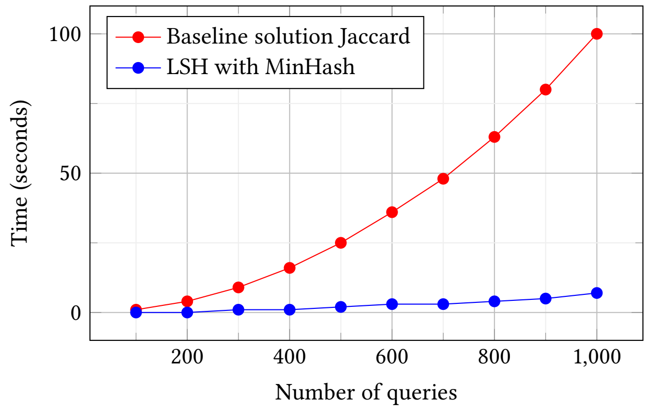

Figure 7: Time performance of the Naive algorithm compared with LSH using MinHash – Variable number of queries

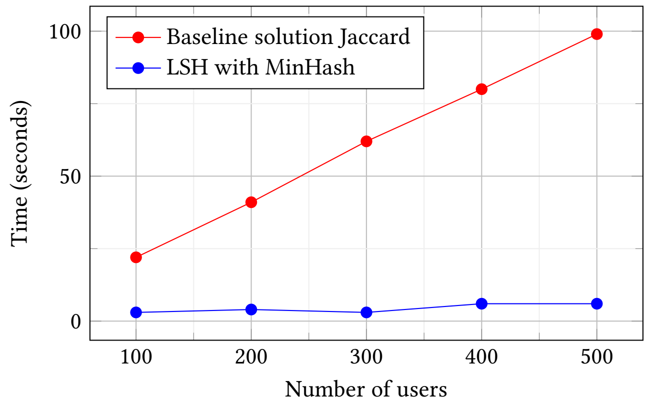

Figure 8: Time performance of the Naive algorithm compared with LSH using MinHash – Variable number of users

5.2.2 LSH with SimHash. The SimHash technique is compared in this section with its naïve counterpart, which computes the cosine similarity between all of the queries in the utility matrix. In this instance, it is possible to observe that leveraging LSH to identify similar queries enhances time performance of the algorithms.

Given that the naïve approach using the Cosine similarity performs better than the technique using the Jaccard similarity, this time the algorithm is evaluated on a dataset with 2000 queries rather than 1000. Furthermore the number of hyperplanes for SimHash has been set at 200 hyperplanes created at random, corresponding to 200 rows of the signature matrix; the latter is divided into 13 bands by LSH increasing the number of rows for each band to 15. The decision to use a high number of rows per band is due to the fact that SimHash creates a signature matrix of zeros and ones, making it simpler for two queries in the same band to hash in the same bucket. Contrarily, MinHash's signature matrix is made up of integers between 1 and the entire number of users in the system (rows of the utility matrix), making it less likely that two columns in the same band would hash into the same bucket.

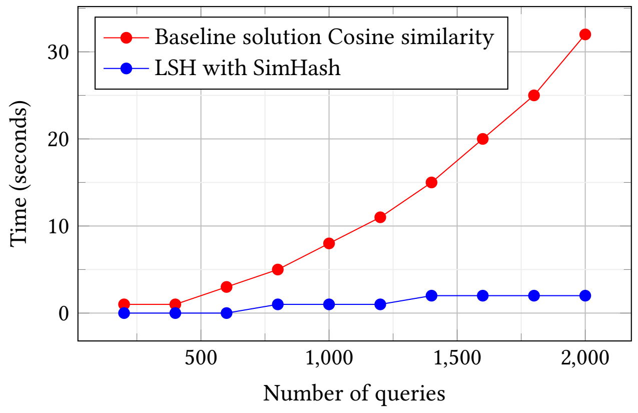

Figure 9: Time performance of the Naive algorithm compared with LSH using SimHash – Variable number of queries

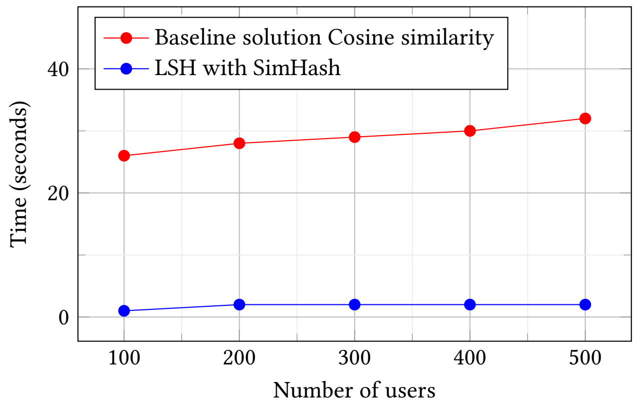

Figure 10: Time performance of the Naive algorithm compared with LSH using SimHash – Variable number of users

5.2.3 LSH parameters tweaks. The parameters specifying the size of each LSH band and the number of rows of the signature matrix have a significant impact on LSH performance. Let's study these two characteristics independently in order to better understand their impact.

Signature matrix rows. The algorithm's time performance will increase as the number of signature matrix's rows decrease, while the probability of finding the wrong most similar query, namely the *error rate*, will decrease as the signature matrix gets smaller. Finding a good balance between the error rate and time performance is crucial, and in border cases it's possible that:

(1) the error rate is low, meaning the algorithm can accurately predict the queries' most similar queries. The issue with this border case is that LSH will likely discover a set of candidate similar pairs corresponding to all the combinations of queries, which means that the algorithm reverts to the naive approach of trying to find the query that is the most similar by comparing the similarities between all the possible pairs of queries.
(2) time performance is the best, meaning the algorithm can find the most similar queries quickly. In this case the issue is related to the accuracy of the result, because with a tiny signature matrix, LSH will probably identify too few candidate similar pairs, which increases the likelihood that LSH will miss pairs of queries that may have been similar, i.e. the error rate is large.

The result is clearly depicted in Figures 11 and Figure 12 where the test was run using LSH with SimHash configured to divide the signature matrix in bands made of a constant number of 12 rows. The time the algorithm took to discover the candidate similar queries as well as the error rate were recorded; the error rate is calculated as the complement of the accuracy, i.e. the fraction of successfully predicted most similar queries over all the other queries.

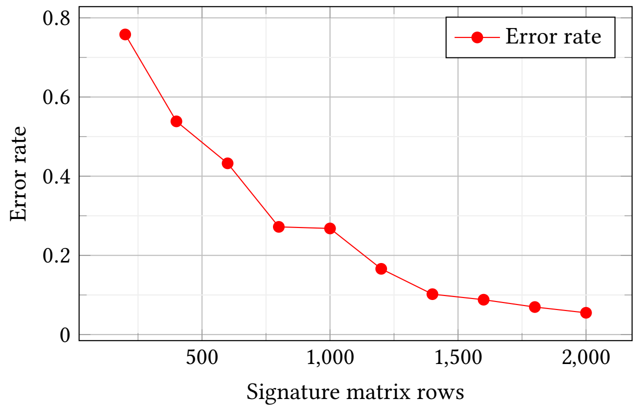

Figure 11: As the number of rows in the signature matrix rises, the error rate reduces.

LSH band size. As already anticipated also in Section 4.2.4 choosing a good number of rows per band or similarly a number of bands that divides the signature matrix are crucial. In particular in the same way shown for the number of rows of the signature matrix it's possible to observe that testing LSH with SimHash configured to have a constant amount of 200 signature rows(number of hyperplanes for SimHash) and a variable number of rows per band, when the number of rows for each band increase:

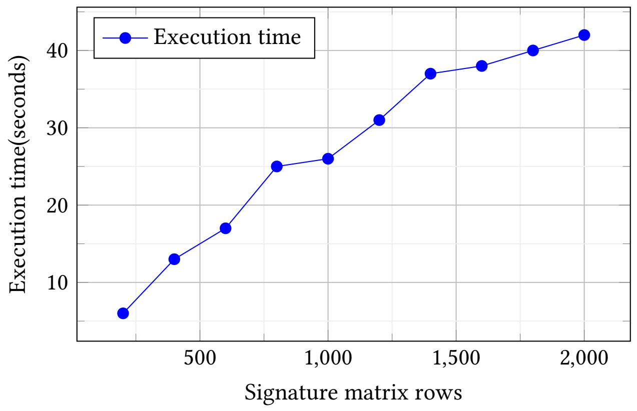

Figure 12: The time required to search for the most similar query grows as the signature matrix's row count rises.

(1) the error rate tend to increase as shown in Figure 13; this is a consequence of the fact that the number of candidate pairs found by LSH decrease.
(2) the algorithm will take less time to find the most similar query as show in Figure 14. This is also caused by the fact that LSH only finds a small number of candidate similar pairs of queries, and thus the time required to compare all the possible candidate pairs is less.

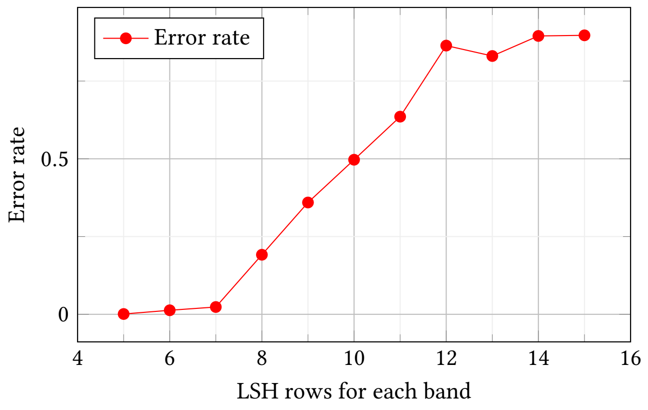

Figure 13: As the size of each band rises, the error rate increases.

As seen in this section, it is crucial to properly adjust the hyperparameters before executing the algorithm. This should be done while taking into account all possible scenarios. The measurements in this section are calculated by iterating over the various value combinations, measuring the time performance of the process step-by-step, and determining the error as the opposite of accuracy.

5.3 Accuracy of the recommendation system

The preceding section analyzed the runtime performance of the algorithm using only collaborative filtering with LSH while tweaking the LSH parameters. However, time performance is not the only aspect that defines a recommendation system's quality. In this section,

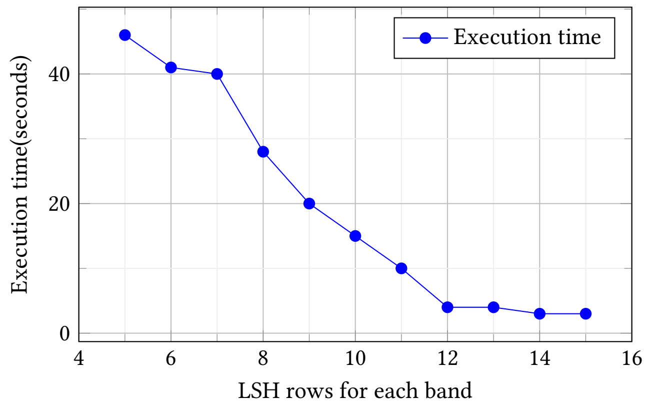

Figure 14: The time required to search for the most similar query decreases as the size of each band increases.

the accuracy of the algorithms described in this paper is measured using two fundamental metrics: *mean absolute error*(MAE) and *root mean square error*(RMSE). The idea is to compare the accuracy on the following versions of the algorithm:

(1) Collaborative filtering recommendation system with LSH using MinHash discussed in Section 4.2.
(2) Collaborative filtering recommendation system with LSH using SimHash discussed in Section 4.3.
(3) Hybrid recommendation system that combines collaborative filtering(using LSH) with a content based approach as discussed in Section 4.4.
(4) Random recommendation system that randomly fills the ratings in the utility matrix.

5.3.1 *MAE and RMSE.*

The mean absolute error and the root mean squared error have been selected as the two metrics to evaluate the model's accuracy. The distinction between the two is that whereas root mean square error measures the average squared difference between the expected and actual ratings, mean absolute error measures the average difference between the predicted and actual ratings. A high value of RMSE or MAE indicates poor performance in terms of the quality of the recommendation system: both should be lowered to have the maximum accuracy.

$$MAE = \frac{\sum_i^n |y_i x_i|}{n} \quad (17) \qquad RMSE = \sqrt{\frac{\sum_i^n (y_i x_i)^2}{n}} \quad (18)$$

5.3.2 *K-fold cross validation.*

The idea for measuring the quality of the recommendations, is to apply the K-fold cross validation approach, which takes a portion of non-zero ratings from the utility matrix to be used as a test set, and then runs the aforementioned algorithms on the utility matrix with the ratings in the test set removed. This total procedure is executed K times, where $K = 10$ has been selected as a suitable value for the number of folds. The average of the two metrics' accuracy throughout the several folds provides the final indication of MAE and RMSE accuracy. Similar results were obtained using the K-folding approach on both the real dataset and the synthetic dataset, indicating the increase in accuracy that follows from using the content-based strategy of Section 4.4 to create a hybrid recommendation system. The histograms of Figures 15 and 16 in particular show the average result of the accuracy metrics mentioned above applied on each fold of both the real and synthetic datasets.

The figures make it clear that the random recommendation system's RMSE and MAE are high in both the datasets, suggesting poor prediction accuracy. The collaborative filtering method is definitely better than the random method, and as the plots demonstrate, the accuracy of the recommendation system increases further when it is combined with a content-based method to form a hybrid recommendation system. The latter is driven by the fact that user ratings are not the only criteria for identifying queries that may be of interest to a user: other aspects of the queries such as the rows they return(their content), are also very important in identifying queries that may be of use to a particular user.

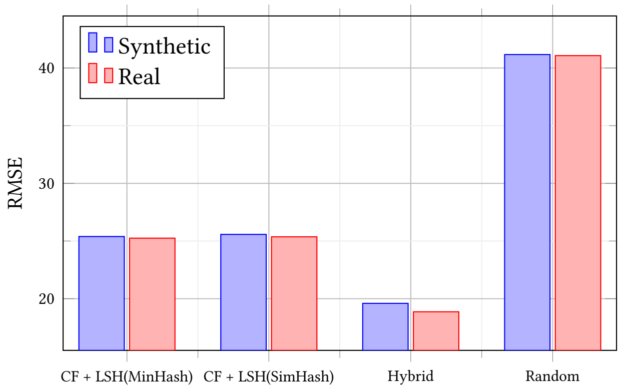

Figure 15: Average RMSE of the different algorithms on the two datasets.

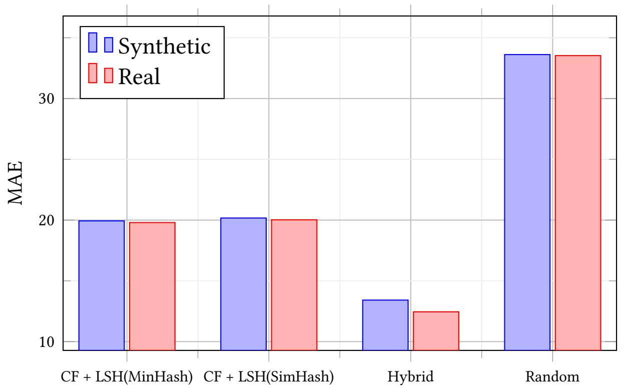

Figure 16: Average MAE of the different algorithms on the two datasets.

It's interesting to note that, for all of the algorithms mentioned above, with the exception of the hybrid algorithm, the accuracy metrics produce results that are identical on both datasets, whereas the hybrid recommendation system appears to produce results that are more accurate on the real dataset. This might be related to the data distribution governing the two datasets.

5.4 Observations

A recommendation system's quality is determined by a number of essential aspects, including time performance and accuracy. Thanks to the experiments conducted in this section, it is possible to conclude that as model accuracy rises, time performance decreases, and vice versa:

- With the random recommendation method, the recommendations are easily calculated and it takes less than one second to fill the utility matrix. However, this approach has the drawback of providing useless recommendations(random).
- When LSH and collaborative filtering are combined together, the recommendation system is more accurate compared to random recommendations, however this comes at the expense of a worsening time performance. In addition, as demonstrated in Section 5.2, SimHash should be preferred over MinHash on big datasets since it allows for a faster execution time while maintaining the same accuracy for the recommendations: as shown in Section 5.3, the accuracy of the two LSH techniques is roughly equivalent.
- The collaborative filtering approach relies solely on information derived from user ratings, without considering the characteristics of the queries. This has motivated the integration of a content-based approach with the already-existing collaborative filtering recommendation system described in Section 4.3 in order to increase the accuracy of the recommendations. The plots in Section 5.3 demonstrate the accuracy improvement and prove that the hybrid recommendation system works better than the approach employing simply collaborative filtering. However this comes at a higher cost to pay in terms of execution time, in fact computing the item and user profiles is costly and may require several minutes compared to the few seconds needed for collaborative filtering.

6 CONCLUSION

The solutions presented and examined in this research produced satisfactory results, demonstrating how the integration of several data mining approaches may result in an advanced recommendation system that can make meaningful recommendations. The final algorithm was built one block at a time, producing two distinct algorithms: collaborative filtering with LSH, which keeps both a high accuracy and good time performance, and another approach, which has an even greater accuracy but worse time performance (hybrid recommendation). The experiments described in Section 5 demonstrated how combining a content-based approach with the currently used collaborative filtering method results in a hybrid recommendation system that generates recommendations that are more accurate. Furthermore it was possible to examine the impact of LSH approaches applied to the collaborative filtering method from the perspective of performance in order to quickly exploit similarities between queries based on the ratings contained in the utility matrix. This methodology demonstrated how using these kinds of strategies enables the collaborative filtering method to operate even on massive datasets. ...

REFERENCES

[1] Barry Becker. 1996. Census Income Data Set . https://archive.ics.uci.edu/ml/datasets/Census%2BIncome.

[2] Andrei Broder. 1997. On the Resemblance and Containment of Documents. *Proceedings of the International Conference on Compression and Complexity of Sequences.* https://doi.org/10.1109/SEQUEN.1997.666900

[3] Edgar F. Codd. 1970. A relational model of data for large shared data banks. *Communication of the ACM* 13, 6 (1970).

[4] E. F. Codd. 1979. Extending the Database Relational Model to Capture More Meaning. *ACM Trans. Database Syst.* 4, 4 (dec 1979), 397–434. https://doi.org/10.1145/320107.320109

[5] Zhiyuan Fang, Lingqi Zhang, and Kun Chen. 2016. Hybrid Recommender System Based on Personal Behavior Mining. https://doi.org/10.48550/ARXIV.1607.02754

[6] Wael H. Gomaa and Aly A. Fahmy. 2013. Article: A Survey of Text Similarity Approaches. *International Journal of Computer Applications* 68, 13 (April 2013), 13–18. Full text available.

[7] Qixia Jiang and Maosong Sun. 2011. Semi-Supervised SimHash for Efficient Document Similarity Search., Vol. 1. 93–101.

[8] J. Prasanna Kumar and P. Govindarajulu. 2013. Near-Duplicate Web Page Detection: An Efficient Approach Using Clustering, Sentence Feature and Fingerprinting. *International Journal of Computational Intelligence Systems* 6, 1 (2013), 1–13. https://doi.org/10.1080/18756891.2013.752657 arXiv:https://doi.org/10.1080/18756891.2013.752657

[9] Jimmie D. Lawson and Yongdo Lim. 2001. The Geometric Mean, Matrices, Metrics, and More. *The American Mathematical Monthly* 108, 9 (2001), 797–812. https://doi.org/10.1080/00029890.2001.11919815 arXiv:https://doi.org/10.1080/00029890.2001.11919815

[10] Gurmeet Singh Manku, Arvind Jain, and Anish Das Sarma. 2007. Detecting Near-Duplicates for Web Crawling. In *Proceedings of the 16th International Conference on World Wide Web* (Banff, Alberta, Canada) *(WWW '07)*. Association for Computing Machinery, New York, NY, USA, 141–150. https://doi.org/10.1145/1242572.1242592

[11] Elaine Rich. October 1979. User modeling via stereotypes. In *Cognitive Science*, Vol. 3. 329–354.

[12] Sadhan Sood and Dmitri Loguinov. 2011. Probabilistic Near-Duplicate Detection Using Simhash. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management* (Glasgow, Scotland, UK) *(CIKM '11)*. Association for Computing Machinery, New York, NY, USA, 1117–1126. https://doi.org/10.1145/2063576.2063737

[13] Du Zou, Wei-jiang Long, and Zhang Ling. 2010. A cluster-based plagiarism detection method - Lab report for PAN at CLEF 2010, Vol. 1176.