The many faces of approximation in KNN graph computation

Olivier Ruas

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

THÈSE DE DOCTORAT DE

L'UNIVERSITE DE RENNES 1
COMUE **U**NIVERSITE **B**RETAGNE **L**OIRE

Ecole Doctorale N°601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *(voir liste des spécialités)*

Par

« Olivier RUAS »

« The many faces of approximation in KNN graph computation »

««»

Thèse présentée et soutenue à RENNES , le 17 décembre 2018
Unité de recherche : INRIA Rennes Bretagne Atlantique
Thèse N° :

Rapporteurs avant soutenance :

Marc Tommasi, Marc Tommasi, Professeur des universités, Université de Lille
Antonio Fernandez, research professor, IMDEA Networks (Madrid)

Composition du jury :

Attention, en cas d'absence d'un des membres du Jury le jour de la soutenance, la composition ne comprend que les membres présents

Président :	Prénom Nom Fonction et établissement d'exercice
Examinateurs :	Marc Tommasi, Professeur des universités, Université de Lille
	Antonio Fernandez, Research Professor, IMDEA Networks (Madrid)
	Sara Bouchenak, Professeure des universités, INSA Lyon
	David Gross-Amblar, Professeur des universités, IRISA (Rennes)
	François Taïani, Professeur des universités, Université de Rennes 1
	Anne-Marie Kermarrec, PDG de Mediego
Dir. de thèse :	François Taïani, Professeur des universités, Université de Rennes 1
Co-dir. de thèse :	Anne-Marie Kermarrec, PDG de Mediego

Invité(s)

ACKNOWLEDGEMENT

Je tiens à remercier
I would like to thank. my parents..
J’adresse également toute ma reconnaissance à ....
....

TABLE OF CONTENTS

INTRODUCTION

The Big Data world

We face today the well-known "Big Data" phenomenon. Every minute, 300 hours of videos are uploaded on YouTube [omnb], 243,000 photos are uploaded on Facebook [omna] and 473,000 tweets are generated on Twitter [sta]. A large part of this data is generated by the users themselves. Users may generate explicit data such as videos, images or posts but they also produce *implicit* data by buying a product or interacting (e.g. liking, sharing, rating) with content. This data (implicit and explicit) is the fuel of modern machine learning techniques. At the same time, such a large volume of data has started to become a problem for the users themselves. Finding the content of interest among the whole dataset is incredibly difficult. A brief description of the content is clearly not enough since going through the entire set of contents is impossible given its size. Consequently users do need help to find interesting content.

How to get around?

This problem is particularly visible in online services. Online services have become the usual way to fulfill many kinds of needs such as watching movies, listening to music or reading the news. These services have to cope with this ever growing amount of available data which prevents users to find the content they are interested in. In this context, personalization is key.

Personalization as a way to find a needle in the haystack for the user. Personalization is the process of adapting a service to each user, based on the data associated with her. For instance, the content of a newsletter may be chosen according to the tastes of a user: a fan of sport will receive sport news, and local news can be added based on the user location. The most widely used approach to personalize is to recommend content to users, be it video, music or news. The most well-known example is probably Netflix, which recommends movies to a user based on her history. Another example is Amazon item recommendation: when you buy a book, Amazon will recommend you the books other users have bought along with this book (Fig. 1). A lot of effort has been put into the development of recommendation algorithms which are becoming increasingly accurate. This interest in this field was increased even further by the Netflix prize [BL+07]. The Netflix prize was a competition launched in 2007 by

Figure 1 – Books recommended by Amazon [ama] when looking at the item "The Art of Computer Programming".

Netflix which aimed at increasing the precision of their item recommendation system by 10%. Every user has a profile which is the set of movies she liked. These movies, which are the features characterizing the user, can be seen as coordinates in a space called feature-space. These coordinates can be assigned a default value (e.g. 1) or be the associated rating the user gave, if available. Every user is then a point in a space which dimension is the total number of movies, i.e. the total number of features. Computations in such a space are expensive, even if the dataset is sparse. It turns out that the method which won the competition and increased by more than 10% the precision was a combination of 107 different algorithms. Netflix implemented the two algorithms providing the best results, but not the rest of them as they estimated that the additional accuracy did not justify the cost of deploying them [net]. This is representative of the trade-off between the quality and complexity faced in machine learning. The more complex the model, the more time spent into learning and the better the quality. The quality is not linear in the complexity nor the computation time. It is important to know when seeking a better quality would result in a prohibitive extra computation time. ***This trade-off between the quality and the complexity is a major concern in machine learning.***

Similar people like similar contents. Many algorithms have been developed to tackle this trade-off, notably in collaborative filtering. Collaborative filtering techniques recommend to a user the items, be them movies, articles, consumer goods, etc..., liked by the users which are similar to her. The intuition is that if Alice had a similar behavior as Bob, then Alice is likely to behave the same way as Bob in the future. For example, we assume that if Alice and Bob liked the same movies, it is likely that Alice and Bob will like the same movies in the future. So any new film liked by Bob can be recommended to Alice. Collaborative filtering differs from content filtering, which predicts the data about users based on the content of the data (e.g. Alice likes action movies, then

she will like the new action movies). Collaborative filtering is widely used to achieve personalization [SLH14] in many contexts (Amazon, Netflix, e-commerce,...), and can be performed by several approaches, the two main ones being matrix factorization and KNN-based approaches.

Matrix factorization [KBV09] (MF) is one of the most famous collaborative filtering techniques. It has been introduced during the Netflix prize and is widely used since then. Matrix factorization tries to predict the rating for each pair (user, movie). Since movies are features which can be seen as coordinates, MF tries to predict the missing coordinates, based on the ones already available. See Sec. 1.2.1 for a detailed presentation of MF.

K-Nearest-Neighbors graphs. The second main class of collaborative filtering approaches rely on K-Nearest-Neighbors (KNN) graphs [DML11, BFG+14]. K-Nearest-Neighbors based approaches build a graph connecting each vertice (representing either an item or a user) to its k closest counterparts according to a given similarity metric. The challenges of the constructions of these graphs lie in finding the nearest neighbors without doing an exhaustive search through the whole dataset. Speeding-up the K-Nearest-Neighbors graph construction is the focus of this thesis.

To take away:
- KNN-based collaborative filtering is a widely adopted approach in online personalization.
- Trading complexity and precision is one of the biggest challenge in the field.

Efficient KNN graph construction

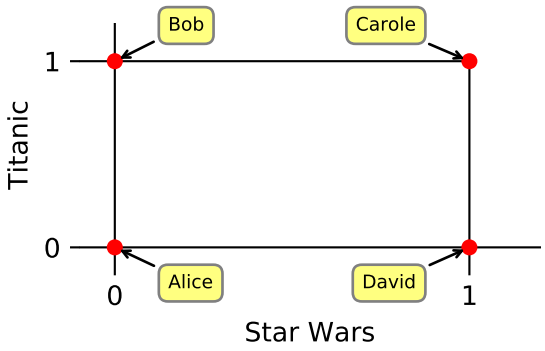
K-Nearest-Neighbors graphs

A K-Nearest-Neighbors (KNN) graph is a directed graph in which every node is connected to its k closest counterparts, called *neighbors*. In other words, every user is linked to the set of k users which are the most similar to her. A similarity metric is used to evaluate how close, i.e. how similar, two users are. The similarity metric appears to be of the utmost importance since it will shape the final KNN graph. The similarity metrics used in practice are based on another set, the feature set. Each feature can be seen as a coordinate, a user is then represented as a point in a space whose dimension is the number of features. The number of features represents the dimension of the problem.

As an example, consider four users, Alice, Bob, Carole and David. They have expressed their tastes about two movies, Star Wars and Titanic, by rating them. The rating

User	Star Wars	Titanic
Alice	0	0
Bob	0	1
Carole	1	1
David	1	0

(a) Complete dataset.

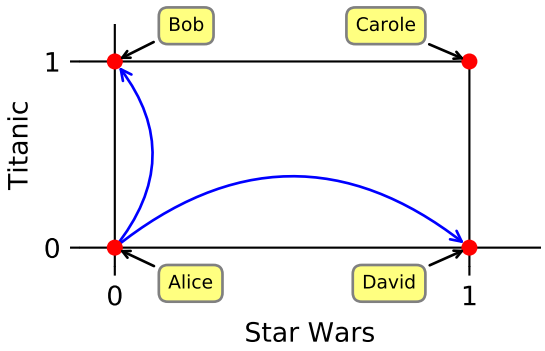


(b) Representation of the users in the feature space.

Figure 2 – Representation of the dataset. Figure 2a shows the users' profiles and Figure 2b represents them in a space where each feature is a dimension, the feature space. Each value in the profile correspond to a coordinate in the feature space.

User	KNN
Alice	{Bob,David}
Bob	{Alice,Carole}
Carole	{Bob,David}
David	{Alice,Carole}

(a) KNN graph of the dataset.



(b) Representation of Alice's 2-NN in the feature space.

Figure 3 – Representation of the KNN graph. Figure 3a shows the complete KNN graph while Figure 3b represents Alice's KNN (in blue in Figure 3a) in the feature space.

1 means they liked the movie and 0 means they did not. The set $\{\text{Star Wars}, \text{Titanic}\}$ is the feature set, so the dimension of the problem is two. The resulting dataset is represented in Figure 2a and 2b. Figure 2a shows a basic representation of the users' profiles. Figure 2b represents their profiles into the feature space: each feature is seen as a dimension and the value of the profiles are coordinates.

The KNN graphs with $k = 2$ are represented in Figures 3a and 3b, according to the classical distance in $\mathbb{R}^2$. Figure 3a shows the KNN of every user. The KNN of Alice (in blue in Figure 3a) is represented in Figure 3b.

KNN for personalization in online services

In online services, KNN graphs are used to associate each user (note that could be item but we focus on user based approaches) to her k most similar users. We

want to compute the KNN graph of users of online services: every node is a user of an online service such as Netflix or Amazon. For example, in Netflix a user would be connected in the graph to the k other users with the most similar tastes. Once the KNN graph is computed, we can perform personalization. Personalization, such as item recommendation, works then by assuming that if most of your neighbors share a common property then there is a high probability that you share it too. For example, if all of the neighbors of a user Alice, which are the users which have the most similar tastes to Alice's, have liked a movie that Alice has not seen yet, then this movie is a good candidate for Alice's Saturday evening. Based on the same assumption, KNN graphs are used for other applications such as classification [GSA+11, NST+14]: a user is classified to the class which is the most represented among its neighbors. Also, KNN graphs can be the first step for more advanced machine learning techniques such as spectral graph analysis or distance-preserving dimension reduction [CFS09].

In online services, the data is highly dynamic and the personalization has to be done in real time. To keep up and produce relevant personalization, the KNN graphs need to be computed as often as possible to be up-to-date. Features are usually the implicit data about the content of the online services: the movies a user has seen, the items she bought, the content she liked and so on. In Netflix the profile of a user is the set of the movies she has seen. Each feature is a movie, and the value is 1 if the user watched it and liked it, 0 otherwise. In Amazon the profile would be the items the user bought. Then, as explained previously, each feature would be an item, the value would be 1 if the user bought it, 0 otherwise. In these cases the dimension of the feature space is the number of items which is high. Also, the number of possible values is two, with most of them set to 0: every user's profile is compound by only a small fraction of the feature set. These datasets are sparse, unlike the datasets used in other field such as image retrieval (see Sec. 1.1.4 for more details).

Because of the sparsity, the closest neighbor may be really far in the feature space. This is one instance of the so-called curse of dimensionality. Most of the existing KNN graphs algorithms were designed for low dimensional and compact datasets. Also the dynamics of the data was not a concern, the computation of the KNN graphs were done offline once and for all. To deal with these constraints, new algorithms and techniques should be used.

Today, building efficiently KNN graphs in sparse and dynamic context remains a very active area of research.

To take away:
Efficient KNN graph computation in large dimension is still an open problem.

Research challenges

The naive way to compute a KNN graph is very simple and consists, for every user, to compute its similarity with every other user in the dataset. Each user then has a list of other users ranked according to their similarity in order to keep the k users with the highest similarity. This brute force approach is optimal but inefficient. This strategy computes a similarity for every pair of users, so it has a quadratic complexity in terms of number of similarity computations. This complexity makes the KNN graph brute force approach impractical for datasets with a huge number of users. Table 1 shows the main characteristics of several widely used datasets. Netflix refers to the dataset provided by the company for the Netflix prize, Amazon refers to a set of reviews crawled on the company website [ML13a] and MNIST [LeC98] is a famous dataset of handwritten digits used for image recognition. The number of items is the number of features: the number of movies for Netflix, the number of items in sale for Amazon and the number of pixels for MNIST. The number of users refers to the number of entities we are looking nearest-neighbors for: users for Netflix, buyers for Amazon and images for MNIST. The number of comparisons is the number of comparisons done to compute an exact KNN graphs using the brute force approach: $\frac{n\times(n-1)}{2}$ where n is the number of users. This number has a quadratic growth and a small dataset such as MNIST requires to compute 10^8 similarities. For larger datasets, the number of required similarities is getting extremely large: 1.15×10^{11} for Netflix and 2.21×10^{13} for Amazon. Since these two datasets are only sampling of the complete datasets these companies have, the real number of similarities is even larger. The brute force approach is clearly not suited for datasets with such a high number of users.

Dataset	Number of items	Number of users	Number of comparisons
MNIST	784	$10,000$	5.00×10^7
Netflix	$17,770$	$480,189$	1.15×10^{11}
Amazon	$2,441,053$	$6,643,669$	2.21×10^{13}

Table 1 – Global characteristics of several datasets.

By using specific datastructures such as KD-trees to organize the dataset, the KNN graph computation can be done efficiently in low dimension [BKL06, LMYG04, Moo00]. Low dimension refers to the dimension of the problem: the dimension of the feature space. In Table 1, the dimension of the dataset is the number of items. MNIST, with its number of items below 1000 has a low dimension, while Netflix and Amazon have a high dimension.

In high dimension, these solutions have the same complexity as the brute force approach: ***computing efficiently a KNN graph in high dimension remains an open problem***. The existing approaches try to overcome the quadratic complexity by not

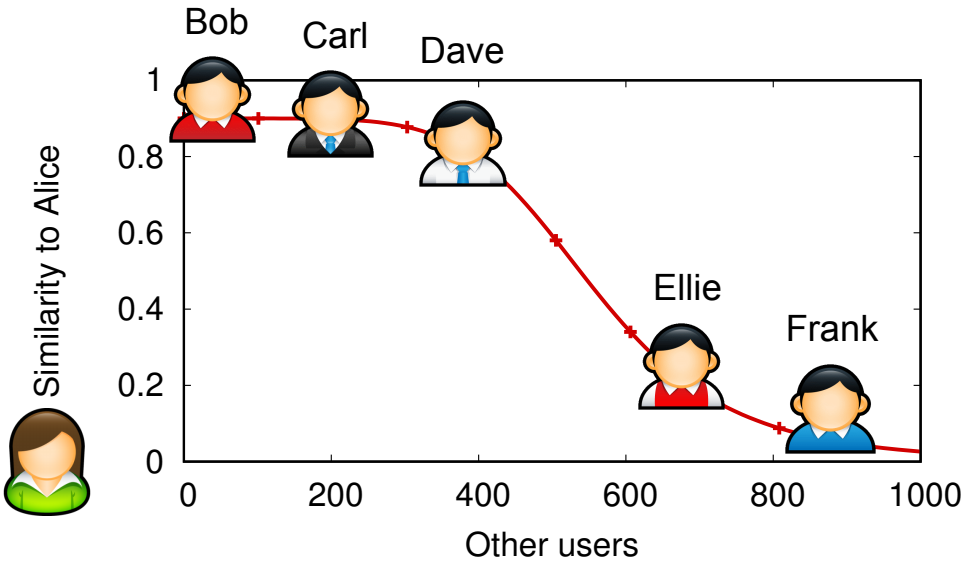


Figure 4 – The first 400 closest users to Alice all show a high similarity with her. If Carl or Dave replaces Bob in Alice's approximate 30-NN neighborhood, Alice's neighborhood quality will only change marginally.

computing all the similarities [IM98, BFG+14, DML11, BKMT16]. This speeds-up the computation but at the risk of missing some neighbors. The obtained graph is not exact but is an approximation: an Approximate-Nearest-Neighbor (ANN) graph.

Constructing an ANN graph aims at finding for each user u a neighborhood that is as close as possible to an exact KNN neighborhood. The meaning of 'close' depends on the context, but in most applications, a good approximate neighborhood is one whose aggregate similarity (its *quality*) comes close to that of an exact KNN set.

For instance, Figure 4 shows the distribution of similarities between a user Alice and other users in a hypothetical dataset. In terms of similarity, the first 400 users (up to Dave) are very close to Alice. If we wish to approximate Alice's 30 closest users, returning any 30 users taken between Bob (the closest) and Carl will yield a neighborhood quality only marginally lower than that of Alice's exact neighborhood.

Though the existing approaches to compute KNN graphs provide approximations, they are still expensive. Even if the computation is done offline and the KNN graphs used only when computed, the cost is still prohibitive for large datasets. The only way to make them work on large datasets is to scale the infrastructure along with the datasets. Yet, even for companies who do not own their infrastructure but rely on the cloud, such scaling can be extremely expensive. Finding more efficient ways to compute KNN graphs for large datasets would grant access to personalization to companies who could not afford it otherwise. Also, computation time has a practical impact when the data freshness is highly valuable. The data may have changed so much during the KNN graph computation that the result itself is already outdated by the end of the computation. In news recommendations for example, users interest may change from

one day to another, with the start of the Olympic games for example. In these cases, it is better to have an approximate but fast-to-compute KNN graph than an exact one.

Existing approaches skip some of the computations of similarity between users, and then eliminate some of the quadratic complexity. Still, their computation times remain high and it seems hard to push in that direction and lower further the number of similarity computations. As the datasets are getting bigger and bigger, whether it be in terms of users or in terms of features, we need to find new, alternative approaches to speed-up the KNN graph computation.

Claim and contributions

Claim:
Existing ANN algorithms base their approximations on the fact that not all candidates are examined. In this thesis we push further the notion of approximation by approximating profiles, the similarity metric and the locality used to cluster the users and run the computations efficiently.

In this thesis, we make the following three contributions:

Approximating the users' profiles: sampling

To illustrate our claim, we first propose an approximation of the profiles through feature sampling. Our first contribution is a new strategy to approximate each user's profile by a smaller one. We sample the users' profiles to keep only a subset of each profile. The goal of sampling is to limit the number of features in the profile of each user, thus limiting the time spent on each similarity computation. We show that keeping the least popular (i.e. the ones with the least number or ratings) features is best of sampling policies. We experimented our new strategy along several other sampling policies and applied them to the state-of-the-art KNN graph algorithms. The less items kept the faster the computation but the lower the quality of the obtained KNN graph. Still, by keeping the 25 least popular features for each user we reduce the computation time by up 63% on AmazonMovies. The resulting KNN graphs are providing recommendations as good as the ones obtained with the exact KNN graphs.

Approximating the similarity: GoldFinger

Our second contribution goes even further: here we **approximate the similarity**. To speed-up the similarity computation, we argue that one should eschew the *extensive*, and often *explicit*, representation of Big Data, and work instead on a *compact*, *binary*,

and *fast-to-compute* representation (i.e. a *fingerprint*) of the entities of a dataset. We propose to fingerprint the set of items associated with each node into what we have termed a *Single Hash Fingerprint* (SHF), a 64- to 8096-bit vector summarizing a node's profile. SHFs are very quick to construct, they protect the privacy of users by hiding the original clear-text information, and provide a sufficient approximation of the similarity between two nodes using extremely cheap bit-wise operations. We propose to use these SHFs to rapidly construct KNN graphs, in an overall approach we have dubbed *GoldFinger*. GoldFinger is *generic* and *efficient*: it can be used to accelerate any KNN graph algorithm relying on Jaccard index, adds close to no overhead, and the size of individual SHFs can be tuned to trade space and time for accuracy. GoldFinger also provides interesting privacy properties, namely k-anonymity and ℓ-diversity. These privacy properties are gained *for free* by compacting the profiles. We show that GoldFinger is able to deliver speed-ups up to 78.9% (on movielens1M) against existing approaches, while incurring a small loss in terms of quality. Despite the small loss in KNN quality, there is close to no loss in the quality of the derived recommendations.

Approximating the locality while clustering: Cluster-and-Conquer

Using the compressed profiles has reduced the computation time of similarity so much that the bottleneck has shifted. Decreasing even further the time spent on similarity computation would result in a negligible improvement compared to the total computation time. The similarity computation is no longer a bottleneck, but memory access can be. Ignoring data locality can be an issue, leveraging it is a solution to minimize the random accesses. Our third contribution is a new algorithm increasing the data locality by relying on a divide-and-conquer strategy. Users are clustered using a new hash function. The hash function is fast-to-compute and approximately preserves the topology of users: similar users tend to be hashed together. Several hash functions used to make sure the users are hashed at least once with their neighbors. KNN graphs are computed locally and independently on the subdataset of each cluster and are then merged together. Cluster-and-Conquer provides speeds-up up to $\times 9$ (on AmazonMovies) compared to the existing approaches relying on raw data. As with GoldFinger, the quality of the derived recommendations is negligible.

Outline

This thesis is organized as follows. We first present in Chapter 1 the useful background about KNN graph computation and compacted datastructures. Then Chapters 2, 3 and 4 present our contributions, respectively approximating the profiles by sampling, approximating the similarity metric with GoldFinger and approximating the

data locality with Cluster-and-Conquer. Finally, we conclude in Chapter 5 and present some perspective and future works.

Publications

1. Nobody cares if you like Star Wars, KNN graph construction on the cheap. Anne-Marie Kermarrec, Olivier Ruas, François Taïani. European Conference on Parallel Processing (Europar), August 29-31 2018.

CHAPTER 1

STATE OF THE ART

In this chapter, we first provide an overview of the existing approaches to build KNN graphs (Sec. 1.1). We then present existing techniques to do item recommendation (Sec. 1.2), which is one of the main applications of KNN graphs. Finally we discuss the main compacted datastructures used in these fields (Sec. 1.3).

1.1 K-Nearest Neighbors Graphs

Intuitively, a KNN graph is a directed graph connecting each user to its most similar counterparts in the dataset. KNN graphs are used in many fields such as item recommendations or classification, but unfortunately they are particularly costly to compute. To reduce the cost, sometimes an approximation of the KNN graph is computed instead of the exact one. In the following we first formally define what the KNN graphs are (Sec. 1.1.1). We then present the metrics usually used to compute how similar users are (Sec. 1.1.2). Finally we explain the existing approaches to compute the approximate KNN graphs and how they speed-up the computation at the cost of a negligible loss of quality in the obtained KNN graphs (Sec. 1.1.3).

1.1.1 Formal definition

We consider a set of users $U = \{u_1, .., u_n\}$ and a set of items $I = \{i_1, .., i_m\}$. For instance, the items might be videos proposed by a streaming website, the books sold on an e-commerce platform or the publications on a social media. We assume every user u has expressed ratings on a subset of I: we note $r(u, i) \in \mathbb{R}$ the rating made by u on the item i. The profile of u, noted P_u, is the set of items for which u has expressed a rating: $P_u = \{i | r(u, i) \text{ is defined}\} \subset I$.

For instance, a dataset with $n = 5$ users and $m = 5$ items is represented in Figure 1.1. The number 5 highlighted in bold in Figure 1.1a means that the user u_3 has given the item i_1 the rating 5, i.e. $r(u_3, i_1) = 5$. A user typically does not rate all the items, for instance the user u_1 did not rate the items i_2, i_4 and i_5. The corresponding profiles are represented in Figure 1.1b. In profiles, only the items with ratings are considered: u_1 has rated two items, i_1 and i_3, its profile is then $\{i_1, i_3\}$. The table in Figure 1.1a may

	i_1	i_2	i_3	i_4	i_5
u_1	1		5		
u_2		3		4	
u_3	**5**		1		
u_4	5		1		2
u_5		3		4	5

(a) Dataset with ratings

u_1	$\{i_1, i_3\}$
u_2	$\{i_2, i_4\}$
u_3	$\{i_1, i_3\}$
u_4	$\{i_1, i_3, i_5\}$
u_5	$\{i_2, i_4, i_5\}$

(b) Profiles.

Figure 1.1 – Dataset with m=5 and n=5. Each line corresponds to a user while each column corresponds to an item.

be represented as the matrix in Fig. 1.2. This matrix is called the user-item matrix. The

$$\begin{bmatrix} 1 & & 5 & & \\ & 3 & & 4 & \\ 5 & & 1 & & \\ 5 & & 1 & & 2 \\ & 3 & & 4 & 5 \end{bmatrix}$$

Figure 1.2 – User-item matrix corresponding to the dataset in Fig. 1.1a

dataset is said to be dense when the matrix is almost full, while it is said to be sparse if the matrix is nearly empty. To estimate the sparsity of the dataset, we compute the density of the dataset: the number of ratings divided by the total number of possible ratings.

$$\text{density} = \frac{\text{\#ratings}}{n \times m}$$

In the previous example we have a density of $\frac{12}{5\times5} = 0.48$, which is high. In this thesis we focus on dataset with low density.

A *k-nearest neighbor* (KNN) graph associates each user u with the set of k other users $knn(u) \subseteq U$ which are closest to u according to a given similarity metric on profiles:

$$\begin{array}{rccc} sim: & U \times U & \rightarrow & \mathbb{R} \\ & (u, v) & & sim(u, v) = f_{sim}(P_u, P_v). \end{array}$$

Any similarity function sim can be used: it can rely on the profiles or the ratings. Intuitively, the higher the similarity, the more similar the profiles and ratings should be. More about usual similarity functions is detailed in Sec. 1.1.2.

Formally, computing the KNN graph results in finding $knn(u)$ for each u such that

$$knn(u) \in \underset{S \subseteq U \setminus \{u\} : |S| = k}{\operatorname{argmax}} \sum_{v \in S} sim(u, v), \tag{1.1}$$

Because a user might have the same similarity with several other users, there might

be more than one such k-tuple. As a result, $knn(u)$ is not necessarily unique for a given u, and some datasets (U, I, r) might possess several KNN graphs for a given similarity function.

The example of the 2009 Netflix dataset:

— U: all Netflix users, $|U| = n = 17,770$
— I: all the movies, $|I| = m = 480,189$
— profiles: all movies watched by users
— #ratings $= 100,480,507$
— density $= 0.000871542$
— KNN: the k other users with the most similar tastes. k is a parameter of the system.

With a density of 0.000871542, this Netflix dataset is said to be sparse.

1.1.2 Similarity metrics

The similarity metric is used to compute how similar two users are and thus, who are a user's nearest neighbors. This similarity metric is of the utmost importance because it determines the topology of the KNN graph. Depending on the data and the objective at hand, many similarity metrics can be used. In this section we present the two main similarity metrics, Jaccard and Cosine.

Jaccard similarity

The Jaccard similarity, also called Jaccard's index or Jaccard's coefficient, has been developed as a metric in statistics to express at what extend two sample sets are similar [Jac01]. It is based on sets of items and does not rely on the ratings. The similarity between two users measures how close their profiles are in terms of size and how much they overlap. More formally the Jaccard similarity between two users u and v is expressed as the size of the intersection of their profiles divided by the size of the union of their profiles:

$$f_{sim}(P_u, P_v) = J(P_u, P_v) = \frac{|P_u \cap P_v|}{|P_u \cup P_v|} \tag{1.2}$$

The similarity ranges from 0 (users do not share any item in common) to 1 (users have exactly the same profile).

For instance, in Figure 1.3 user u_1 has a similarity of 1 with herself and u_3 because they have the same profile. She has no item in common with both u_2 and u_4 so her similarity with them is 0. If we take into account the ratings we observe that u_1 and u_3 have opposed tastes. They have rated the same items but they expressed opposed

u	$f_{sim}(P_{u_1}, P_u)$
u_1	1
u_2	0
u_3	1
u_4	$2/3$
u_5	0

Figure 1.3 – Jaccard similarity of the users with u_1.

tastes about them. Still, their Jaccard similarity is 1 because Jaccard similarity does not rely on ratings but on profiles. The Jaccard similarity is fast to compute compared to other similarity because all it does is computing a set intersection and a set union. It is widely used in document classification [GRS99, Hua08], DNA recognition [SWR+09] and recommender systems [SVZ08].

Cosine similarity

Another widely used similarity is the cosine similarity [AT05]. It is the cosine of the angle between the two profiles, vectors into the item space, in which the ratings play the role of coordinates. A missing rating is interpreted as a coordinate set to 0.

$$f_{sim}(P_u, P_v) = cosine(P_u, P_v) = \frac{\mathbf{r(u)} \cdot \mathbf{r(v)}}{\|\mathbf{r(u)}\|_2 \times \|\mathbf{r(v)}\|_2} = \frac{\sum_{i \in P_u \cap P_v} r(u,i) \times r(v,i)}{\sqrt{\sum_{i \in P_u} r(u,i)^2} \times \sqrt{\sum_{i \in P_v} r(v,i)^2}}$$

where $\mathbf{r(u)}$ is the scalar representation of the ratings of P_u in which every missing ratings is represented by a 0. The $\cdot$ operator is the usual scalar product.

As a cosine, the values range from -1 (opposite ratings) to $+1$ (same ratings), unlike the Jaccard similarity. The similarity $cosine(P_u, P_v)$ between two users u and v is equal to -1 when u and v have the same profile but with opposite ratings: $r(u,i) = -r(v,i)$ for every item in their profiles. In that case, their Jaccard similarity would be 1 because the profiles are the same even though the ratings are not. Unfortunately, the computation of the cosine similarity is more expensive because of the ratings.

To take away:
There are many existing similarity metrics but the most widely used are **Jaccard** and **cosine** similarities. We focus in this thesis on Jaccard because cosine is more expensive to compute.

1.1.3 How to compute KNN graphs: a wide range of algorithms

There are many approaches to compute a KNN graph. Most of them rely on the same process: organizing the data so that the neighbors are easily found, without requiring to go through the whole dataset. When the datasets are too large to fit in the main memory, out-of-core techniques [KBG12, RMZ13, CKO16] optimize the sequential readings. They typically reorganize the data on the disk in such a way that the neighbors are likely to be in the same chunk of data. On the other hand, in-memory approaches maintain datastructures in which neighbors are close to each other. Because of the important memory new hardwares are provided with, in-memory approaches are our main focus in this thesis. In this chapter, we present the main in-memory approaches used to compute a KNN graph in online services. We first present the naive approach which is exact but has a prohibitive complexity. We then discuss the existing solutions used to compute the KNN graph in small dimensions before talking about LSH, an effective approach to compute KNN graphs in online services. We finally present the greedy approaches, based on local search, which are the most efficient in our context and KIFF, an approach specialized in very sparse datasets.

Brute force

The Brute Force algorithm naively computes the similarities between every pair of profiles as shown in Algorithm 1. Each user u has a list $knn(u)$ of other users ranked according to their similarity in order to keep the k users with the highest similarity. The function add updates the knn and return 0 if the knn remains the same, 1 if it has changed. The Brute Force algorithm performs a number of similarity computations equal to $\frac{n\times(n-1)}{2}$. Since the Jaccard similarity between two users u and v is proportional to $O(|P_u \cap P_v|)$ the worst case complexity of the Jaccard similarity is $O(m)$. The overall approach has a complexity of $O(n^2 \times m)$. Fortunately, in practice the profiles' sizes are lower than m so the similarity's complexity is lower than $O(m)$. While this algorithm produces an exact KNN graph, the approach is computationally intensive.

Algorithm 1 Brute force algorithm

```
for i ∈ [1, n] do
    for j ∈ [i + 1, n] do
        s ← sim(u_i, u_j)
        knn(u_i).add(u_j, s)
        knn(u_j).add(u_i, s)
    end for
end for
return knn
```

Brute force approach:
- **Neighbors selection**: computes the similarity with every other user.
- **Number of similarities**: $\frac{n \times (n-1)}{2}$.
- Produces the exact KNN graph.

Small dimensions

In small dimensions, i.e. when the item set I is small, several approaches yielding good performances exist.

As an example, the Recursive Lanczos Bisection algorithm [CFS09] is a divide and conquer method. It divides the dataset into small subdatasets using the Lanczos algorithm, then computes small and inexpensive KNN graphs on the subdataset, and finally merges them. The key element of the algorithm is that the divide keeps neighbors together thanks to the Lanczos algorithm, separating users based on the eigen values of the users-ratings matrix. For more detail on Lanczos algorithm, see Section 1.3.3. The complexity of the Recursive Lanczos Bisection is $O(n \times m^t)$, where n is the number of users and m the number of items, with t higher than one. This complexity makes it inefficient in high dimension, when m becomes large.

Other methods rely on specific datastructures to compute the KNN graphs. The main idea is to rely on a tree, similar to a decision tree, which organizes the users in a spatial hierarchical manner. The root node represents all users, and the space is cut in half at each node in order to group users by similarity. Many datastructures have been develop in such manner: Cover-trees [BKL06], Spill-trees [LMYG04] or KD-trees [Ben75].

Unfortunately, in high dimension these approaches are outperformed by the Brute Force approach in terms of computation time.

Locality sensitive hashing

When the dimension is large, the previous techniques are not efficient. In order to speed-up the KNN graph computation, existing techniques focus on decreasing the quadratic complexity. Not computing all the similarities makes the computation faster but at the risk of missing some neighbors. The resulting KNN graphs are approximate.

Locality-Sensitive-Hashing (LSH) [IM98, GIM+99] is an approach originally designed to answer a KNN request: given a user u, which may not be included in U, we want to return the k most similar users to u in U. KNN request is treated more precisely in Sec. 1.1.4. The algorithm can be used to compute a complete KNN graph by requesting a KNN request for every user of U.

LSH is an approach which lowers the number of similarity computations by hashing each user into several buckets. Every user is put into several buckets and the neighbors are then selected among the users found in the same buckets. The main idea is to use well-chosen hash functions such that similar neighbors have a high probability of ending up into the same buckets. These functions are called Locality-Sensitive-Hashing (LSH) functions. More formally, given a distance d on I and a bucket space B, the hash function $h : I \rightarrow B$ is a LSH function if it satisfies the following properties:

— $d(p, q) \leq R$ then $h(p) = h(q)$ with a probability at least p_1.

— $d(p, q) \geq R$ then $h(p) \neq h(q)$ with a probability at most p_2.

The function h is interesting if $p_1 > p_2$.

LSH functions, in the case of Jaccard similarity are relying on permutations of the item set I, called min-wise independent permutations [Bro97, BCFM00]. LSH functions have the property that the probability that u and v are hashed into the same bucket is thus proportional to their Jaccard similarity: $\mathbb{P}(h(u) = h(v)) = \frac{|P_u \cap P_v|}{|P_u \cup P_v|}$. Each function h is associated with a random permutation p of I. The hash functions are pair wise independent. The permutation p is used to define a total order on the items: for two items i_1 and i_2 we have $i_1 < i_2$ if and only if $p(i_1) < p(i_2)$. The hash of a user is the item corresponding to the minimum of its profile's items by the permutation: $h(u) = min_p(i|i \in P_u)$. Note that users are hashed into buckets corresponding to items they have in their profiles. The probability that two users u and v are hashed into the same bucket is the probability that their minimum by p is the same: there are $|P_u \cap P_v|$ possible items out of their $|P_u \cup P_v|$ total items. The previous property that the probability that u and v are hashed into the same bucket is thus proportional to their Jaccard similarity: $\mathbb{P}(h(u) = h(v)) = \frac{|P_u \cap P_v|}{|P_u \cup P_v|}$.

p	$i_2 < i_5 < i_3 < i_4 < i_1$

Figure 1.4 – Example of permutation p of $I = \{i_1, i_2, i_3, i_4, i_5\}$. The permutation can be seen as a total order on I.

users	profiles	new order by p	hash
u_1	$\{i_1, i_3\}$	$i_3 < i_1$	i_3
u_2	$\{i_2, i_4\}$	$i_2 < i_4$	i_2
u_3	$\{i_1, i_3\}$	$i_3 < i_1$	i_3
u_4	$\{i_1, i_3, i_5\}$	$i_5 < i_3 < i_1$	i_5
u_5	$\{i_2, i_4, i_5\}$	$i_2 < i_5 < i_4$	i_2

Figure 1.5 – Hashing process by using p. The profiles of the set of users $U = \{u_1, u_2, u_3, u_4, u_5\}$ are permuted using p and then the position in the permutation of the minimum is used as the hash of the profile.

Figures 1.4, 1.5 and 1.6 represent this process on $I = \{i_1, i_2, i_3, i_4, i_5\}$. The permutation $p = (i_2, i_5, i_3, i_4, i_1)$ shown in Figure 1.4 can be seen as a total order on I:

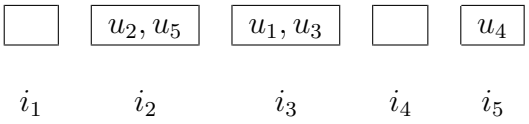


Figure 1.6 – Resulting buckets of the hashing defined in Fig 1.5. u_1 and u_3 are hashed into the same bucket since they have the same profiles while u_4 is not hashed in the same bucket as them even though her profile differs by only one item.

$i_2 < i_5 < i_3 < i_4 < i_1$. The profiles are ordered by the order defined by p and the minimum item, by p, is the hash of the profile. The resulting buckets are represented in Fig. 1.6. If two users have the same profiles, such as u_1 and u_3, they are necessarily hashed into the same bucket. Unfortunately, u_4 is not hashed into the same bucket as them, even though her profile differs from their by only one item, i_5 which was lower by p than i_1 and i_3. To increase the probability that similar users are hashed into the same buckets, we use several hash functions. We note H the number of hash functions used.

p_2	$i_5 < i_4 < i_1 < i_3 < i_2$

Figure 1.7 – Second permutation p_2 of $I = \{i_1, i_2, i_3, i_4, i_5\}$. The permutation can be seen as a total order on I.

users	profiles	new order by p_2	hash
u_1	$\{i_1, i_3\}$	$i_1 < i_3$	i_1
u_2	$\{i_2, i_4\}$	$i_4 < i_2$	i_4
u_3	$\{i_1, i_3\}$	$i_1 < i_3$	i_1
u_4	$\{i_1, i_3, i_5\}$	$i_5 < i_1 < i_3$	i_5
u_5	$\{i_2, i_4, i_5\}$	$i_5 < i_4 < i_2$	i_5

Figure 1.8 – Hashing process by using p_2. The profiles of the set of users $U = \{u_1, u_2, u_3, u_4, u_5\}$ are permuted using p_2 and then the position in the permutation of the minimum is used as the hash of the profile.

Figure 1.7 represents another permutation p_2 and Fig. 1.8 the hashing it provokes. Figure 1.9 represents the final buckets, taking into account the hashing using p. The red users are the users added thanks to p_2, while the black are the ones previously hashed by p. u_4 has been hashed twice into the same buckets so she is only in one buckets. By using p_2, u_4 is not alone in its buckets anymore. The more hash functions, i.e. the higher H, the higher the probabilities that similar users are hashed into the same buckets.

To compute the KNN graphs, we compute the KNN of each user by computing its similarity with all the users which are in the same buckets as her. Being hashed into the bucket associated to the item i requires to have i into one's profile. All the users which are in the same buckets share some items, therefore their similarity is non zero. LSH relies on the hash functions to avoid to compute useless similarity computations,

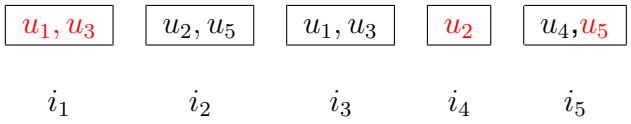


Figure 1.9 – Resulting buckets of the hashing defined in Fig 1.5. Now $H = 2$. Increasing the number of hash functions increases the probability that the neighbors are hashed together at least once.

thus lower the $O(n^2)$ complexity. There is no study of an exact complexity for LSH. We provide a rough estimation of it. Each user is in at most H buckets, filled in average with n/m users, thus a rough estimation of the average number of similarities is $O(n \times H \times \frac{n}{m})$.

Unfortunately, the hash functions and the hash themselves are extremely expensive to compute. Also hashing every user requires to compute the image by a permutation of all users' items. LSH is better for datasets where the item set and the users' profiles are reasonably small.

LSH:
- **Neighbors selection**: computes the similarity with users hashed into the same buckets.
- **Number of similarities**: $O(n \times H \times \frac{n}{m})$ in average.
- Relies on a heavy precomputation and produces an ANN graph.

Greedy approaches: the friend of your friend is probably your friend

In the type of datasets we are interested in this thesis, the item set is particularly large. Also, the number of ratings is so large that LSH is more expensive than the greedy approaches. For the datasets we are interested in, new approaches have been developed: greedy approaches.

The greedy approaches compute the KNN graphs relying on a local search which can be synthesized by the famous saying 'the friend of your friend is probably your friend'. In the context of KNN graphs construction, they assume that the neighbors of a neighbor are more likely to be neighbors than any random user. Greedy approaches start from a random graph and then iteratively refine the neighborhood of each user by computing their similarity with the neighbors' neighborhoods and keeping the most similar among the neighbors of neighbors and the ones in the current KNN. In the brute force approach, the KNN are computed sequentially, user by user. The KNN of the first user will be ready even if the KNN of the last of user is not. In the greedy approaches, the KNN of each user are computed at the same time. The KNN graphs are computed globally, the neighborhoods are converging at the same time for every

user. The neighborhood of a user is able to converge because the neighborhoods of its own neighbors are converging at the same time. We present Hyrec [BFG+14] and NNDescent [DML11], two of such approaches.

Hyrec, presented in Algorithm 2, starts from an initial random graph, which is then iteratively refined to converge to a KNN graph. At each iteration, for each user u, Hyrec compares all the neighbors' neighbors of u with u. The algorithm stops either when the number of updates c during one iteration is below the value $\delta \times k \times n$, with a fixed δ, or after a fixed number of iterations.

Algorithm 2 Hyrec algorithm

```
for u ∈ U do
    knn(u) ← Sample(U, k)
end for
for t ← 1..iter do
    c ← 0
    for u ∈ U do
        for v ∈ knn(u) do
            for w ∈ knn(v) do
                s ← sim(u, v)
                c ← c + knn(u).add(w, s)
            end for
        end for
    end for
    if c ≤ δ × k × n then
        return knn
    end if
end for
```

NNDescent, presented in Algorithm 3, uses a similar strategy to that of Hyrec, exploiting the fact that a neighbor of a neighbor is likely to be a neighbor. As Hyrec, NNDescent starts with a random graph which is then refined. NNDescent primarily differs from Hyrec in its iteration strategy. During each iteration, for each user u, NNDescent compares all the pairs (u_i, u_j) among the neighbors of u, and updates the neighborhoods of u_i and u_j accordingly. NNDescent includes a number of optimizations to avoid computing the same similarities several times such as maintaining update flags to know which users were already in one's neighborhood in the previous iteration. It also reverses the current KNN approximation to increase the space search among neighbors. To avoid to double the number of similarities computed, the reverse graph is sampled. As Hyrec, it stops when the number of changes c is below the value $\delta \times k \times n$, with a fixed δ, or after a fixed number of iterations. NNDescent is shown in Algorithm 3, where $Sample(U, k)$ returns a sample of U of size k. The function $Reverse$ returns, given a graph knn, a graph knn' in which $u \in knn'(v)$ if and only if $v \in knn(u)$.

Greedy approaches approximate the KNN graph to speed-up the computation.

Algorithm 3 NNDescent algorithm

for $u \in U$ **do**
  $knn(u) \leftarrow Sample(U, k)$
end for
for $t \leftarrow 1..iter$ **do**
  $c \leftarrow 0$
  $knn' \leftarrow Reverse(knn)$
  for $u \in U$ **do**
    for $u_1, u_2 \in knn(u) \cup knn'(u)$ **do**
      $s \leftarrow sim(u_1, u_2)$
      $c \leftarrow c + knn(u_1).add(u_2, s)$
      $c \leftarrow c + knn(u_2).add(u_1, s)$
    end for
  end for
  if $c \leq \delta \times k \times n$ **then**
    return knn
  end if
end for

Thanks to the local search, Hyrec and NNDescent highly decrease the number of computed similarities. Their speed-up is due to the fact that the number of candidates for each user's neighborhood has been reduced a lot. The candidate set for the neighborhood of each user is approximated: instead of the whole user set, it is reduced to the neighbors' neighborhoods. In the brute force approach, each user is compared to the $n-1$ others while in the greedy approaches, the candidates are selected locally among the neighbors' neighbors. Since there is at most k^2 neighbors of neighbors for every user, the number of computed similarities in the greedy approaches is $O(n \times k^2 \times iter)$ where $iter$ is the number of iterations. Still, these algorithms are substantially faster than the brute force approach. They have also been shown to work better than the other approach in our context [DML11].

Greedy approaches:
- **Neighbors selection**: compute the similarity with the neighbors of the neighbors.
- **Number of similarities**: $O(n \times k^2 \times iter)$ with $iter$ being the number of iterations.
- Fastest approaches so far, but produce an ANN graph.

KIFF

K-nearest neighbor Impressively Fast and eFficient [BKMT16] (KIFF) is an algorithm for very sparse datasets. Based on the observation that, with many similarities like

Jaccard or Cosine, two users have a non-zero similarity if and only if their profiles share some common items. In very sparse datasets, each item has been rated by few users. By selecting the neighbors among the users having rated the same items, KIFF drastically improves the KNN graph computation.

Conclusions

Table 1.1 summarizes the characteristics of the existing approaches to compute KNN graphs. It turns out that in high dimension greedy approaches are the best so far.

algo	Complexity	precomp	exact	High Dim	Scale
Brute force	$O(n^2 \times m)$	None	✓	×	×
Rec. Lanczos	?	Heavy	×	×	×
KD-trees	?	Heavy	✓	×	×
LSH	$O(n \times H \times \frac{n}{m} \times m)$	Heavy	×	✓	✓
Greedy	$O(k^2 \times n \times iter \times m)$	None	×	✓	✓

Table 1.1 – Characteristics of the existing approaches to compute KNN graphs.

1.1.4 KNN queries

Related but different, answering to KNN queries is a well-known problem. A KNN query [GIM+99] is a request of the k most similar users to a user specified in the request: for a given user u, the KNN of u in U should be returned. It differs from the KNN graph problem because the queries are done upon another set of users V which is unknown beforehand. The main challenge here is to reply to the queries as fast as possible. Preprocessing a datastructure to answer the queries is the general approach to solve this problem.

While in the context of this thesis, we are interested in online services personalization, KNN queries has been widely studied in the context of image retrieval and the challenges wary a lot depending on the context. In image retrieval, KNN queries are used to identify the most similar pictures to a picture specified in the request. In that context, the features are the pixels of the images or the image descriptors (e.g. SIFT [Low04] or GIST [OT01]) [ML09, JDS11]. The feature space is then very dense (as opposed to online services): every image has few undefined coordinates, and each feature has a large set of possible values, i.e. a pixel value is three numbers between 0 and 255.

If the set V of users in which the queries is done is included in the set U of users of the dataset, i.e. $V \subset U$, then for every user of u, computing the KNN graph over U and returning the KNN of u works. But if u is in V but not in U, such KNN does not exists. The KNN graphs are not the most suitable datastructure for the KNN requests.

Greedy approaches, especially cannot be used since they compute the KNN graphs as a whole, with a global convergence of the KNNs.

To speed-up the requests, powerful datastructures other than KNN graphs are computed, such as hash tables [GIM+99, DIIM04] or KD-trees [FBF77, SAH08]. These datastructures are usually computed offline. The most important here is the response time between the query and retrieval of the neighbors, not the computation time of the datastructure. The environment is static, the set of users of the dataset U does not change through time.

For instance, the LSH approach has been developed to answer KNN queries. The LSH buckets are computed offline, before receiving the requests. When a request is received for a user u, this user is hashed into some buckets. Users in those buckets are retrieved and the similarities between them and u are computed to keep the closest ones, which are returned.

Along with KNN graphs and LSH, one commonly used technique to answer the KNN queries for image retrieval is to use product quantization [JDS11]. Images descriptors are compressed using quantization into highly compacted descriptors. These compacted descriptors are used to approximate the similarity between two images. An inverted index is used to avoid the exhaustive search. One key advantage of this technique is that compacted descriptors fit into the main memory, which is not the case for the whole dataset. This lead to fast KNN neighbors retrieval.

The datastructures the KNN queries rely on are costly to compute. If they can be used to compute KNN graphs, their preprocessing time is too long to be used in practice for KNN graph computations, especially in the context of online services.

1.2 Item recommendation

One of the main application of KNN graphs is personalization. There are many personalization techniques but the most widely used is item recommendation. Item recommendation schemes help the users to find the interesting contents among the total item set I. It consists in providing every user with a list of items she is likely to rate positively. The recommended items should not already be in the user's profile.

Most of the item recommendation schemes work as follows: for a given user u, the items u has not been exposed to yet are given a score based on u's profile. The items with the highest scores are recommended to u. First we present Matrix Factorization (Sec. 1.2.1), which models the user-feature matrix in order to fill all the missing ratings. We then discuss KNN based item recommendation (Sec. 1.2.2), which recommends items based on the profiles of the neighbors.

	i_1	i_2	i_3	i_4	i_5
u_1	1		5		
u_2		3		4	
u_3	5		1		
u_4	5		1		2
u_5		3		4	5

(a) Raw dataset

$$\begin{bmatrix} 1 & & 5 & & \\ & 3 & & 4 & \\ 5 & & 1 & & \\ 5 & & 1 & & 2 \\ & 3 & & 4 & 5 \end{bmatrix}$$

(b) User-feature matrix

Figure 1.10 – Dataset with $m = 5$ and $n = 5$. Each line corresponds to a user while each column corresponds to an item.

$$\begin{bmatrix} 1 & r_{12} & 5 & r_{14} & r_{15} \\ r_{21} & 3 & r_{23} & 4 & r_{25} \\ 5 & r_{32} & 1 & r_{34} & r_{35} \\ 5 & r_{42} & 1 & r_{44} & 2 \\ r_{51} & 3 & r_{53} & 4 & 5 \end{bmatrix}$$

Figure 1.11 – Matrix factorization aims at finding the missing values r_{xy}.

1.2.1 Matrix Factorization

We present matrix factorization for completeness, it will not be mentioned any further in the thesis.

Matrix factorization [KBV09] (MF) is one of the most famous collaborative filtering techniques. Matrix factorization tries to predict the rating for each pair (user, item) by assuming that the ratings are the results of a small number of "latent" features, specific to each user and items. For instance, the latent factors in a dataset where the features are movies can be types of movies, e.g. action movies, or the presence of a given actor. Once the factors are found, it is easy to predict the rating for each pair (user, item). This rating will be used as the score to recommend the items: the items with the highest estimated ratings will be recommended. Since movies are features which can be seen as coordinates, MF tries to predict the missing coordinates, based on the ones already available.

Figure 1.10 shows two representations of the dataset introduced in Sec. 1.1.1: Figure 1.10a represents the usual dataset while Figure 1.10b represents the corresponding user-feature matrix. The blanks in the matrix are the missing ratings MF tries to estimate. In Figure 1.11, these ratings are given labels. Matrix factorization aims at finding the values of the r_{xy}.

To fill the missing ratings, we proceed to the decomposition of the matrix into a product of two matrices. The Matrix M is represented as a product of two matrices M_I and M_U. The matrix M_U represents the users and M_I the features (a.k.a. the movies) in a new space of dimension d:

$$M = M_U \times M_I^t$$

$$\begin{bmatrix} 1 & r_{12} & 5 & r_{14} & r_{15} \\ r_{21} & 3 & r_{23} & 4 & r_{25} \\ 5 & r_{32} & 1 & r_{34} & r_{35} \\ 5 & r_{42} & 1 & r_{44} & 2 \\ r_{51} & 3 & r_{53} & 4 & 5 \end{bmatrix} = \begin{bmatrix} v_{11} & v_{12} \\ v_{21} & v_{22} \\ v_{31} & v_{32} \\ v_{41} & v_{42} \\ v_{51} & v_{52} \end{bmatrix} \times \begin{bmatrix} j_{11} & j_{21} & j_{31} & j_{41} & j_{51} \\ j_{12} & j_{22} & j_{32} & j_{42} & j_{52} \end{bmatrix}$$

Figure 1.12 – Matrix factorization aims at finding the r_{xy} solving this linear system.

$$M_U = \begin{bmatrix} v_{11} & v_{12} \\ v_{21} & v_{22} \\ v_{31} & v_{32} \\ v_{41} & v_{42} \\ v_{51} & v_{52} \end{bmatrix}$$

(a) User matrix

$$M_I = \begin{bmatrix} j_{11} & j_{12} \\ j_{21} & j_{22} \\ j_{31} & j_{32} \\ j_{41} & j_{42} \\ j_{51} & j_{52} \end{bmatrix}$$

(b) Item matrix

Figure 1.13 – Representations of the users and the items in the new space of dimension $d = 2$.

Both of these matrices have a small and fixed dimension, which is a parameter chosen arbitrarily by the designer of Matrix Factorization. The values of the matrices M_u and M_I are the parameters of our model. We want to set them to fit the known ratings. Once M_U and M_I are set, the missing values are found by doing the matrix product. Intuitively, the dimension of the matrices represents the dimension of the dataset. If the chosen dimension is too small, then we will have a too simplistic model, thus the model will not be able to fit the data. On the other hand, if the chosen dimension is too big, we will face overfitting: it will fit well with observed data but will poorly predict the missing ratings.

Figure 1.12 shows the decomposition of the matrix of Fig. 1.11 as a product of two matrices of dimension $d = 2$. Figure 1.13 shows the representations of the users and the items in this space.

The interest of MF comes from the fact that the parameters to find by computing the matrices M_U and M_I is $d \times (n + m)$ which is much smaller than the number of missing values, even though it is not true in our example.

A matrix decomposition requires the matrix not to have missing values. That is why the matrices M_U and M_I are computed by doing model optimization. They are typically computed by a gradient descent [KBV09], by minimizing the error between the actual available ratings and their estimations provided by the model. Gradient descent is a wide and active field of research [MS07, RRWN11, ZCJL13, ZCJL13, SBS+13, LWS15].

Once the matrices M_U and M_I are computed, we obtain estimations of the missing ratings. These obtained ratings can be interpreted as the ratings users are likely to give to the items they did not rate yet. Items with the highest estimated ratings are recommended to each user.

The main interest of MF is the small dimension: the computations are much faster in the latent factor space that in the regular feature space. To be accurate, the matrix factorization requires dense dataset. Also, the users and the items (e.g. movies, books) need to be static: the addition of one user or item requires the whole model to be computed from scratch again. MF does not optimize item recommendation but rating prediction. Most of the ratings computed are useless since only a small part of the items which are not rated by a user will be recommended to her.

As said previously, we discussed Matrix Factorization for the sake of completeness, given that the main application to KNN graphs is item recommendation. Even though MF is widely used to do item recommendation, it is not used alone. In practice, hybrid recommender systems are used [Bur02]. They rely on the outputs of several of several "basic recommender systems" such as MF, KNN-based item recommender systems or popularity-based item recommender systems (i.e. the most popular items are recommended). The good performances of MF does not exempt the KNN graph computations. The two approaches are complementary.

A weak point of MF compared to other approaches (e.g. KNN-based or popularity-based) is that the results of the recommendations cannot be explained. Because of the nature of model optimization, there is no clear explanation of why the recommended items were chosen: it can be seen as a black box outputting the recommendations. In popularity-based recommender systems the explanation is simple: these items were recommended because they were the most popular ones, many users liked them and thus it is probable that the user will like them too. The provided motivation helps the users to trust the recommendations.

1.2.2 KNN-based recommenders

Along with MF, KNN graphs are widely used to do item recommendation. Two kinds of KNN graphs can be considered when doing item recommendation: user-based KNN graphs and item-based KNN graphs.

User-based KNN graphs connect each user to its most similar counterparts in the dataset, as explained in Section 1.1.1. The recommended items will be selected among the items in the profiles of the neighbors. Intuitively, the items of the neighbors of a user u should interest u since they have similar interests. More formally, to do item recommendation for u we first compute the KNN graph. Then we select the candidate items $cand(u)$ which can be recommended to u: we gather the items present in the profiles of u's neighbors and which are not in u's profile:

$$cand(u) = \{i | \exists v \in knn(u), i \in P(v) \wedge i \neq P_u\}$$

	u_1	u_2	u_3	u_4	u_5
i_1	1		5	5	
i_2		3			3
i_3	5		1	1	
i_4		4			4
i_5				2	5

(a) Item oriented dataset

i_1	$\{i_1, i_3, i_4\}$
i_2	$\{i_2, i_5\}$
i_3	$\{i_1, i_3, i_4\}$
i_4	$\{i_2, i_5\}$
i_5	$\{i_4, i_5\}$

(b) Profiles of items.

Figure 1.14 – Dataset with m=5 and n=5. Each line corresponds to an item while each column corresponds to a user.

Then we compute for each item i of $cand(u)$ a score $score(u, i)$, using a weighted average of the ratings given by other users in u's KNN:

$$\forall i \in cand(u), score(u, i) = \frac{\sum_{v \in knn(u)} r(u, i) \times sim(u, v)}{\sum_{v \in knn(u)} sim(u, v)}.$$

The items with the highest score are recommended to u.

Unlike Matrix Factorization, recommending items to a user does not require to compute a score for every item. Only the items in the profiles of her neighbors are considered. Also the recommendations are easily explained: the recommended items were chosen for a user u because they are the items the users the most similar to u have liked the most. Still, the whole process of computing the KNN graphs and then make recommendations is expensive.

Item-based KNN Item-based KNN graphs are graphs in which the nodes are not the users but the items: in these graphs, each item is connected to its most similar counterparts. It is the same as user-based KNN graphs but the roles of users and items are inverted. Each item has a profile, which is the set of the users which have rated it. Similarly to user-based KNN graphs, we use a metric to compute a distance between two items. Distances are usually based on the items profiles, or the associated ratings [SKKR01, LSY03]. The same techniques to compute user-based KNN graphs are used to compute item-based KNN graphs.

Figure 1.14 represents the item dataset, based on the same data as Fig. 1.1. Figure 1.14a shows the ratings while figure 1.14b shows the corresponding profiles of the items. By using the Jaccard similarity, $sim(i_1, i_3) = 1$ so the nearest neighbor, i.e. the most similar item, of i_1 is i_3.

The intuition behind item recommendation using item-based KNN graphs is that if a user has liked an item, she should like similar items. For example, a user u whose profile is $\{i_1\}$ would be recommended i_3 since the most similar item to i_1 is i_3. More formally, for a user u, we define a set of candidates $cand(u)$ to recommend, which are

the items the most similar to those in the profile P_u of u:

$$cand(u) = \{i | \exists j \in P_u, i \in knn(j) \wedge i \neq P_u\}$$

The candidates are then ranked by their score, which is the weighted average of the ratings:

$$\forall i \in cand(u), score(u,i) = \frac{\sum_{j \in P_u} r(u,j) \times sim(i,j)}{\sum_{j \in P_u} sim(i,j)}$$

The items with the highest scores are recommended to u.

Similarly to user-based KNN, only a small part of the items the user did not rated receive a score. The recommended items are the ones which are the most similar to the items of the user's profile. In terms of computation time, in the case of a dataset with a lot of users but a small item set, the item-based KNN approach is better than the user-based one. In such case the computation of the KNN graphs on items is faster since the number of items is lower than the number of users.

1.3 Compacted Datastructures

Whether it is for item recommendation of other applications, KNN graphs are expensive to compute. The existing approaches try to limit the number of similarities while computing a KNN graph. An orthogonal approach consists in compacting the data to make it easier to process.

Compacted datastructures are widely used when the available data is too large to be easily accessed, stored or analyzed. How a compacted datastructure works strongly depends on the type of operation is seeks to optimize. In the following we discuss three typical operations for which compacted datastructures have been developed and discuss their performances: set membership (Sec. 1.3.1), frequency estimation (Sec. 1.3.2) and dimension reduction (Sec. 1.3.3). Finally we introduce MinHash (Sec. 1.3.4), a datastructure made to estimate the Jaccard similarity.

1.3.1 Set membership

Set membership query is a widely used set operation. Given an item i and a set S, a set membership query returns whether the item belongs to the set. It is used a lot in online services to check if an item is already in a user's profile, in order not to recommend this item for example. It can also be used to compute the intersection of two sets, e.g. to compute the Jaccard similarity. For large set, storing the whole set and finding a particular item is difficult. To overcome those difficulties, we can use a specific datastructure such as a Bloom filter.

Bloom filters

A Bloom filter [Blo70] is a probabilistic datastructure introduced to test whether an item i is present in a set S. The Bloom filter is probabilistic: when a set membership query returns true, then the item is present in the set *with some probability*. The request may returns true although the item does not belong to the set. On the other hand, if the answer is false, then the item is definitively absent.

Composition: a Bloom filter is composed by a bit array and several hash functions which map the items to the bits.

— $B = (\beta_x)_{x \in [\![1..b]\!]}$: a bit array of b bits, which are all initialized to 0.

— $(h_j)_{1 \leq j \leq p}$: p hash functions of $I \to [\![1..b]\!]$.

The bit array is a compact representation of the set S. Usually, the size b of the bit array is much smaller than the size of S. The hash functions are used to hash the items to the bit array.

Operations: a Bloom filter represents a set in which we can add items and check whether an item is present.

— **Add**: to add an item i to the Bloom filter, the item is hashed using all the hash functions. We obtain p hashes: $(h_j(i))_{1 \leq j \leq p}$. These hashes are indexes in the bit array. The bits corresponding to these indexes are set to 1.

For instance, an empty Bloom filter B with $b = 10$ and $p = 2$ is shown in Figure 1.15. Then an item i_1 is added to B (Fig. 1.16). The item i_1 is hashed by the two hash functions to obtain two indexes, 3 and 6, and the corresponding bits are set to 1. Similarly, an item i_2 is added to B (Fig. 1.17). One of the corresponding bits was already set to 1: there was a collision.

The only operation which updates the bits is the addition of one item, which switch some bit to 1. A bit set to 1 cannot be set back to 0. The bit at the index x is set to one if and only if some item, for which one hash was x, was inserted:

$$\beta_x = \begin{cases} 1 & \text{if } \exists e \in S : \exists j \in [\![1..p]\!], h_j(e) = x, \\ 0 & \text{otherwise,} \end{cases}$$

— **Set membership**: to check if an item i is in the Bloom filter, we proceed as for the addition. We hash the item with all the hash functions, obtaining p indexes of the bit array: $(h_j(i))_{1 \leq j \leq p}$. We return true if all the indexes are set to 1: $\bigwedge_{1 \leq j \leq p} \beta_{(h_j(i))} = 1$.

Figure 1.18 represents the request of set membership of i_3 in B. The item i_3 is hashed by the two hash functions to obtain two indexes: 1 and 7. The corresponding bits are set to 0 so the answer to the query is false.

Set membership properties:

— **No is no**: if a query returns false about the item i, that means that one of the

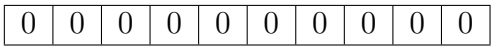


Figure 1.15 – Initial Bloom filter B of size $b = 10$.

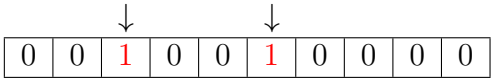


Figure 1.16 – Addition of the item i_1 to the Bloom filter B. where $b = 10$ and $p = 2$. $h_1(i_1) = 3$ and $h_2(i_1) = 6$. The corresponding bits are set to 1.

bits which are associated by a hash function to i is set to 0. It could not have been the case if i were inserted. If the answer is false, then the item is absent.

— **Yes may be no**: if a query returns true about the item i, that means that all the bits which are associated by a hash function to i are set to 1. Unfortunately, it can be caused by collisions. If other items have been inserted and their hashes are the same as the ones associated to i, then the answer is true despite i being absent of the set.

Figure 1.19 represents a set membership query on B for the item i_4. Because of the collisions, all the corresponding bits are set to 1. Despite not being present the answer is true.

With a uniform random hash function, the probability that a given bit is left to 0 while hashing an item i is $1 - \frac{1}{b}$. The probability that this bit is left to 0 by all the hash functions while hashing i is $(1 - \frac{1}{b})^p$. The probability that this bit is left to 0 while hashing N items is $(1 - \frac{1}{b})^{p \times N}$. Thus the probability that this bit is set to one is $1 - (1 - \frac{1}{b})^{p \times N}$. The probability of a false positive, in a Bloom filter in which N items has been inserted is $(1 - (1 - \frac{1}{b})^{p \times N})^p$. For a given b and N, we can compute the optimal p to minimize false positive, usually more than 1. If there are too few hash functions, the collisions are unlikely to happen but have an important impact on the set membership queries. On the other hand, too many hash functions would result in a Bloom filter filled with ones and only returning true to set membership.

The main interest of the Bloom filters is its **constant size**. The size will not change while items are added. Adding items increases the false positive rate of the add operation though.

Bloom filters are widely used and one of their most emblematic use is to optimize cache filtering [BM04]. Using a Bloom filter provides an effective datastructure to know if an URL has already been accessed, which means that the next time the web page can be cached. Other types of Bloom filter can be used such as counting Bloom filter [FCAB00] which replace the bits by counters in order to have a remove operation (the counters are incremented when an item is inserted, decremented when removed), Bloomiers filters [CKRT04] which generalize the standard Bloom filters to

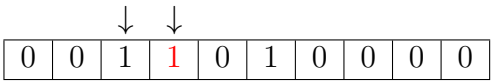


Figure 1.17 – Addition of the item i_2 to the Bloom filter B. $h_1(i_2) = 4$ and $h_2(i_2) = 3$. The bit corresponding to $h_2(i_2)$ was already set to 1: there is a collision.

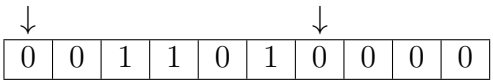


Figure 1.18 – Set membership of the item i_3 to the Bloom filter B. $h_1(i_3) = 1$ and $h_2(i_3) = 7$. The set membership query returns false since the bit corresponding to the hashes are not all set to 1.

support generic set operations or Scalable Bloom filters [ABPH07] which adapt their size while items are inserted, providing a minimum false positive probability.

Set intersection. Bloom filters can be used to estimate the cardinality of the set intersection of two sets. Given the Bloom filters B and B' representing the sets S and S' respectively, we want to estimate the size of the set intersection $S \cap S'$. We assume that B and B' share the same parameters b and p. First the cardinality n_S of S can be estimated by [SB07]:

$$n_S = -\frac{b}{p}\log[1 - \frac{||B||_1}{b}]$$

with $||B||_1$ being the cardinality of B, i.e. its number of bits set to 1. Similarly, with $n_{S'}$ being the cardinality of S' we have:

$$n_{S'} = -\frac{b}{p}\log[1 - \frac{||B'||_1}{b}]$$

From these formulas, and given that $|S \cap S'| = |S| + |S'| - |S \cup S'|$ we can estimate the size $n_{S \cap S'}$ of the intersection $S \cap S'$ by using the corresponding Bloom filters B and B' [SB07]:

$$n_{S \cap S'} = -\frac{b}{p}\log[1 - \frac{||B||_1}{b}] - \frac{b}{p}\log[1 - \frac{||B'||_1}{b}] + \frac{b}{p}\log[1 - \frac{||B^{S \cup S'}||_1}{b}]$$

with $||B^{S \cup S'}||_1$ being the cardinality of $B^{S \cup S'} = (\beta_x^{S \cup S'})_{x \in [\![1..b]\!]}$ the union of the arrays of B and B' defined by:

$$\forall x \in [\![1..b]\!], \beta_x^{S \cup S'} = \beta_x \vee \beta'_x$$

1.3.2 Frequency estimation

Another common problem in the database community is to find the most requested items and computing their frequency. We have a stream of items $S = (i_j)_{1 \leq j \leq q}$ for some unknown q and we want to estimate statistics about the streams: what are the

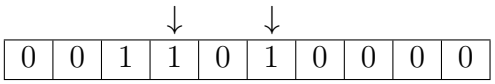


Figure 1.19 – Set membership of the item i_4 to the Bloom filter B. $h_1(i_4) = 7$ and $h_2(i_4) = 4$. Because of the collisions, the set membership will return true even though i_4 has never been added to B.

most frequent items, how many times did they appear or how many distinct items were in the stream. Depending on the statistic we are interested in, specific datastructures have been designed to provide accurate estimations while minimizing the storage.

Count sketches

The count sketch [CCFC02] has been introduced to estimate the most frequent item of the stream while using a very limited storage. We have a stream of items $S = (i_j)_{1 \leq j \leq q}$ for some unknown and large q. We want to estimate, for a given i, the number of times n_i this item appears in the stream. A count sketch is a datastructure designed to provide a good estimation of n_i without storing the whole stream S, neither a counter for each item $i \in I$. A count sketch supports two operations: the addition of an item of the stream and the estimation of the number of times a given item had been added.

Composition: a count sketch is composed by p arrays and hash functions, each hash function mapping the items to a counter of the corresponding array. An extra hash function is used to determine for each item by how much the counters will be updated.

— $(B^j)_{1 \leq j \leq p}$: p arrays. For all $j \in [\![1..p]\!]$, the array $(\beta_x^j)_{x \in [\![1..b]\!]}$ is an array of b counters, which are all initialized to 0.

— $(h_j)_{1 \leq j \leq p}$: p pair wise independent hash functions of $I \to [\![1..b]\!]$.

— $(s^j)_{1 \leq j \leq p} : I \to \{-1; +1\}$: p pair wise independent hash functions. All the hash are independent from each other.

The counter arrays are the compact representation of the stream S. Each array B^j is associated to the hash functions h_j and s_j with the same index. Similarly to the Bloom filter, the hash functions h_j are used to map the items to particular indexes of the array B_j. The difference is that in the count sketch, there is only one hash function per array. A count sketch can be seen as a set of p Bloom filters with only one hash function each.

Figure 1.20 represents an empty count sketch cs with $p = 2$ arrays (shown as horizontal bars) of size $b = 5$.

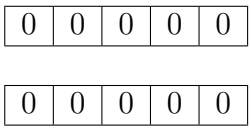


Figure 1.20 – Initial Count-sketch cs with $b = 5$ and $p = 2$.

Operations:

— **Add**: to add an item i to the count sketch, the item is added independently to each array, i.e. to each row. For each row, the item is hashed by the corresponding hash function: for the row B^j, with $1 \leq j \leq p$, we obtain the hash $h_j(i)$. These hashes correspond to positions in the rows: the value $h_j(i)$ corresponds to an index in the table B^j. The value $s_j(i) \in \{-1; +1\}$ is added to the value at the index $h_j(i)$ of the row B^j:

$$\forall j \in [1..p], \beta^j_{h_j(i)} \leftarrow \beta^j_{h_j(i)} + s_j(i)$$

Figures 1.21 and 1.22 show the successive additions to cs of two distinct items i_1 and i_2. The hashes of the items i_1 and i_2 are computed and correspond to indexes in the rows: the first function refers to the first row and so on. The corresponding values are then increased by $s(i_1)$ and $s(i_2)$. Since the two items have the same hash by h_1 there is a collision. The addition of i_2 cancels the effect of the addition of i_1 on the arrays there is the collision because $h(i_2) = -h(i_1)$.

— **Frequency estimation**: for a given item i, each value stored in the position $h_j(i)$, multiplied by $s_j(i)$, of the row j is an estimator of the number of times the item i has been added to the count sketch. Still, these individual estimators have a high variance. To reduce the variance, the final estimator $\widehat{n}_i$ of the frequency of the item i is the median value of the $\beta^j_{h_j(i)} \times s_j(i)$:

$$\widehat{n}_i = \text{median}_{1 \leq j \leq p}(\beta^j_{h_j(i)} \times s_j(i)))$$

Figure 1.23 shows the query on cs of the item i_1. The item is hashed by each hash function and the corresponding values are retrieved: $\{-1, -2\}$. The result is the median of these values, after multiplication by $s_j(i_1)$: median$(\{1, 2\}) = 1.5$

Frequency estimation properties:

— The individual estimators $\beta^j_{h_j(i)} \times s_j(i)$ have a expected value equal to n_i:

$$\mathbb{E}_j[\beta^j_{h_j(i)} \times s_j(i)] = n_i$$

Indeed:

$$\mathbb{E}_j[\beta^j_{h_j(i)} \times s_j(i)] = \mathbb{E}_j[(\sum_{o \in h_j^{-1}(h_j(i))} n_o \times s_j(o)) \times s_j(i)]$$

with o being the items of S whose hash is equal to $h_j(i)$.

$$\mathbb{E}_j[(\sum_{o \in h_j^{-1}(h_j(i))} n_o \times s_j(o)) \times s_j(i)] = \mathbb{E}_j[(\sum_{o \in h_j^{-1}(h_j(i)) \neq i} n_o \times s_j(o)) \times s_j(i)] + \mathbb{E}_j[n_i \times s_j(i) \times s_j(i)]$$

We have

$$\mathbb{E}_j[(\sum_{o \in h_j^{-1}(h_j(i)) \neq i} n_o \times s_j(o)) \times s_j(i)] = \sum_{o \in h_j^{-1}(h_j(i)) \neq i} n_o \times \mathbb{E}_j[s_j(o) \times s_j(i)] = 0$$

by pairwise independence and

$$\mathbb{E}_j[n_i \times s_j(i)) \times s_j(i)] = \mathbb{E}_j[n_i] = n_i$$

— The variance of the estimators $\beta^j_{h_j(i)} \times s_j(i)$ are bounded by $\sum_{o \in h_j^{-1}(h_j(i)) \neq i} n_o^2$ where o are the items of S which hash is equal to $h_j(i)$.

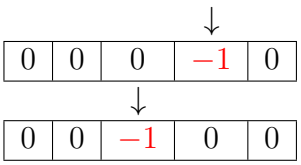


Figure 1.21 – Addition of an item i_1 to cs whose hashes are $h_1(i_1) = 4$ and $h_2(i_1) = 3$, with $s_1(i_1) = s_2(i_1) = -1$. The hash of the first hash function corresponds to an index of the first array, the hash of the second hash function is an index of the second array. The corresponding values are increased by $s_1(i_1) = s_2(i_1) = -1$.

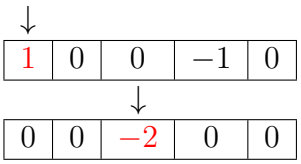


Figure 1.22 – Addition of an item i_2 to cs whose hashes are $h_1(i_2) = 1$ and $h_2(i_2) = 3$, with $s_1(i_2) = 1$ and $s_2(i_2) = -1$. The corresponding values are increased by the $s_j(i_2)$.

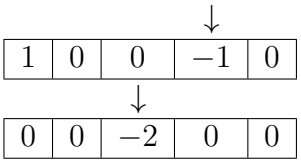


Figure 1.23 – Frequency estimation query of the item i_1 to cs whose hashes are $h_1(i_1) = 4$ and $h_2(i_2) = 3$. The result is the median of the corresponding values weighted by $s_1(i_1) = s_2(i_1) = -1$: 1.5.

The number of hash functions p is usually $O(\log(n))$. The size b of each row depends on the needed accuracy. The larger b, the more accurate the estimation. As for Bloom filters, the benefit of count sketches is their **constant size**: the size does not change when adding items. Still, an increase in the stream will lower the quality of the estimation.

Many other datastructures have been developed to increase the quality of the estimation. One of the most famous ones is the count-min sketch [CM05]. It does not

use the $(s_j)_{1\leq j\leq p}$ hash functions, when an item i is added, the associated counters are increased by one: $\forall j \in [\![1..p]\!], \beta^j_{h_j(i)} \leftarrow \beta^j_{h_j(i)} + 1$. The counters are containing only positive values, which was not the case in the count sketch since the $s_j(i)$ can be negative. The estimation is not obtained by the medium anymore but by returning the minimum of the counters associated to i: $min_j(\beta^j_{h_j(i)})$. For a given error on the accuracy of the estimators, the space needed by count-min sketches is an order of magnitude lower than the one needed by count sketch [CM05].

HyperLogLog

HyperLogLog [FFGM07] is a datastructure developed to estimate the number of distinct elements in a stream S. For a given stream of items $S = (i_j)_{1\leq j\leq q}$ we are interested in the number n of distinct elements in S, without storing the whole stream neither having a counter for each item in the stream. HyperLogLog is an extremely efficient datastructure to solve that problem in one pass on the stream.

Intuition: HyperLogLog relies on the observation that the cardinality of a set of numbers in a binary representation can be estimated by considering the maximum of the position of the leftmost bit set to one. Given a random real number of the unit interval, we can map it into $\{0,1\}^\infty$. We are interested in the probability that the first bit set to 1 is at a given position n. The probability that the rightmost bit is set to 1 is $1/2$. The probability that its first bit set to 1 is at the position 2 is $1/4$ because it is the probability that the first bit is set to 0 multiplied by the probability that the second bit is set to 1. More generally, if we have uniformly distributed real numbers of the unit interval, mapped into $\{0,1\}^\infty$, the probability that the first bit set to one is at the position n is $\frac{1}{2^n}$.

Now consider a set T of uniformly distributed real numbers of the unit interval. They are mapped into $\{0,1\}^\infty$ and the probability of their first bit set to one is at the position n is thus $\frac{1}{2^n}$. We are interested in the maximum of these positions. For this maximum to be n, it requires, in average, the set to have 2^n numbers. Then if the maximum is n, an estimation of the cardinality is 2^n.

HyperLogLog maps each item of S to a real number of the unit interval, which is then mapped into $\{0,1\}^\infty$. The hashing is done uniformly on the unit interval. The maximum n left most position bit set to 1 to each number is kept. The estimator of the cardinality of S is then 2^n. To lower the high variance of that estimator, the items are randomly grouped and a maximum is stored for each group. The harmonic mean of the group estimator is returned.

Composition:

— M: an array of m integers, which are used to store the maximum of the positions of the leftmost bits encountered. Several integers are used to decrease the variance by using several estimators. $m = 2^b$ with $b \in \mathbb{N}^*$.

— $h : I \to \{0,1\}^\infty$: a uniform hash function. h is used to transform the items in the stream into a set of binary numbers, uniformly distributed in $\{0,1\}^\infty$.
— $\rho : \{0,1\}^\infty \to \mathbb{N}^*$: $\rho(i)$ returns the position of the first bit set to 1 of i.

Operations:

— **Add**: To add the item i, we hash it using h. The first b bits of $h(i)$ are used as an index to select which integer in M to modify. The maximum between $M[h(i)_{[1..b]}]$ and the position of the first one in the rest of the bits is assigned to $M[h(i)_{[1..b]}]$: $M[h(i)_{[1..b]}] \leftarrow max(M[h(i)_{[1..b]}], \rho(h(i)_{[b+1..]}))$
— **Cardinality estimation**: returns $\alpha_m \times m^2 \times (\sum_{1 \leq j \leq m} 2^{-M[j]})^{-1}$, with α_m a value which depends on m which corrects a multiplicative bias.

Each value in M is an estimator of a substream of approximately $\frac{n}{m}$ distinct items. Then their values should be close to $\log_2(\frac{n}{m})$. The harmonic mean (which is $m \times (\sum_{1 \leq j \leq m} 2^{-M[j]})^{-1}$) of the $2^{M[j]}$ is of the order $\frac{n}{m}$. By multiplying the harmonic mean by m we obtain an estimator of the order of n. Only the number of distinct items will be counted since the addition of an item i done a second time will not change anything: the hash will be the same and so will be the value of the position leftmost bit set to 1.

HyperLogLog is extremely compact in terms of storage: it can estimate cardinalities beyond 10^9 with accuracy of 2% with only 1.5 kilobytes [FFGM07].

HyperLogLog is used to estimate extremely large stream of items. It is widely used in networking and traffic monitoring. For instance, it can be used to estimate the number of unique visitors on a webpage for example.

1.3.3 Dimension reduction

Previous datastructures aimed at solving very specific problems while using a small amount of memory. More general techniques exist to compress the data for general purposes. Dimension reduction is a common technique to compress data. It is based on Johnson-Lindenstrauss lemma [Ach03, DG03] which states that if n points are represented in a high dimensional Euclidian space, then they can be mapped into a $O(\log(n)/\epsilon^2)$ space in which the distances between the points are changed by a factor at most $(1 \pm \epsilon)$. Out of the many ways to reduce the dimension of the data, we focus here on the main techniques used in personalization techniques: random projection, Lanczos algorithm and Matrix Factorization.

Random projection

Random projection is the simplest dimension reduction technique. It randomly projects the feature space onto a smaller space of dimension d. The n points in the initial space of dimension m are represented by an $n \times m$ matrix M. The points in the compressed

$$M_U = \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ u_4 \\ u_5 \end{bmatrix} = \begin{bmatrix} U_{11} & U_{12} \\ U_{21} & U_{22} \\ U_{31} & U_{32} \\ U_{41} & U_{42} \\ U_{51} & U_{52} \end{bmatrix}$$

Figure 1.24 – Compact representations of the users from Fig. 1.13 in the new space of dimension $d = 2$.

space are obtained by multiplying M by a random projection matrix R. R has a dimension $m \times d$ and each column has unit length. In practice, R can be obtained using a Gaussian distribution [AC09] or more complex schemes [Ach01]. Random projection is simple and computationally efficient. One of contribution, GoldFinger (see Chap. 3), rely on a simpler and binarized version of random projection.

Lanczos algorithm

Lanczos algorithm [Lan50] projects the feature space onto the subspace defined by the d most useful eigenvalues of the feature space. First the algorithm finds the singular-values of the user-feature matrix and then projects the matrix on the space generated by the d highest singular-values. Usually d is an order of magnitude smaller than the feature space dimensions. Lanczos algorithm is more costly than random projection but the projection is more interesting because the new dimensions are not random, they are the most useful eigenvalues of the feature space.

Matrix Factorization

Matrix Factorization (see Sec. 1.2.1) can be used to do dimension reduction. MF aims at computing two matrices M_U and M_I such as:

$$M = M_U \times M_I^t$$

where M is the user-item matrix. The dimension of M_U is $n \times d$ with d an arbitrary value. d represents the intrinsic dimension of the data, the dimension of the space in which U and I will be embedded into.

M_U is a compressed representation of the data of the users, the i^{th} line of M_U is the compacted representation, in d dimension, of the user u_i.

Figure 1.24 shows the resulting M_U of the example in Fig. 1.13. Each line corresponds to the compacted data of a user.

1.3.4 MinHash (BBMH)

Having two sets $A \subset I$ and $B \subset I$, MinHash [Bro97] is a datastructure designed to estimate the Jaccard similarity $J(A,B) = \frac{|A \cap B|}{|A \cup B|}$, using the less storage possible. It can be used to compress the profiles in a computation of KNN graphs based on the Jaccard similarity. MinHash is based on the same hashing mechanism as LSH (Sec 1.1.3). In LSH, the hash of the profile is used to map the user into a given bucket while in MinHash the hash is used as compacted representation of the profiles.

Intuition: MinHash relies on minwise independent permutations [Bro97, BCFM00] $(\pi_j)_{1 \leq j \leq p}$: each π_j is a permutation on I which provides a new total order on I, thus on A and B. We have $\mathbb{P}(min(\pi_j(A)) = min(\pi_j(B))) = \frac{|A \cap B|}{|A \cup B|}$. By using all the permutations, $\frac{1}{p} \sum_{1 \leq j \leq p} \mathbb{1}(min(\pi_j(A)) = min(\pi_j(B)))$ is an unbiased estimator of $J(A,B)$. These functions are the same as the ones used in LSH [IM98, GIM+99].

MinHash stores, for every set A, a fixed number p of items. Every item stored is the minimum by the order induced by a minwise independent permutation of the set: $(\pi_j(A))_{1 \leq j \leq p}$. As explained previously (Sec. 1.1.3), the estimator of the Jaccard similarity between A and B is:

$$\frac{1}{p} \sum_{1 \leq j \leq p} \mathbb{1}(min(\pi_j(A)) = min(\pi_j(B)))$$

The variance is $\frac{1}{p} \times J(A,B) \times (1 - J(A,B))$ and decreases when the number of permutations used increases.

In order to avoid storing the item entirely, b-bit Minwise Hashing [LK11] (BBMH) introduced an improvement to compact even further the representation of the sets. Instead of storing the item entirely, BBMH only stores the lowest b bits of it. The estimator becomes:

$$\frac{1}{p} \sum_{1 \leq j \leq p} (\prod_{1 \leq x \leq b} \mathbb{1}(min(\pi_j(A))_x = min(\pi_j(B))_x))$$

with $min(\pi_j(A)_x)$ being the x^{th} lowest bit of $min(\pi_j(A))$. Unfortunately, this estimator is biased. To remove the bias, the results returned is $\frac{e - c_1}{1 - c_2}$ with c_1 and c_2 two values which depends on the $|I|$, $|A|$, $|B|$ and b. Storing only b bits introduces a trade-off for the variance. Li et al. shows that in practice, when the similarity is high enough, BBMH can provide a 21 fold improvement in storage space by storing 1 bit per item instead of 64.

MinHash and BBMH can be used anytime the Jaccard similarity between two sets are required such as similarity search. In particular they can be used to compress the dataset while computing a KNN graph relying on the Jaccard similarity, the topic of this thesis.

1.3.5 Conclusion

Table 1.2 summarizes the compacted datastructures presented. We can observe that none of the datastructures can be used to speed-up the computation of KNN graphs. MinHash is the only datastructure estimating Jaccard similarity but its pre-computation time is prohibitive.

Datastructure	Jaccard	Set inter	Cst size	Add user	Add item	Add rating	precomp
Bloom filter	×	✓	✓	✓	✓	✓	Light
Count sketches	×	×	✓	✓	✓	✓	Light
HyperLogLog	×	×	✓	✓	✓	✓	Light
MinHash	✓	×	✓	✓	×	✓	Heavy
MF	×	×	✓	×	×	×	Heavy
Q	×	×	✓	✓	×	×	Heavy

Table 1.2 – Characteristics of the existing compacted datastructures.

1.4 Conclusion

The datasets we focus on in this thesis are so large in terms of number of users, items and ratings that using a brute force approach is prohibitive. Existing approaches seek to lower of the number of comparisons between users to speed-up the computation. The greedy approaches are the fastest so far. Still, they spend up to 90% of their total computation time computing similarities [BKMT16]. An orthogonal approach would be to decrease the computation time of each similarity computation by using compacted datastructures. Among all the existing datastructures, only MinHash is designed to provide an estimation of the Jaccard similarity. Unfortunately, the preprocessing required in MinHash is prohibitive when computing a KNN graph.

CHAPTER 2

SAMPLING: WHY NOBODY CARES IF YOU LIKE STAR WARS

2.1 Introduction

In Chapter 1.1, we surveyed the existing approaches to compute KNN graphs. They are using pre-indexing mechanisms or greedy incremental strategies to approximate the set of candidates for each users' neighborhood by small subset of users. These set of candidates are so small that the number of similarity computations is highly reduced. However, it seems hard to lower even further that number.

In this chapter we focus on an orthogonal approach, and instead of approximating the set of candidates, we **approximate the users' profiles**. We leverage *sampling* as a preliminary pruning step to accelerate the time to compute similarities between two entities. Our proposal stems from the observation that many KNN graphs computations are performed on entities (users, documents, molecules) linked to items (e.g. the web pages a user has viewed, the terms of a document, the properties of a molecule). In these KNN graphs, the similarity function is expressed as a set similarity between bags of items (possibly weighted), such as Jaccard's coefficient or cosine similarity. The only existing compacted datastructure designed to estimate the Jaccard similarity is MinHash (see Sec. 1.3.4). Unfortunately, it was designed to optimize space and not computation time: its preprocessing is prohibitive. To avoid such a preprocessing we perform the simplest compacting scheme possible: sampling. The goal of sampling is to limit the size of these bags of items on which is the similarity is computed and thus the time to compute the similarity. Each profile is approximated by a small subset of the items initially present.

Sampling might however degrade the resulting approximated KNN graph to a point where it becomes unusable, and must therefore be performed with care. In this paper we propose to sample the bags of items associated with each entity to a common fixed size s, by keeping their s *least popular* items. Our intuition is that less popular items are more discriminant when comparing entities than more popular or random items. For instance, the fact that Alice enjoys the original 1977 *Star Wars* movie tells us less about her tastes than the fact she also loves the 9 hour version of Abel Gance's 1927

Napoléon movie because many other people liked Star Wars.

We compare our novel policy against three other sampling policies: (i) keeping the s most popular items of each entity, (ii) keeping s random items of each entity, and (iii) sampling the universe of items, independently of the entities. We evaluate these four sampling policies on four representative datasets. As a case study, we finally assess the effects of these strategies on recommendation. Our evaluation shows that our sampling policy clearly outperforms the other ones both with respect to computation time and resulting quality: keeping the 25 least popular items reduces the computational time by up to 63%, while producing a KNN graph close to the ideal one. The recommendations achieved by using the resulting KNN graphs are moreover as good as the one relying on the exact KNN graph on all datasets.

The rest of this chapter is organized as follows. In Section 2.2 we formally define the context of our work and our approach. The evaluation procedure is described in Section 2.3. Section 2.4 presents our experimental results and we conclude in Section 2.5.

2.2 Approximating the profiles to reduce KNN computation time.

2.2.1 Intuition

A large portion of a KNN graph's construction time (be the graph approximate or exact) often comes from computing individual similarity values (up to 90% of the total construction time in some recent approaches [BKMT16]). This is because computing explicit similarity values on even medium-size profiles can be relatively expensive. For instance, Figure 2.1 shows the time required to compute Jaccard's index

$$J(P_1, P_2) = \frac{|P_1 \cap P_2|}{|P_1 \cup P_2|}$$

between two random user profiles of the same size, depending on this size. The profiles are randomly selected from a universe of 1000 items, and the measures taken on an Intel Xeon E5420@2.50GHz. The cost of computing a single index is relatively high even for medium-size profiles: 2.7 ms for two random profiles of 80 items, which is typical of the average profile size of the datasets we have considered.

As said previously (see Section 1.1.3), existing KNN graph construction approaches only perform a fraction of the similarity computations required by an exhaustive search and are easily parallelizable, but it is now difficult to see how their greedy component could be further improved.

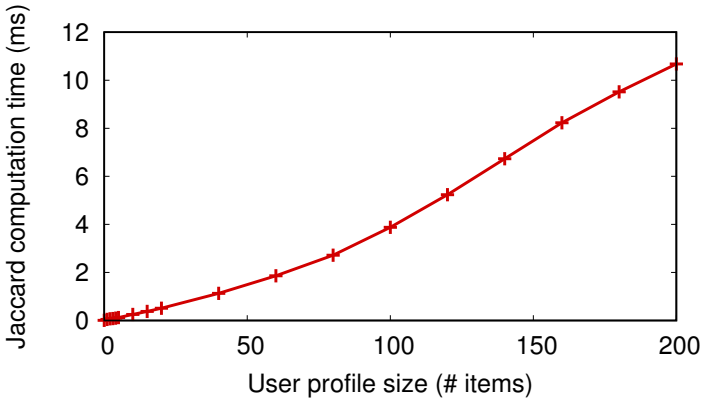


Figure 2.1 – The cost of computing Jaccard's index between explicit user profiles is relatively high (a few *ms*) for average-size profiles. Cost averaged over 4.9 million computations between randomly generated profiles on an Intel Xeon E5420@2.50GHz.

In order to overcome the inherent cost of similarity computations, we therefore propose to target the data on which computations run, rather than the algorithms that drive these computations. This strategy stems from the observation the set intersection is the only non-trivial operation required to compute the Jaccard similarity. Indeed, $|P_u \cup P_v| = |P_u| + |P_v| - |P_u \cap P_v|$ and we can store $|P_u|$ for every user. By approximating the profiles, we have smaller profiles and thus a faster similarity.

2.2.2 Gance's Napoléon tells us more than Lucas's Star Wars

Computing the intersection $P_u \cap P_v$ is time consuming for large sets and is the main bottleneck of Jaccard's similarity. To reduce the complexity of this operation, we propose to sample each profile P_u into a subset $\widehat{P}_u$ in a preparatory phase applied when the dataset is loaded into memory, and to compute an approximated KNN graph on the approximated profiles.

Although simple, this idea has surprisingly never been applied to the computation of KNN graphs on entity-item datasets. Sampling carries however its own risks: if the items that are most characteristic of a user's profile get deleted, the KNN neighborhood of this user might become irremediably degraded. To avoid this situation, we adopt a *constant-size sampling* that strives to retain the *least popular items* in a profile.

The intuition is that unpopular items carry more information about a user's tastes than other items: if Alice and Bob have both enjoyed Abel Gance's *Napoléon*—a 1927 silent movie about Napoléon's early years—they are more likely to have similar tastes, than if they have both liked *Star Wars: A New Hope*—the 1977 first installment of the series, enjoyed by 96% of users [1].

1. `https://www.rottentomatoes.com/m/star_wars`, accessed 21 Feb. 2018

2.2.3 Our approach: Constant-Size Least Popular Sampling (LP)

More formally, if the size of the profile of a user u is larger than a parameter s, we only keep its s least popular items

$$\widehat{P}_u \in \underset{S \in \mathcal{P}_u^s}{\operatorname{argmin}} \sum_{i \in S} pop(i), \tag{2.1}$$

where $\mathcal{P}_u^s$ is the set of subsets of P_u of a given size s, i.e. $\mathcal{P}_u^s = \{S \in \mathcal{P}(I) : |S| = s \wedge S \subseteq P_u\}$, and $pop(i)$ is the popularity of item $i \in I$ over the entire dataset:

$$pop(i) = |\{u \in U : i \in P_u\}|. \tag{2.2}$$

If the profile's size is below s, the profile remains the same: $\widehat{P}_u = P_u$.

In terms of implementation, we compute the popularity of every item when reading the dataset from disk. We then use Eq. (2.1) to sample the profile of every user in a second iteration. The sampled profiles are finally used to estimate Jaccard's similarity between users when the KNN graph is constructed:

$$\widehat{J}(P_u, P_v) = J(\widehat{P}_u, \widehat{P}_v) = \frac{|\widehat{P}_u \cap \widehat{P}_v|}{|\widehat{P}_u| + |\widehat{P}_v| - |\widehat{P}_u \cap \widehat{P}_v|} \tag{2.3}$$

2.3 Experimental Setup

2.3.1 Baseline algorithms and competitors

Our Constant-Size Least Popular sampling policy (*LP* for short) can be applied to any KNN graph construction algorithm such as Hyrec, NNDescent (Sec. 1.1.3) or LSH (Sec. 1.1.3). For simplicity, we apply it to a brute force approach (Sec. 1.1.3) that compares each pair of users and keeps the k most similar for each user. This choice helps focusing on the raw impact of sampling on the computation time and KNN quality, without any other interfering mechanism.

We use full profiles for our baseline, and compare our approach with three alternative sampling strategies: *constant-size most popular*, *constant-size random*, and *item sampling*.

Baseline: no sampling

We use our brute force algorithm (Sec. 1.1.3) without sampling as our baseline. This approach yields an exact result, which we use to assess the approximation introduced by sampling, and provide a reference computing time.

Dataset	Users	Items	Scale	Ratings > 3	$\lvert P_u \rvert$	$\lvert P_i \rvert$	Density
movielens1M	$6,038$	$3,533$	1-5	$575,281$	95.28	162.83	2.697%
movielens10M	$69,816$	$10,472$	0.5-5	$5,885,448$	84.30	562.02	0.805%
movielens20M	$138,362$	$22,884$	0.5-5	$12,195,566$	88.14	532.93	0.385%
AmazonMovies	$57,430$	$171,356$	1-5	$3,263,050$	$56,82$	19.04	0.033%
DBLP	$18,889$	$203,030$	5	$692,752$	36.67	3.41	0.018%
Gowalla	$20,270$	$135,540$	5	$1,107,467$	54.64	8.17	0.040%

Table 2.1 – Description of the datasets used in our experiments

Constant-size most popular sampling (MP)

Similarly to LP, MP only keeps the s most popular items of each profile P_u:

$$\widehat{P}_u \in \operatorname*{argmax}_{\mathcal{S} \in \mathcal{P}_u^s} \sum_{i \in S} pop(i). \tag{2.4}$$

As with LP, we do not sample the profile if its size is lower than s.

Constant-size random sampling (CS)

This sampling policy randomly selects s items from P_u, with a uniform probability. As above, there is no sampling if the size of the profile is lower than s. In terms of implementation, this policy only requires one iteration over the data.

Item Sampling (IS)

This last policy uniformly removes items from the complete dataset. More precisely, each item $i \in I$ is kept with a uniform probability p to construct a reduced item universe $\hat{I}$ (i.e. $\forall i \in I : \mathbb{P}(i \in \hat{I}) = p$). The sampled profiles are then obtained by keeping the items of each profile that are also in $\hat{I}$: $\widehat{P}_u = P_u \cap \hat{I}$. On average, the profile of all users is reduced by a factor of $\frac{1}{p}$, but this policy does not adapt to the characteristics of individual profiles: small profiles run the risk of losing too much of their content to maintain good quality results.

2.3.2 Datasets

We use six publicly available datasets containing movie ratings (Table 2.1). Ratings range from disliking (0.5 or 1) to liking (5). To apply Jaccard similarity, we binarize the datasets by keeping only ratings that reflect a positive opinion (i.e. > 3), *before* performing any sampling.

For instance, consider the example of Figure 1.1 (see Sec. 1.1.1). Figure 2.2 represents the same dataset as previously, but with the binarization for the profiles. The

	i_1	i_2	i_3	i_4	i_5
u_1	1		**5**		
u_2		3		**4**	
u_3	**5**		1		
u_4	**5**		1		2
u_5		3		**4**	**5**

(a) Dataset with ratings. The ratings in bold are the ones kept after binarization.

u_1	$\{i_3\}$
u_2	$\{i_4\}$
u_3	$\{i_1\}$
u_4	$\{i_1\}$
u_5	$\{i_4, i_5\}$

(b) Profiles after binarization.

Figure 2.2 – Dataset with m=5 and n=5. After binarization, a lot of ratings are not considered anymore.

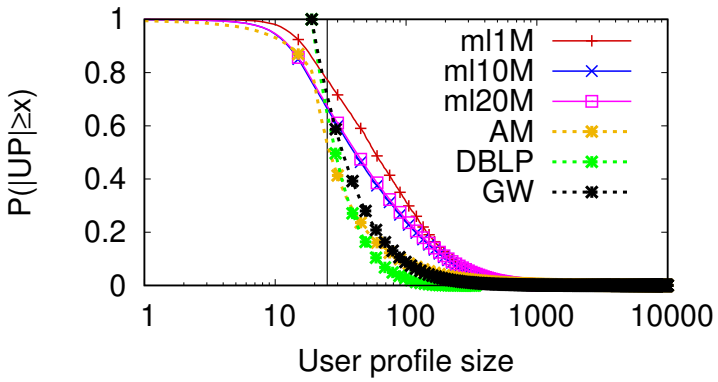


Figure 2.3 – CCDF of user profile sizes on the datasets used in the evaluation (positive ratings only). Between 77% (movielens1M) and 53% (AmazonMovies) of profiles are larger than the default cut-off value 25 (marked as a vertical bar).

profiles in Figure 2.2b are the ones obtained after binarization. They are smaller than the ones in Figure 1.1b. In particular, the users u_1 and u_3, which had the same profiles but with opposite ratings, have different profiles after binarization.

Figure 2.3 shows the resulting Complementary Cumulative Distribution Functions (CCDF) of profile sizes for each dataset. For instance, more than 66% of users have profiles larger than 25 in movielens10M (ml10M). This means that a constant-size sampling with $s = 25$ on movielens10M removes more than 3 million ratings (-69.23%).

The three Movielens datasets

Movielens [HK15] is a group of anonymous datasets containing movie ratings collected on-line between 1995 and 2015 by GroupLens Research [RIS+94]. The datasets (before binarization) contain movie ratings on a 0.5-5 scale by users who have at least performed more than 20 ratings. We use three versions of the dataset, movielens1M (ml1M), movielens10M (ml10M) and movielens20M (ml20M), containing between $575,281$ and $12,195,566$ positive ratings (i.e. higher than 3).

The AmazonMovies dataset

AmazonMovies [ML13b] (AM) is a dataset of movies reviews from Amazon which spans from 1997 to 2012. Ratings range from 1 to 5. We restrain our study to users with at least 20 ratings (positive and negative ratings) to avoid to deal with users with not enough data (this problem, called the *cold start problem*, is generally treated separately [LVLD08]). After binarization, the resulting dataset contains $57,430$ users; $171,356$ items; and $3,263,050$ ratings.

DBLP

DBLP [YL12] is a dataset of co-authorship from the DBLP computer science bibliography. In this dataset, both the user set and the item set are subsets of the author set. If two authors have published at least one paper together, they are linked, which is expressed in our case by both of them rating each other with a rating equal to 5. As with AM, we only consider users with at least 20 ratings: the others are removed from the user set but are still part of the item set. The resulting dataset contains $18,889$ users, $203,030$ items; and $692,752$ ratings.

Gowalla

Gowalla [CML11] (GW) is a location-based social network. As DBLP, both user set and item set are subsets of the set of the users of the social network. The undirected friendship link from u to v is represented by u rating v with a 5. As previously, only the users with at least 20 ratings are considered, the resulting dataset contains $20,270$ users, $135,540$ items; and $1,107,467$ ratings.

2.3.3 Evaluation metrics

We measure the effect of sampling along two main metrics: *(i)* their computation *time*, and *(ii)* the *quality* ratio of the resulting KNN graph.

The time is measured from the beginning of the execution of the algorithm, until the KNN graph is computed. It does not take into account the preprocessing of the dataset, which is evaluated separately in Section 2.4.2.

When applying sampling, the resulting KNN graph is an approximation of the exact one. In many applications such as recommender systems, this approximation should provide neighborhoods of high quality, even if those do not overlap with the exact KNN. To gauge this quality, we introduce the notion of *similarity ratio*, which measures how well the average similarity of an approximated graph compares against that of an exact KNN graph. Formally we define the *average similarity* of an approximate KNN graph

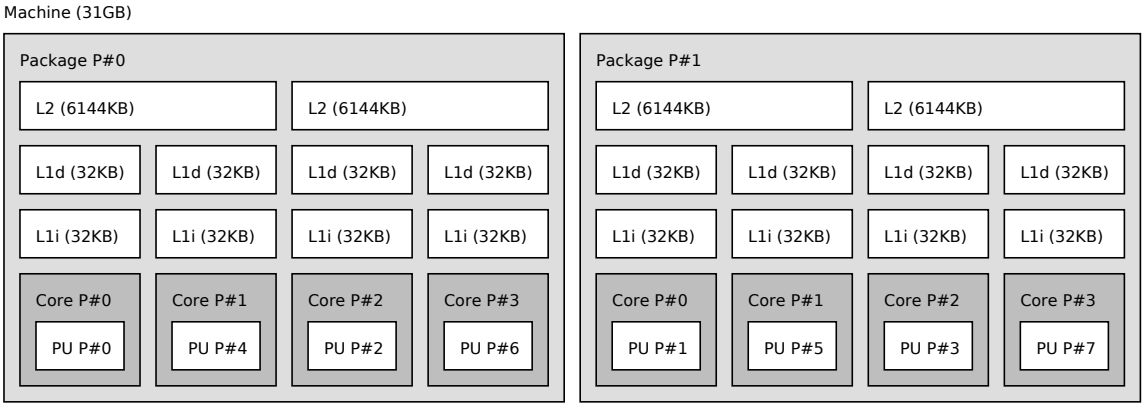


Figure 2.4 – Representation of the server we use for our experiments.

$\widehat{G}_{\mathrm{KNN}}$ as

$$avg\_sim(\widehat{G}_{\mathrm{KNN}}) = \mathop{\mathbb{E}}_{(u,v)\in U^2 : v\in\widehat{knn}(u)} f_{sim}(P_u, P_v), \tag{2.5}$$

i.e. as the average similarity of the edges of $\widehat{G}_{\mathrm{KNN}}$, and we define the *quality* of $\widehat{G}_{\mathrm{KNN}}$ as its *normalized* average similarity

$$quality(\widehat{G}_{\mathrm{KNN}}) = \frac{avg\_sim(\widehat{G}_{\mathrm{KNN}})}{avg\_sim(G_{\mathrm{KNN}})}, \tag{2.6}$$

where G_{KNN} is an ideal KNN graph, obtain without sampling.

A quality close to 1 indicates that the approximate neighborhoods of $\widehat{G}_{\mathrm{KNN}}$ present a similarity that is very close to that of ideal neighborhoods, and can replace them with little loss in most applications, as we will show in the case of recommendations in our evaluation.

Throughout our experiments, we use a 5-fold cross-validation procedure which creates 5 training sets composed of 80% of the ratings. The remaining 20%, i.e. the training sets, are used for recommendations in Section 2.4.5. Our results are the average on the 5 resulting runs.

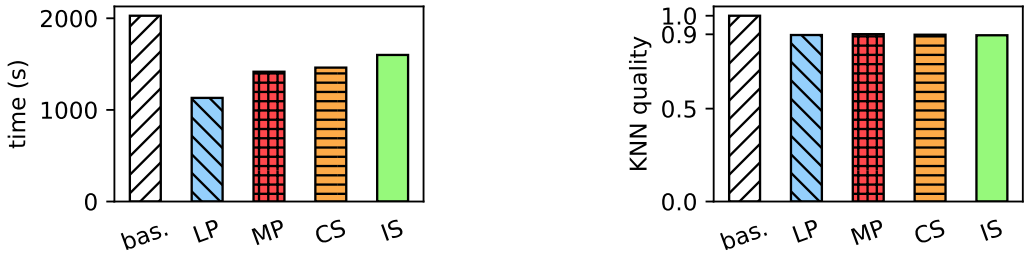
2.3.4 Experimental setup

The details of the implementation are in Appendix A. We ran our experiments on a 64-bit Linux server with two Intel Xeon E5420@2.50GHz, totaling 8 hardware threads, 32GB of memory, and a HHD of 750GB. Figure 2.4 shows a representation of the server, obtained using *lstopo*[2]. We use all 8 threads. In our experiments, we compute KNN graphs with k set to 30, which is a standard value.

2. https://linux.die.net/man/1/lstopo

Dataset	Base.	LP	Δ (%)	MP	Δ (%)	CS	Δ (%)	IS	Δ (%)
ml1M	19	**11**	**−40.5**	14.3	−24.7	14.2	−25.3	12.9	−32.1
ml10M	2028	**1131**	**−44.2**	1416.6	−30.1	1461.6	−27.9	1599.8	−21.1
ml20M	8393	**4865**	**−42.0**	5766.0	−31.3	5965.0	−28.9	6535.3	−22.1
AM	1862	**687**	**−63.1**	817.8	−56.1	748.1	−59, 8	850.0	−54.4
DBLP	100	72.2	**−27.8**	**61.2**	**−38.8**	84.8	−15.2	65.5	−24.5
GW	160	**106.1**	**−33, 7**	111.6	−30, 3	112.8	−29, 5	114.5	−28, 4

Table 2.2 – Computation time (s) of the baseline and the four sampling policies. The parameters were chosen to have a quality equal to 0.9. LP reduces computation time by 27% (DBLP) to 63% (AM), and outperforms other sampling policies on all datasets.



(a) Computation time (lower is better)

(b) KNN quality (higher is better)

Figure 2.5 – Computation time and KNN quality of the baseline and the sampling policies on movielens10M, when quality is set to 0.9. LP yields a reduction of 44.2% in computation time, outperforming other sampling policies.

2.4 Experimentations

2.4.1 Reduction in computing time, and quality/speed trade-off

The baseline algorithm (without sampling) produces an exact KNN graph, with a quality of 1. To compare the different sampling policies (LP, MP, CS and IS) on an equal footing, we configure each of them on each dataset to achieve a quality of 0.9. The resulting parameter s ranges from 10 (MP on DBLP) to 75 (MP on movielens1M), while p (for IS) varies between 0.35 (on AmazonMovies) and 0.68 (on movielens20M). Table 2.2 summarizes the computation times measured on the six datasets with the percentage time reduction obtained against the baseline (Δ columns), while Figure 2.5 shows the results on movielens10M. Except on DBLP where MP performs the best, LP outperforms all other policies on most datasets, reaching a reduction of up to 63% on AM.

Because they reduce the size of profiles, sampling policies exchange quality for speed. To better understand this trade-off, Figure 2.6 plots the evolution of the computation time and the resulting quality when s ranges from 5 to 200 for LP, MP, and CS ($s \in \{5, 10, 15, 20, 30, 40, 50, 75, 100, 200\}$), and p ranges from 0.1 to 1.0 for IS ($p \in \{0.1, 0.2, 0.4, 0.5, 0.75, 0.9, 1.0\}$).

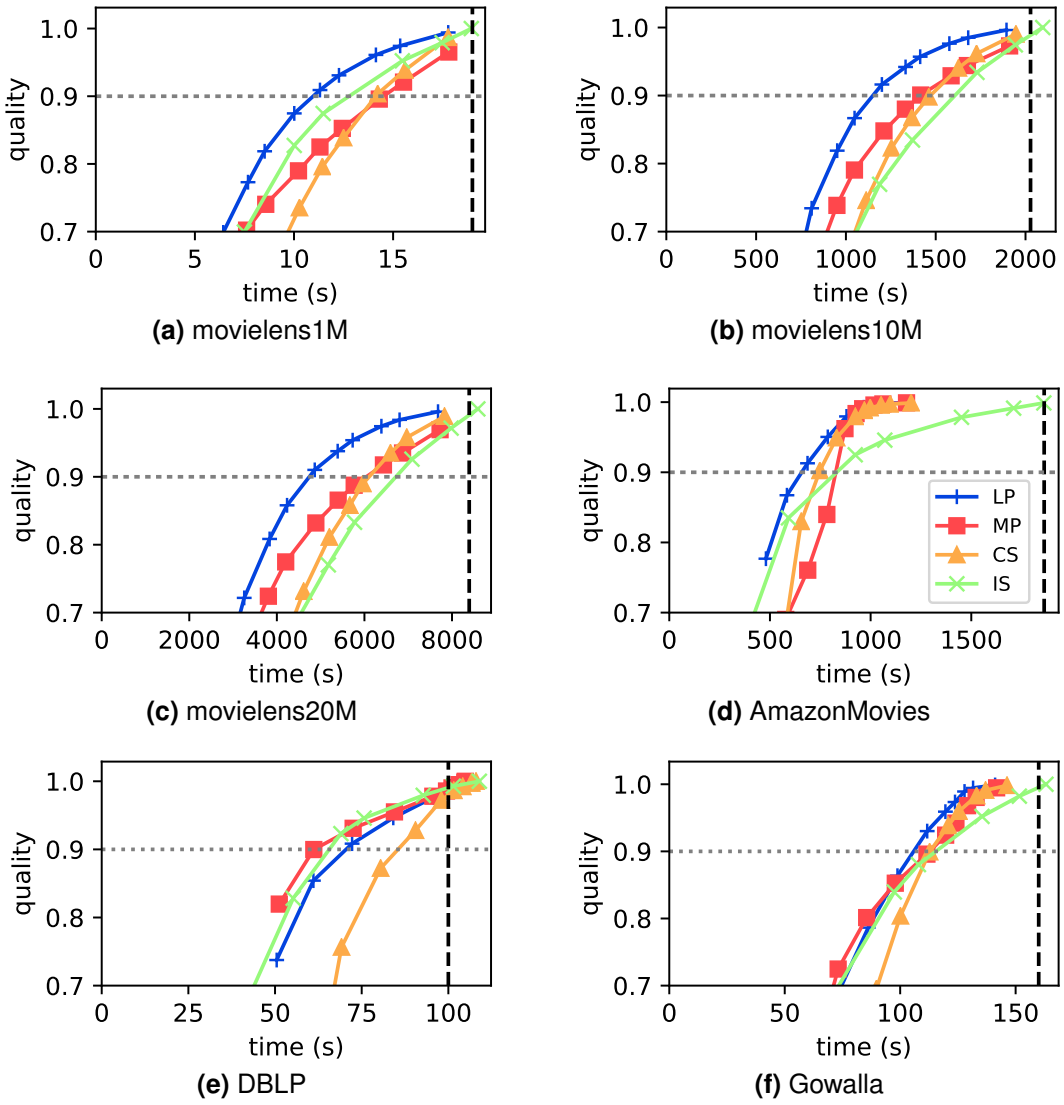


Figure 2.6 – Trade-off between computation time and quality. Closer to the top-left corner is better. LP clearly outperforms all other sampling policies on all datasets.

For clarity, we only display points with a quality above 0.7, corresponding to the upper values of s and p. The dashed vertical line on the right shows the computation time of the baseline (producing a quality of 1), while the dotted horizontal line shows the quality threshold of 0.9 used in Table 2.2 and Figure 2.5.

Lines closer to the top-left corner are better. The figures confirm that our contribution, LP, outperforms other sampling policies on all datasets but DBLP and GW.

On these two datasets, there is no winner: LP, MP and IS are similar on DBLP and on low values on GW while LP is better by a low margin on the high values in GW. These difference on these two datasets can be explained by their high sparsity and the fact that item set is the user set before removing users with small profiles. We end up in DBLP with items which have been rated in average only 3.41 times. In that case, keeping the least popular item of user u would mean that in average u would have a

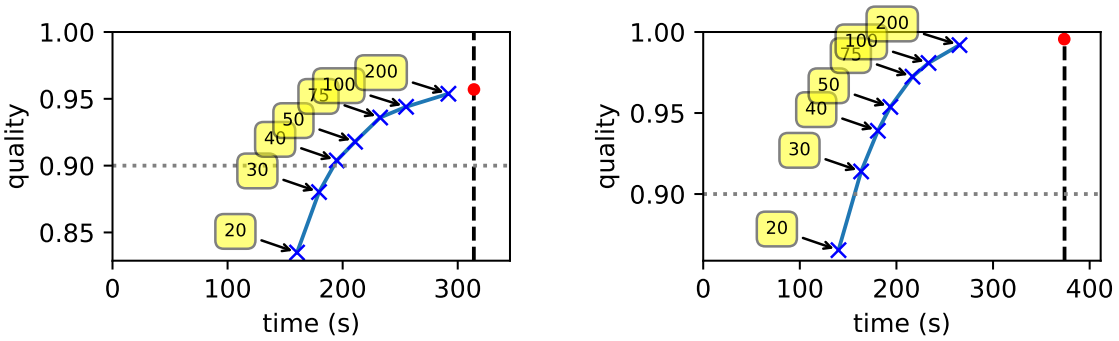
non-zero similarity with only 2.41 other users. We end-up with 2.41 neighbors instead of 30, resulting in a drop in quality. More generally, to have 30 other users with a non-zero similarity, it would require to keep the $30/2.41 = 12,44813278$ least popular items. It does not mean that these users have a high real similarity. To have more candidates we need to keep more items: on DBLP the similarity of 0.9 is reached by LP with a sampling size of 15. In average each user have a non-zero similarity with $36,15$ users. This number is close to 30 meaning that LP is a good representation of the profiles, since the resulting KNN graph has a high quality. And this number is a high estimation since the average of 3.41 ratings per item would decrease by removing the most similar items of each profile.

On the other datasets, there is however no clear winner among the remaining policies: IS performs well on movielens1M, but arrives last on the other datasets, and MP and CS show no clear order, which depends on the dataset and the quality considered.

Impact on greedy approaches

If LP is better than the other sampling policies, it is not clear if it can be used to compute KNN graphs. The brute force approach is too impractical and should not be used for big datasets. We computed KNN graphs on ml10M with LP using Hyrec and NNDescent. The parameter δ of Hyrec and NNDescent is set to 0.001, and their maximum number of iterations to 30. The sampling size s ranges from 20 to 200. Figure 2.7 shows the results. As previously, the dashed vertical line on the right shows the computation time of the baseline (i.e. using Hyrec or NNDescent without sampling), while the dotted horizontal line shows the quality threshold of 0.9. The red dot represents the baseline: unlike the brute force approach, Hyrec and NNDescent without sampling does not necessarily produce KNN graph of quality equal to 1. Lines closer to the top-left corner are better. The same trade-off as with the brute force approach between quality and computation time is obtained. LP does not alter the convergence of the greedy approaches, the speed-up is correlated to how small the sampling size is. The quality provided by NNDescent is similar of those of the brute force approach. On the other hand, Hyrec provide a lower quality. For Hyrec to achieve the same quality of 0.9, the value of the sampling size is higher for Hyrec than the value for brute force, $s = 25$. The reason of that difference is explained by the performances of Hyrec without sampling. Without sampling, the resulting KNN graphs computed by Hyrec have a lower quality: Hyrec without sampling provokes a loss of 0.04 in quality. This loss is reverberated on the quality when using LP.

LP has the same effect on greedy approaches it has on the brute force approach. Sampling can be used to compute KNN graphs with the most up-to-date KNN graph algorithm.



(a) Hyrec **(b)** NNDescent

Figure 2.7 – Trade-off between computation time and quality of Hyrec and NNDescent on ml10M with LP. Closer to the top-left corner is better.

Dataset	Base.	LP	Δ (s)	MP	Δ (s)	CS	Δ (s)	IS	Δ (s)
ml1M	0.36	0.50	+0.14	0.49	+0.13	0.46	+0.10	0.33	−0.03
ml10M	4.03	5.49	+1.46	5.67	+1.64	4.99	+0.96	3.98	−0.05
ml20M	8.55	11.95	+3.40	12.35	+3.80	11.05	+2.50	8.71	+0.16
AM	3.42	4.90	+1.48	4.70	+1.28	4.32	+0.90	2.41	−1.01
DBLP	0.42	0.79	+0.37	0.75	+0.33	0.57	+0.08	0.63	+0.21
GW	0.47	0.91	+0.44	0.90	+0.43	0.64	+0.17	0.63	+0.15

Table 2.3 – Preprocessing time (seconds) for each dataset, and each sampling policy, with parameters set so that the resulting KNN quality is 0.9. The preprocessing times are negligible compared to the computation times.

2.4.2 Preprocessing overhead

As is common with KNN graph algorithms [BKMT16, DML11], the previous measurements do not include the loading and preprocessing time of the datasets, which is typically dominated by I/O rather than CPU costs. Sampling adds some overhead to this preprocessing, but Table 2.3 shows that this extra cost (Δ columns) remains negligible compared to the computation times of Table 2.2. For instance, LP adds 3.4 s to the preprocessing of movielens20M, which only represents 0.07% of the complete execution time of the algorithm ($4865\text{s} + 11.95\text{s} = 4877\text{s}$). IS even decreases the preprocessing time on three datasets out of six, by starkly reducing the bookkeeping costs of profiles while introducing only a low extra complexity.

2.4.3 Influence of LP on the topology

Figure 2.8 shows how LP tends to distort estimated similarity values between pairs of users in ml10M, when $s = 25$ ($q = 0.9$ with brute force). The x-axis represents the real similarity of a pair of users $(u, v) \in U^2$, the y-axis represents its similarity obtained

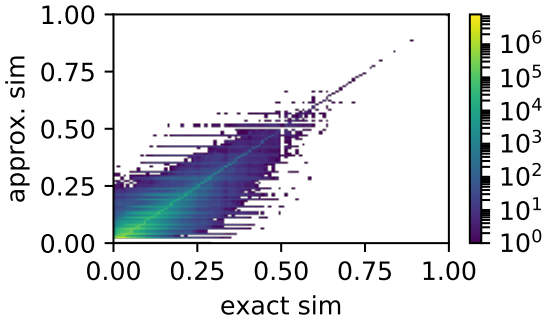


Figure 2.8 – Heatmap similarities on movielens10M with LP ($s = 25$). The majority of pairs are concentrated around the diagonal: LP has a low impact on the topology of the dataset.

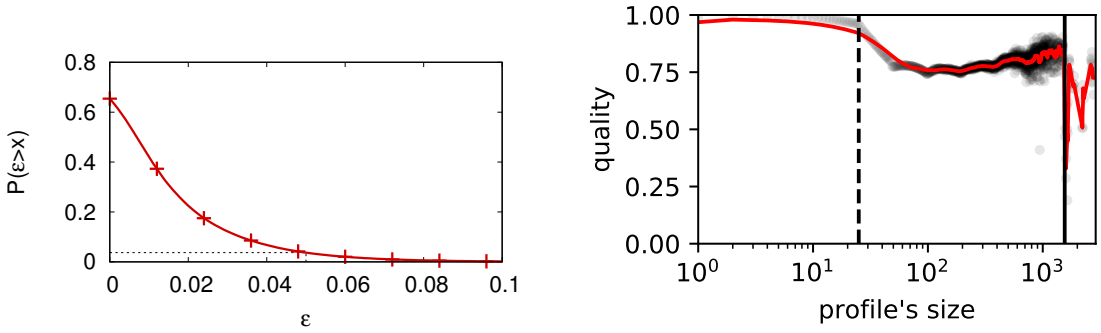
using LP, and the z-axis represents the number of pairs whose real similarity is x and estimated similarity y. (Note the log scale for z values.) These figures were obtained by sampling 10^8 pairs of users of ml10M. Due to technical reasons we had to divide each z-value by 10.

2.4.4 Influence of LP at the user's level

Constant size sampling has a different influence on each user, depending on this user's profile's size. Profiles whose sizes are below the parameter s remain unchanged while larger profiles are truncated, thus losing information.

Figure 2.9 investigates the impact of this loss with our approach, LP, on movielens10M with $s = 25$ (corresponding to a quality of 0.9). Figure 2.9a plots the distribution of the similarity error $\epsilon = |J(P_u, P_v) - J(\widehat{P}_u, \widehat{P}_v)|$ introduced by sampling when ϵ is computed for each pair of users (u, v). The figure shows that 35% of pairs experience no error ($\epsilon = 0$), and that 96% have an error below 0.05 (dotted vertical line), confirming that our sampling only introduces a limited distortion of similarities.

Figure 2.9b represents the impact of LP on the quality of users' neighborhoods, according to the initial profile size of users. For every user u with an initial profile size of $|P_u|$, we compute the average similarity of u's approximated neighborhood $\widehat{knn}(u)$, and normalize this similarity with that of u's exact neighborhood $knn(u)$. The closest to 1 the better. We then average this normalized similarity for users with the same profile size $\{u \in U : |P_u| = P\}$. These points are displayed as a scatter plot (in black, note the log scale on the x axis), and using a moving average of width 50 (red curve). The first dashed vertical line is the value of the truncation parameter s ($x = 25$). The points after the second vertical line (at $x = 1553$) represent 24 users (out of 69816) and thus are not statistically significant. As expected, there is a clear threshold affect around the truncation value $s = 25$, yet even users with much larger profiles retain a high



(a) CCDF of the similarity's error. Only 4% of the users have their similarities changed by more than 0.05.

(b) Quality per user as a function of a user's profile size (note the log scale for x).

Figure 2.9 – Influence on the similarity and the quality of sampling with LP with $s = 25$ on movielens10M (total KNN quality equal to 0.9).

neighborhood quality, that remains on average above 0.75.

2.4.5 Recommendations

We want to evaluate the impact of the loss in quality on a practical use of the KNN graphs. To do so we perform item recommendations using the exact KNN graphs and the approximated graphs produced with LP. We recommend the items that a user u is more likely to like. This likelihood is expressed as a weighted average of the ratings the items received by the neighbors of u, weighted by the similarity of u with them. We use the real profiles, without sampling nor binarization, to compute these predicted ratings. After computing the score of every item, we recommend to u a set R_u composed by 30 items with the highest scores:

$$R_u \in \underset{R \subseteq I \setminus P_u : |R|=30}{\operatorname{argmax}} \sum_{i \in S} \frac{\sum_{v \in knn_u} sim(u,v) \times r_{v,i}}{\sum_{v \in knn_u} sim(u,v)}, \tag{2.7}$$

where $r(i, v)$ is the rating made by the user v on the item i. We use the same 5-fold cross-validation as used for the KNN graph computation. We consider a recommendation successful when a recommended item is found within the 20% removed ratings (the testing set) with a rating above 3 ($r(i, u) > 3$). The quality of the recommendation is measured using *recall*, the proportion of successful recommendations among all recommendations.

Table 2.4 shows the recall we obtain by using the exact KNN graphs obtained with the baseline and with LP using when the KNN quality is set to 0.9. In spite of its approximation, LP introduces no loss in recall, and even achieves slightly better scores than the baseline, which shows that our sampling approach can be used with little impact in

Dataset	Base.	LP	Δ
movielens1M	0.218	0.220	$+0.002$
movielens10M	0.273	0.275	$+0.002$
movielens20M	0.256	0.258	$+0.002$
AmazonMovies	0.595	0.596	$+0.001$
DBLP	0.360	0.356	-0.004
Gowalla	0.268	0.265	-0.003

Table 2.4 – Recommendation recall without sampling (*Base.*) and using the *Least Popular* (LP) policy (total KNN quality set to 0.9).

concrete applications.

2.5 Conclusion

In this chapter, we have proposed *Constant-Size Least Popular Sampling* (LP) to speed up the construction of KNN graphs on entity-item datasets. By keeping only the least popular items of users' profiles, we make them shorter and thus faster to compare. Our extensive evaluation on four realistic datasets shows that LP outperforms more straightforward sampling policies. More precisely, LP is able to decrease the computation time of KNN graphs by up to 63%, while providing a KNN graph close to the ideal one, with no observable loss when used to compute recommendations.

CHAPTER 3

GOLDFINGER: THE SIMPLER THE FASTER

3.1 Introduction

In the previous chapter, approximations of the profiles were used to speed-up the KNN graph computation. Sampling limits the size of the profile of every user thus speeds-up the similarity computation. The smaller the profiles, the faster the set intersection, which is at the core of the Jaccard similarity. Still, the speed-up is limited: we only have a gain of 34% on Gowalla. It is because approximating the profiles does not entirely solve the problem: the Jaccard similarity is costly. Performing on less items speeds-up the computation but it cannot change the inherent cost of this operation.

Instead of the profiles, the similarity itself should be approximating. In this chapter, we **approximate the Jaccard similarity**, by relying on a specific compacted datastructure. We advocate the use of *fingerprints*, a *compact*, *binary*, and *fast-to-compute* representation of data. Compacted datastructures are already used to performed specific operations with highly compacted representation of the data. Our focus is the computation time, not the memory space, while compacted datastructures' goal is to gain space while providing the best accuracy: MinHash (Sec. 1.3.4) provides a good estimator of the Jaccard similarity but its pre-processing is too long to be used in KNN graphs constructions.

We propose to fingerprint the set of items associated with each node into what we have termed a *Single Hash Fingerprint* (SHF), a 64- to 8096-bit vector summarizing a node's profile. SHFs are very quick to construct, they protect the privacy of users by hiding the original clear-text information, and provide a sufficient approximation of the similarity between two nodes using extremely cheap bit-wise operations. We propose to use these SHFs to rapidly construct KNN graphs by approximating the similarity, in an overall approach we have dubbed *GoldFinger*. As sampling, GoldFinger is *generic* and *efficient*: it can be used to accelerate any KNN graph algorithm relying on Jaccard index, adds close to no overhead, and the size of individual SHFs can be tuned to trade space and time for accuracy.

The rest of this chapter is organized as follows. The intuition and the approach

are detailed in Section 3.2. The experimental setup are presented in Section 3.3. The results are discussed in Section 3.4 while a sensitivity analysis is presented in Section 3.5. We conclude in Section 3.6.

3.2 Intuition and Approach

3.2.1 Intuition

Our motivation is the same as with sampling: a large portion of a KNN graph's construction time (whether approximate or exact) often comes from computing individual similarity values (up to 90% of the total construction time in some recent approaches [BKMT16]). This is because computing explicit similarity values on even medium-size profiles can be relatively expensive (see Section 2.2.1).

Our intuition is that, with almost no overhead, fingerprints can capture enough of the characteristics of the data to provide a good approximation of similarity values, while drastically reducing the cost of computing these similarities. This strategy is further orthogonal to the actual algorithm used to compute an approximate KNN graph, opening up a wide range of design choices and trade-offs.

3.2.2 GoldFinger and Single Hash Fingerprints

Our approach, dubbed *GoldFinger*, extracts from each user's profile a *Single Hash Fingerprint* (SHF for short). An SHF is a pair $(B, c) \in \{0,1\}^b \times \mathbb{N}$ comprising a bit array $B = (\beta_x)_{x \in [\![0, b-1]\!]}$ of b bits, and an integer c, which records the number of bits set to 1 in B (its L1 norm, which we call the *cardinality* of B in the following). The SHF of a user's profile P is computed by hashing each item of the profile into the array and setting to 1 the associated bit

$$\beta_x = \begin{cases} 1 & \text{if } \exists e \in P : h(e) = x, \\ 0 & \text{otherwise,} \end{cases}$$
$$c = \left\| (\beta_x)_x \right\|_1$$

where $h()$ is a uniform hash function from all items to $[\![0, b-1]\!]$, and $\| \cdot \|_1$ counts the number of bits set to 1.

Benefits in terms of space and speed The length b of the bit array B is usually much smaller than the total number of items, causing collisions, and a loss of information. This loss is counterbalanced by the highly efficient approximation SHFs can provide of any set-based similarity. The Jaccard's index of two user profiles P_1 and P_2 can be

SHF length (bits)	Comp. Time (ms)	Speedup $\lvert P\rvert = 80$
64	0.011	253
256	0.032	84
1024	0.120	23
4096	0.469	6

Table 3.1 – Jaccard's index computation time between SHFs, and speed-up against an explicit computation (80 items, Fig. 2.1). SHFs are typically 1 to 2 orders of magnitude faster.

estimated from their respective SHFs (B_1, c_1) and (B_2, c_2) with

$$\widehat{J}(P_1, P_2) = \frac{\|B_1 \textbf{ AND } B_2\|_1}{c_1 + c_2 - \|B_1 \textbf{ AND } B_2\|_1}, \tag{3.1}$$

where B_1 **AND** B_2 represents the bitwise AND of the bit-arrays of the two profiles. This formula exploits two observations that hold generally with no or few collisions in the bit arrays (a point we return to below). First, the size of a set of items P can be estimated from the cardinality of its SHF (B_P, c_p)

$$|P| \approx \|B_P\|_1 = c_p. \tag{3.2}$$

Second, the bit array $B_{(P_1 \cap P_2)}$ of the intersection of two profiles $P_1 \cap P_2$ can be approximated with the bitwise AND of their respective bit-arrays, B_1 and B_2:

$$B_{(P_1 \cap P_2)} \approx (B_1 \textbf{ AND } B_2). \tag{3.3}$$

Equation (3.1) combines these two observations along with some simple set algebra ($|P_1 \cup P_2| = |P_1| + |P_2| - |P_1 \cap P_2|$) to obtain the final formula.

The computation incurred by (3.1) is much faster than on explicit profiles, and is independent of the actual size of the explicit user profiles. This is illustrated in Table 3.1 which shows the computation time of Equation (3.1) on the same profiles as Figure 2.1 (Section 2.2.1) for SHFs of different lengths (as in Figure 2.1, the values are averaged over 4.9 million computations). For instance, estimating Jaccard's index between two SHFs of 1024 bits (the default length used in our experiments) takes 0.120 ms, which is 23 times faster than computing Jaccard's index on two explicit profiles of 80 items each.

The link with Bloom Filters and collisions SHFs can be interpreted as a highly simplified form of Bloom filters, and suffer from errors arising from collisions, as Bloom filters do. However, the two structures serve different purposes: whereas Bloom filters are designed to test whether individual elements belong to a set, SHFs are designed to approximate set similarities (in this example Jaccard's index). Still, Bloom filters can

	GolFi	BF	gain(%)
ml10M	**2216.32**	6444.52	**65.61**
AM	**2103.98**	5886.98	**64.26**

Table 3.2 – Comparison of the time spend (ms) to compute 10000 similarity computations on movielens10M and AmazonMovies using GoldFinger and regular Bloom filters, with $b = 1024$ bits each. GoldFinger achieves speeds-up up to 65%.

be used to estimate set intersection (see Sec. 1.3.1). We compared the computation time of the Jaccard's index using both datastructures: *(i)* a standard implementation of Bloom filters[1] (labeled *BF*) and *(ii)* the SHFs (labeled *GolFi*). Both of them use 1024 bits and the Bloom filters only have one hash function. We compute 10000 similarities between pairs of users of movielens10M and AmazonMovies. The results are displayed in Table 3.2. SHFs outperforms Bloom filters.

In addition of computation time, Bloom filters differ by the use of multiple hash functions. Indeed, Bloom filters often employ multiple hash functions to minimize false positives for the set membership queries. By contrast, multiple hash functions increase single-bit collisions, and therefore degrade the approximation provided by SHFs.

More precisely, SHFs suffer from two types of collisions: two items e_1 and e_2 of the same profile P_1 might be mapped to the same bit position: $h(e_1) = h(e_2)$. In P_1's SHF (B_1, c_1), this type of *intra-profile collision* will cause c_1 to underestimate the cardinality of P_1, and can lead to an over- or under-approximation of the approximated similarity $\widehat{J}(P_1, P_2)$ with a second profile P_2, depending whether e_1 and e_2 also appear in P_2 or not. A second type of *inter-profile collision* occurs when two items $e'_1 \in P_1$ and $e'_2 \in P_2$ present in two different profiles P_1 and P_2 collide ($h(e_1) = h(e_2)$). In this case, the binary **AND** operation overestimate the cardinality of the intersection, leading to an overestimation of $\widehat{J}(P_1, P_2)$.

The probability of both types of collision decreases with the length b of SHFs. There is therefore a natural trade-off between the quality of the estimation provided by SHFs and the time needed to compute this estimation. This trade-off is governed by the length b of the bit arrays: longer bit-arrays deliver a better estimation (with the extreme case of one bit for each item) at the cost of a slower computation. We discuss in more detail in Section 3.5 this trade-off, and its impact on the KNN graph computation.

3.2.3 Analysis of Single Hash Fingerprints

The construction of the SHF follows a classical *balls into bin* procedure in which items play the role the balls, and the bits of SHFs the role of bins. We are interested in analysis the behavior of the L_1 norm of the bit-wise intersection of SHFs

1. https://github.com/Baqend/Orestes-Bloomfilter

$\|\mathbf{B}_A$ **AND** $\mathbf{B}_B\|_1$. We can do this by considering three kinds of 'balls' (items): (a) items that belong to both profiles, (b) items that belong only to P_A, (c) items that belong only to P_B.

We will use the following notation to capture these three cases:

$$P_\cap = P_A \cap P_B$$
$$P_{A'} = P_A \setminus P_\cap$$
$$P_{B'} = P_B \setminus P_\cap$$

and

$$\alpha = |P_\cap| = |P_A \cap P_B|$$
$$\gamma_A = |P_{A'}| = |P_A| - \alpha$$
$$\gamma_B = |P_{B'}| = |P_B| - \alpha$$

In particular we have:

$$|P_A| = \gamma_A + \alpha$$
$$|P_B| = \gamma_B + \alpha$$

Let us denote I_x (resp. A_x, B_x) the event that one of the items of $P_\cap$ (resp. $P_{A'}$, $P_{B'}$) is mapped to bit x through the hash function h. Formally, the events I_x, A_x, B_x are defined as

$$S_x = (\exists e \in S : h(e) = x),$$

where $S \in \{P_\cap, P_{A'}, P_{B'}\}$.

$$(\mathbf{B}_A \textbf{ AND } \mathbf{B}_B)_x = [(I_x \vee A_x) \wedge (I_x \vee B_x)] \tag{3.4}$$
$$= [I_x \vee (\neg I_x \wedge A_x \wedge B_x)], \tag{3.5}$$

where $[P]$ denotes Iverson's bracket for predicate P, i.e. $[P] = 1$ if P is true, $[P] = 0$ otherwise. Equation (3.5) captures the fact that a bit x of $\mathbf{B}_A$ **AND** $\mathbf{B}_B$ might be set to 1 either if at least one of the elements of $A = P_A \cap P_B$ is mapped to x, or if some elements of $B = P_A \setminus P_B$ and some elements of $C = P_B \setminus P_A$ collide on x.

$$\mathbb{P}\Big((\mathbf{B}_A \textbf{ AND } \mathbf{B}_B)_x = 1\Big) = \mathbb{P}(I_x \vee (\neg I_x \wedge A_x \wedge B_x)) \tag{3.6}$$
$$= \mathbb{P}(I_x) + \mathbb{P}(\neg I_x \wedge A_x \wedge B_x) \tag{3.7}$$
$$= \mathbb{P}(I_x) + \mathbb{P}(\neg I_x) \times \mathbb{P}(A_x) \times \mathbb{P}(B_x) \tag{3.8}$$
$$= (1 - q^\alpha) + q^\alpha \times (1 - q^{\gamma_A}) \times (1 - q^{\gamma_B}) \tag{3.9}$$
$$= 1 + q^\alpha \times (q^{\gamma_A + \gamma_B} - q^{\gamma_A} - q^{\gamma_B}) \tag{3.10}$$

where $p = \frac{1}{b}$ is the probability that the hash function h sends an item to a particular bit x, and $q = 1 - p$ its complement.

Distribution of $\widehat{J}(P_A, P_B)$

We start by computing the joint distribution of four quantities that we will then use to compute $\widehat{J}(P_A, P_B)$.

We define the following notations to denote the images by h of different parts of the profiles P_A and P_B:

— $\widehat{P}_{A'} = h(P_{A'}) = h(P_A \setminus P_B)$, the set of bits that receive items that are only in P_A ;
— $\widehat{P}_{B'} = h(P_{B'}) = h(P_B \setminus P_A)$, the set of bits that receive items that are only in P_B ;
— $\widehat{P}_\cap = h(P_\cap) = h(P_B \cap P_A)$, the set of bits that receive items that are both in P_A and P_B ;
— $\widehat{P}_\cup = h(P_\cup)$.

Although $P_{A'}$, $P_{B'}$ and $P_\cap$ are disjoint by definition, $\widehat{P}_{A'}$, $\widehat{P}_{B'}$, and $\widehat{P}_\cap$ are usually not. To capture this overlap we introduce the following sets:

— $\widehat{P}_{\widehat{\eta}_A} = \widehat{P}_{A'} \setminus \widehat{P}_\cap = h(P_{A'}) \setminus h(P_\cap)$,
— $\widehat{P}_{\widehat{\eta}_B} = \widehat{P}_{B'} \setminus \widehat{P}_\cap = h(P_{B'}) \setminus h(P_\cap)$.
— $\widehat{P}_{\widehat{\beta}} = \widehat{P}_{A'} \cap \widehat{P}_{B'} \setminus \widehat{P}_\cap = h(P_{A'}) \cap h(P_{B'}) \setminus h(P_\cap)$,

and the following quantities:

— $\widehat{u} = |\widehat{P}_\cup|$,
— $\widehat{\alpha} = |\widehat{P}_\cap|$,
— $\widehat{\eta}_A = |\widehat{P}_{\widehat{\eta}_A}|$,
— $\widehat{\eta}_B = |\widehat{P}_{\widehat{\eta}_B}|$,
— $\widehat{\beta} = |\widehat{P}_{\widehat{\beta}}| = \widehat{\eta}_A + \widehat{\eta}_B + \widehat{\alpha} - \widehat{u}$.

The relationship between the above variables can illustrated on the Venn Diagram of Figure 3.1.

— $\widehat{u}$ counts the bits that are set to 1 in $\mathbf{B}_A$ **OR** $\mathbf{B}_B$;
— $\widehat{\alpha}$ counts the bits that receive elements that belong to the intersection $P_\cap = P_A \cap P_B$ (possibly in addition to elements that belong to only one of the two profiles).

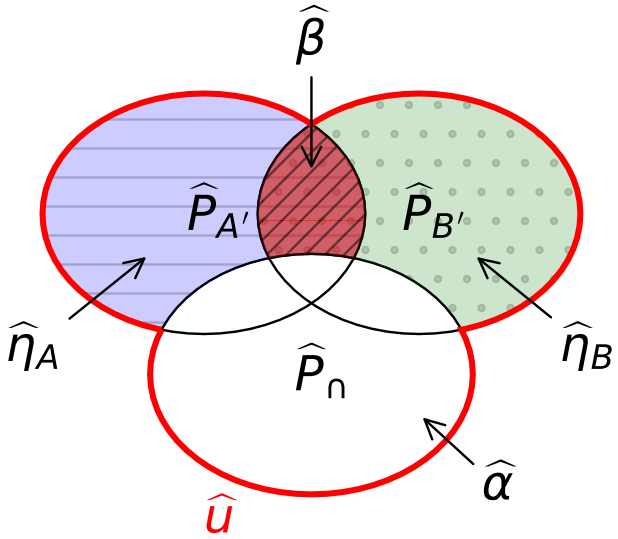


Figure 3.1 – Variables $\widehat{\eta}_A, \widehat{\eta}_B, \widehat{\alpha}, \widehat{\beta}$ and $\widehat{u}$

— $\widehat{\eta}_A$ (resp. $\widehat{\eta}_B$) counts the bits that receive elements that are only in P_A (resp. P_B), possibly along with elements that are only in P_B (resp. P_A), but no elements from the intersection $P_\cap$.

— $\widehat{\beta}$ counts the bits that receive elements that are only in P_A *and* elements that are only in P_B, but no elements from the intersection $P_\cap$.

With the above quantities we have

$$\widehat{J}(P_A, P_B) = \frac{\widehat{\alpha} + \widehat{\beta}}{\widehat{u}} = \frac{2\widehat{\alpha} + \widehat{\eta}_A + \widehat{\eta}_B}{\widehat{u}} - 1.$$

To compute the distribution of $\widehat{J}(P_A, P_B)$ we first compute the joint distribution of the values $(\widehat{u}, \widehat{\alpha}, \widehat{\eta}_A, \widehat{\eta}_B)$ when h is chosen uniformly randomly among all functions mapping $P_\cup = P_A \cup P_B$ onto $[\![0, b-1]\!]$ ($\widehat{\beta}$ is completely determined by the four other values, and used for readability).

We use a counting strategy: given a tuple $(\widehat{u}, \widehat{\alpha}, \widehat{\eta}_A, \widehat{\eta}_B)$, how many random functions h produce these values? To count these functions, we use a constructive argument. Because $P_\cap$, $P_{A'}$ and $P_{B'}$ are disjoint, h can be seen as the piece-wise combination of three independent random functions $h_{|P_\cap}$, $h_{|P_{A'}}$, and $h_{|P_{B'}}$, where $h_{|X}$ denotes the restriction of h to a set X.

To construct h, we start by choosing $\widehat{P}_\cup = h(P_\cup)$ within $[\![0, b-1]\!]$. As $\widehat{u} = |h(P_\cup)|$, there are $\binom{b}{\widehat{u}}$ such choices.

Within $\widehat{P}_\cup$, we then choose

— $\widehat{P}_\cap$ (middle bottom circle in Figure 3.1, containing $\widehat{\alpha}$ bits),

— $\widehat{P}_{\widehat{\beta}}$ (containing $\widehat{\beta}$ bits),

— $\widehat{P}_{\widehat{\eta}_A} \setminus \widehat{P}_{\widehat{\beta}}$ (containing $\widehat{\eta}_A - \widehat{\beta}$ bits).

Note that once the 3 above sets have been chosen, only one possibility remains for

$\widehat{P}_{\widehat{\eta}_B} \setminus \widehat{P}_\cap$, since $\widehat{P}_\cap$, $\widehat{P}_{\widehat{\beta}}$, $\widehat{P}_{\widehat{\eta}_A} \setminus \widehat{P}_{\widehat{\beta}}$ and $\widehat{P}_{\widehat{\eta}_B} \setminus \widehat{P}_{\widehat{\beta}}$ partition $\widehat{P}_\cup$.

In total, the number of choices for $\widehat{P}_\cup$, $\widehat{P}_\cap$, $\widehat{P}_{\widehat{\beta}}$, $\widehat{P}_{\widehat{\eta}_A} \setminus \widehat{P}_\cap$ (and $\widehat{P}_{\widehat{\eta}_B} \setminus \widehat{P}_\cap$) is

$$\binom{b}{\widehat{u}}\binom{\widehat{u}}{\widehat{\alpha}}\binom{\widehat{u}-\widehat{\alpha}}{\widehat{\beta}}\binom{\widehat{u}-\widehat{\alpha}-\widehat{\beta}}{\widehat{\eta}_A-\widehat{\beta}}$$

One these supporting sets have been chosen, we pick $h_{|P_\cap}$, $h_{|P_{A'}}$, and $h_{|P_{B'}}$.

$h_{|P_\cap}$ is a surjection from $P_\cap$ to $\widehat{P}_\cap$. There are

$$\widehat{\alpha}!\left\{{\alpha \atop \widehat{\alpha}}\right\}$$

such surjections, where $\left\{{\alpha \atop \widehat{\alpha}}\right\}$ is Stirling's number of the second type.

$h_{|P_{A'}}$ maps $P_{A'}$ onto $\widehat{P}_\cup$, but only needs to be surjective on $\widehat{P}_{A'} \setminus \widehat{P}_\cap = \widehat{P}_{\widehat{\eta}_A}$. In addition, $h_{|P_{A'}}$ does only map elements unto $\widehat{P}_{A'} \subseteq \widehat{P}_{\widehat{\eta}_A} \cup \widehat{P}_\cap$, whose cardinal is $|\widehat{P}_{A'}| = \widehat{\eta}_A + \widehat{\alpha}$. To compute the number of such functions we introduce $\xi(x, y, z)$, the number of function $f : X \mapsto Y$ from a finite set X onto a finite set Y, that is surjective on a subset $Z \subseteq Y$ of Y, with $x = |X|$, $y = |Y|$, $z = |Z|$.

Using an inclusion-exclusion argument we have

$$\xi(x, y, z) = \sum_{k=0}^{z} (-1)^k \binom{z}{k} (y-k)^x$$

There are therefore

$$\xi(\gamma_A, \widehat{\eta}_A + \widehat{\alpha}, \widehat{\eta}_A)$$

functions $h_{|P_{A'}}$.

Similarly there are

$$\xi(\gamma_B, \widehat{\eta}_B + \widehat{\alpha}, \widehat{\eta}_B)$$

functions $h_{|P_{B'}}$,

so in total there are

$$\binom{b}{\widehat{u}}\binom{\widehat{u}}{\widehat{\alpha}}\binom{\widehat{u}-\widehat{\alpha}}{\widehat{\beta}}\binom{\widehat{u}-\widehat{\alpha}-\widehat{\beta}}{\widehat{\eta}_A-\widehat{\beta}}\widehat{\alpha}!\left\{{\alpha \atop \widehat{\alpha}}\right\}\xi(\gamma_A, \widehat{\eta}_A + \widehat{\alpha}, \widehat{\eta}_A)\xi(\gamma_B, \widehat{\eta}_B + \widehat{\alpha}, \widehat{\eta}_B)$$

functions h producing $(\widehat{u}, \widehat{\alpha}, \widehat{\eta}_A, \widehat{\eta}_B)$.

Because there are in total $b^{(\alpha+\gamma_A+\gamma_B)}$ possible random functions from $P_\cup$ onto $[\![0, b-1]\!]$ (where $\alpha + \gamma_A + \gamma_B = |P_\cup|$), we have

$$\begin{aligned}\mathbb{P}(\widehat{u}, \widehat{\alpha}, \widehat{\eta}_A, \widehat{\eta}_B | \alpha, \gamma_A, \gamma_B) = \frac{1}{b^{(\alpha+\gamma_A+\gamma_B)}} &\times \binom{b}{\widehat{u}}\binom{\widehat{u}}{\widehat{\alpha}}\binom{\widehat{u}-\widehat{\alpha}}{\widehat{\beta}}\binom{\widehat{u}-\widehat{\alpha}-\widehat{\beta}}{\widehat{\eta}_A-\widehat{\beta}} \\ &\times \widehat{\alpha}!\left\{{\alpha \atop \widehat{\alpha}}\right\}\xi(\gamma_A, \widehat{\eta}_A + \widehat{\alpha}, \widehat{\eta}_A)\xi(\gamma_B, \widehat{\eta}_B + \widehat{\alpha}, \widehat{\eta}_B)\end{aligned}$$

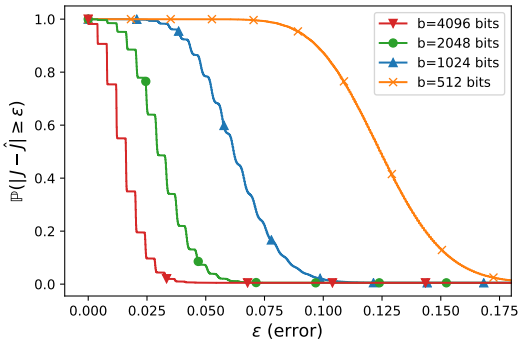


Figure 3.2 – Distribution of the error for two profiles of 128 items, with no overlap. The larger the SHFs the lower the error.

which can also be expressed as

$$\mathbb{P}(\widehat{u},\widehat{\alpha},\widehat{\eta}_A,\widehat{\eta}_B|\alpha,\gamma_A,\gamma_B) = \frac{1}{b^{(\alpha+\gamma_A+\gamma_B)}} \times \frac{b!}{(b-\widehat{u})!\widehat{\beta}!(\widehat{\eta}_B-\widehat{\beta})!(\widehat{\eta}_A-\widehat{\beta})!} \times \begin{Bmatrix}\alpha\\ \widehat{\alpha}\end{Bmatrix} \xi(\gamma_A,\widehat{\eta}_A+\widehat{\alpha},\widehat{\eta}_A)\xi(\gamma_B,\widehat{\eta}_B+\widehat{\alpha},\widehat{\eta}_B)$$

after substituting the binomial coefficients and simplifying, provided that the following conditions are met

$$\widehat{u},\widehat{\alpha},\widehat{\beta},\widehat{\eta}_A,\widehat{\eta}_B \geq 0$$

$$b \geq \widehat{u} \geq \widehat{\beta}+\widehat{\alpha},\ \widehat{\eta}_A+\widehat{\alpha},\ \widehat{\eta}_B+\widehat{\alpha}.$$

The probability density function $\mathbb{P}(\widehat{u},\widehat{\alpha},\widehat{\eta}_A,\widehat{\eta}_B|\alpha,\gamma_A,\gamma_B)$ can be used to compute the probability of error produced by the estimator $\widehat{J}$, using a discrete sum on the corresponding domain of $(\widehat{u},\widehat{\alpha},\widehat{\eta}_A,\widehat{\eta}_B)$.

$$\mathbb{P}\Big(\ |J(P_A,P_B)-\widehat{J}(P_A,P_B)| > \epsilon \ \Big|\ \alpha,\gamma_A,\gamma_B\ \Big) = \sum_{\substack{(\widehat{u},\widehat{\alpha},\widehat{\eta}_A,\widehat{\eta}_B)\in\mathbb{N}^4:\\ \left|\frac{\alpha}{\alpha+\gamma_A+\gamma_B}-\frac{\widehat{\alpha}+\widehat{\beta}}{\widehat{u}}\right|>\epsilon}} \mathbb{P}(\widehat{u},\widehat{\alpha},\widehat{\eta}_A,\widehat{\eta}_B|\alpha,\gamma_A,\gamma_B)$$

Figure 3.2 shows the distribution of the error for two profiles with 128 items. There is no overlap between the two profiles so their real Jaccard similarity is 0. The estimation of the Jaccard similarity is done using SHFs of size 512, 1024, 2048 and 4096. The larger the SHFs the lower the error.

Figure 3.3 represents the distribution of the error for two profiles of same size. Their mutual size is varying from 32 to 128 items. As previously, their intersection is null. The estimation of the Jaccard similarity is done using SHFs of size $b = 1024$. The larger the size of the real profiles the larger the error.

Figure 3.3 shows the probability of the estimation of two profiles of same size to be

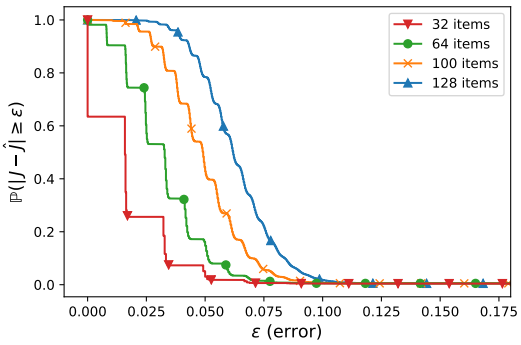


Figure 3.3 – Distribution of the error for two profiles of same size with no overlap, with $b = 1024$. The larger the profiles the larger the error.

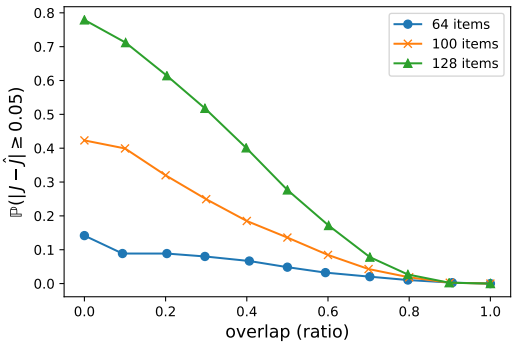


Figure 3.4 – Probability for two profiles of same size, to have an error higher than 0.005 when compacted using SHFs with $b = 1024$. The more similar the profiles the lower the error.

higher than 0.005. The profiles sizes ranges from 64 to 128. The estimation is obtained with SHF of size $b = 1024$. As previously, the larger the profiles the larger the probability that the error is higher than 0.005. Still, the probability decreases as the overlap between the profiles increases.

3.2.4 Privacy guarantees of GoldFinger

The noise introduced by collisions brings additional privacy benefits: collisions obfuscate a user's profile, and thus make it harder to guess this original profile from its compacted SHF. This obfuscation can be exploited by allowing users to compute locally their SHF before sending it to some untrusted KNN-construction service.

We characterize the level of protection granted by GoldFinger along two standard measures of privacy, *k-anonymity* [Sam01], and *ℓ-diversity* [MGKV06]. For this analysis, we assume an honest but curious attacker who wants to discover the profile P_u of a particular user u, knowing its corresponding SHF (B_u, c_u). We assume the attacker knows the item set I, the user set U and the hash function h. More importantly, for a given bit position $x \in [\![0, b-1]\!]$, we assume the attacker can compute $H_x = h^{-1}(x)$, the preimage of x by h. How much information does (B_u, c_u) leak about the initial profile P_u?

k-anonymity

Definition 3.1. *Consider an obfuscation mechanism $obf : \mathcal{X} \mapsto \mathcal{Y}$ that maps a cleartext input $x \in \mathcal{X}$ to an obfuscated value in $\mathcal{Y}$. $obf()$ is* k-anonymous *for $x \in \mathcal{X}$, if the observed obfuscated value $obf(x)$ is indistinguishable from that of at least $k-1$ other explicit input values. Expressed formally, $obf()$ is k-anonymous for $x \in \mathcal{X}$ iff*

$$\left| obf^{-1}\left(obf\left(x\right)\right) \right| \geq k. \tag{3.11}$$

Theorem 1. *GoldFinger ensures $(2^{\frac{m}{b} \times c_u})$-anonymity for a given SHF (B_u, c_u) of length b, and cardinality c_u, where $m = |I|$ is the size of the item set.*

Proof. Let x be the index of a bit set to 1. Let $H_x = h^{-1}(x)$ the set of all the items which are hashed by h to x, it is on average of size $\frac{m}{b}$ with a uniformly random hashing function. Thus $\mathcal{P}(H_x)$ (the powerset of H_x, sometimes noted $\{0,1\}^{H_x}$), whose cardinality is $2^{\frac{m}{b}}$, is the set of all possible sub-profiles that will set the bit x to 1. All of these sub-profiles are indistinguishable once hashed, hashing ensures $(2^{\frac{m}{b}})$-anonymity for this bit. For every bit set to one, there are $2^{\frac{m}{b}}$ possible set of items, leading to a $(2^{\frac{m}{b}})^{c_u}$-anonymity, since all pre-images $(H_x)_x$ are pair-wise disjoint. □

This means that having a compacted profile of cardinality c_u cannot allow an attacker to distinguish the actual profiles which was used to generate (B_u, c_u) between the $(2^{\frac{m}{b} \times c_u} - 1)$ others.

We are not considering empty profiles, so every SHF has at least one bit set to one, so SHF ensures at least $(2^{\frac{m}{b}})$-anonymity for the whole dataset. As the cardinality of the item set $m = |I|$, the anonymity granted by GoldFinger increases.

For instance, one of the datasets we consider, AmazonMovies, has $171,356$ items. With 1024 bit long SHFs (the typical size we use), GoldFinger provides 2^{167}-anonymity, meaning each compacted profile is indistinguishable from at least $2^{167} \approx 1.87 \times 10^{50}$ possible profiles.

ℓ-diversity

Although k-anonymity provides a measure of the difficulty to recover the complete profile P_u of a user u, it does not cover cases in which an attacker would seek to guess some partial information about u. This type of question is better captured by a second metric, *ℓ-diversity* [MGKV06]. The ℓ-diversity model ensures that, for a given SHF (B_u, c_u), the actual profile P_u it was created from is indistinguishable from $\ell - 1$ other profiles $\{P_i\}_{i \in [\![1, \ell-1]\!]}$, and that these profiles form a *well-represented* set.

The difference with k-anonymity lies in this notion of *well-representedness*. In our case it means that we cannot infer any taste from possible profiles: for example in a movie dataset, if all the profiles in the preimage of a given SHF (B_u, c_u) include science-fiction movies, you can infer that the user enjoys science fiction. ℓ-diversity ensures that the possible set of profiles are really diverse.

Definition 3.2. *Consider an obfuscation mechanism $obf : \mathcal{P}(I) \mapsto \mathcal{Y}$ that maps a sets of items $P \subseteq I$ to an obfuscated value in $\mathcal{Y}$. $obf()$ is* ℓ-diverse *for $P \subseteq I$, if the observed obfuscated value $obf(P)$ is indistinguishable from that of at least $\ell - 1$ other explicit profiles $\mathcal{Q} = \{P_i\}_{i \in [\![1, \ell-1]\!]}$ that are pair-wise disjoint: $\forall P_1, P_2 \in \mathcal{Q} : P_1 \cap P_2 = \emptyset$. Expressed formally[2], $obf()$ is ℓ-diverse for $P \subseteq I$ iff*

$$\max_{\substack{\mathcal{Q} \subseteq obf^{-1}(obf(P)) \setminus \{P\}: \\ \forall P_1, P_2 \in \mathcal{Q}: P_1 \cap P_2 = \emptyset}} |\mathcal{Q}| \geq \ell - 1. \tag{3.12}$$

Theorem 2. *For a given SHF (B_u, c_u) of length b and cardinality c_u, SHF ensures $(\frac{m}{b})$-diversity for (B_u, c_u).*

Proof. The reasoning is similar to that of k-anonymity. Let x be the index of a bit set to 1. $|H_x| = \frac{m}{b}$ items are hashed into this bit in average. Assuming an arbitrary order on

2. This definition, adapted to our context, differs slightly from that of the original paper, but leads in practice to the same result.

Dataset	Native	MinHash	GoldFinger	speedup (×)
ml1M	0.37s	6.24s	0.31s	20.1
ml10M	3.90s	203s	3.24s	62.7
ml20M	8.71s	820s	7.06s	116.1
AM	3.40s	3250s	1.92s	1692.7
DBLP	0.42s	944s	0.29s	3255.2
GW	0.47s	594s	0.40s	1485.0

Table 3.3 – Preparation time of each dataset for the native approach, b-bit minwise hashing (MinHash) & GoldFinger. GoldFinger is orders of magnitude faster than MinHash, whose overhead is prohibitive.

items, let us note i_j^x the j^{th} element of the pre-image H_x for each bit x set to 1 in B_u. Without loss of generality, we can choose our order so that $i_0^x \in P_u$ for all x. Consider now the profiles $Q_j = \cup_{x:B_u[x]=1} i_j^x$ for $j \in [\![1, \frac{m}{b} - 1]\!]$. By construction *(i)* $P_u \neq Q_j$ for $j \geq 1$, *(ii)* the $\{Q_j\}_{j \in [\![1, \frac{m}{b}-1]\!]}$ are pair-wise disjoint, and *(iii)* they are all indistinguishable from P_u once mapped onto their SHF. □

For instance, in the dataset AmazonMovies, using 1024 bit long SHFs, we insure 167-diversity.

Since our hashing is deterministic, we do not have a stronger notion of privacy such as differential privacy [Dwo08]. It can be easily obtained by inserting random noise to the SHF [AGK12].

3.3 Experimental Setup

3.3.1 Datasets

We use the same datasets as in the previous chapter: ml1M, ml10M, ml20M, DBLP and GW. For more details, see Section 2.3.2.

3.3.2 Baseline algorithms and competitors

We apply GoldFinger to four existing KNN algorithms: Brute Force (as a reference point, Sec.1.1.3), NNDescent, Hyrec (Sec. 1.1.3) and LSH (Sec. 1.1.3). We compare the performance and results of each of these algorithms in their native form (*native* for short) and when accelerated with GoldFinger. For completeness, we also discuss *b-bit minwise hashing* (Sec. 1.3.4), a binary sketching technique proposed to estimate Jaccard's index between sets, albeit in a different context than ours.

b-bit minwise hashing (MinHash)

A standard technique to approximate Jaccard's index values between sets is the *MinHash* algorithm and its compacted version *b-bit minwise hashing* (see Sec. 1.3.4). In the following, we refer to b-bit minwise hashing as MinHash for short, even though it is an improvement of the original algorithm.

Unfortunately, computing MinHash summaries is extremely costly (as it requires creating a large number of permutations on the entire item set), which renders the approach self-defeating in our context. Table 3.3 summarizes the time required to load and construct the internal representation of each dataset when using a native (explicit) approach, GoldFinger (using Jenkins' hash function [Jen97]), and MinHash. We use 1024 bits for GoldFinger (a typical value), and $b = 4$ and 256 permutations for BBHM (configuration which provides the best trade-off between time and KNN quality). Whereas GoldFinger is slightly faster than a native approach (as it does not need to create extensive in-memory objects to store the dataset), MinHash is one to three orders of magnitude slower than GoldFinger (1692 times slower on AmazonMovies for instance). This kind of overhead makes it impractical for environments with limited resources, and we therefore do not consider MinHash in the rest of our evaluation.

3.3.3 Parameters

We set k to 30 (the neighborhood size). The parameter δ of Hyrec and NNDescent is set to 0.001, and their maximum number of iterations to 30. The number of hash functions for LSH is 10. GoldFinger uses 1024 bits long SHFs computed with Jenkins' hash function [Jen97].

3.3.4 Evaluation metrics

We measure the effect of GoldFinger on Brute Force, Hyrec, NNDescent and LSH along two main metrics: *(i)* their computation *time* (measured from the start of the algorithm, once the dataset as been prepared), and *(ii)* the *quality* ratio of the resulting KNN (Sec. 2.3.3). When applying GoldFinger to recommendation, we also measure the *recall* obtained by the recommender. Throughout our experiments, we use a 5-fold cross-validation procedure, and average our results on the 5 resulting runs.

3.3.5 Implementation details and hardware

The details of the implementation are in Appendix A. Our experiments run on a 64-bit Linux server with two Intel Xeon E5420@2.50GHz, totaling 8 hardware threads,

32GB of memory, and a HHD of 750GB. A graphic representation is available in Chap. 2 (Sec. 2.3 Fig. 2.4). Unless stated otherwise, we use all 8 threads.

3.4 Evaluation Results

We first discuss the impact of GoldFinger on computation time and KNN quality (Sec. 3.4.1) before analyzing in more detail its impact on the type of computation performed (Sec. 3.4.1), and on L1 cache accesses (Sec. 3.4.4). We then evaluate its influence on scalability (Sec. 3.4.5), and assess its effect on a KNN-base recommender (Sec. 3.4.6)

3.4.1 Computation time and KNN quality

The performance of GoldFinger (*GolFi*) in terms of execution time and KNN quality is summarized in Table 3.4 for the four KNN algorithms and four datasets. The columns marked *nat.* indicate the results with the native algorithms, while those marked *GolFi* contain those with GoldFinger. The columns in italics show the gain in computation time brought by GoldFinger (*gain %*), and the loss in quality (*loss*). The fastest time for each dataset is shown in bold. Excluding LSH for space reasons, the same results are shown graphically in Figures 3.5 (time) and 3.6 (quality).

Overall, GoldFinger delivers the fastest computation times across all datasets, for a small loss in quality ranging from 0.22 (with Brute Force on Gowalla) to an improvement of 0.11 (Hyrec on AmazonMovies). Excluding LSH on AmazonMovies, DBLP and Gowalla for the moment, GoldFinger is able to reduce computation time substantially, from 42.1% (NNDescent on ml1M) to 78.9% (Brute Force on ml1M), corresponding to speedups of 1.72 and 4.74 respectively.

GoldFinger only has a limited effect on the execution time of LSH on the AmazonMovies, DBLP and Gowalla datasets. This lack of impact can be explained by the characteristics of LSH and the datasets. LSH must first create user buckets using permutations on the item universe, an operation that is proportional to the number of items. At the same time, because AmazonMovies, DBLP and Gowalla are comparatively very sparse (last column of Table 2.1), the buckets created by LSH tend to contain few users. As a result the overall computation time is dominated by the creation of buckets, and the effect of GoldFinger becomes limited.

In spite of these results, GoldFinger consistently outperforms native LSH on these datasets for instance taking 62s (with Hyrec) instead of 141s with LSH on AmazonMovies (a speedup of $\times 2.27$), for a comparable quality.

datasets	algo	*comp. time (s)* **nat.**	**GolFi**	*gain%*	*KNN quality* **nat.**	**GolFi**	*loss*
ml1M	Brute Force	19.0	4.0	78.9	1.00	0.93	0.07
	Hyrec	14.4	4.4	69.4	0.98	0.92	0.06
	NNDescent	19.0	11.0	42.1	1.00	0.93	0.07
	LSH	9.5	**3.0**	68.4	0.98	0.92	0.06
ml10M	Brute Force	2028	606	70.1	1.00	0.94	0.06
	Hyrec	314	**110**	65.0	0.96	0.90	0.06
	NNDescent	374	147	60.7	1.00	0.93	0.07
	LSH	689	255	63.0	0.99	0.94	0.06
ml20M	Brute Force	8393	2616	68.8	1.00	0.92	0.08
	Hyrec	842	**289**	65.7	0.95	0.88	0.07
	NNDescent	919	383	58.3	0.99	0.92	0.07
	LSH	2859	1060	62.9	0.99	0.93	0.06
AM	Brute Force	1862	435	76.6	1.00	0.96	0.04
	Hyrec	235	**62**	73.6	0.82	0.93	−0.11
	NNDescent	324	91	71.9	0.98	0.95	0.03
	LSH	144	141	2.1	0.98	0.96	0.02
DBLP	Brute Force	100	46	54.0	1.0	0.82	0.18
	Hyrec	46	27	41.3	0.86	0.81	0.05
	NNDescent	31	**24**	22.6	0.98	0.82	0.16
	LSH	40	38	2.6	0.87	0.86	0.01
Gowalla	Brute Force	160	54	66.3	1.0	0.78	0.22
	Hyrec	39	**22**	43.6	0.95	0.78	0.17
	NNDescent	45	26	42.2	1.0	0.79	0.21
	LSH	30	27	3.7	0.87	0.82	0.05

Table 3.4 – Computation time and KNN quality with native algorithms (*nat.*) and GoldFinger (GolFi). GoldFinger yields the shortest computation times across all datasets (in bold), yielding gains (*gain*) ranging of up to 78.9% against native algorithms. The loss in quality is moderate to nonexistent, ranging from 0.22 to an improvement of 0.11.

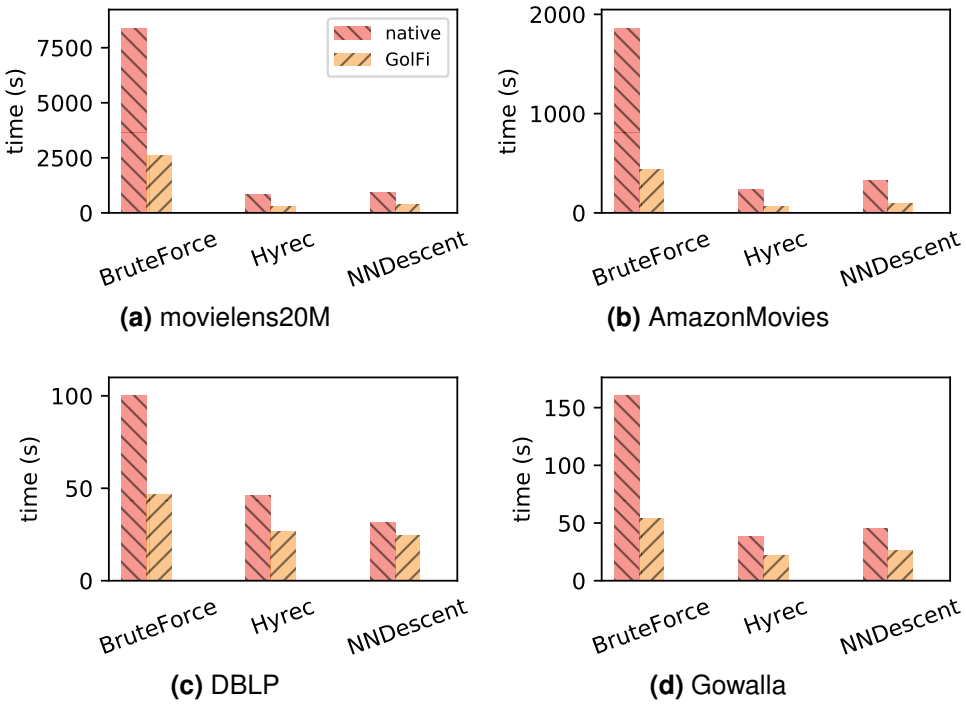


Figure 3.5 – Execution time using a 1024 bits SHF (lower is better). GoldFinger (GolFi) outperforms Brute Force, Hyrec and NNDescent in their native version on the four datasets.

3.4.2 Comparison with LP

A natural question is how GoldFinger performs compared to LP. Because of the high influence of the sampling parameter for LP, we compared them for a similar size. With $b = 1024$ bits SHFs, GoldFinger uses a total of 1068 bits per user since the cardinality is also stored. For LP, 1068 represents between 30 and 40 items. So we compare GoldFinger (labeled *GolFi*) with both these configurations (labeled $s = 30$ and $s = 40$) of LP, and with the baseline (labeled *bas.*). The results on ml10M with the brute force approach are represented on Figure 3.7. For a similar similar, GoldFinger outperforms LP.

To estimate the raw speed-up, we compare the three approaches while computing 10000 similarity computations on movielens10M and AmazonMovies. Table 3.5 shows the results. GoldFinger achieves speeds-up up to 8.94 compared to LP, while using approximately the same number of bits.

3.4.3 Breakdown of execution time

Figure 3.8 shows the breakdown of the execution time of Hyrec and NNDescent on ml10M, with and without GoldFinger, in terms of similarity computations (*sim*) and bookkeeping (*BK*). In both cases, GoldFinger is able to drastically reduce the cost of

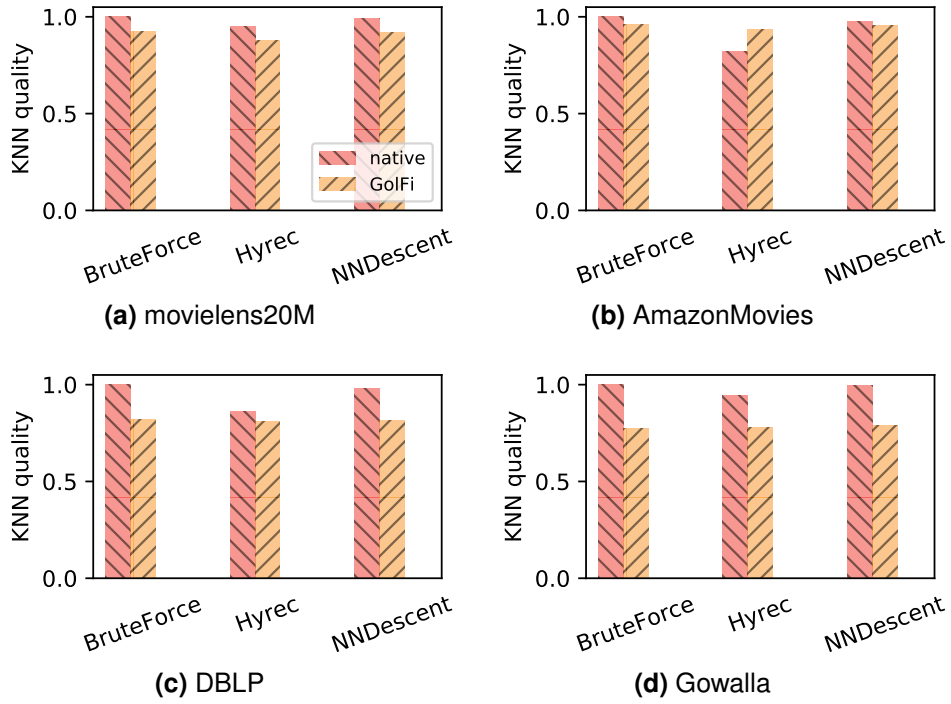
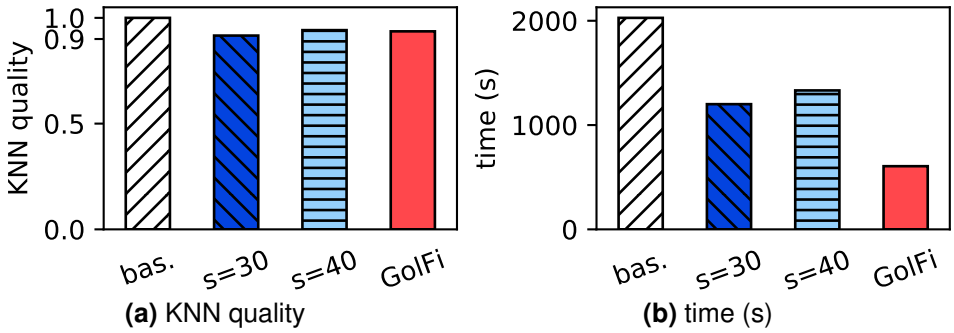


Figure 3.6 – KNN quality using a 1024 bits SHF (higher is better). GoldFinger (GolFi) only experiences a small decrease in quality.

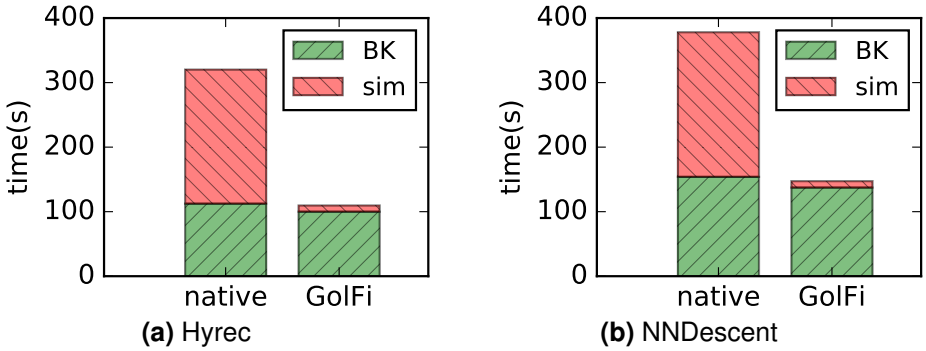
	GoldFinger	LP 30	Speed-up	LP 40	Speed-up
ml10M	2216.32	14149.76	$\times\mathbf{6.38}$	19808.74	$\times\mathbf{8.94}$
AM	2103.98	14264.16	$\times\mathbf{6.78}$	17172.8	$\times\mathbf{8.16}$

Table 3.5 – Comparison of the time spend (ms) to compute 10000 similarity computations on movielens10M and AmazonMovies using LP ($s = 30$ and $s = 40$) and GoldFinger ($b = 1024$). GoldFinger achieves speeds-up up to $\times 8.94$ compared to LP, while using approximately the same number of bits.



(a) KNN quality **(b)** time (s)

Figure 3.7 – Comparisons of the performances of LP with a sampling size of 30 and 40 with GoldFinger of $b = 1024$ bits. For a similar quality, GoldFinger outperforms LP.



(a) Hyrec **(b)** NNDescent

Figure 3.8 – Breakdown of computation time between the similarities computation (*sim*) and bookkeeping (*BK*) on ml10M. GoldFinger (GolFi) substantially reduces the computation part of similarities.

similarity computations, as anticipated. The computation time of similarities goes on average from 216s without GoldFinger to 9.8s with GoldFinger (a speedup of 22, in line with the micro-benchmark of Table 3.1), while the bookkeeping part remains roughly constant.

3.4.4 Memory and cache accesses

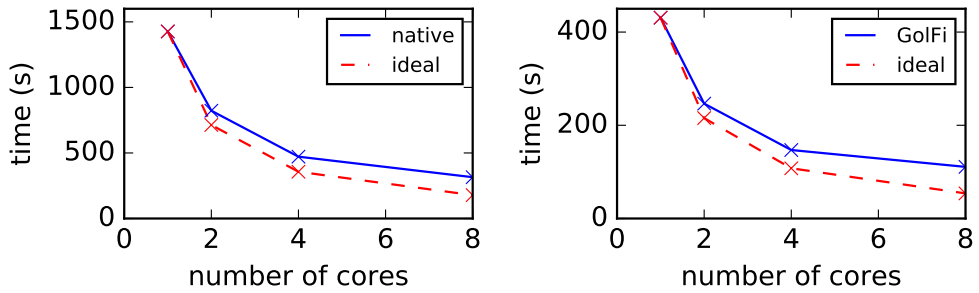
By compacting profiles, GoldFinger reduces the amount of memory needed to process of dataset. To gauge this effect, we use the performance tool `perf` [3] to observe the behavior of GoldFinger in terms of memory accesses. `perf` uses hardware counters to measure the accesses to the cache hierarchy (L1, LLC, and physical memory). To eliminate accesses performed during the dataset preparation, we subtract the values returned by `perf` when only preparing the dataset from the values obtained on a full execution.

Table 3.6 summarizes the measures obtained on Brute Force, Hyrec, NNDescent

3. https://perf.wiki.kernel.org/index.php/Main_Page

	L1 stores ($\times 10^{12}$)			*L1 loads* ($\times 10^{12}$)		
algo	**nat.**	**GolFi**	*gain%*	**nat.**	**GolFi**	*gain%*
Brute Force	2.82	0.34	87.9	8.26	1.08	86.9
Hyrec	0.35	0.08	77.1	1.14	0.28	75.4
NNDescent	0.57	0.16	71.9	1.93	0.59	69.4
LSH	0.84	0.85	−1.19	2.96	2.90	2.03

Table 3.6 – L1 stores and L1 loads with the native algorithms (*nat.*) and GoldFinger (GolFi) on ml10M. GoldFinger drastically reduces the number of L1 accesses, yielding reductions (*gain*) ranging from 67.2% to 87.7%.



(a) Native Hyrec **(b)** Hyrec + GoldFinger

Figure 3.9 – Effect of the number of cores on Hyrec for ml10M. GoldFinger (GolFi) preserves the scalability of the algorithm.

and LSH on ml10M, both without (*native*) and with GoldFinger (*GolFi*). We only show L1 accesses for space reasons, since LLC and RAM accesses are negligible in comparison. Except on LSH, GoldFinger significantly reduces the number of L1 cache loads and stores, confirming the benefits of GoldFinger in terms of memory footprint and usage. For LSH, L1 accesses are almost not impacted by GoldFinger. Again, we conjecture this is because memory accesses are dominated by the creation of buckets, rather by similarity computations, even if on this dataset, the acceleration provided by GoldFinger remains visible on the global computation time.

3.4.5 Scalability: number of cores

Because GoldFinger modifies both the memory accesses and the breakdown between different computation activities, it could perturb the native scalability of the algorithm it is applied to. Figure 3.9 shows this is not the case, by plotting the execution time of Hyrec on ml10M both with and without GoldFinger when increasing the number of available cores from 1 to 8. The dotted-line (labeled *ideal*) represents an algorithm that would scale perfectly: for a given number of cores, its represented value is the value for one core divided by the number of cores. Clearly GoldFinger, in addition of

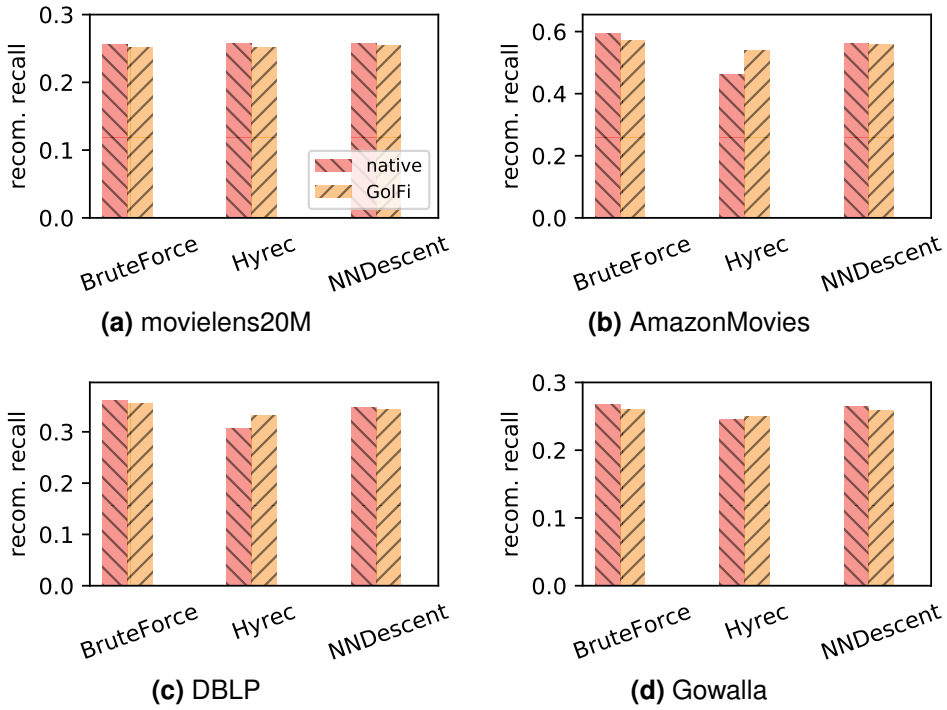


Figure 3.10 – Recommendation quality using a 1024 bits SHF (higher is better). GoldFinger's (GolFi) recall loss is negligible.

the speed-up, preserves the scalability of the underlying algorithm.

3.4.6 GoldFinger in action

As with LP (Sec. 2.4.5), we evaluate the applicability of GoldFinger in the context of a concrete application, namely a recommender. Item recommendation is one of the main applications of KNN graphs, and consists in providing every user with a list of items she is likely to rate positively. To do so, we compute for each user u and each item i not known to u that is present in u's KNN neighborhood a score $score(u,i)$, using a weighted average of the ratings given by other users in u's KNN:

$$score(u,i) = \frac{\sum_{v \in \widehat{knn}(u)} r(u,i) \times sim(u,v)}{\sum_{v \in \widehat{knn}(u)} sim(u,v)}.$$

Using the KNN graphs computed for the previous sections, we recommend 30 items to each user in every dataset. Since we use a 5-fold cross validation, we use the 1/5 of each dataset not used in an experiment as our testing set, and consider a recommendation successful if the user has produced a positive rating for the recommended item in the testing set. We evaluate the quality recommendation using *recall*, i.e. the number of successful recommendations divided by the number of positively rated items hidden in the testing set.

Figure 3.10 shows the recall of the recommendation made with the native algorithms and with their GoldFinger counterparts on all datasets. These results clearly show that the small drop in quality caused by GoldFinger has no impact on the outcome of the recommender, confirming the practical relevance of GoldFinger as a generic acceleration technique for KNN computation.

3.5 Sensitivity Analysis

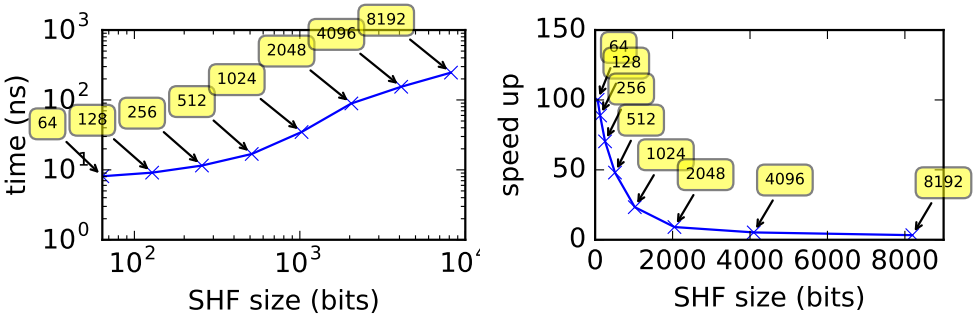
The performance of GoldFinger depends on a number of key parameters and external factors: the size of the SHFs, the chosen hash function and the properties of the dataset. In the following, we study the impact of each of these factors in turn. Unless stated otherwise, the parameters are the same as in the previous section: in particular we set the default size of SHFs to 1024. Except when considering how the characteristics of the underlying dataset impacts GoldFinger, we also focus on ml10M.

3.5.1 Size of the SHFs

As explained in Section 3.2, the SHF size determines the number of collisions occurring when computing SHFs, and when intersecting them. It thus affects the obtained KNN quality. Shorter SHFs also tend to deliver higher speedups, resulting in an inherent trade-off between execution time and quality.

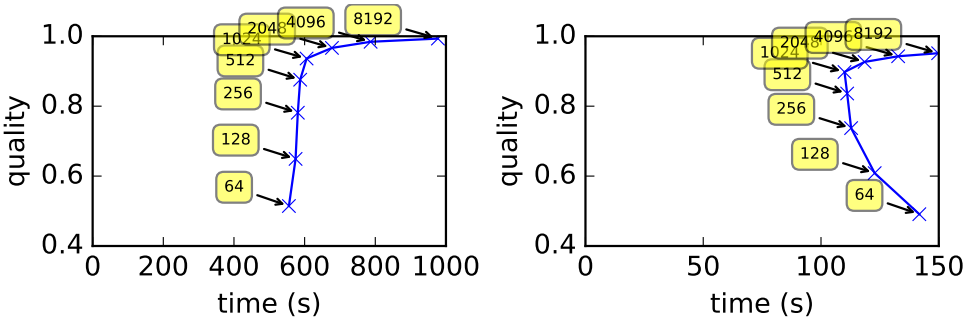
Impact on the similarity computation time

SHFs aim at drastically decreasing the cost of individual similarity computations. To assess this effect, Figure 3.11 shows the average computation time of one similarity computation when using SHFs (Equation 3.1), and its associated speed-up. The measures were obtained by computing with a multithreaded program the similarities between two sets of 5×10^4 users, sampled randomly from ml10M. The first set of users is divided in several parts, one for each thread. On each thread, each user of the part of the first set is compared to every user of the second set. The total of time needed is divided by the total number of similarities computed, 2.5×10^9. The presented results are averaged over four runs. As expected, the computation time is linear in the size of the SHF. Computation time spans from 8 nanoseconds to 250 nanoseconds using SHF, against 800 nanoseconds with real profiles. The other datasets show similar results.



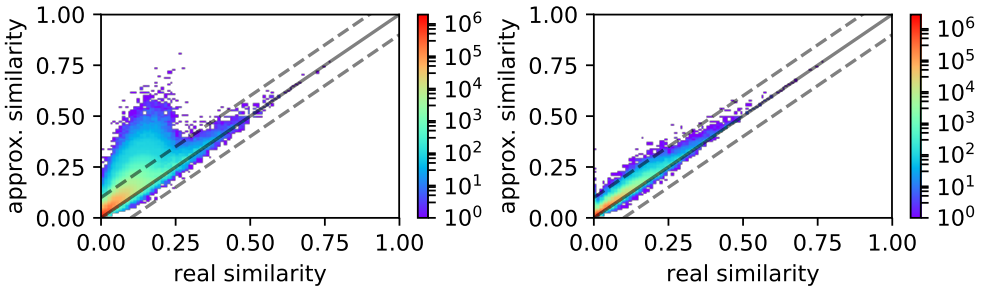
(a) Time of one similarity computation **(b)** Speed-up

Figure 3.11 – Effect of the size of the SHF on the similarity computation time, on ml10M. The computation time is roughly proportional to the size of SHFs.



(a) Brute Force **(b)** Hyrec

Figure 3.12 – Relation between the execution time and the quality in function of the size of SHF.



(a) 1024 bits **(b)** 4096 bits

Figure 3.13 – Heatmaps similarities on ml10M. The distortion of the similarity decreases when the size of SHF augments.

Impact on the execution of the algorithm

Figure 3.12 shows how the overall execution time and the quality of Brute Force and Hyrec evolve when we increase the size of the SHFs. (LSH presents a similar behavior to that of Brute Force, and NNDescent to that of Hyrec.)

As expected, larger SHFs cause Brute Force to take longer to compute, while delivering a better KNN quality (Figure 3.12a). The overall computation time does not exactly follow that of individual similarity computations (Figure 3.11a), as the algorithm involves additional bookkeeping work, such as maintaining the KNN graph, and iterating over the dataset.

The KNN quality of Hyrec shows a similar trend, increasing with the size of SHFs. The computation time of Hyrec presents however an unexpected pattern: it first *decreases* when SHFs grow from 64 to 1024 bits, before increasing again from 1024 to 4096 bits (Figure 3.12b). This difference is due to the different nature of the two approaches. The Brute Force algorithm computes a fixed number of similarities, which is independent of the distribution of similarities between users. By contrast, Hyrec adopts a greedy approach: the number of similarities computed depends on the iterations performed by the algorithm, and these iterations are highly dependent on the distribution of similarity values between pairs of users (what we have termed the *similarity topology* of the dataset), a phenomenon we return to in the following section.

Impact on estimated similarity values

Figure 3.13 shows how SHFs tend to distort estimated similarity values between pairs of users in ml10M, when using 1024 (Figure 3.13a) and 4096 bits (Figure 3.13b). The x-axis represents the real similarity of a pair of users $(u, v) \in U^2$, the y-axis represents its similarity obtained using GoldFinger, and the z-axis represents the number of pairs whose real similarity is x and estimated similarity y. (Note the log scale for z values.) The solid diagonal line is the identity function ($x = y$), while the two dashed lines

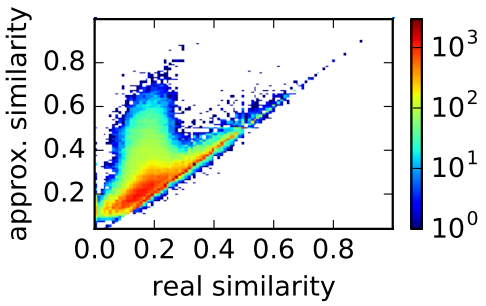


Figure 3.14 – Heatmap of the pair of the KNN graph obtained on ml10M with GoldFinger. The majority of pairs are concentrated around the diagonal, higher than the pairs of the total dataset.

demarcate the area in which points are at a distance lower than 0.1 from the diagonal. These figures were obtained by sampling 10^8 pairs of users of ml10M. Due to technical reasons we had to divide each z-value by 10.

The closer points lay to the diagonal ($x = y$), the more accurate the estimation of their similarity by GoldFinger. Points above $x = y$ indicate that the SHFs are over-approximating Jaccard's index, while points below correspond to an under-approximation. Collisions between items within the same profile tend to cause over-approximations, and thus to move pairs over $x = y$, while collisions between items of different profiles tend to cause under-approximations, and thus to move pairs under $x = y$. Figure 3.13 shows that the size of SHFs strongly influences the accuracy of the Jaccard estimation: while points tend to cluster around $x = y$ with 4096-bits SHFs (Figure 3.13a), the use of 1024 bits generate collisions that lead GoldFinger to overestimate low similarities (Figure 3.13b).

To understand why GoldFinger performs well despite this distortion, we analyze more carefully the distribution of pairs in Figure 3.13a. Most of the pairs (94%) of users have an exact similarity below 0.1. Even with 1024 bits, an overwhelming majority (92%) of these turn out to also have an approximated similarity below 0.1. This confirms our initial intuition (Section 3.2): two users with low similarity are likely to get a low approximation using GoldFinger.

Although the area $[0, 0.1] \times [0, 0.1]$ is where most of the pairs of users lay, interesting pairs are however not concentrated there. Indeed, the pairs of users present in the KNN (as directed edges) show higher similarities: less than 1% can be found in the area $[0, 0.1] \times [0, 0.1]$. To understand what happens for the rest of the pairs, we focus on the number of total pairs which are at a distance lower than ϵ of the diagonal. The distribution with $b = 1024$ and 4096 is represented in Figure 3.15. First, Figure 3.15 confirms the impact of the size on the similarity: the larger the SHFs, the lower the error. Then, with SHFs of size 1024, 52% of the pairs are a distance lower than $\epsilon = 0.01$, 75% for $\epsilon = 0.02$, 94% for $\epsilon = 0.05$ and 99% for $\epsilon = 0.1$. This means that a large majority

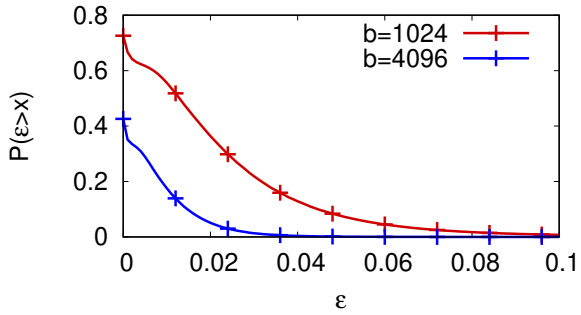
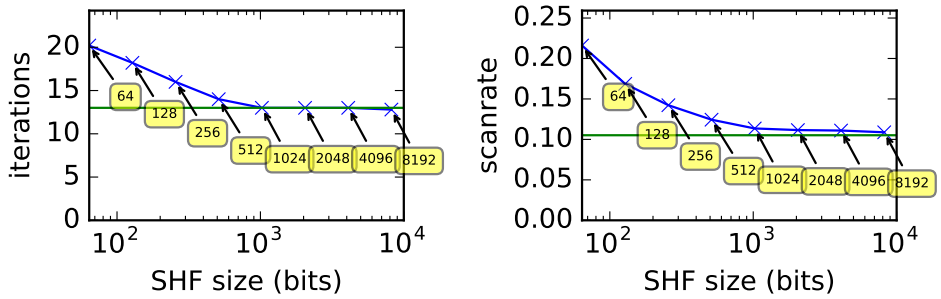


Figure 3.15 – Distribution of the error of the similarity while using GoldFinger with $b = 1024$ and $b = 4096$ on movielens10M. The shorter the profiles the larger the error.



(a) Number of iterations

(b) Scanrate

Figure 3.16 – Effect of compression on the convergence of Hyrec on ml10M. GoldFinger converges to the native approach when the size of SHF augments.

of pairs do not see their similarity changed much by the use of SHFs. The pairs that experience a large variation between their real and their estimated similarity are too few in numbers to have a decisive impact on the quality of the resulting KNN graph.

To confirm this observation, we focus on the KNN graph obtained by GoldFinger with 1024-bit SHFs and the Brute Force algorithm. Figure 3.14 only contains pairs of users that are present in the neighborhood of each other (with $k = 30$ neighbors): I.e. a pair (u, v) is only present if $v \in \widehat{knn}(u)$. Figure 3.14 confirms that only a minority of KNN neighbors experience a large similarity distortion.

The distortion of the similarity is not sufficient to significantly change the neighborhood of users resulting from an exhaustive search. This is why the only effect on the Brute Force algorithm is a decrease in execution time along with a small drop in quality.

Hyrec and NNDescent, however, iterate recursively on node neighborhoods, and are therefore more sensitive to the overall distribution of similarity values. The recursive effect is the reason why—somewhat counter-intuitively—Hyrec and NNDescent's execution time first decreases as SHFs grow in size (as mentioned when we discussed Figure 3.12).

To shed more light on this effect, Figure 3.16 shows the number of iterations and the corresponding scanrate performed by Hyrec for SHF sizes varying from 64 to 8192 bits.

The scanrate is the number of similarity computations executed by Hyrec+GoldFinger divided by the number of comparisons performed by the Brute Force algorithm, $n \times (n-1)/2$. The green horizontal line represents the results when using native Hyrec. As expected, the behavior of the GoldFinger version converges to that of native Hyrec as the size of the SHFs increases. Interestingly, short SHFs (< 1024 bits) cause Hyrec to require more iterations to converge (Figure 3.16a), leading to a higher scanrate (Figure 3.16b), and hence more similarity computations. When this occurs, the performance gain on individual similarity computations (Figure 3.11) does not compensate this higher scanrate, explaining the counter-intuitive pattern observed in Figure 3.12b.

3.5.2 The hash function

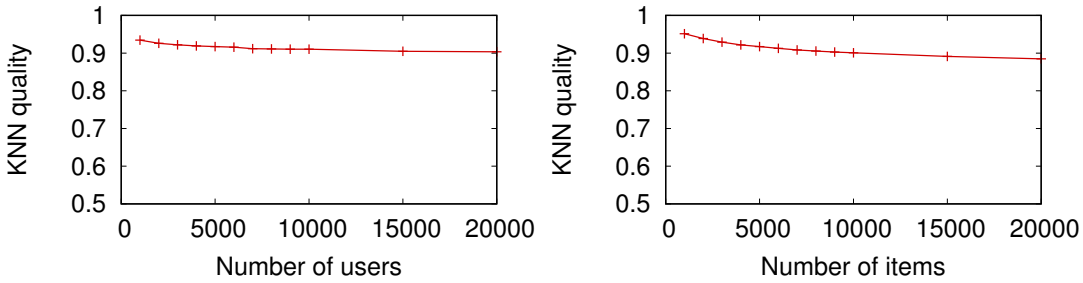
The hash function used to compute SHFs is central to GoldFinger's behavior, as it determines the index of each item in an SHF. In particular, the hash function controls the number of collisions, and thus the distortion on the similarity. Such a distortion has an impact on the resulting KNN graph and the execution time as seen in the previous sections.

We studied the influence on ml10M of three distinct common hash functions: Jenkins' hash function [Jen97] (which we have used in our experiments), the modulo function applied to item IDs, and SHA-512. The main influence of the hash function is on the creation time: using SHA-512 increases by more than 10 times the dataset creation time reported in Table 3.3 (Section 3.3) because of the high cost of hashing items. All other metrics (execution time, quality and recall) remain however similar across all three hashing methods.

Let us note that uniform random hashing is not the only available strategy. Although we did not test it, it may be possible to design a biased hash function that reduces the distortion introduced by SHFs, thus improving quality.

3.5.3 Impact of the dataset

As seen in Section 3.4, GoldFinger provides a lower KNN quality on DBLP and Gowalla although these two datasets appear similar to AmazonMovies. To better understand how the characteristics of a dataset impact the behavior of GoldFinger, we apply GoldFinger to a series of synthetic datasets in which we modify only one parameter at a time. The parameters we are interested in are the number of users, the number of items, and the spread of ratings among items.



(a) Impact of $|U|$ **(b)** Impact of $|I|$

Figure 3.17 – GoldFinger maintains a high quality when increasing the number of users or items. (Brute Force+GoldFinger, $b = 1024$ bits, synthetic datasets with default values $|U| = 5,000$, $|I| = 5,000$, and Zipfian distribution of users and items, with exponent 1)

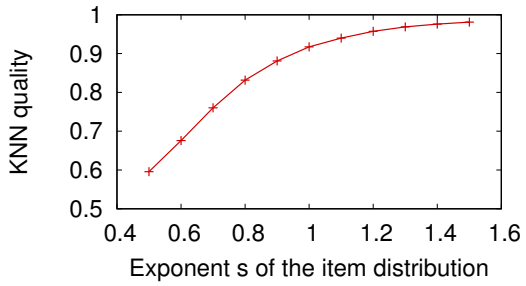


Figure 3.18 – GoldFinger exploits the concentration of ratings among a few items found in many datasets, shown here when increasing the exponent s of the item distribution.

Experimental methodology

We generate synthetic datasets by fixing a number of users $|U|$ and a number of items $|I|$. We then generate ratings by drawing user-item pairs $(u, i) \in I \times U$ according a Zipfian distribution on both items and users. We use a Zipfian exponent of 1 for users, and s for items. To avoid the cold start problem, we further add 20 ratings to every users (still drawing items according to a Zipfian distribution). In total, we draw $80 \times |U|$ ratings.

By default, we set $|U| = 5,000$, $|I| = 5,000$, and $s = 1$. Starting from this default configuration, we vary $|U|$, $|I|$ and s while keeping the other parameters at their default value. We vary $|U|$ and $|I|$ from $1,000$ to $20,000$, and s from 0.5 (corresponding to a higher spreading of ratings among items) to 1.5 (corresponding to a higher concentration of ratings on popular items).

We study the impact of the changes in the dataset on the KNN quality by using the brute force algorithm only with SHFs of $b = 1024$ bits. The brute force algorithm allows us to focus on the raw impact of the changes in the dataset, without any interfering mechanism such as the convergence speed of Hyrec or NNDescent that we discussed in Section 3.5.1.

Number of items matter more than number of users

Figures 3.17a and 3.17b show the impact of the number of users and items on the KNN quality. In both cases, a larger set lowers the KNN quality. This is because increasing the number of users or items increases the chances of collisions. On average, $|I|/b$ items are hashed onto each bit, increasing the distortion introduced by SHFs as $|I|$ grows (see Section 3.2.2). Similarly, when computing the KNN of a user u, a larger user set U increases the chances that a user v that is dissimilar from u will obtain a compacted profile that is close enough to that of u to end up in u's approximated KNN, thus increasing the number of *inter-profile collisions*.

Although both curves show the same trend, the impact on the KNN quality remains limited: increasing the number of items from 1000 to 20000 decreases the quality by 0.066, while increasing the number of users over the same range causes a drop of 0.031. The impact of items is also stronger than that of users. This difference explains why GoldFinger performs particularly well on movielens10M, while showing higher quality losses on DBLP and Gowalla.

Unbalanced item popularity is better

Figure 3.18 shows the impact of the distribution of the ratings among items on the KNN quality. GoldFinger leverages the inherently skewed distribution of many entity-item datasets, which often contain a few highly popular items and a long tail of less frequent items. With higher s values, more ratings are concentrated into a small subset of items, reducing the frequency of collisions, and helping GoldFinger achieve a higher quality.

3.6 Conclusion

In this chapter, we have proposed *fingerprinting*, a new technique that consists in constructing *compact*, *fast-to-compute* and *privacy-preserving* representation of datasets. We have illustrated the effectiveness of our approach on the emblematic big-data problem of KNN graph construction, and proposed *GoldFinger*, a novel generic mechanism to accelerate the computation of Jaccard's index while protecting users' privacy. GoldFinger exploits *Single Hash Fingerprints*, a compact, binary, and fast-to-compute representation (i.e. a *fingerprint*) of the entities of a dataset.

Our extensive evaluation shows that GoldFinger is able to drastically accelerate the construction of KNN graphs against the native versions of prominent KNN construction algorithms such as NNDescent or LSH while incurring a small to moderate loss in quality, and close to no overhead in dataset preparation compared to the state of the

art. We have also precisely characterized the privacy protection provided by GoldFinger in terms of k-anonymity and ℓ-diversity. These properties are obtained for free.

CHAPTER 4

CLUSTER-AND-CONQUER: WHEN GRAPH LOCALITY MEETS DATA LOCALITY

4.1 Introduction

Approximating the set of candidates while building a KNN graph allows a substantial decrease in the computation time compared to a brute force approach. Yet, most of the remaining computation time is spent computing similarities. In the previous chapter we have seen that approximating the similarity shifts the bottleneck. The new bottleneck is no longer related to the similarities: neither to their number nor to how they are computed. The new bottleneck is due to *memory accesses*. In the greedy approaches, updating the neighborhood of a user requires to retrieve the KNNs of its current neighborhood and the associated k^2 profiles. All these data accesses create data contention. In this chapter we address this new bottleneck by **approximating the graph locality** to increase data locality and decrease memory contention.

GoldFinger has reduced the time spent on computing similarities to a point where it is negligible compared to the overall computation time. The bottleneck has shifted now from computation time to memory accesses. As shown on Fig. 3.8 (Sec. 3.4.3), the large majority of the total computation time is now due to bookkeeping operations, i.e. involving updates to shared data structures and accessing the users' profiles and datastructures associated to KNN. By language abuse, we use KNN for the datastructure storing the list of neighbors, see the Appendix A for more details on this datastructure. The new issue is now the access to the users' profiles and the datastructures storing the KNNs in memory. The time spent accessing the data may arise from two phenomenons: the synchronization overheads and the fact that memory accesses are random. Synchronization is used to prevent the KNN to be concurrently changed by two threads in inconsistent ways. Only the accesses to the KNNs are synchronized, however, the profiles are not since they are only read, never altered.

By removing all the synchronization the computation time does not change, only the obtained KNN graph has changed, suffering from some erroneous values because of the concurrent modifications of the KNNs. This result shows that the new bottleneck is the memory accesses done while accessing the profiles and the KNNs. Although

the greedy approaches rely on a strategy which can be seen as "local", in practice the profiles and KNNs are randomly stored in memory, regardless of the graph topology of the KNN once computed. Accessing the profiles of the neighbors results in random accesses which significantly slow down the computation. Decreasing these random accesses is possible by improving the data locality, i.e. by storing data which will be used at the same time close to each other. Improving the locality of data is one of the main concern of out-of-memory algorithms which try to organize the data the same way it is used and therefore accessed: for KNN graph computation, they try to gather neighbors' data at the same place.

For in-memory algorithm, a generic way to increase data locality consists in clustering the users and then solving locally the problem, using a divide and conquer strategy. The challenge is how to cluster the users while keeping the neighbors together without doing any similarity computations. Interestingly enough, finding the right way to partition the data is trying to find which users are close to each other: this is precisely what building the KNN graph is about. This circular dependency is the reason why a clustering which is good and fast is difficult to achieve. There are many existing clustering techniques but most of them are expensive in term of computations. At one end of the spectrum lies the k-means algorithm [M$^+$67]. In this approach the clustering relies on many similarity computations but with no preprocessing. At the other end there is LSH [IM98, GIM$^+$99] which clusters without any similarity computations but relies on costly hash functions which makes it unusable for datasets with large item sets.

In this chapter we present ***Cluster-and-Conquer***, a novel KNN graph algorithm which approximates the graph locality by clustering users into small groups of users which are likely to be neighbors in the final KNN graph. In such small groups, there is a high spatial data locality when computing a KNN graph. Cluster-and-Conquer follows a divide and conquer strategy relying on FastMinHash, a novel, fast and accurate clustering scheme which takes the best of both the k-means world and the LSH one: it does not require any similarity computations (as LSH) nor preprocessing (as k-means). The users are clustered and KNN graphs are computed locally in the clusters, using GoldFinger, and then merged together. We present an extensive evaluation of the algorithm, performed on several real and artificial datasets, which confirms the gain of locality of the clustering scheme.

4.2 Our approach: Cluster-and-Conquer

4.2.1 Intuition

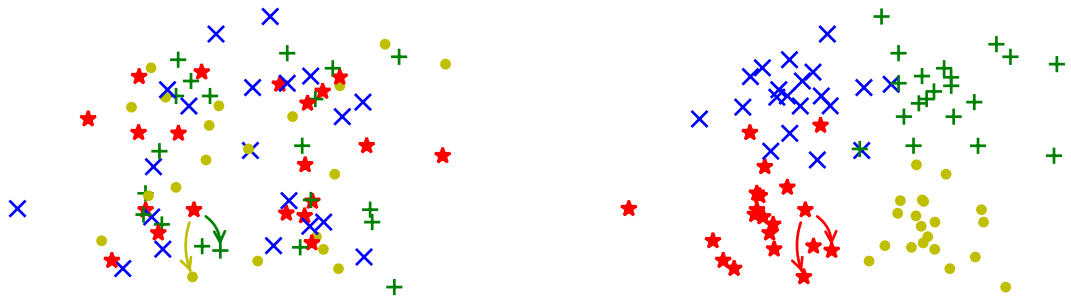
Memory accesses are the main bottleneck, not similarities. The GoldFinger approach presented in the previous chapter shifts the bottleneck from similarity compu-

tation to memory accesses. As a result, approximate KNN graph algorithms, because GoldFinger successfully achieved ultra-fast computations, spend most of their computation time accessing the profiles and KNNs of individual users. The main bottleneck is now memory accesses. The memory accesses are problematic for two reasons: *(i)* their high number and *(ii)* the fact that they are random.

— *(i)* In greedy approaches, users iteratively refine their neighborhoods by checking the neighbors of their neighbors. The approach is using basic multi-threading: each thread is assigned a given set of users and computes their KNNs. Each thread has to process the KNN of $\frac{n}{p}$ users, where p is the number of threads, and k^2 users are used to refine the KNN of each user. In total, each thread has to access up to $k^2 \times \frac{n}{p}$ profiles, per iteration. For large datasets, these profiles cannot fit into cache, resulting in many accesses to main memory causing contention.

— *(ii)* Even though the greedy scheme is labeled as a "local" search, locality here refers to a proximity in the KNN graph, it is actually ***graph locality***. The graph locality refers to the content of the profiles, but is completely orthogonal to their locations in memory, i.e. to ***data locality***. From a memory standpoint, graph locality is invisible: two users with the exact same profile will have their profiles stored in two different locations in the memory, at random. The many accesses to the memory are not sequential but random and thus costly.

Cluster-and-Conquer aligns data locality with an approximate graph locality to limit the number of memory accesses and make them sequential.



(a) With greedy approaches (such as Hyrec and NNDescent), close users are on different threads.

(b) With Cluster-and-Conquer, close users are in the same clusters.

Figure 4.1 – Two dimensional illustration of users of an artificial dataset. Each figure represents the dispersion of the users on four threads (represented by four colors) with locality unaware traditional KNN approach and Cluster-and-Conquer. Each color and marker represent a different thread.

Figure 4.1 illustrates the relation between graph locality and data locality in a standard greedy approach (such as Hyrec or NNDescent) and Cluster-and-Conquer. Each point represents a user of the dataset in a two dimensional space. Each color and

marker represent a thread: every user, represented by a blue cross, is assigned to the same thread. Figure 4.1a represents the distribution of each user in four threads while computing the KNN graph with a standard locality unaware greedy approach. Close users (graph locality) are not treated by the same threads (data locality). To compute the KNN of a user, the thread has to access the data of users which have been assigned to other threads. In total each thread will have to access not only the data of the users it is in charge of, but many other profiles and KNNs, provoking data contention. ***There is a clear mismatch between graph locality and the data locality***. Figure 4.1b represents the distribution achieved by Cluster-and-Conquer. Close users are globally in the same thread, each thread will access mostly only data associated to its users. In Cluster-and-Conquer graph locality meets data locality.

Cluster-and-Conquer clusters users into small subdatasets to increase data locality. The partial KNN graphs of these small subdatasets are computed independently for each subdataset. The higher the locality of the subdatasets the faster these KNN graphs computation. Then simply merging the obtained KNN subgraphs provides a good approximation of the complete KNN graph of the whole dataset.

Clustering is key to process smaller datasets. The real challenge is to store close to each other, in memory, the profiles which are close to each other in the graph. The difficulty is to do this grouping before computing the KNN graph because the goal of KNN is precisely to compute who is close to whom. A bad clustering will separate neighbors into different clusters, leading to a poor KNN quality. This clustering should also use as few similarities computations as possible, which are particularly costly. This eliminates approaches such as k-means which require many similarity computations. The clustering should not be based on comparisons between users and thus it should be based on the users' profiles only. Still the clustering should not rely on a costly preprocessing. This excludes approaches such as LSH which cluster users based on their profiles only, but at the expense of a prohibitive preprocessing.

Fast but approximate clustering does the job. We propose FastMinHash, a fast-to-compute hashing scheme. FastMinHash functions are used to cluster the users, by hashing each user into a cluster. FastMinHash functions are ***profile-based*** to hash users into buckets without computing any similarity between users. They rely on random hash functions for the hashes to be ***fast to compute***. Random hashing means that there is no correlation between the input and the output of the hash function. Still, the hash functions are ***deterministic***: hashing the same element twice with the same hash function will provide the same result. Random hash functions provoke collisions: users with no item in common may be hashed into the same cluster. This was impossible in LSH, in which all the users in the same buckets share at least a common item, thus have a non-zero similarity. In LSH every similarity computed was strictly superior to zero. Because of the collisions, we lose that property and we have to compute use-

less similarities. This is the price to pay for having a fast-to-compute, but approximate, hashing scheme.

Extra useless similarities is a price we can afford. Our intuition is that we can keep these extra similarities negligible. The higher the number of clusters, the less users there should be, on average, in each of them. In a small cluster, computing a KNN graph locally is fast, even if there are unnecessary similarity computations. Small clusters should then make the extra similarities negligible.

Introducing redundancy to compensate approximate clustering. Unfortunately, close users, no matter how close they are, may end up within different clusters if they do not have the same profile. Multiple clustering schemes, using different hash functions, are performed to increase the probability that neighbors end-up within the same cluster at least once. If each hash function creates b clusters, t hash functions create $t \times b$ clusters. Each of these clusters is small, thus their KNN graphs are fast to compute. The use of multiple clustering schemes should ensure that, for each user, nearly every neighbor is found at least in one partial KNN graph. Once the partial KNN graphs are merged, the complete KNN graph should have a good quality.

4.2.2 FastMinHash: fast and coarse hashing

For each user the choice of the cluster is based on the hash of her profile. No similarity computations are performed, in contrast to k-means. Two users with the same profiles should be put in the same cluster, so the hash functions should be deterministic. Also, the hashes have to be fast to compute, unlike LSH. To do so, we use random hash functions similar to the one used in GoldFinger. Then the FastMinHash function H is based on hash of every item of the profiles of a user.

Consider h a uniform random hash function on an interval of integers of size b: $[\![1, b]\!] \subset \mathbb{N}$

$$h : I \to [\![1, b]\!] \subset \mathbb{N} \tag{4.1}$$

$$i \to h(i) \in \mathbb{N} \tag{4.2}$$

For a user u, every item of u's profile is hashed using h. The hash of u using the FastMinHash function, written $H(u)$, is defined as the minimum of the hashes of its items. Choosing the minimum makes the hash function deterministic and ensure that if two users have the same profiles they will necessarily have the same hash.

$$H : U \to [\![1, b]\!] \tag{4.3}$$

$$u \to H(u) = min_{i \in P_u} h(i) \tag{4.4}$$

Every user is hashed through FastMinHash and the obtained hash is the index of the cluster the user is assigned to.

For example, we consider the hash function h with an item set $I = \{i_1, i_2, i_3, i_4, i_5\}$ and $b = 3$ clusters. We compute, using the associated FastMinHash H, the hash of two users u and v:

$$h \begin{cases} i_1 \to 2 \\ i_2 \to 3 \\ i_3 \to 2 \\ i_4 \to 1 \\ i_5 \to 3 \end{cases}$$

$$P_u = \{i_1, i_2, i_3\}$$
$$P_v = \{i_3, i_4, i_5\}$$
$$H(u) = min\{h(i_1), h(i_2), h(i_3)\} = min\{2, 3, 2\} = 2$$
$$H(v) = min\{h(i_3), h(i_4), h(i_5)\} = min\{2, 1, 3\} = 1$$

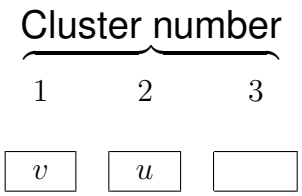


Figure 4.2 – Clustering of u and v with $b = 3$ clusters.

User u will be in the cluster whose index is 2 while v will be in the cluster of index 1. Figure 4.2 shows the clustering of u and v. Taking the minimum of the items' hashes introduces a bias towards the clusters of low index, especially if a popular item is hashed into one of the first cluster. For instance, in our example, consider a user u' with a profile equal to $\{i_1, i_2, i_3, i_4\}$. This user would have been hashed in the cluster with the index 1. She would have been in a different cluster than u, although the presence of i_4 is the only difference in their profiles. This is caused by i_4 being hashed to a lower number than the other elements. Because of the minimum, the high hashes are not as often selected as the lower ones. The corresponding clusters are filled with less users than the clusters with low indexes. We discuss the way to overcome this unbalance in Section 4.2.3.

Note that we reuse the notation b which was previously used as the size of the SHFs in the GoldFinger approach (see Chap. 3). Until the end of the chapter, b will refer to the number of clusters generated by FastMinHash functions. When we refer to GoldFinger, we assume a constant size of 1024. The use of the same notation is not fortuitous: both values refers to the cardinal of the image of hash functions.

Property of FastMinHash functions

In the following we study the property of the FastMinHash functions. More specifically we are interested in the probability of two users being hashed in the same cluster. This probability is lower-bounded by a quantity proportional to the size of the intersection of their profile. More formally, for two users u and v, we have:

$$\mathbb{P}[\min h(P_u) = \min h(P_v)] = \Omega(|P_u \cap P_v|)$$

In other words, the more items the two users have in common, the more likely they are to end up into the same cluster.

Proof:

Notation 1. *As in Chapter 3, let I be the set of items.*
Let P_A and P_B be two subsets of I.
Let P_c a subset of I of size c.
The intersection of P_A and P_B is $P_\cap = P_A \cap P_B$ and their union is $P_\cup = P_A \cup P_B$.
We note $P_\uplus = P_\cup \setminus P_\cap$, $P_{A'} = P_A \setminus P_\cap$ and $P_{B'} = P_B \setminus P_\cap$.
We define $|P_\cap| = \alpha$, $|P_\uplus| = \gamma$, $|P_{A'}| = \gamma_A$ and $|P_{B'}| = \gamma_B$.
Let b be a constant in $\mathbb{N}$.

Notation 2. *Let h be a random hash function from I to $\{1, 2, .., b\}$.*
The function is totally random so the hashing of two different items is modeled by independent events.
We extend h to $\mathcal{P}(I)$ by defining $h(P \in \mathcal{P}(I)) = \{x | \exists i \in P, h(i) = x\}$.

The problem We want to compute, for two profiles P_A and P_B, what are the probabilities that the minimum of their image by h has the same minimum.
In other words, we want to compute:

$$\mathbb{P}[\min h(P_A) = \min h(P_B)] \tag{4.5}$$

Lemmas

Lemma 1. *The probability for a profile of cardinality c to have an image whose minimum is less or equal to m is $1 - (1 - \frac{m}{b})^c$. More formally:*

$$\mathbb{P}[\min h(P_c) \leq m] = 1 - (1 - \frac{m}{b})^c \tag{4.6}$$

Proof:

$$\begin{aligned}\mathbb{P}[\min h(P_c) \leq m] &= 1 - \mathbb{P}[\min h(P_c) > m] \\ &= 1 - \prod_{i \in P_c} (\mathbb{P}[h(i) > m]) \\ &= 1 - \prod_{i \in P_c} (1 - \mathbb{P}[h(i) \leq m]) \\ &= 1 - \prod_{i \in P_c} (1 - \frac{m}{b}) \\ &= 1 - (1 - \frac{m}{b})^c\end{aligned}$$

□

Lemma 2. *For a value m such as $\frac{m}{b} \sim 0$, the probability for a profile of cardinality c to have an image whose minimum is less or equal to m can be approximated by $\frac{cm}{b}$. More precisely:*

$$\mathbb{P}[\min h(P_c) \leq m] = \frac{cm}{b} + o(\frac{m}{b}) \tag{4.7}$$

Proof:

$$\begin{aligned}\mathbb{P}[\min h(P_c) \leq m] &= 1 - (1 - \frac{m}{b})^c \\ &= 1 - (1 - \frac{cm}{b} + o(\frac{m}{b})) \\ &= \frac{cm}{b} + o(\frac{m}{b}))\end{aligned}$$

□

Lemma 3. *The probability for a profile of cardinality c to have an image whose minimum is exactly equal to m is $(1 - \frac{m}{b})^c - (1 - \frac{m-1}{b})^c$. More formally:*

$$\mathbb{P}[\min h(P_c) = m] = (1 - \frac{m}{b})^c - (1 - \frac{m-1}{b})^c \tag{4.8}$$

Proof:

$$\begin{aligned}\mathbb{P}[\min h(P_c) = m] &= \mathbb{P}[\min h(P_c) \leq m] - \mathbb{P}[\min h(P_c) \leq m-1] \\ &= (1 - (1 - \frac{m}{b})^c) - (1 - (1 - \frac{m-1}{b})^c) \\ &= (1 - \frac{m}{b})^c - (1 - \frac{m-1}{b})^c\end{aligned}$$

□

Lemma 4. *For a value m such as $\frac{m}{b} \sim 0$, the probability for a profile of cardinality c to have an image whose minimum is exactly equal to m can be approximated by $\frac{c}{b}$. More*

precisely:

$$\mathbb{P}[\min h(P_c) = m] = \frac{c}{b} + o(\frac{m}{b}) \tag{4.9}$$

Proof:

$$\begin{aligned}
\mathbb{P}[\min h(P_c) \leq m] &= (1 - \frac{m}{b})^c - (1 - \frac{m-1}{b})^c \\
&= (1 - (1 - \frac{cm}{b} + o(\frac{m}{c}))) - (1 - (1 - \frac{c(m-1)}{b} + o(\frac{m}{b}))) \\
&= -(1 - \frac{cm}{b} + o(\frac{m}{c})) + (1 - \frac{c(m-1)}{b} + o(\frac{m}{b})) \\
&= \frac{c}{b} + o(\frac{m}{b})
\end{aligned}$$

□

Property 1. *The probability for two profiles to be hashed to sets which have the same minimum is greater or equal to a value which is increasing proportionally to the cardinality of their intersection. More formally, we have:*

$$\mathbb{P}[\min h(P_A) = \min h(P_B)] = \Omega(\alpha) \tag{4.10}$$

Proof:

$$\begin{aligned}
\mathbb{P}[\min h(P_A) = \min h(P_B)] &= \mathbb{P}[\min h(P_\cap) \leq \min h(P_\uplus)] \\
&\quad + \mathbb{P}[\min h(P_\cap) > \min h(P_\uplus) \wedge \min h(P_{A'}) = \min h(P_{B'})]
\end{aligned}$$

Let us focus on the first term of the sum.

$$\begin{aligned}
\mathbb{P}[\min h(P_\cap) \leq \min h(P_\uplus)] &= \sum_{m=0}^{b-1} \mathbb{P}[\min h(P_\cap) \leq m | \min h(P_\uplus) = m] \times \mathbb{P}[\min h(P_\uplus) = m] \\
&= \sum_{m=0}^{b-1} \mathbb{P}[\min h(P_\cap) \leq m] \times \mathbb{P}[\min h(P_\uplus) = m]
\end{aligned}$$

Because $P_\cap$ and $P_\uplus$ are disjoint. There exists m_1 such as $\frac{m_1}{b} \sim 0$, so by Lemma 4:

$$\begin{aligned}
\mathbb{P}[\min h(P_\cap) \leq \min h(P_\uplus)] &\geq \sum_{m=0}^{m_1-1} \mathbb{P}[\min h(P_\cap) \leq m] \times \mathbb{P}[\min h(P_\uplus) = m] \\
&\geq \sum_{m=0}^{m_1-1} (\frac{\alpha m}{b} + o(\frac{m}{b})) \times (\frac{\gamma}{b} + o(\frac{m}{b})) \\
&\geq \sum_{m=0}^{m_1-1} \frac{\alpha m \gamma}{b^2} + o(\frac{m}{b}) \\
&\geq \sum_{m=0}^{m_1-1} \Omega(\alpha\gamma) \\
&= \Omega(\alpha\gamma)
\end{aligned}$$

Now, let us estimate the second term. Since $\min h(P_\uplus) = \min(h(P_{A'}), h(P_{B'}))$ we have:

$$\begin{aligned}
&\mathbb{P}[\min h(P_\cap) > \min h(P_\uplus) \wedge \min h(P_{A'}) = \min h(P_{B'})] \\
&\qquad = \mathbb{P}[\min h(P_\cap) > \min h(P_{A'}) \wedge \min h(P_{A'}) = \min h(P_{B'})]
\end{aligned}$$

We sum for all the possible values K of $h(P_\uplus)$:

$$\begin{aligned}
&\mathbb{P}[\min h(P_\cap) > \min h(P_\uplus) \wedge \min h(P_{A'}) = \min h(P_{B'})] \\
&\qquad = \sum_{K=0}^{b-1} \mathbb{P}[\min h(P_\cap) = K \wedge \min h(P_\cap) > \min h(P_{A'}) \wedge \min h(P_{A'}) = \min h(P_{B'})] \\
&\qquad = \sum_{K=0}^{b-1} \mathbb{P}[\min h(P_\cap) = K \wedge K > \min h(P_{A'}) \wedge \min h(P_{A'}) = \min h(P_{B'})] \\
&\qquad = \sum_{K=0}^{b-1} \mathbb{P}[\min h(P_\cap) = K] \times \mathbb{P}[K > \min h(P_{A'}) \wedge \min h(P_{A'}) = \min h(P_{B'})] \\
&\qquad = \sum_{K=0}^{b-1} \mathbb{P}[\min h(P_\cap) = K] \times \left(\sum_{m=0}^{K-1} \mathbb{P}[\min h(P_{A'}) = m] \times \mathbb{P}[\min h(P_{B'}) = m] \right)
\end{aligned}$$

There exists m_2 such as $\frac{m_2}{b} \sim 0$, so by Lemma 4:

$$\begin{aligned}
&\mathbb{P}[\min h(P_\cap) > \min h(P_\uplus) \wedge \min h(P_{A'}) = \min h(P_{B'})] \\
&\quad \geq \sum_{K=0}^{m_2-1} \mathbb{P}[\min h(P_\cap) = K] \times \left(\sum_{m=0}^{K-1} \mathbb{P}[\min h(P_{A'}) = m] \times \mathbb{P}[\min h(P_{B'}) = m] \right) \\
&\quad \geq \sum_{K=0}^{m_2-1} (\frac{\alpha}{b} + o(\frac{K}{b})) \times \left(\sum_{m=0}^{K-1} \mathbb{P}[\min h(P_{A'}) = m] \times \mathbb{P}[\min h(P_{B'}) = m] \right)
\end{aligned}$$

There exists m_3 such as $\frac{m_3}{b} \sim 0$, so by Lemma 4:

$$\begin{aligned}
&\mathbb{P}[\min h(P_\cap) > \min h(P_\uplus) \wedge \min h(P_{A'}) = \min h(P_{B'})] \\
&\quad \geq \sum_{K=0}^{m_2-1} (\frac{\alpha}{b} + o(\frac{K}{b})) \times \left(\sum_{m=0}^{m_3-1} \mathbb{P}[\min h(P_{A'}) = m] \times \mathbb{P}[\min h(P_{B'}) = m] \right) \\
&\quad \geq \sum_{K=0}^{m_2-1} (\frac{\alpha}{b} + o(\frac{K}{b})) \times \left(\sum_{m=0}^{m_3-1} (\frac{\gamma_A m}{b} + o(\frac{m}{b})) \times (\frac{\gamma_B}{b} + o(\frac{m}{b})) \right) \\
&\quad \geq \sum_{K=0}^{m_2-1} (\frac{\alpha}{b} + o(\frac{K}{b})) \times \left(\sum_{m=0}^{m_3-1} (\frac{\gamma_A m \gamma_B}{b^2} + o(\frac{m}{b})) \right) \\
&\quad \geq \sum_{K=0}^{m_2-1} (\frac{\alpha}{b} + o(\frac{K}{b})) \times \left(\sum_{m=0}^{m_3-1} (\frac{\gamma_A m \gamma_B}{b^2} + o(\frac{m}{b})) \right) \\
&\quad \geq \sum_{K=0}^{m_2-1} \left(\sum_{m=0}^{m_3-1} (\frac{\alpha \gamma_A m \gamma_B}{b^3} + o(\frac{m+K}{b})) \right) \\
&\quad \geq \sum_{K=0}^{m_2-1} \left(\sum_{m=0}^{m_3-1} (\frac{\alpha \gamma_A m \gamma_B}{b^3} + o(\frac{m}{b})) \right) \\
&\quad \geq \sum_{K=0}^{m_2-1} \left(\sum_{m=0}^{m_3-1} \Omega(\alpha \gamma_A \gamma_B) \right) \\
&\quad = \Omega(\alpha \gamma_A \gamma_B)
\end{aligned}$$

So we have:

$$\begin{aligned}
\mathbb{P}[\min h(P_A) = \min h(P_B)] &= \mathbb{P}[\min h(P_\cap) \leq \min h(P_\uplus)] \\
&\quad + \mathbb{P}[\min h(P_\cap) > \min h(P_\uplus) \wedge \min h(P_{A'}) = \min h(P_{B'})] \\
&\geq \Omega(\alpha\gamma) + \Omega(\alpha \gamma_A \gamma_B) \\
&= \Omega(\alpha)
\end{aligned}$$

□

This result suggests that, for two users, the more items they have in common the more likely they are to be hashed into the same cluster.

4.2.3 Clustering: FastMinHash in action

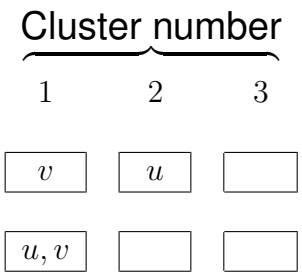
The clustering is achieved through hashing all the users with several FastMinHash functions. For each function H, every user is associated to a number between 1 and b which represents its associated cluster by H.

Hashing a user with a FastMinHash hash function H may prevent neighbors from being in the same cluster: for example if their profiles differ by one item which is hashed into a cluster lower than the rest of the items. Consider the same example as in Section 4.2.2: u and u' have a high Jaccard similarity of 0.75. Still, they are not within the same cluster because the extra item in the profile of u' has a hash lower than any other items. To reduce the probability that neighbors are hashed into different clusters, we use multiple hash functions. In practice, we change the seed of the hash function used to hash the items to produce different FastMinHash functions. For every FastMinHash function, there are b new clusters, thus the total number of clusters is $t \times b$, with t being the number of hash functions. **The probability that two neighbors are never hashed into the same cluster decreases exponentially when the number of hash functions increases**.

As an example, consider the earlier example of Section 4.2.2. We still have $I = \{i_1, i_2, i_3, i_4, i_5\}$ and $b = 3$ and we are still interested in the two users u and v. We rename the hash function h by h_1 and H by H_1. We consider another hash function h_2 and its associated FastMinHash H_2:

$$h_2 \begin{cases} i_1 \to 1 \\ i_2 \to 3 \\ i_3 \to 3 \\ i_4 \to 2 \\ i_5 \to 1 \end{cases}$$

$$H_2(u) = min\{h_2(i_1) = 1, h_2(i_2) = 3, h_2(i_3) = 3\} = 1$$
$$H_2(v) = min\{h_2(i_3) = 3, h_2(i_4) = 2, h_2(i_5) = 1\} = 1$$



The users u and v were hashed into different clusters by H_1 but they are within the same cluster when using H_2. The more hash functions we use, the higher the probability that neighbors will be hashed into the same cluster at least once.

Because of the bias introduced by the min used in the FastMinHash function, the clusters are unbalanced. Especially, if highly popular items are hashed into one of the first clusters, this cluster is likely to be filled with users while most of the following would be empty. The KNN graphs in those clusters would be so costly to compute that this might slow down the whole computation. To avoid this situation, we compute more hash functions that required, and discard the ones that perform worst in terms of item balancing. We use T hash functions, with $T > t$. We keep the t cluster configurations which provide the smallest clusters: we keep the cluster configurations with the smallest biggest clusters.

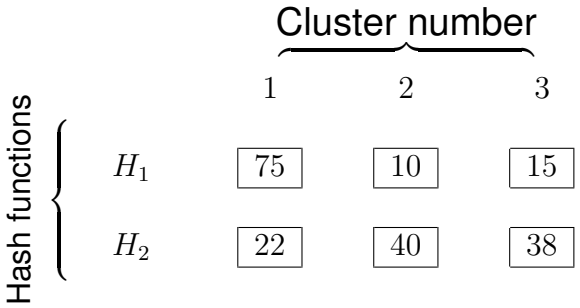


Figure 4.3 – Example of two cluster configurations of 100 users with $b = 3$. Each line represent the hashing done by a different FastMinHash function. Some functions produce really unbalanced cluster configurations.

Figure 4.3 illustrates this selection of the fittest on two cluster configurations of users with $|U| = 100$ and $b = 3$. Each line represents the hashing produced by a different FastMinHash function. The boxes represent the clusters, and the number in each cluster represents the number of users within this cluster. If we want to use only one clustering, i.e. $t = 1$ and only compute H_1, we would end up with the first one. The problem is that this clustering is highly unbalanced: the first cluster contains most of the users while the others are nearly empty. We would prefer to use the second one. The idea is to do both cluster configurations, i.e. $T = 2$, and to choose the cluster configuration whose biggest cluster is the smallest. In this example we will choose the second clustering because its most filled cluster has a size of 40 which is lower than 75.

The higher T, the number of candidate clustering configuration we test, the more balanced the final t cluster configurations, thus the faster the computation. Still, computing too many tentative clustering configurations can be time consuming. In practice we choose $T = 2 \times t$.

The pseudocode of the clustering scheme is shown in Algorithm 4. The variables

Algorithm 4 Cluster-and-Conquer clustering scheme

Step 1: the clustering
for $i \in [\![1, t]\!]$ **do**
  $B_i \leftarrow$ new $Set\langle U\rangle[b]()$
end for
for $u \in U$ **do**
  for $i \in [\![1, t]\!]$ **do**
    $B_i[H_i(u)] \leftarrow B_i[H_i(u)] \cup \{u\}$
  end for
end for
return $(B_i)_{i\in[\![1,t]\!]}$

$(B_i)_{i\in[\![1,t]\!]}$ are arrays of clusters. There are t of them, one for each hash function. Each B_i is of size b, the number of cluster per hash function H_i. Each $B_i[j]$ is a cluster, represented by a set of users, so that $\bigcup_{j=1}^{b} B_i[j] = U$.

4.2.4 Scheduling: everybody is equal

Testing more hash functions lowers the computation time but the clusters remain unbalanced. To prevent an unbalanced workload among the threads, we do some light-weight work scheduling.

We want to have the same computation time for each thread. To do so we use a basic thread pool. The clusters are stored in a priority queue. The priority of each cluster is its number of users. They are stored in a decreasing order: the largest clusters are processed first.

Each thread accesses the thread pool, retrieves the first subdataset and then computes the associated KNN. The computed KNN is stored in a list of KNNs. Then the thread starts again the process, until the priority queue is empty. To avoid any concurrent access, the priority queue uses synchronization.

4.2.5 The local KNN computation

In each cluster c obtained through the previous step, we compute the KNN graph of the subdataset. Any approach can be used to compute their KNN graph. Depending on the number of users n_c we use either the brute force algorithm or a KNN graph algorithm. In practice we use Hyrec but any other KNN graph algorithm can be used. We chose the approach with the lower number of similarities. The brute force approach computes $\frac{n_c \times (n_c-1)}{2}$ similarities, while with Hyrec the number of computed similarities can be approximated by (in practice there is a flag mechanism to avoid redundant similarity computations, so the real number is lower) $\frac{\rho \times k^2 \times n}{2}$, where ρ is the number of iterations. So when $n_c < \rho \times k^2$ we chose the brute force approach, Hyrec otherwise.

In practice we take $\rho = 5$. To speed-up the computation, we use GoldFinger with SHFs of size 1024. The pseudo-code is represented in Algorithm 5.

Algorithm 5 Cluster-and-Conquer algorithm local computing scheme

```
Step 3: computing the KNN graphs
for c ∈ C_i do
    if n_c < ρ × k² then
        return BruteForce(c)
    else
        return Hyrec(c)
    end if
end for
```

4.2.6 The conquer step: merging the KNN graphs

We merge the KNN graphs obtained in each cluster, one by one, into a unique KNN graph knn. The merging is done at the granularity of each user. Each user is in t different clusters, so for each user is connected to up to $t \times k$ other users. For each cluster c, we retrieve the corresponding KNN graph knn'. The neighborhood $knn'(u)$ of each user u is obtained, and each neighbor $v \in knn'(u)$ is added to u's neighborhood in knn. The similarities of the neighbors are stored along the user in each $knn'(u)$ so there is no similarity computation performed while adding neighbors to $knn(u)$. Only the ones with the neighbors with the highest scores are kept. The resulting KNN graph knn is returned. The pseudocode is presented in Algorithm 6. The KNN $knn(u)$ of each user u is seen as a heap of pairs (user,similarity) (represented as (v, s) in Algo. 6).

Algorithm 6 Cluster-and-Conquer algorithm merging scheme

```
Step 4: the conquer step
knn ← new knn()
for i ∈ [[1, t]] do
    for c ∈ C_i do
        knn' ← c.knn()
        for u ∈ knn' do
            for (v, s) ∈ knn'(u) do
                knn(u).add(v, s)
            end for
        end for
    end for
end for
return knn
```

4.2.7 Putting all the pieces together

Cluster-and-Conquer is the sequential combination of all previous parts. We summarize the execution of overall approach:

— **Step** 1: the clustering. The dataset is clustered. T cluster configurations are tested, using FastMinHash functions. The t best are kept, along their $t \times b$ clusters.

— **Step** 2: the scheduling. Each of the p thread is assigned some of the $t \times p$ largest clusters, and many smaller ones.

— **Step** 3: the KNN graph computation. Each thread computes independently the KNN of the subdataset associated to the clusters it is in charge of.

— **Step** 4: the conquer step. The resulting partial KNN graphs are merged.

The resulting KNN graph is returned as the KNN graph of the whole dataset.

4.3 Experimental setup

4.3.1 Datasets

We use the same datasets as in the previous chapters: movielens1M, movielens10M, movielens20M, AmazonMovies, DBLP and Gowalla. For more details, see Section 2.3.2.

Except for the main results (Sec. 4.4.1), we will focus on movielens10M and AmazonMovies because they have a similar number of users ($69,816$ for ml10M, $57,430$ for AM) and ratings ($5,885,448$ for ml10M, $3,263,050$ for AM) but they differ by the size of their item set ($10,472$ for ml10M against $171,356$ for AM): movielens10M is dense while AmazonMovies is sparse. The differences or similarities between the results on these two datasets give an indication on how the sparsity impacts on the results of Cluster-and-Conquer.

4.3.2 Parameters

As with LP and GoldFinger, we compute KNN graphs with a neighborhood size k equal to 30. The parameters for Hyrec, NNDescent and LSH are the same as in the GoldFinger evaluation (Sec. 3.3.3). When using Cluster-and-Conquer, the number of clusters is set to 4096. The number of hashing functions used is 8, except of movielens10M and movielens20M for which there are 4 hashing functions. This difference is explained by the higher density and the large number of users of movielens10M and movielens20M: some of their clusters are extremely large so using more hash functions drastically increases the computation time. The impact of the number of hash function is discussed in Section 4.5.1. The number of hashing functions tried is twice the num-

ber of hashing functions really used: 16 for all the datasets except for movielens10M and movielens20M for which it is 8.

4.3.3 Evaluation metrics

We measure the effect of Cluster-and-Conquer along the same metrics as before: *(i)* their computation *time*, and *(ii)* the *quality* ratio of the resulting KNN (Sec. 2.3.3). When applying Cluster-and-Conquer to recommendation, we also measure the *recall* obtained by the recommender. Throughout our experiments, we use a 5-fold cross-validation procedure, and average our results over the 5 resulting runs.

4.3.4 Implementation details and hardware

The details of the implementation are in Appendix A. Our experiments run on a 64-bit Linux server with two Intel Xeon E5420@2.50GHz, totaling 8 hardware threads, 32GB of memory, and a HHD of 750GB. A graphic representation is available in Chap. 2 (Sec. 2.3 Fig. 2.4). Unless stated otherwise, we use all 8 threads.

4.4 Evaluation

We first discuss the raw performances of Cluster-and-Conquer, compared to the same algorithms already introduced in Chapter 1, the brute force approach, LSH, NNDescent and Hyrec, which we execute either in native mode (i.e. without GoldFinger) or with GoldFinger (Sec. 4.4.1). Then we show the impact of Cluster-and-Conquer on the memory locality (Sec. 4.4.2) and the scanrate (Sec. 4.4.3). Finally we evaluate the scalability of our new approach (Sec. 4.4.4) and the performances of the obtained KNN graphs while performing recommendation (Sec. 4.4.5).

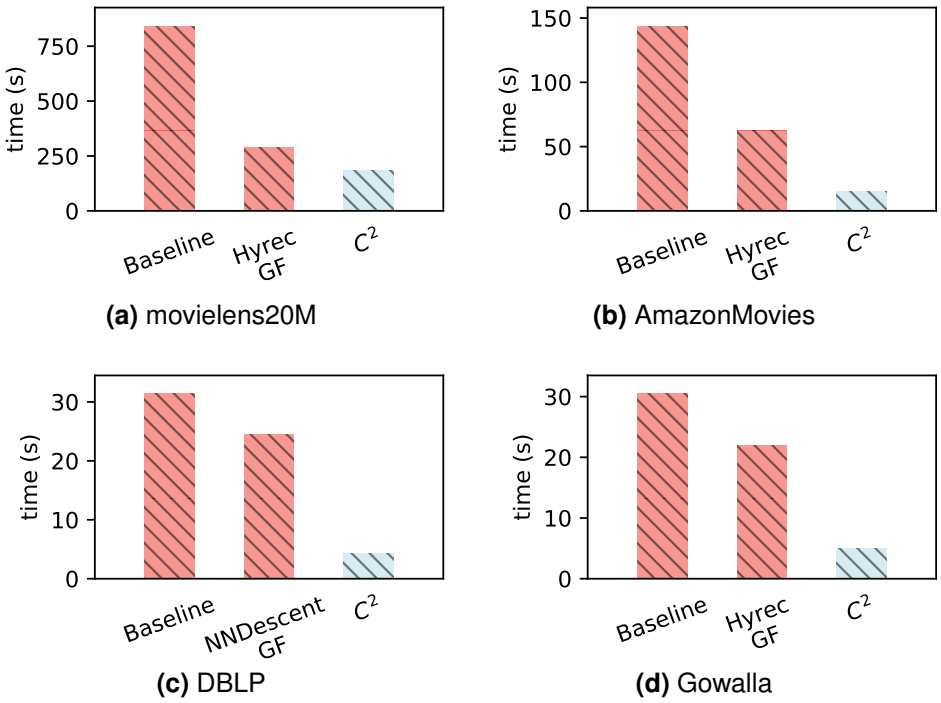
4.4.1 Computation time and KNN quality

The performances of Cluster-and-Conquer are summarized in Table 4.1 over the six datasets. A part of the results are shown graphically in Figures 4.4 and 4.5. In addition of those of Cluster-and-Conquer (noted C^2), the performances of three other approaches are also displayed for each dataset. The brute force approach and the best native approach for the dataset (labeled baseline), both without GoldFinger nor sampling, are represented. The last approach of each dataset is either Hyrec, NNDescent or LSH *with* GoldFinger: the displayed approach is the one yielding the best computation time on the dataset. The fastest computation time is shown in bold.

datasets	Algo	**Time**	**Speed-up**	**KNN quality**	Δ
ml1M	BruteForce	19.0	-	1.0	-
	Baseline[LSH]	9.54	-	0.98	-
	LSH/GF	2.96	$\times 3.23$	0.92	-0.05
	C^2	**2.33**	$\times 4.1$	0.9	-0.07
ml10M	BruteForce	2027.8	-	1.0	-
	Baseline[Hyrec]	314.12	-	0.96	-
	Hyrec/GF	109.98	$\times 2.86$	0.9	-0.06
	C^2	**63.23**	$\times 4.97$	0.9	-0.06
ml20M	BruteForce	8393.07	-	1.0	-
	Baseline[Hyrec]	841.72	-	0.95	-
	Hyrec/GF	289.23	$\times 2.91$	0.88	-0.07
	C^2	**185.5**	$\times 4.54$	0.89	-0.06
am	BruteForce	1861.98	-	1.0	-
	Baseline[LSH]	143.77	-	0.98	-
	Hyrec/GF	62.41	$\times 2.3$	0.93	-0.04
	C^2	**15.21**	$\times 9.45$	0.95	-0.03
DBLP	BruteForce	99.95	-	1.0	-
	Baseline[NNDescent]	31.36	-	0.98	-
	NNDescent/GF	24.43	$\times 1.28$	0.82	-0.16
	C^2	**4.36**	$\times 7.2$	0.8	-0.18
GW	BruteForce	160.17	-	1.0	-
	Baseline[LSH]	30.44	-	0.87	-
	Hyrec/GF	21.88	$\times 1.39$	0.78	-0.1
	C^2	**5.01**	$\times 6.08$	0.73	-0.14

Table 4.1 – Computation time and KNN quality of the brute force approach (without GoldFinger), the best approach without GoldFinger (labeled Baseline), the best approach relying on GoldFinger and our approach. Cluster-and-Conquer clearly outperforms the other approaches, yielding speed-ups up to $\times 9.45$ (DBLP) against the baseline.

Cluster-and-Conquer consistently achieves the best computation time on all the datasets. Cluster-and-Conquer clearly outperforms all the approaches, providing speed-up from $\times 4.1$ (ml1M) to $\times 9.45$ (AM) compared to the native baselines (i.e. without GoldFinger). The KNN quality provided by Cluster-and-Conquer is slightly lower than the one provided by the best approach using GoldFinger: it goes from a loss of 0.18 (on DBLP) to a loss of 0.03 (on AmazonMovies). When the loss is slightly higher, it is because of the choice of the parameters: the number of hash functions has a tremendous effect on both computation time and KNN quality. A more detailed study of the effect of the parameters is performed in Section 4.5.1.



(a) movielens20M **(b)** AmazonMovies

(c) DBLP **(d)** Gowalla

Figure 4.4 – Execution time of Cluster-and-Conquer and the best existing approach for each dataset (lower is better). Baseline refers to the fastest native approach (i.e. without GoldFinger), while Hyrec GF (resp. NNDescent GF) refers to the use of Hyrec (resp. NNDescent) using GoldFinger, the fastest of the two is chosen. Cluster-and-Conquer (C^2) outperforms the best existing approaches on the four datasets.

4.4.2 Memory and cache accesses

By clustering the users into small subdatasets, Cluster-and-Conquer decreases the data accesses and increase data locality. As with GoldFinger (see Sec 3.4.4), we use `perf` [1] to measure the impact of our algorithm on memory accesses. We studied the impact of Cluster-and-Conquer on multiple cache operations compared to Hyrec with GoldFinger, on movielens10M and AmazonMovies.

Table 4.2 summarizes the number of loads and store performed during the execution, Table 4.3 shows the corresponding number of misses and Table 4.4 represents their percentage.

Not only Cluster-and-Conquer decreases the total number of cache accesses but it also decreases the relative number of misses. These results validate our claim that Cluster-and-Conquer increases data locality.

1. https://perf.wiki.kernel.org/index.php/Main_Page

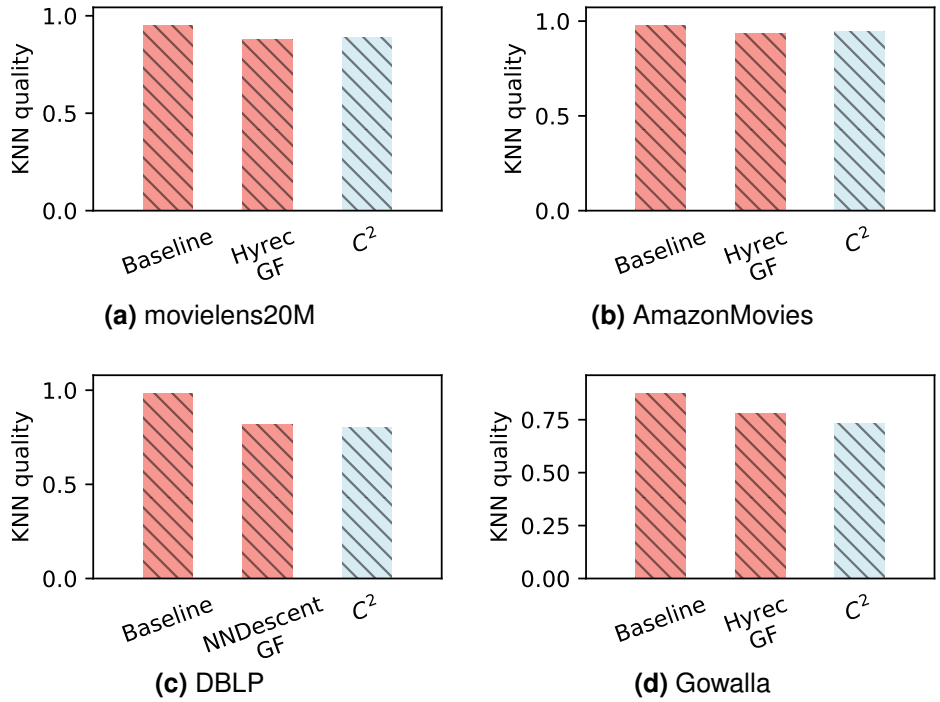


Figure 4.5 – KNN quality using Cluster-and-Conquer and the best existing approach for each dataset (higher is better). Baseline refers to the fastest native approach (i.e. without GoldFinger), while Hyrec GF (resp. NNDescent GF) refers to the use of Hyrec (resp. NNDescent) using GoldFinger, the fastest of the two is chosen. Cluster-and-Conquer (C^2) provides a similar KNN quality.

	movielens10M ($\times 10^9$)			*AmazonMovies* ($\times 10^9$)		
Operation	**Hyrec/GF**	$\mathbf{C^2}$	*gain*	**Hyrec/GF**	$\mathbf{C^2}$	*gain*
cache-references	10.28	7.58	-26.24%	6.10	2.58	-57.69%
L1-dcache-loads	274.40	222.46	-18.93%	182.05	91.16	-49.93%
L1-dcache-stores	84.41	69.07	-18.17%	55.46	30.41	-45.17%
LLC-loads	9.50	6.93	-27.01%	5.50	2.17	-60.50%
LLC-stores	0.34	0.31	-8.36%	0.27	0.2	-24.24%

Table 4.2 – Cache operations ($\times 10^9$) with Hyrec with GoldFinger (labeled Hyrec/GF) and Cluster-and-Conquer (C^2) on ml10M and AM. Cluster-and-Conquer reduces the number of cache accesses, yielding reductions (*gain*) up to 60.50%.

	movielens10M ($\times 10^9$)			*AmazonMovies* ($\times 10^9$)		
Type of misses	**Hyrec/GF**	$\mathbf{C^2}$	*gain*	**Hyrec/GF**	$\mathbf{C^2}$	*gain*
cache-references	3.36	1.85	-45.06%	1.91	0.40	-78.85%
L1-dcache-loads	14.91	10.90	-26.89%	8.65	3.34	-61.38%
L1-dcache-stores	3.90	3.72	-4.63%	2.85	3.12	$+9.62\%$
LLC-loads	3.17	1.71	-46.12%	1.77	0.33	-81.33%
LLC-stores	0.10	0.08	-15.8%	0.07	0.05	-26.20%

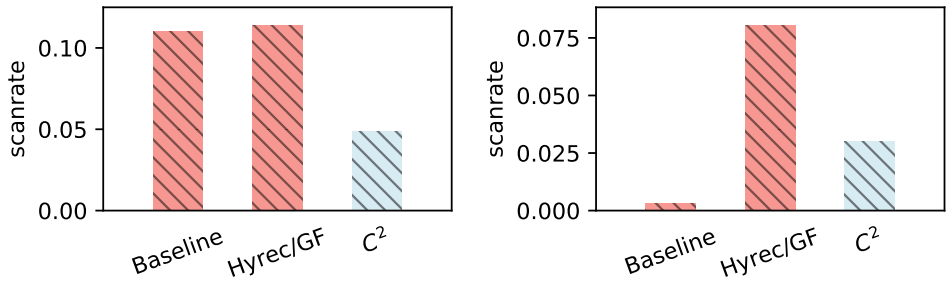
Table 4.3 – Cache operations misses ($\times 10^9$) with Hyrec with GoldFinger (labeled Hyrec/GF) and Cluster-and-Conquer (C^2) on ml10M and AM. Cluster-and-Conquer highly reduces the number of misses in the cache accesses, yielding reductions (*gain*) up to 81.33%.

4.4.3 Scanrate and distribution of computation time

Figure 4.6 shows the scanrate of the baseline, Hyrec and Cluster-and-Conquer approaches on movielens10M and AmazonMovies. As previously, the baseline approach is the native algorithm (among LSH, NNDescent or Hyrec) that is yielding the best computation time on the dataset *without* GoldFinger nor sampling. Also, Hyrec uses GoldFinger. For movielens10M, the scanrate increases when using Cluster-and-Conquer, which explains the similar computation time. For AmazonMovies, the scanrate highly decreases compared to Hyrec, which explains the huge decrease in computation time: from 62.41s to 15.21s. Still the scanrate is still higher than the baseline for AmazonMovies: LSH which has a more complex hashing scheme at the cost of an intensive precomputation.

	movielens10M			*AmazonMovies*		
Type of misses	**Hyrec/GF**	$\mathbf{C^2}$	*gain*	**Hyrec/GF**	$\mathbf{C^2}$	*gain*
cache-references	32.69%	24.35%	−8.34	31.30%	15.65%	−15.65
L1-dcache-loads	5.43%	4.90%	−0.53	4.75%	3.67%	−1.09
L1-dcache-stores	4.62%	5.39%	+0.76	5.14%	10.28%	+5.14
LLC-loads	33.40%	24.65%	−8.74	32.20%	15.22%	−16.98
LLC-stores	28.67%	26.35%	−2.33	27.56%	26.85%	−0.71

Table 4.4 – Percentage of missed cache operations with Hyrec with GoldFinger (labeled Hyrec/GF) and Cluster-and-Conquer (C^2) on ml10M and AM. Cluster-and-Conquer reduces ratio of hits and misses of cache accesses, yielding reductions (*gain*) up to 17.37%.



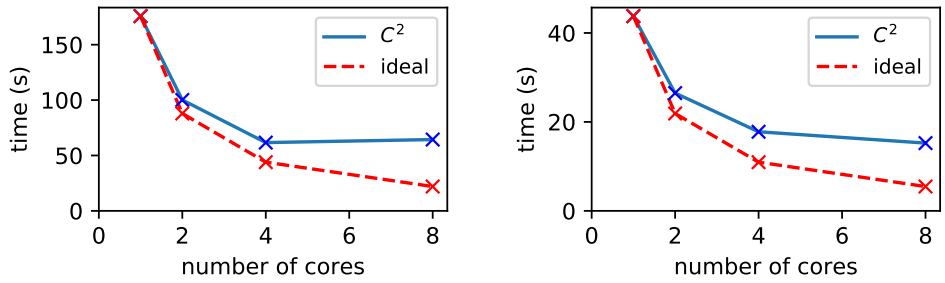
(a) movielens10M **(b)** AmazonMovies

Figure 4.6 – Scanrate of the baseline, Hyrec with GoldFinger and Cluster-and-Conquer on movielens10M and AmazonMovies. Baseline refers to the fastest native approach (i.e. without GoldFinger). Cluster-and-Conquer highly decreases the scanrate.

4.4.4 Scalability

Cluster-and-Conquer relies on a divide-and-conquer strategy. Intuitively, such local computations of small KNN graphs should scale. Figure 4.7 shows the computation time of Cluster-and-Conquer on movielens10M and AmazonMovies when increasing the number of cores from 1 to 8. The dotted line (labeled as *ideal*) represents the result we would obtain if the algorithms were to scale perfectly: its values are the value with 1 core divided by the number of cores. It seems that Cluster-and-Conquer does not scale linearly as there is not such a great improvement between 4 and 8 cores. We conjecture that this is due to data contention. Each thread has its own subdataset to work on. Individually they fit into the higher level of memory but not the whole, resulting in accesses to lower level of memory. The bigger the dataset, the bigger the data contention.

To verify our conjecture we measure multiple metrics while using Cluster-and-Conquer with different number of cores: the cache-references, the L1-dcache-stores, the LLC-stores and their corresponding misses metrics. Table 4.5 summarizes the results. In-



(a) movielens10M **(b)** AmazonMovies

Figure 4.7 – Effect of the number of cores on Cluster-and-Conquer for ml10M and AM. Cluster-and-Conquer seems to struggle to scale from 4 cores to 8.

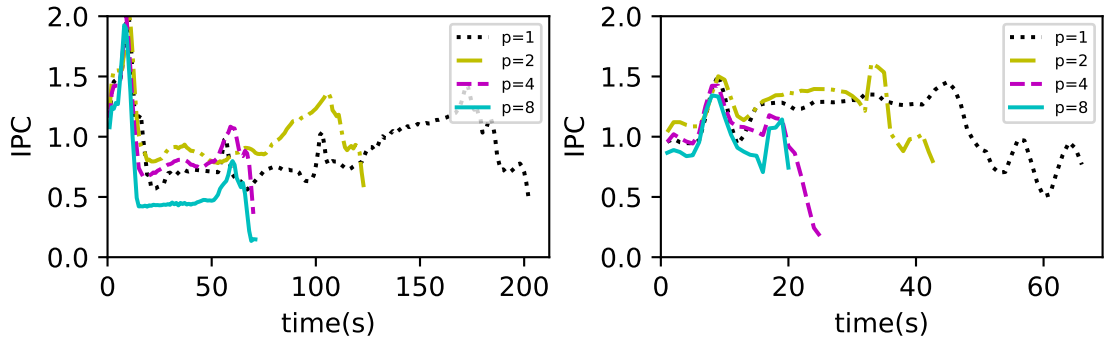
	movielens10M				*AmazonMovies*			
Operation	1	2	4	8	1	2	4	8
cache-references	–	-2.80%	$+2.97\%$	$+15.38\%$	–	-1.13%	-4.30%	$+9.76\%$
cache-misses	–	-12.66%	$+2.37\%$	$+64.60\%$	–	-9.02%	$+7.43\%$	$+37.72\%$
L1-dcache-stores	–	$+19.05\%$	$+25.41\%$	$+37.73\%$	–	$+17.63\%$	$+22.64\%$	$+33.97\%$
L1-dcache-store-misses	–	$+31.29\%$	$+58.31\%$	$+182.85\%$	–	$+31.3\%$	$+67.54\%$	$+209.71\%$
LLC-stores	–	$+16.03\%$	$+36.47\%$	$+61.26\%$	–	-4.34%	$+23.54\%$	$+48.77\%$
LLC-store-misses	–	$+9.83\%$	$+46.29\%$	$+171.68\%$	–	$+3.04\%$	$+43.11\%$	$+148.92\%$

Table 4.5 – Percentage of increase of cache operations while changing the number of cores in Cluster-and-Conquer on ml10M and AmazonMovies. When many cores are available, the data contention increases the number of cache operations and miss rates.

creasing the number of cores increases the cache operations, in particular it highly increases the misses, comforting our hypothesis of data contention. At the same time we monitor the number of instructions per cycle (IPC) of the machine, using *tiptop* [Roh12]. Each second *tiptop* measures the current IPC, and then we computed a sliding average: the value at time t is the average of the values of time $t-1$, t and $t+1$. The results are shown in Figure 4.8. The more cores, the lower the IPC. A lower IPC can be caused by processes waiting for data. Along with the increasing cache misses, it is highly probable that the poor scalability between 4 and 8 cores is due to data contention. Using a machine with more memory would solve the problem.

4.4.5 Cluster-and-Conquer and item recommendation

We study the practical impact of the small loss in KNN quality incurred by using Cluster-and-Conquer on the iconic item recommendation problem. We compare the recommendations made using the exact KNN graphs and the ones obtained with Cluster-and-Conquer. Figure 4.9 represents the results of the recall of the recommen-



(a) movielens10M **(b)** AmazonMovies

Figure 4.8 – Effect of the number of cores on the number of instructions per cycle (IPC) on the machine while executing Cluster-and-Conquer for ml10M and AM. The more cores the lower the IPC, suggesting an increasing data contention.

dation on movielens20M, AmazonMovies, DBLP and Gowalla. The results of the exact KNN graphs are labeled "BruteForce" while the ones obtained with Cluster-and-Conquer are labeled C^2. The loss in recall is negligible: we obtain a maximum loss of 6.18% on Gowalla. The use of Cluster-and-Conquer provides KNN graphs good enough to be used to perform item recommendation.

4.4.6 Impact of the different mechanisms of Cluster-and-Conquer: FastMinHash, the independent clusters and GoldFinger

Cluster-and-Conquer combines several key mechanisms. In this section we study the impact of the several mechanisms of Cluster-and-Conquer on its total execution. Figure 4.10 represents the execution of Cluster-and-Conquer and some of the utmost important mechanisms it uses. ① The users of the datasets are clustered using **FastMinHash** (labeled **FMH**). ② The clusters are **handled independently**, there are 8 of them in the example (C1 up to C8). ③ For each cluster, the associated KNN is computed locally, using **GoldFinger** (labeled **GolFi**). The resulting KNNs (KNN1 up to KNN8) are then merged together in the final KNN.

In the following, we study the influence of the use of FastMinHash functions, independent clusters and GoldFinger on Cluster-and-Conquer. The improvement provided by each mechanism is measured by replacing them, one at a time, by an alternative taken from the state of the art. FastMinHash functions are replaced by LSH functions, shared clusters are used instead of independent ones and raw profiles are replacing the SHFs of GoldFinger. Each mechanism is studied independently, Table 4.6 summarizes the results of all corresponding experiments. Cluster-and-Conquer is referred as **complete** when it uses all three mechanisms: FastMinHash, independent clusters and GoldFinger.

① **FastMinHash versus LSH:** to measure the influence of the FastMinHash func-

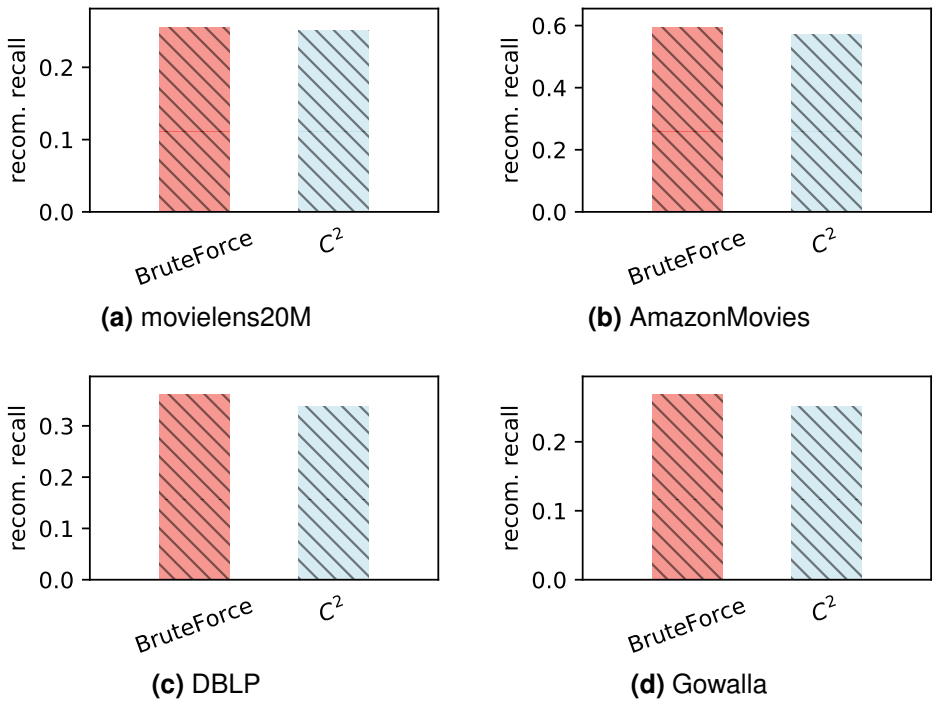


Figure 4.9 – Recommendation quality using Cluster-and-Conquer. We observe that the loss in recall is negligible.

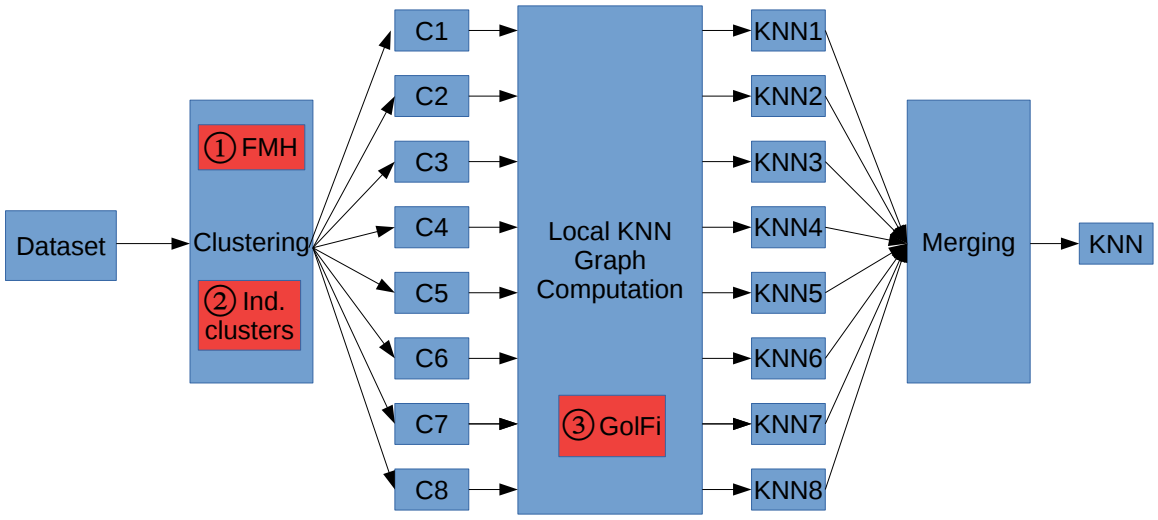


Figure 4.10 – During the execution of Cluster-and-Conquer, several mechanisms are used: FastMinHash during the clustering (labeled FMH), the choice of clustering in independent clusters (labeled Ind. clusters) and the use of GoldFinger (labeled GolFi).

datasets	Mechanisms	**KNN quality**	**time (s)**
ml10M	Complete	0.90	64
	① LSH hash	0.90	**54**
	② Merged clusters	0.90	126
	③ Raw data	**0.95**	312
AM	Complete	0.95	**15**
	① LSH hash	0.95	200
	② Merged clusters	0.95	25
	③ Raw data	**0.97**	36

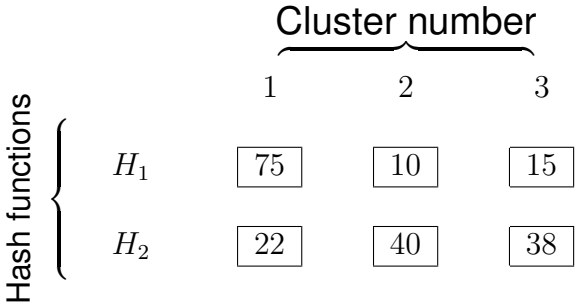
Table 4.6 – Impact of the FastMinHash, the independence of the clusters and the use of GoldFinger on the computation time and the KNN quality. The use of our fast hashing scheme instead of the LSH hash function slows down the computation on movielens10M ($+18.5\%$) but provides an important speed-up on AmazonMovies (-92.5%). The independent clusters highly decreases the computation time on movielens10M while slightly speeding-up the computation time for AmazonMovies. GoldFinger highly improves the computation time for both dataset while slightly decreasing the KNN quality.

tions, we compare Cluster-and-Conquer while using FastMinHash functions and by using the LSH hash functions (labeled as ***LSH Hash***). By design, the LSH hash functions create one cluster for every item in the item set. Two users are hashed in the same cluster with a probability proportional to their Jaccard similarity.

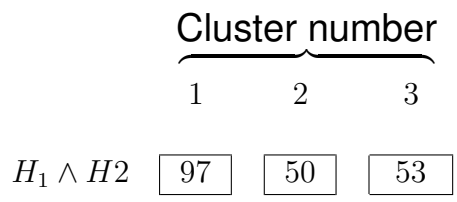
The use of our FastMinHash functions decreases the computation time for AmazonMovies by 92.5% while it increases it for movielens10M by 18.5%. This difference is explained by the size of the item set. The LSH hash functions are extremely costly when the item set is large as it is the case for AmazonMovies ($171,356$ items). In the case of movielens10M, the item set is small ($10,472$ items), the FastMinHash functions do not fully compensate the advantage of the LSH hash functions: the clusters are smaller and filled with users which have items in common. However, the computation times are very close: 64s for FastMinHash against 54s for LSH. Interestingly the KNN quality is roughly the same, meaning that the nice property of LSH hash functions is compensated by the collisions due to the fast hash functions.

② **Independent clusters versus shared clusters:** to study the impact of having independent clusters, we launched Cluster-and-Conquer where the clusters were shared by all the hash functions (labeled ***Shared clusters***): the first clusters of all the hash functions are merged together, then the second clusters and so on. This process is used by the standard LSH algorithm while clustering the users.

Figure 4.11 represents the clustering of 100 users obtained using 2 hash func-



(a) Independent clusters. Each clustering configuration results in 3 new clusters.



(b) Shared clusters. Each clustering configuration cluster the users into existing clusters. This is the scheme used by LSH.

Figure 4.11 – Example of two cluster configurations of 100 users with $b = 3$ when using shared and independent clusters. The shared clusters can be obtained by merging the independent clusters with the same cluster number.

tions (H_1 and H_2) and 3 clusters, with both independent and shared clusters. In each cluster (represented by a box) the number of its associated users is written. Figure 4.11a shows the clustering obtained with independent clusters. Each hash function results in $b = 3$ new clusters. Each user is in two different clusters, one for H_1 and one for H_2. Figure 4.11b represents the same clustering but using shared clusters. The same clusters are used for both hash functions H_1 and H_2. The users in the first clusters are the users of both the two clusters of cluster number equal to 1 in the independent clusters. For clarity reason the sum of users of each cluster in the clustering using shared clusters is the sum of the number of the associated independent clusters. In practice they would be redundancy, the users which have been hashed in the same cluster with two different hash function, so these numbers would be lower.

The independence of clusters is what makes possible the local computations of KNN graphs on subdatasets. Since the clusters are not handled independently anymore, the KNNs are not locally computed for each user in each cluster independently anymore. For each user, we select all the other users in the same clusters (possibly multiple) as her and compute the similarities and select the best. In other words, for each user we use a brute force approach on the subset of the users which are in the same clusters as her. We directly obtain a full KNN graph, we do not need any merging.

The use of independent clusters decreases the computation time by 49.2% for movielens10M and by 40% for AmazonMovies while providing the same KNN quality. The difference between movielens10M and AmazonMovies can be explained by their sparsity. The average size of a user's profile is 84.30 in movielens10M, while it is 56.82 for AmazonMovies. The higher the average size of profiles the more important the bias towards the first clusters. The first clusters are larger, and this difference is increased when merged. Thus the clusters

are more unbalanced for movielens10M than AmazonMovies, leading to longer computing time.

③ **GoldFinger versus raw data:** the influence of GoldFinger is studied by using Cluster-and-Conquer without GoldFinger (labeled as ***Raw data***).
The use of GoldFinger decreases the computation time by 79.5% for movielens10M and 58.3% for AmazonMovies. The loss in KNN quality is small, 0.05 for movielens10M and 0.02 for AmazonMovies. GoldFinger is a key contribution of Cluster-and-Conquer, independently of the dataset.

We observe that Cluster-and-Conquer is adapted to the performances of both sparse and dense datasets, movielens10M and AmazonMovies being representative of both categories (see Sec. 4.3.1). Using independent clusters avoid cluster from dense dataset to be too unbalanced, in addition of allowing local KNN graph computations. The FastMinHash functions avoid expensive computation when the dataset is too sparse. And GoldFinger speeds-up the similarity computations, especially for dense datasets.

4.5 Parameters sensitivity analysis

The performances of Cluster-and-Conquer depend on many parameters: the number of clusters, the number of hash functions and the number of tried hash functions and the characteristics of the dataset. In this section we study the influence of each of these parameters on the performances. Unless stated otherwise, the parameters are the same as in the previous sections: more specifically the number of clusters is 4096, the number of hash functions is 8 and the number of hash functions tried is 16, except for movielens10M and movielens20M for which the number of hash function is 4 and the number of hash function tried is 8. Except in Sec. 4.5.3 where the impact of the characteristics of the dataset are studied, we focus on movielens10M and AmazonMovies.

4.5.1 Number of clusters and number of hash functions

We evaluated the performances of Cluster-and-Conquer when the number of clusters belongs to $\{512, 1024, 2048, 4096, 8192\}$ and the number of hash functions belongs to $\{1, 2, 4, 8, 10\}$. In these experiments, the number of hash functions tried is twice the number of hash functions.

We represent the results in two different kind of graphs. The results are first displayed in a plot like the one displayed in Figure 4.12. The black dot represent the result of the baseline. Each curve represents the results for a constant number of clusters. The parameter here is the number of hash functions. The general tendency is that the more hash functions, the better the quality but the larger the computation time.

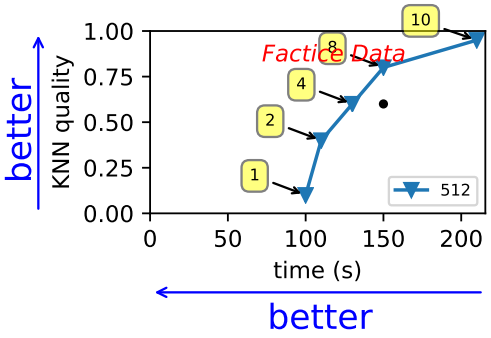
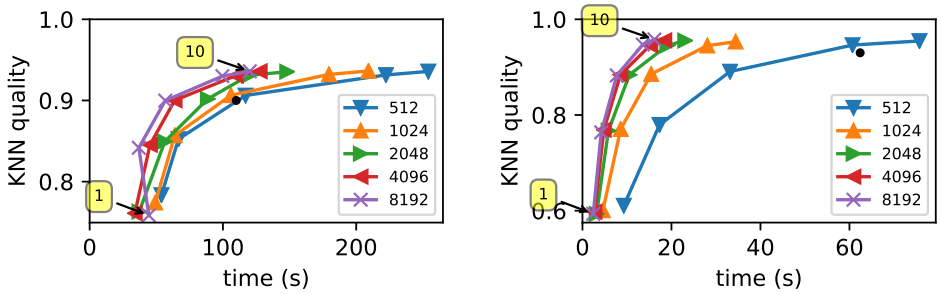


Figure 4.12 – Representation of the effect of the number of clusters on Cluster-and-Conquer. A curve represents the results for a given number of clusters, here 512. The points of the curve represent the results for the different number of hash functions, ranging from 1 to 10 in this example. The more hash functions, the better the quality, but the larger the computation time. The black dot represents the baseline, with GoldFinger.



(a) movielens10M **(b)** AmazonMovies

Figure 4.13 – Effect of the number of hash functions on Cluster-and-Conquer for ml10M and AM. The more hash functions the higher the KNN quality but the higher the computation time.

Then, the results are represented in a figure such as Figure 4.14. The same points are represented, but each curve represents the results for a constant number of hash functions. The parameter here is the number of clusters. Generally, the more clusters, the lower the computation time and the better the quality.

The results of our experiments are displayed in Figures 4.13 and 4.15. Figure 4.13 displays the impact of the number of hash functions, for a given number of clusters (similarly as Figure 4.12). Figure 4.15 displays the impact of the number of clusters, for a given number of hash functions (similarly as Figure 4.14). The black dot represents the value of Hyrec with GoldFinger, the best configuration on these datasets.

Figure 4.13 shows that the number of hash functions has an important impact on the performances both on the KNN quality and the computation time. At first, increasing the number of hash functions highly increases the KNN quality but the increase decreases, following a logarithmic growth. The differences of KNN quality between the values 8 and 10 are negligible, showing that increasing it further would not bring interesting

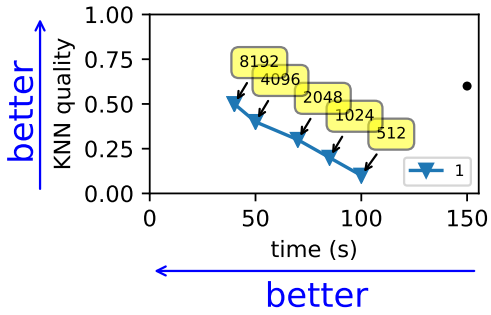


Figure 4.14 – Representation of the effect of the number of clusters on Cluster-and-Conquer. A curve represents results for a given number of hash function, here 1. The points of the curve represent the results for the different number of clusters, ranging from 512 to 8162 in this example. The more clusters, the better the quality and the lower the computation time. The black dot represents the baseline, with GoldFinger.

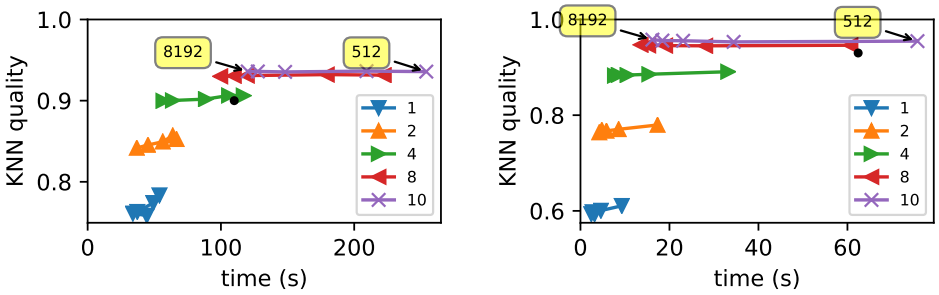
improvement. In term of computation time, the more hash functions the longer the computation. If the KNN quality seems to converge, the computation time keeps on increasing.

Figure 4.15 shows that, given a number of hash functions, increasing the number of clusters decreases the computation time. The decrease is important at first and lower and lower as the number of clusters increases. Increasing even more than 8192 seems useless since the differences between 4096 and 8192 is negligible in most of the cases. Interestingly enough, in movielens10M, the KNN quality slightly increases along the number of clusters while in AmazonMovies it seems to stagnate or even decrease. Increasing the number of clusters speeds-up the computation but also helps to put the users into small enough clusters. In such clusters, Hyrec is not used, but the brute force approach which is more precise. That is why the quality first increases. At the extreme, the clusters are so small that the users have no chance be with their neighbors. To illustrate that point we did the same experiments on ml1M. Figure 4.16 represents the results. Increasing the clusters still speeds-up the computation but at the cost of a loss in the quality.

The number of clusters and the number of hash functions are two key parameters of Cluster-and-Conquer. They represent a trade-off between computation time and KNN quality: the higher they are, the higher the quality and the computation time. With small values we can increase the speed-up.

4.5.2 The selection of the fittest

We study the impact of the number of tried hash functions on the computation time and KNN quality. Figures 4.17a and 4.17b represent the trade-off between the computation time and the KNN quality on movielens10M and AmazonMovies. For AmazonMovies (resp. movielens10M), the number of hash functions tried T varies



(a) movielens10M **(b)** AmazonMovies

Figure 4.15 – Effect of the number of clusters on Cluster-and-Conquer for ml10M and AM. Increasing the number of clusters decreases the computation time but it slightly decreases the KNN quality.

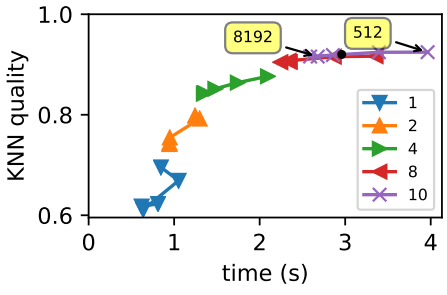
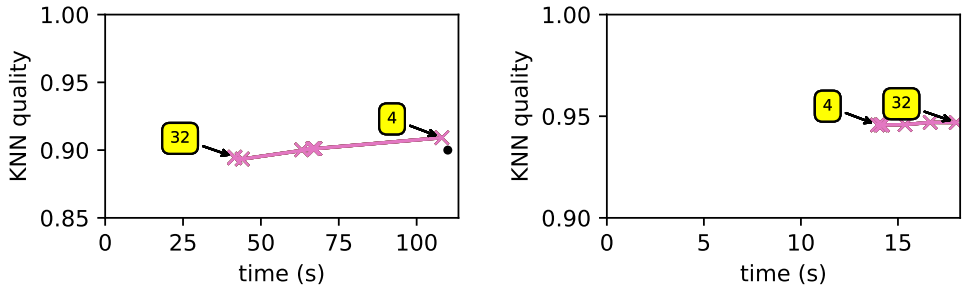


Figure 4.16 – Effect of the number of clusters on Cluster-and-Conquer for ml1M. Increasing the number of clusters decreases the KNN quality.



(a) movielens10M **(b)** AmazonMovies

Figure 4.17 – Effect of the number of hash functions tried on Cluster-and-Conquer for ml10M and AM. Increasing the number of hash functions tried decreases the computation time for ml10M while slightly increasing it for AmazonMovies.

in $\{8, 9, 10, 16, 24, 32\}$ (resp. $\{4, 8, 9, 10, 16, 24, 32\}$), while the number of hash functions kept is 8 (resp. 4). On movielens10M, increasing the T increases the KNN quality while globally decreasing the computation time. On AmazonMovies, the computation time is so low that the computation time needed to try the extra hash function is not negligible and is not compensated by the reduction of the cluster size. Also the KNN quality of AmazonMovies does not increase, because of its high initial value. In practice in such case we can deactivate this process and take only the hash functions needed.

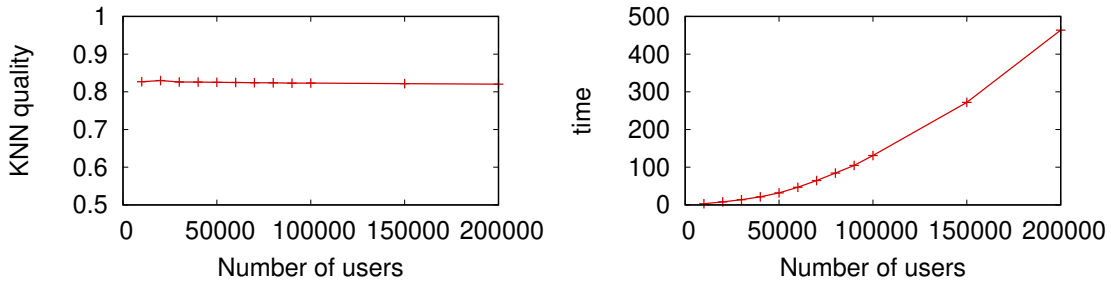
4.5.3 Impact of the dataset

To better understand the impact of the characteristics of the dataset on Cluster-and-Conquer, we study its performances on a series of synthetics datasets. In each dataset, only one characteristic is changed at a time. The parameters we are interested in are the number of users, the number of items, and the spread of ratings among items.

Experimental methodology

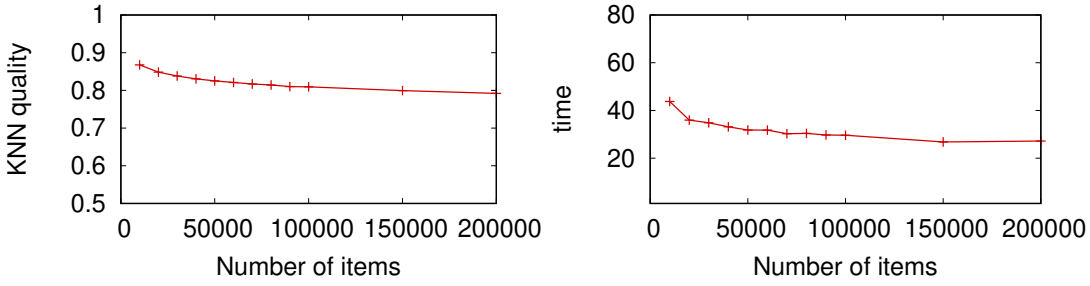
We generate synthetic datasets by fixing a number of users $|U|$ and a number of items $|I|$. We then generate ratings by drawing user-item pairs $(u, i) \in I \times U$ according a Zipfian distribution on both items and users. We use a Zipfian exponent of 1 for users, and s for items. To avoid the cold start problem, we further add 20 ratings to every users (still drawing items according to a Zipfian distribution). In total, we draw $r \times |U|$ ratings, with r a parameters representing the average profile's size.

By default, we set $|U| = 50,000$, $|I| = 50,000$, $r = 80$ and $s = 1$. Starting from this default configuration, we vary $|U|$, $|I|$ and s while keeping the other parameters at their default value. We vary $|U|$ and $|I|$ from $10,000$ to $200,000$, r from 20 to 500 and s from 0.5 (corresponding to a higher spreading of ratings among items) to 1.5 (corresponding to a higher concentration of ratings on popular items).



(a) Impact of $|U|$ on the KNN quality **(b)** Impact of $|U|$ on the computation time

Figure 4.18 – Cluster-and-Conquer maintains a high quality when increasing the number of users. (Synthetic datasets with default values $|U| = 50,000$, $|I| = 50,000$, and Zipfian distribution of users and items, with exponent 1)



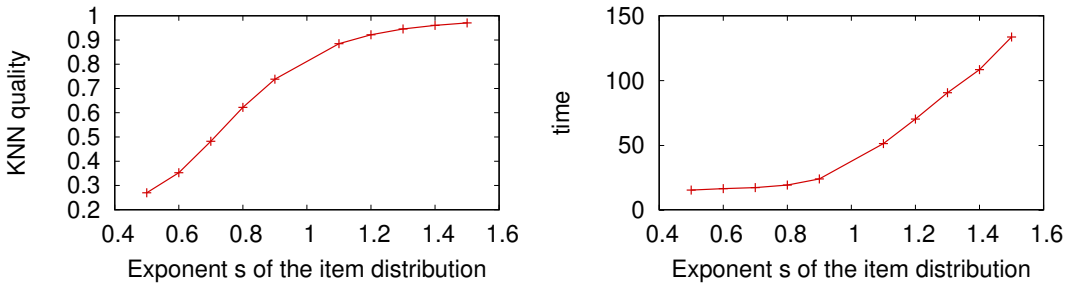
(a) Impact of $|I|$ on the KNN quality **(b)** Impact of $|I|$ on the computation time

Figure 4.19 – Cluster-and-Conquer maintains a high quality when increasing the number of users or items. (Synthetic datasets with default values $|U| = 50,000$, $|I| = 50,000$, and Zipfian distribution of users and items, with exponent 1)

We study the impact of the changes in the dataset on the KNN quality and computation of Cluster-and-Conquer with 4096 clusters, 8 hash functions and 16 hash functions tried.

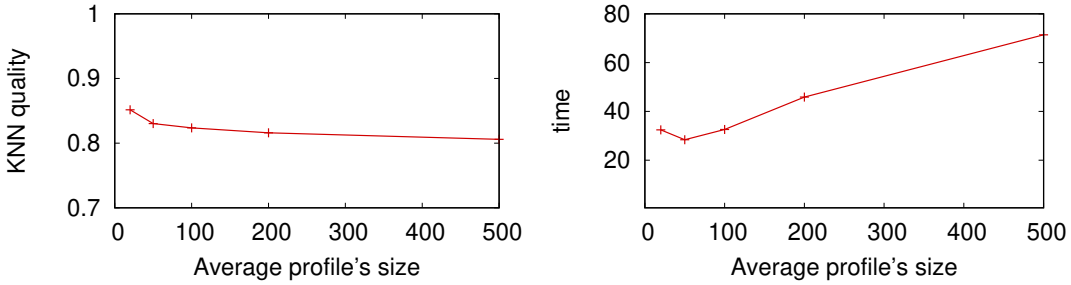
Impact of $|U|$ and $|I|$ on the computation. Figures 4.18 and 4.19 show the impact of $|U|$ and $|I|$ on the KNN quality and the computation time using Cluster-and-Conquer. The KNN quality seems to be independent, for a given configuration, of the number of users $|U|$. However, the larger I the lower the KNN quality. Still the global impact is low. In terms of computing time, increasing the size of U highly increases the computation time, but the increase is linear, not quadratic. On the other hand, increasing $|I|$ provokes a decrease in computation time. We believe this is due to a better distribution in the clusters. The parameters values (number of clusters and number of hash functions) should be adjusted when the number of users and the number of items change.

Impact of the distribution of the ratings on the computation Figure 4.20 shows the impact of the distribution of the ratings among the items on the KNN quality and the computation time. Figure 4.20a shows that the more unbalanced the item popularity is, the better for KNN quality. At the same time Figure 4.20b shows that the increase



(a) Impact of s on the KNN quality **(b)** Impact of s on the computation time

Figure 4.20 – Cluster-and-Conquer exploits the concentration of ratings among a few items found in many datasets, shown here when increasing the exponent s of the item distribution.



(a) Impact of r on the KNN quality **(b)** Impact of r on the computation time

Figure 4.21 – Cluster-and-Conquer exploits the concentration of ratings among a few items found in many datasets, shown here when increasing the exponent s of the item distribution.

in s increases the computation time. The less uniform the distribution, the more users will be hashed into the same clusters, increasing the KNN quality but at the same time it will increase the number of similarity computed, thus the computation time. If an unbalanced distribution was a good thing for GoldFinger, high values of s are slowing Cluster-and-Conquer.

Impact of r on the computation Figure 4.21 shows the impact of the average profile's size on the KNN quality and the computation time. Increasing the number of ratings decreases the KNN quality because neighbors are less likely to be hashed together. Also it has a negative impact on GoldFinger which may requires a bigger size to sustain such profiles.

4.6 Conclusions

We have proposed Cluster-and-Conquer, a novel algorithm to compute KNN graphs. Cluster-and-Conquer approximates the graph locality to increase the data locality when

computing the KNN graph. Cluster-and-Conquer relies on a divide-and-conquer approach that clusters users, computes locally the KNN graphs in each cluster and then merges them. The novelties of the approach are the hash functions used to cluster, the fact that the clusters are computed independently and the use of GoldFinger. FastMinHash is a fast-to-compute hash function introduced to hash user into the clusters. Computing the KNN graphs locally allows to use a standard KNN approach in big clusters and to keep the computation fast.

The local computation of small subdatasets increases the data locality and highly reduces the number of memory accesses.

We extensively evaluated Cluster-and-Conquer both on real and synthetic datasets. We studied the breakdown of impact of each mechanism and conducted a sensitivity analysis. Then to study some impact with respect to the number of users, the number of items and the ratings distribution that were not exhibited by real datasets we conducted experiments on synthetic data.

Cluster-and-Conquer speeds-up the KNN graph computation, in particular on spare dataset, while incurring a negligible loss in the KNN quality. We showed that the obtained KNN graphs can replace the exact one in item recommendation without loss.

CHAPTER 5

CONCLUSION AND PERSPECTIVES

Context

With the perpetual increase of available content, online services desperately need tools to help users to find their way. The most emblematic way to do so is personalization. More specifically, item recommendation is very popular on video platforms such as Netflix, YouTube, online media or even commerce web sites. K-Nearest-neighbors graphs are used to recommend items by aggregating the items in the profiles of the users with similar interests and are of the most used approach.

Unfortunately computing KNN graphs is costly since it requires to compute the similarity between each pair of users. For datasets with a large number of users, the number of similarities to compute becomes easily intractable. To overcome this cost, approximations are needed.

The existing approaches approximate the set of potential neighbors to small set of users. By doing so, they highly reduce the number of computed similarities. The resulting KNN graphs are approximations of the exact KNN graphs. In many applications such as news recommender systems, the volatility of the data makes the freshness of the model vital. An approximated KNN graph based on fresh data is a preferable option than having an exact KNN graph based on outdated data. Despite the high speed-up they provide, these approaches remain expensive and it seems difficult to push further in that direction.

Computing the similarities between the users represents up to 90% of the total computation time of the existing approaches. An orthogonal approach is then to decrease the cost of each similarity computation instead of their number. Compacting the data of each user to make it more tractable should allow interesting speed-ups. Unfortunately the existing compacted datastructures aim at optimizing the memory space, not the computation time.

Contributions

In this thesis, we pushed further the notion of approximation to speed-up the similarity computation and then design a new algorithm for KNN graph computation.

Approximating the profiles of users: the existing datastructures such as MinHash are expensive to compute because of the complex preprocessing. To increase the computation time, we aimed at a datastructures with the simplest preprocessing as possible. We introduced ***sampling***: by approximating the profiles to only a small subset of items, we limit the complexity of each similarity computation. We found that the best sampling policy is to keep only the ***least popular items*** of each profile. Despite its simplicity, it has never been done before and yields interesting results: by keeping the 25 least popular items of each profile we reduces the computation time by 63% on AmazonMovies while producing a good approximation of the KNN graph.

Approximating the similarity metric: sampling speeds-up the computation of the similarity but it cannot change the inherent cost of the similarity. To reduce the similarity complexity, we approximate the similarity computation itself. We use a compacted representation of the users' profiles, called SHF, and then approximate the costly set intersection of the similarity by a cheap bit-wise AND operation. Our approach **GoldFinger**, drastically reduces the computation time, providing speeds-up up to 78.9% compared to the use of raw data. The resulting KNN graphs suffers from a negligible to moderate loss in terms of KNN quality but show this loss does not impact item recommendation.

Approximating the graph locality: using GoldFinger shifts the bottleneck from the similarity computation to the data accesses. The lack of data locality slows down the computation: each thread has to not only load the data of the users it is in charge of, it also need to access the data of their neighbors which are treated by other threads. By clustering the similar users together, **Cluster-and-Conquer** increases the data locality by approximating graph locality and ensures that the KNNs are computed within the same clusters. Cluster-and-Conquer drastically speeds-up the KNN graph computation, providing speeds-up ratios up to 9.

Approximations are the solution to the so-called curse of dimensionality. Existing KNN graph algorithms are already relying on approximations: they approximate the set of candidates for each user's KNN by a small set. In this thesis we pushed further the notion of approximation by approximating the profiles, the similarity and the graph locality. The approach we designed clearly outperforms all the other ones, providing high speeds-up. The obtained KNN graphs are still very good approximations and can be used in application such as item recommendation.

Perspectives

Lowering the effect of popular items to have more balanced clusters: a short-term perspective is to have balance clusters. Cluster-and-Conquer limits the unbalance between clusters but it does not remove it completely. We intend to design a better cluster scheme to have smaller and more balanced clusters. Splitting the large clusters into smaller ones, using recursively our clustering scheme should work. The main issue is to prevent users of these large clusters to end-up in nearly empty ones with the splitting. From having too much candidates for their neighbors, they would end-up not having enough.

Aiming at stronger privacy properties: GoldFinger provides k*-anonymity* and ℓ*-diversity* for free. In term of privacy, these properties are weak. Increasing the privacy is another interesting short-term perspective. By relying on deterministic hash functions, we cannot have interesting properties such as Differential Privacy. Introducing random noise to the hashing scheme would be a way to have stronger privacy properties. By randomly flipping some bits, BLIP [AGK12] provides Differential Privacy to a Bloom-filter. We believe this work would be easily adapter to GoldFinger. It would be interesting to study the resulting trade-off between the privacy and the performances: the more noise the better the privacy but the worse the approximation of the Jaccard similarity by GoldFinger.

GoldFinger beyond the KNN graph computation: GoldFinger is *generic*, in the sense that it can be used in any algorithm relying on Jaccard similarity. Studying the effect of GoldFinger on other algorithms, such as k-means, is another short-term perspective. The resulting trade-off between time and quality would be of the utmost importance.

Distributing Cluster-and-Conquer: by design Cluster-and-Conquer is following a map-reduce scheme, it is thus highly compatible to a distributed environment. An interesting mid-term perspective is to distribute Cluster-and-Conquer in several machines. It is very likely that the new bottleneck would be the communications. A classical thread-pool may not be the most suited in such a configuration: a scheduling taking into account the distribution of the data would be required.

Pushing, once again, the frontiers of approximations: as a more long-term perspective, it would be interesting to see how far we can push the notion of approximations in KNN graph computation.

— Can we have more balanced clusters by approximating even more the graph

locality?

— Can we have stronger privacy properties by approximating even more the users' profiles?
— Can we have a communication-efficient distributed Cluster-and-Conquer by approximating the distribution of the data on the machines?

More generally, to what extent can we push the notion of approximations to solve the successive bottlenecks we face while computing a KNN graph? It would be interesting to study what kind of problems can be solved by relying on approximations? It seems necessary for big data problems where the data is too large to be treated thoroughly.

APPENDIX A

IMPLEMENTATION

All the algorithms and techniques used in this thesis have been implemented in Java 1.8. Altogether, the implementation represents more than 5000 lines of code, and 200 java classes. The code is available here:

https://gitlab.inria.fr/oruas/SamplingKNN

Every algorithm (Brute force, Hyrec, NNDescent, LSH and Cluster-and-Conquer) and every datastructure (the regular one, LP, MP, CS, IS and GoldFinger) have been implemented in a generic way. Each algorithm relies on an instance of the interface *dataset*, so launching Hyrec with two different datastructures is easy:

```
Dataset dataset = new DatasetLeastPop("movielens10M", 25);
Hyrec algo = new Hyrec(dataset);
algo.doKNN();
```

Figure A.1 – Code launching Hyrec with LP, with a sampling size $s = 25$.

```
Dataset dataset = new DatasetGoldFinger("movielens10M", 1024);
Hyrec algo = new Hyrec(dataset);
algo.doKNN();
```

Figure A.2 – Code launching Hyrec with GoldFinger, with a size $b = 1024$.

Figures A.1 and A.2 shows the constructions of a KNN graph on movielens10M, with LP and GoldFinger respectively. Only the dataset definition changes. For clarity reason, the code is simplified.

In the following the implementations of two core components are presented, the KNN graph and GoldFinger.

KNN graph

The KNN graph is implemented as a hash map containing for each user a KNN. Each user is thus associated to its KNN. Every KNN contains k elements: each element

represents a neighbor and its score, i.e. its similarity with the user associated to the KNN. The KNN is a ***binary min heap***: the root represents the user with the lowest similarities among the neighbors, and the two successors of any node have a higher similarity than this node. It is implemented as an array in which the first element is the root, and for an element at the position i, its two successors are at the position $2i + 1$ and $2i + 2$. The KNN has a ***constant size***: the size is k and never change. In addition of the heap, a hash map is maintained to keep track of the currents users of the KNN.
Addition: The main operation of the KNN is the addition of a pair (user,similarity). To add a pair (v, s) to the KNN of u we first check if the user is not already in the hash set (and thus in the KNN). If not, we compare the similarity s to the lowest of the similarity s_m of the neighbors in the KNN. If $s < s_m$ then all the neighbors of the current KNN have a higher similarity with u so the KNN is not changed. If $s \geq s_m$ then u is added to the KNN and the user associated to s_m is ejected from the KNN and from the hash map. The interest of having a min heap is that the element associated to s_m is the root and thus stored at the first position in the array. Accessing the first element to know if v can be added is fast. The operation returns true if v has been added, false otherwise.

GoldFinger

In GoldFinger, each is associated to SHF, the fingerprint of its profile. The SHF is implemented as an array of Longs. Both the cardinality and the bit array are implemented in the same array. The first Long is used to store the cardinality, the others the bit array β. Each Long represents a 64-bits subpart of the total array. A SHF of 1024 bits is thus represented by an array of $1024/64 + 1 = 17$ Longs. The intersection of two SHFs is computed by a bit-wise AND operation between the arrays, let aside the first element.

BIBLIOGRAPHY

[ABPH07] Paulo Sérgio Almeida, Carlos Baquero, Nuno M Preguiça, and David Hutchison. Scalable bloom filters. *Information Processing Letters*, 101(6):255–261, 2007.

[AC09] Nir Ailon and Bernard Chazelle. The fast johnson–lindenstrauss transform and approximate nearest neighbors. *SIAM Journal on computing*, 39(1):302–322, 2009.

[Ach01] Dimitris Achlioptas. Database-friendly random projections. In *Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 274–281. ACM, 2001.

[Ach03] Dimitris Achlioptas. Database-friendly random projections: Johnson-lindenstrauss with binary coins. *Journal of computer and System Sciences*, 66(4):671–687, 2003.

[AGK12] Mohammad Alaggan, Sébastien Gambs, and Anne-Marie Kermarrec. Blip: Non-interactive differentially-private similarity computation on bloom filters. In *SSS*, pages 202–216. Springer, 2012.

[ama] Amazon amazon. `https://www.amazon.com/`. Accessed: 2018-08-01.

[AT05] Gediminas Adomavicius and Alexander Tuzhilin. Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *Knowledge and Data Engineering, IEEE Transactions on*, 17(6):734–749, 2005.

[BCFM00] Andrei Z Broder, Moses Charikar, Alan M Frieze, and Michael Mitzenmacher. Min-wise independent permutations. *Journal of Computer and System Sciences*, 60(3):630–659, 2000.

[Ben75] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[BFG+14] Antoine Boutet, Davide Frey, Rachid Guerraoui, Anne-Marie Kermarrec, and Rhicheek Patra. Hyrec: leveraging browsers for scalable recommenders. In *Middleware*, 2014.

[BKL06] Alina Beygelzimer, Sham Kakade, and John Langford. Cover trees for nearest neighbor. In *ICML*, 2006.

[BKMT16] Antoine Boutet, Anne-Marie Kermarrec, Nupur Mittal, and François Taïani. Being prepared in a sparse world: the case of knn graph construction. In *ICDE*, 2016.

[BL+07] James Bennett, Stan Lanning, et al. The netflix prize. In *Proceedings of KDD cup and workshop*, volume 2007, page 35. New York, NY, USA, 2007.

[Blo70] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[BM04] Andrei Broder and Michael Mitzenmacher. Network applications of bloom filters: A survey. *Internet mathematics*, 1(4):485–509, 2004.

[Bro97] Andrei Z Broder. On the resemblance and containment of documents. In *Compression and Complexity of Sequences 1997.*, 1997.

[Bur02] Robin Burke. Hybrid recommender systems: Survey and experiments. *User modeling and user-adapted interaction*, 12(4):331–370, 2002.

[CCFC02] Moses Charikar, Kevin Chen, and Martin Farach-Colton. Finding frequent items in data streams. *Automata, languages and programming*, pages 784–784, 2002.

[CFS09] Jie Chen, Haw-ren Fang, and Yousef Saad. Fast approximate knn graph construction for high dimensional data via recursive lanczos bisection. *Journal of Machine Learning Research*, 10(Sep):1989–2012, 2009.

[CKO16] Nitin Chiluka, Anne-Marie Kermarrec, and Javier Olivares. The out-of-core knn awakens. In *International Conference on Networked Systems*, pages 295–310. Springer, 2016.

[CKRT04] Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, and Ayellet Tal. The bloomier filter: an efficient data structure for static support lookup tables. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 30–39. Society for Industrial and Applied Mathematics, 2004.

[CM05] Graham Cormode and S Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[CML11] Eunjoon Cho, Seth A. Myers, and Jure Leskovec. Friendship and mobility: User movement in location-based social networks. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '11, pages 1082–1090, New York, NY, USA, 2011. ACM.

[DG03] Sanjoy Dasgupta and Anupam Gupta. An elementary proof of a theorem of johnson and lindenstrauss. *Random Structures & Algorithms*, 22(1):60–65, 2003.

[DIIM04] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*, pages 253–262. ACM, 2004.

[DML11] Wei Dong, Charikar Moses, and Kai Li. Efficient k-nearest neighbor graph construction for generic similarity measures. In *WWW*, 2011.

[Dwo08] Cynthia Dwork. Differential privacy: A survey of results. In *International Conference on Theory and Applications of Models of Computation*, pages 1–19. Springer, 2008.

[FBF77] Jerome H Friedman, Jon Louis Bentley, and Raphael Ari Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software (TOMS)*, 3(3):209–226, 1977.

[FCAB00] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking (TON)*, 8(3):281–293, 2000.

[FFGM07] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. In *Discrete Mathematics and Theoretical Computer Science*, pages 137–156. Discrete Mathematics and Theoretical Computer Science, 2007.

[GIM+99] Aristides Gionis, Piotr Indyk, Rajeev Motwani, et al. Similarity search in high dimensions via hashing. In *VLDB*, 1999.

[GRS99] Sudipto Guha, Rajeev Rastogi, and Kyuseok Shim. Rock: A robust clustering algorithm for categorical attributes. In *Data Engineering, 1999. Proceedings., 15th International Conference on*, pages 512–521. IEEE, 1999.

[GSA+11] MR Gorai, KS Sridharan, T Aditya, R Mukkamala, and S Nukavarapu. Employing bloom filters for privacy preserving distributed collaborative knn classification. In *WICT*, 2011.

[HK15] F. Maxwell Harper and Joseph A. Konstan. The movielens datasets: History and context. *ACM Trans. Interact. Intell. Syst.*, 5(4):19:1–19:19, December 2015.

[Hua08] Anna Huang. Similarity measures for text document clustering. In *Proceedings of the sixth new zealand computer science research student conference (NZCSRSC2008), Christchurch, New Zealand*, pages 49–56, 2008.

[IM98] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *STOC*, 1998.

[Jac01] Paul Jaccard. Étude comparative de la distribution florale dans une portion des alpes et des jura. *Bull Soc Vaudoise Sci Nat*, 37:547–579, 1901.

[JDS11] Herve Jegou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence*, 33(1):117–128, 2011.

[Jen97] Bob Jenkins. Hash functions. *Dr Dobbs Journal*, 22(9):107–+, 1997.

[KBG12] Aapo Kyrola, Guy E Blelloch, and Carlos Guestrin. Graphchi: Large-scale graph computation on just a pc. In *OSDI*, volume 12, pages 31–46, 2012.

[KBV09] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, (8):30–37, 2009.

[Lan50] Cornelius Lanczos. *An iteration method for the solution of the eigenvalue problem of linear differential and integral operators*. United States Governm. Press Office Los Angeles, CA, 1950.

[LeC98] Yann LeCun. The mnist database of handwritten digits. *http://yann. lecun. com/exdb/mnist/*, 1998.

[LK11] Ping Li and Arnd Christian König. Theory and applications of b-bit minwise hashing. *Communications of the ACM*, 2011.

[LMYG04] Ting Liu, Andrew W Moore, Ke Yang, and Alexander G Gray. An investigation of practical approximate nearest neighbor algorithms. In *NIPS*, 2004.

[Low04] David G Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91–110, 2004.

[LSY03] Greg Linden, Brent Smith, and Jeremy York. Amazon. com recommendations: Item-to-item collaborative filtering. *Internet Computing, IEEE*, 2003.

[LVLD08] Xuan Nhat Lam, Thuc Vu, Trong Duc Le, and Anh Duc Duong. Addressing cold-start problem in recommendation systems. In *Proceedings of the 2nd international conference on Ubiquitous information management and communication*, pages 208–211. ACM, 2008.

[LWS15] Ziqi Liu, Yu-Xiang Wang, and Alexander J Smola. Fast differentially private matrix factorization. *arXiv preprint arXiv:1505.01419*, 2015.

[M+67] James MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297. Oakland, CA, USA, 1967.

[MGKV06] A. Machanavajjhala, J. Gehrke, D. Kifer, and M. Venkitasubramaniam. L-diversity: privacy beyond k-anonymity. In *22nd International Conference on Data Engineering (ICDE'06)*, 2006.

[ML09] Marius Muja and David G Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. *VISAPP (1)*, 2(331-340):2, 2009.

[ML13a] Julian McAuley and Jure Leskovec. Hidden factors and hidden topics: understanding rating dimensions with review text. In *Proceedings of the 7th ACM conference on Recommender systems*, pages 165–172. ACM, 2013.

[ML13b] Julian John McAuley and Jure Leskovec. From amateurs to connoisseurs: modeling the evolution of user expertise through online reviews. In *WWW*, 2013.

[Moo00] Andrew W Moore. The anchors hierarchy: Using the triangle inequality to survive high dimensional data. In *UAI*, 2000.

[MS07] Andriy Mnih and Ruslan Salakhutdinov. Probabilistic matrix factorization. In *Advances in neural information processing systems*, pages 1257–1264, 2007.

[net] Netflix Technology Blog netflix recommendations: Beyond the 5 stars. `https://medium.com/netflix-techblog/netflix-recommendations-beyond-the-5-stars-part-1-55838468f429`. Accessed: 2018-07-11.

[NST+14] N. Nodarakis, S. Sioutas, D. Tsoumakos, G. Tzimas, and E. Pitoura. Rapid aknn query processing for fast classification of multidimensional data in the cloud. *CoRR*, 2014.

[omna] Omnicore omnicore agency: Facebook by the numbers: Stats, demographics & fun facts. `https://www.omnicoreagency.com/facebook-statistics/`. Accessed: 2018-07-11.

[omnb] Omnicore omnicore agency: Youtube by the numbers: Stats, demographics & fun facts. `https://www.omnicoreagency.com/youtube-statistics/`. Accessed: 2018-07-11.

[OT01] Aude Oliva and Antonio Torralba. Modeling the shape of the scene: A holistic representation of the spatial envelope. *International journal of computer vision*, 42(3):145–175, 2001.

[RIS+94] Paul Resnick, Neophytos Iacovou, Mitesh Suchak, Peter Bergstrom, and John Riedl. Grouplens: an open architecture for collaborative filtering of netnews. In *Proceedings of the 1994 ACM conference on Computer supported cooperative work*, pages 175–186. ACM, 1994.

[RMZ13] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 472–488. ACM, 2013.

[Roh12] Erven Rohou. Tiptop: Hardware performance counters for the masses. In *41st International Conference on Parallel Processing Workshops (ICPPW)*, pages 404–413. IEEE, 2012.

[RRWN11] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in Neural Information Processing Systems*, pages 693–701, 2011.

[SAH08] Chanop Silpa-Anan and Richard Hartley. Optimised kd-trees for fast image descriptor matching. In *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*, pages 1–8. IEEE, 2008.

[Sam01] Pierangela Samarati. Protecting respondents identities in microdata release. *IEEE transactions on Knowledge and Data Engineering*, 13(6):1010–1027, 2001.

[SB07] S Joshua Swamidass and Pierre Baldi. Mathematical correction for fingerprint similarity measures to improve chemical retrieval. *Journal of chemical information and modeling*, 47(3):952–964, 2007.

[SBS+13] Sebastian Schelter, Christoph Boden, Martin Schenck, Alexander Alexandrov, and Volker Markl. Distributed matrix factorization with mapreduce using a series of broadcast-joins. In *Proceedings of the 7th ACM Conference on Recommender Systems*, pages 281–284. ACM, 2013.

[SKKR01] Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl. Item-based collaborative filtering recommendation algorithms. In *Proceedings of the 10th international conference on World Wide Web*, pages 285–295. ACM, 2001.

[SLH14] Yue Shi, Martha Larson, and Alan Hanjalic. Collaborative filtering beyond the user-item matrix: A survey of the state of the art and future challenges. *ACM Computing Surveys (CSUR)*, 47(1):3, 2014.

[sta] statistica the statistics portal: Media usage in an internet minute as of june 2018. `https://www.statista.com/statistics/195140/new-user-generated-content-uploaded-by-users-per-minute/`. Accessed: 2018-07-11.

[SVZ08] Börkur Sigurbjörnsson and Roelof Van Zwol. Flickr tag recommendation based on collective knowledge. In *Proceedings of the 17th international conference on World Wide Web*, pages 327–336. ACM, 2008.

[SWR+09] Patrick D Schloss, Sarah L Westcott, Thomas Ryabin, Justine R Hall, Martin Hartmann, Emily B Hollister, Ryan A Lesniewski, Brian B Oakley, Donovan H Parks, Courtney J Robinson, et al. Introducing mothur: open-source, platform-independent, community-supported software for describing and

comparing microbial communities. *Applied and environmental microbiology*, 75(23):7537–7541, 2009.

[YL12] Jaewon Yang and Jure Leskovec. Defining and evaluating network communities based on ground-truth. *CoRR*, abs/1205.6233, 2012.

[ZCJL13] Yong Zhuang, Wei-Sheng Chin, Yu-Chin Juan, and Chih-Jen Lin. A fast parallel sgd for matrix factorization in shared memory systems. In *Proceedings of the 7th ACM conference on Recommender systems*, pages 249–256. ACM, 2013.

LIST OF FIGURES

LIST OF TABLES

RESUMÉ EN FRANÇAIS

Le monde du Big Data

Nous sommes confrontés aujourd'hui au phénomène bien connu du "Big Data". Non seulement 300 heures de vidéos sont téléchargées chaque minute sur YouTube [omnb], mais aussi 243 000 photos sont téléchargées sur Facebook [omna] et 473 000 tweets sont générés sur Twitter [sta]. Une grande partie de ces données sont générées par les utilisateurs eux-mêmes. Les utilisateurs peuvent générer des données explicites telles que des vidéos, des images ou des messages, mais ils produisent également des données implicites en achetant un produit ou en interagissant (par exemple, aimer, partager, évaluer) avec le contenu. Ces données (implicites et explicites) sont le moteur des techniques modernes d'apprentissage automatique. Dans le même temps, un si grand volume de données a commencé à devenir un problème pour les utilisateurs eux-mêmes. Trouver le contenu intéressant parmi l'ensemble des données est incroyablement difficile. Une brève description du contenu est clairement insuffisant car parcourir l'ensemble du contenu est impossible compte tenu de sa taille. Par conséquent, les utilisateurs ont besoin d'aide pour trouver un contenu intéressant.

Comment s'en sortir ?

Ce problème est particulièrement visible dans les services en ligne. Les services en ligne sont devenus la manière habituelle de répondre à de nombreux besoins tels que regarder des films, écouter de la musique ou en lire les nouvelles. Ces services doivent faire face à ce nombre croissant de données disponibles qui empêchent les utilisateurs de trouver le contenu qui les intéresse. Dans ce contexte, la personnalisation est la clé.

La personnalisation comme moyen de trouver une aiguille dans la botte de foin pour l'utilisateur. La personnalisation est le processus d'adaptation d'un service à chaque utilisateur, en fonction des données qui lui sont associées. Par exemple, le contenu d'une newsletter peut être choisi en fonction des goûts d'un utilisateur : un fan de sport recevra des nouvelles sportives, et les nouvelles locales peuvent être ajoutées en fonction de l'emplacement de l'utilisateur. L'approche la plus utilisée pour personnaliser est de recommander du contenu aux utilisateurs, que ce soit de la vidéo, de la musique ou des nouvelles. L'exemple le plus connu est probablement Netflix, qui recommande des films à un utilisateur en fonction de son historique.

Les personnes similaires aiment des contenus similaires. De nombreux algorithmes ont été développés pour faire de la recommandation, notamment dans le filtrage collaboratif. Les techniques de filtrage collaboratif recommandent à un utilisateur les éléments, que ce soit des films, des articles, des biens de consommation, etc ..., aimé par les utilisateurs qui lui sont semblables. L'intuition est que si Alice avait un comportement similaire à Bob, alors Alice est susceptible de se comporter de la même manière que Bob dans l'avenir. Par exemple, nous supposons que si Alice et Bob ont aimé les mêmes films, il est probable qu'Alice et Bob aimeront les mêmes films à l'avenir. Donc, tout nouveau film aimé par Bob peut être recommandé à Alice.

K-Nearest-Neighbors graphs. Une des principales classes de techniques de filtrage collaboratif s'appuie sur les graphes des K-plus-proches-voisins (K-Nearest-Neighbors, KNN) [DML11, BFG+14]. Les approches basées sur les graphes KNN construisent un graphe reliant chaque noeud (représentant soit un élément ou un utilisateur) à ses homologues les plus proches selon une métrique de similarité donnée. Les défis de la construction de ces graphes résident dans la recherche du voisinage le plus proche sans faire une recherche exhaustive dans l'ensemble des données. Accélérer la construction du graphe KNN est au centre de cette thèse.

La construction efficace des graphes KNN

Graphes des K-plus-proches-voisins

Un graphe KNN est un graphe dans lequel chaque noeud est connecté à ses k homologues les plus proches, appelés voisins. En d'autres termes, chaque utilisateur est lié à l'ensemble de k utilisateurs qui lui ressemblent le plus. Une métrique de similarité est utilisée pour évaluer la proximité, c'est-à-dire à quel point deux utilisateurs sont similaires. La métrique de similarité semble être de la plus haute importance, car elle façonnera le graphe final du KNN. Les métriques de similarité utilisées dans la pratique sont basées sur un autre ensemble, l'ensemble des caractéristiques. Chaque caractéristique peut être vue comme une coordonnée, un utilisateur est alors représenté comme un point dans un espace dont la dimension est le nombre de caractéristiques. Le nombre de caractéristiques représente la dimension du problème.

Défis

La manière naïve de calculer un graphe KNN est très simple et consiste, pour chaque utilisateur, à calculer sa similarité avec tous les autres utilisateurs du jeu de données. Chaque utilisateur a alors une liste d'autres utilisateurs, classés en fonction

de leur similarité afin de garder les k utilisateurs avec la plus haute similarité. Cette approche naïve est optimale mais inefficace. Cette stratégie calcule une similarité pour chaque paire d'utilisateurs. Le nombre de comparaisons est le nombre de comparaisons effectuées pour calculer des graphes KNN exacts utilisant l'approche naïve : $\frac{n\times(n-1)}{2}$ où n est le nombre d'utilisateurs. Cette complexité rend l'approche naïve impossible pour les jeux de données avec un grand nombre d'utilisateurs.

En utilisant des structures de données spécifiques pour organiser le jeu de données, le calcul du graphe KNN peut être effectué efficacement en faibles dimensions [BKL06, LMYG04, Moo00]. Faible dimension fait référence à la dimension du problème : la dimension de l'espace des caractéristiques.

En haute dimension, ces solutions ont la même complexité que l'approche naïve : **le calcul efficace d'un graphe KNN en grande dimension reste un problème ouvert**. Les approches existantes tentent de surmonter la complexité quadratique en ne calculant pas toutes les similarités [IM98, BFG+14, DML11, BKMT16]. Cela accélère le calcul mais au risque de manquer certains voisins. Le graphe obtenu n'est pas exact mais est une approximation.

Bien que les approches existantes pour calculer les graphes KNN fournissent des approximations, elles sont toujours coûteuses. Même si le calcul est effectué hors ligne et les graphes KNN ne sont utilisés une fois calculés, le coût reste prohibitif pour les jeux de données volumineux. La seule façon de les faire travailler sur de grands ensembles de données consiste à adapter l'infrastructure avec le jeux de données. Pourtant, même pour les entreprises qui ne sont pas propriétaires de leur infrastructure mais qui utilisent le cloud, une telle mise à l'échelle peut être extrêmement coûteuse. Trouver des moyens plus efficaces pour calculer les graphes KNN pour les grands ensembles de données donnerait accès à la personnalisation aux entreprises qui ne pouvaient pas se le permettre autrement. En outre, le temps de calcul a un impact pratique lorsque la fraîcheur des données est très précieuse. Les données peuvent avoir tellement changé au cours du calcul du graphe KNN que le résultat lui-même est déjà inutile à la fin du calcul. Dans les recommandations par exemple, l'intérêt des utilisateurs peut changer d'un jour à l'autre, avec le début des jeux olympiques par exemple. Dans ces cas, il est préférable d'avoir un graphe KNN approximatif mais rapide à calculer qu'un graphe exact.

Les approches existantes ignorent certains calculs de similarité entre les utilisateurs pour éliminer une partie de la complexité quadratique. Pourtant, leurs temps de calcul restent élevés et il semble difficile d'aller encore plus loin dans cette direction et d'abaisser davantage le nombre de calculs de similarités. Comme les ensembles de données deviennent de plus en plus grands, que ce soit en termes d'utilisateurs ou de caractéristiques, nous devons trouver de nouvelles approches alternatives pour accélérer le calcul du graphe KNN.

Thèse et contributions

Claim :
Les algorithmes existants basent leurs approximations sur le fait que tous les candidats ne sont pas tous examinés. Dans cette thèse, nous poussons plus loin la notion d'approximation en approximant les profils, la métrique de similarité et la localité utilisée pour regrouper les utilisateurs et exécuter les calculs efficacement.

Dans cette thèse, nous apportons les trois contributions suivantes :

Approximation des profils d'utilisateurs : échantillonnage

Pour illustrer notre affirmation, nous proposons d'abord une approximation des profils en échantillonant les caractéristiques. Notre première contribution est une nouvelle stratégie visant à approximer le profil de chaque utilisateur par un plus petit. Nous échantillonnons les profils des utilisateurs pour ne conserver qu'un sous-ensemble de chaque profil. L'échantillonnage a pour but de limiter le nombre d'entités dans le profil de chaque utilisateur afin de limiter le temps passé sur chaque calcul de similarité. Nous montrons que garder les caractéristiques les moins populaires est la meilleur des façon d'échantillonner. Nous avons expérimenté notre nouvelle stratégie avec plusieurs autres façons d'échantillonner et les avons appliquées à l'état de l'art des algorithmes de calcul de graphe KNN. Plus on échantillonne, plus rapide est le calcul mais plus la qualité du graphe obtenu est faible. Néanmoins, en gardant les 25 fonctionnalités les moins populaires pour chaque utilisateur nous réduisons le temps de calcul de 63%. Les graphes KNN obtenus fournissent des recommandations aussi bonnes que celles obtenues avec les graphes KNN exacts.

Approximation de la similarité : GoldFinger

Notre deuxième contribution va encore plus loin : nous **approximons la similarité**. Pour accélérer le calcul de la similarité, proposons une représentation compacte de l'ensemble des caractéristiques associées à chaque noeud. Cette structure, appelée *Single Hash Fingerprint* (SHF), est un vecteur de 64 à 8096 bits résumant le profil. Les SHF sont très rapides à construire, ils protègent la confidentialité des utilisateurs en cachant l'information originale, et fournit une approximation suffisante de la similarité entre deux noeuds en utilisant des opérations peu coûteuses. Nous proposons d'utiliser ces SHF pour construire rapidement des graphes KNN, dans une approche globale que nous avons surnommée GoldFinger. GoldFinger est *générique* et *efficace* : il peut être utilisé pour accélérer n'importe quel algorithme de construction de graphe

KNN reposant sur l'index Jaccard. Il ajoute peu de temps et la taille des SHF individuels peut être ajustée pour avoir un compromis entre l'espace et le temps de calcul. GoldFinger fournit également des propriétés de confidentialité intéressantes, à savoir du k-anonymat et de la ℓ-diversity. Ces propriétés sont obtenues *gratuitement* en compactant les profils. Nous montrons que GoldFinger est capable de fournir des accélérations allant jusqu'à 78,9% par rapport aux approches existantes, tout en souffrant d'un petite perte en termes de qualité. Malgré la faible perte de qualité, il n'y a aucune perte de qualité des recommandations.

Approximation de la localité : Cluster-and-Conquer

L'utilisation des profils compressés a tellement réduit le temps de calcul de la similarité que le goulot d'étranglement a changé. Diminuer encore plus le temps passé sur la similarité entraînerait une amélioration négligeable par rapport au calcul total. L'accès à la mémoire est le nouveau goulot d'étranglement. Ignorer la localisation des données peut être un problème, et l'optimization de cette-ci est la solution pour minimiser les accès aléatoire à la mémoire. Notre troisième contribution est un nouvel algorithme augmentant la localité des données en s'appuyant sur une stratégie de diviser pour régner. Les utilisateurs sont regroupés en utilisant une nouvelle fonction de hachage. La fonction de hachage est rapide à calculer et préserve approximativement la topologie des utilisateurs : les utilisateurs similaires ont tendance à être hachés ensemble. Plusieurs fonctions de hachage permettent de s'assurer que les utilisateurs sont hachés au moins une fois avec leurs voisins. Les graphes KNN sont calculés localement et indépendamment sur le sous-jeu de données de chaque cluster et sont ensuite fusionnés. Cluster-and-conquer produit des accélérations jusqu'à 9 fois supérieures par rapport aux approches existantes s'appuyant sur des données brutes. Comme avec GoldFinger, la perte de qualité des recommandations dérivées est négligeable.

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

Titre : Les multiples facettes des approximations dans la construction de graphes KNN.

Mot clés : graphe KNN, apprentissage, Base de données, Recommandations

Resumé : La quantité incroyable de contenu disponible dans les services en ligne rend le contenu intéressant incroyablement difficile à trouver. La manière la plus emblématique d'aider les utilisateurs consiste à faire des recommandations. Le graphe des K-plus-proches-voisins (K-Nearest-Neighbours (KNN)) connecte chaque utilisateur aux k autres utilisateurs qui lui sont les plus similaires, étant donnée une fonction de similarité. Le temps de calcul d'un graphe KNN exact est prohibitif dans les services en ligne. Les approches existantes approximent l'ensemble de candidats pour chaque voisinage pour diminuer le temps de calcul. Dans cette thèse, nous poussons plus loin la notion d'approximation : nous approximons les données de chaque utilisateur, la similarité et la localité de données. L'approche obtenue est nettement plus rapide que toutes les autres.

Title : The many faces of approximation in KNN graph computing.

Keywords : KNN graph, Machine learning, Database, Recommendations

Abstract : The incredible quantity of available content in online services makes content of interest incredibly difficult to find. The most emblematic way to help the users is to do item recommendation. The K-Nearest-Neighbors (KNN) graph connects each user to its k most similar other users, according to a given similarity metric. The computation time of an exact KNN graph is prohibitive in online services. Existing approaches approximate the set of candidates for each user's neighborhood to decrease the computation time. In this thesis we push farther the notion of approximation : we approximate the data of each user, the similarity and the data locality. The resulting approach clearly outperforms all the other ones.