Fingerprinting Big Data: The Case of KNN Graph Construction

Rachid Guerraoui, Anne-Marie Kermarrec, Olivier Ruas, François Taïani

Fingerprinting Big Data: The Case of KNN Graph Construction

Rachid Guerraoui Anne-Marie Kermarrec, Olivier Ruas, François Taïani

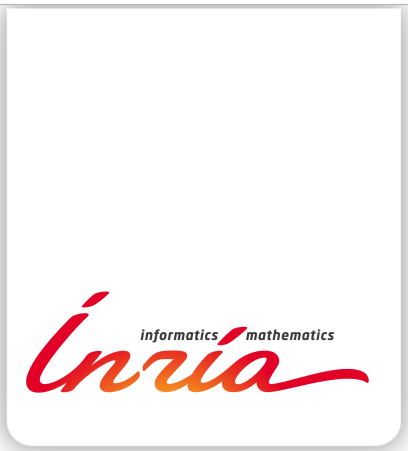
RESEARCH REPORT

N° 9218

October 2018

Project-Teams WIDE

ISSN 0249-6399 ISRN INRIA/RR--9218--FR+ENG



Fingerprinting Big Data: The Case of KNN Graph Construction

Rachid Guerraoui* Anne-Marie Kermarrec†, Olivier Ruas‡, François Taïani‡

Project-Teams WIDE

Research Report n° 9218 — October 2018 — 30 pages

Abstract: We propose *fingerprinting*, a new technique that consists in constructing *compact*, *fast-to-compute* and *privacy-preserving* binary representations of datasets. We illustrate the effectiveness of our approach on the emblematic big data problem of K-Nearest-Neighbor (KNN) graph construction and show that fingerprinting can drastically accelerate a large range of existing KNN algorithms, while efficiently obfuscating the original data, with little to no overhead. Our extensive evaluation of the resulting approach (dubbed GoldFinger) on several realistic datasets shows that our approach delivers speedups of up to 78.9% compared to the use of raw data while only incurring a negligible to moderate loss in terms of KNN quality. To convey the practical value of such a scheme, we apply it to item recommendation, and show that the loss in recommendation quality is negligible.

Key-words: KNN graphs, fingerprint, similarity

* EPFL, Switzerland
† Mediego / EPFL, Switzerland
‡ Inria, Univ Rennes, CNRS, IRISA (France)

RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

Empreintes numériques et Big Data: le cas de la construction des graphes KNN

Résumé : Nous proposons une nouvelle sorte d'empreintes numériques qui est une représentation binaire des données qui est *compacte*, *rapide à calculer* et qui *protège la vie privée*. Nous illustrons l'efficacité de notre approche sur l'emblématique problème de la construction des graphes des k-plus-proches-voisins (KNN) et nous montrons que les empreintes numériques peuvent fortement accélérer de nombreux algorithmes existants tout en brouillant efficacement les données. Notre évaluation poussée de l'approche résultante (appelée GoldFinger) sur plusieurs datasets réels montre que notre approche produit une accélération atteignant 78.9% comparée à l'utilisation des données brutes, tout en ne souffrant que d'une légère perte en qualité. Pour montrer l'utilité pratique de GoldFinger, nous l'appliquons à la recommandation et montrons que la perte en qualité de recommandation est négligeable.

Mots-clés : graphes KNN, empreinte numérique, similarités

1 Introduction

1.1 Overcoming the machine learning cost barrier

Today's ever growing volumes of data have prompted the development of increasingly powerful learning and analysis platforms [35]. Our own experience within a start-up active in this area is that, unfortunately, the cost of running these solutions (whether in a cloud or on dedicated servers) can rapidly become a substantial drag for small businesses. As a result, small companies cannot often expect to recoup the cost of high-end machine-learning techniques running on well-provisioned machines. They have paradoxically more data than they can economically fully exploit.

To overcome this cost barrier we propose a strategy we have called *big data fingerprinting*. Our approach rapidly reduces the size of data into a *compact* and efficient *binary* format that becomes tractable to process on small machines. As an additional benefit, our solution also obfuscates the original clear-text information, at no additional cost. This obfuscation brings useful protection against the privacy risks associated with sensitive data that many organizations must address.

In this paper, we illustrate the potential of our approach on the emblematic problem in big data of K-Nearest-Neighbor (KNN) graph computation.

1.2 K-Nearest-Neighbor (KNN) graphs

K-Nearest-Neighbor (KNN) graphs[1] play a fundamental role in many big data applications, including search [5, 6], recommendation [8, 34, 38] and classification [46]. A KNN graph is a directed graph of entities (e.g., users, documents etc.), in which each entity (or *node*) is connected to its k most similar counterparts or *neighbors*, according to a given *similarity metric*. In many applications, this similarity metric is computed from a second set of entities (termed *items*) associated with each node in a bipartite graph (often extended with weights, such as ratings or frequencies). For instance, in a movie rating database, nodes are users, and each user is associated with the movies (items) she has already rated [25].

Being able to compute a KNN graph efficiently is crucial in situations that are constrained, either in terms of time or resources. This is the case of *real time*[2] web applications, such as news recommenders and *trending* services, that must regularly recompute their suggestions in short intervals on fresh data to remain relevant. This is also the case of privacy-preserving personal assistants executing learning tasks on personal devices with limited resources [1].

Computing an exact KNN graph rapidly becomes intractable on large datasets: under a brute force strategy, a dataset with few thousands of nodes requires tens of billions of similarity computations. Many applications, however, only require a good approximation of the KNN graph [29, 31]. Recent KNN construction algorithms [8, 22] have therefore sought to reduce the number of similarity computations by exploiting a *greedy strategy*. These approaches start from an initial

[1] Note that the problem of computing a complete KNN graph (which we address in this paper) is related but different from that of answering a sequence of KNN queries.

[2] *Real time* is meant in the sense of *web real-time*, i.e. the proactive push of information to on-line users.

random graph that is iteratively refined towards an approximate nearest neighbors (ANN) graph. These techniques are among the most efficient to date, but they seem to have reached their limits, and it is now difficult to see how to further reduce the number of comparisons pursuing this paradigm.

1.3 Fingerprinting Big Data for space and speed

In this paper, rather than reducing an algorithm's complexity (e.g. by decreasing the number of similarity computations), we propose to pursue an orthogonal strategy that is motivated by the system bottlenecks created by Big Data algorithms: large amounts of data not only stress complex algorithms, they also choke the underlying computation pipelines these algorithms execute on. To avoid these costs, we argue that one should avoid 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.

Instantiated in the context of KNN graph construction, 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 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, at close to no overhead, and can be tuned to trade space and time for accuracy.

1.4 Contributions

In this paper we make the following contributions:

- We propose a formal analysis of GoldFinger's estimation mechanism.
- We formally analyze the privacy protection properties of GoldFinger in terms of k-anonymity and ℓ-diversity.
- We extensively evaluate our approach on a range of state-of-the-art KNN graph algorithms such as Locality Sensitive Hashing (LSH), on six representative datasets. We show that GoldFinger is able to deliver speedups of up to 78.9% against existing approaches, while incurring a small loss in terms of quality.
- As a case-study, we use the constructed graphs to produce recommendations, and show that despite the small loss in KNN quality, there is close to no loss in the quality of the derived recommendations.

In the following, we first present the context of our work and our approach (Sec. 2). We then present our evaluation procedure (Sec. 3) and our results (Sec. 4); we report on factors impacting our approach (Sec. 5), before discussing related work (Sec. 6), and concluding (Sec. 7).

2 Problem, Intuition, and Approach

For ease of exposition, we consider in the following that nodes are *users* associated with *items* (e.g. web pages, movies, locations), without loss of generality.

2.1 Notations and problem definition

We note $U = \{u_1, ..., u_n\}$ the set of all users, and $I = \{i_1, ..., i_m\}$ the set of all items. The subset of items associated with user u (a.k.a. its *profile*) is noted $P_u \subseteq I$. P_u might contain for instance the web pages visited by u, and is generally much smaller than I (the universe of all items).

Our objective is to approximate a k-nearest-neighbor (KNN) graph over U (noted G_{KNN}) according to some similarity functions sim computed over user profiles:

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

f_{sim} may be any similarity function over sets that is positively correlated with the number of common items between the two sets, and negatively correlated with the total number of items present in both sets. These requirements cover some of the most commonly used similarity functions in KNN graph construction applications, such as cosine or Jaccard's index. We use Jaccard's index in the rest of the paper [51].

Formally, a KNN graph G_{KNN} connects each user $u \in U$ with a set $knn(u)$ of k other users that maximize the similarity function $sim(u, -)$:

$$knn(u) \in \underset{v \in U \setminus \{u\}}{\text{argtop}^k} f_{sim}(P_u, P_v) \tag{1}$$

where argtop^k returns the set of k-tuples of $U \setminus \{u\}$ that maximize the similarity function $sim(u, -)$[3].

Computing an exact KNN graph is particularly expensive: a brute-force exhaustive search requires $O(|U|^2)$ similarity computations. Many scalable approaches therefore seek to construct *an approximate KNN graph* $\widehat{G}_{\text{KNN}}$, i.e., to find for each user u a neighborhood $\widehat{knn}(u)$ that is as close as possible to an exact KNN neighborhood [8, 22]. The meaning of 'close' depends on the context, but in most applications, a good approximate neighborhood $\widehat{knn}(u)$ is one whose aggregate similarity (its *quality*) comes close to that of an exact KNN set $knn(u)$.

We capture how well the average similarity of an approximated graph $\widehat{G}_{\text{KNN}}$ compares against that of an exact KNN graph G_{KNN} with the *average similarity* of $\widehat{G}_{\text{KNN}}$:

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

i.e. the average similarity of the edges of $\widehat{G}_{\text{KNN}}$. We then define the *quality* of $\widehat{G}_{\text{KNN}}$ as

$$quality(\widehat{G}_{\text{KNN}}) = \frac{avg\_sim(\widehat{G}_{\text{KNN}})}{avg\_sim(G_{\text{KNN}})}. \tag{3}$$

[3] In other words, argtop^k generalizes the concept of argument of the maximum (usually noted argmax) to the k top values of a function over a finite discrete set.

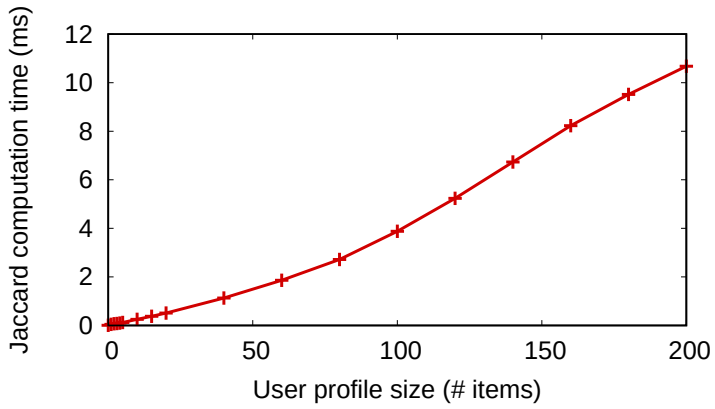


Figure 1: The cost of computing Jaccard's index between explicit user profiles is relatively high (a few *ms*) for average-size profiles. Cost averaged over 4.9 millions computations between randomly generated profiles on a Intel Xeon E5420@2.50GHz.

A quality close to 1 indicates that the approximate neighborhoods have a quality close to that of ideal neighborhoods, and can replace them with little loss in most applications.

With the above notations, we can summarize our problem as follows: for a given dataset $(U, I, (P_u)_{u \in U})$ and item-based similarity f_{sim}, we wish to compute an approximate $\widehat{G}_{\text{KNN}}$ in the shortest time with the highest overall quality.

2.2 Intuition

A large portion of a KNN graph's construction time often comes from computing individual similarity values (up to 90% of the total construction time in some recent approaches [9]). This is because computing explicit similarity values on even medium-size profiles can be relatively expensive. For instance, Figure 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. 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, a typical profile size of the datasets we have considered.

Earlier KNN graph construction approaches have therefore sought to limit the number of similarity computations [8, 22]. They typically adopt a greedy strategy, starting from a random graph, and progressively converging to a better KNN approximation by navigating neighbor-of-neighbor relationships. They only perform a fraction of the similarity computations required by an exhaustive search, show a high memory locality, and are easily parallelizable, but it is now difficult to see how their greedy component could be further improved.

In order to overcome the inherent cost of similarity computations, we therefore propose to target the data on which computations run, rather than the

Table 1: Effect of SHFs on computation time of Jaccard's index, compared to Fig. 1 (80 items).

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*

algorithms that drive these computations. This strategy stems from the observation that *explicit* datastructures (hash tables, arrays) incur substantial costs. To avoid these costs, we advocate the use of *fingerprints*, a *compact*, *binary*, and *fast-to-compute* representation of data.

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.

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

$$\widehat{J}(P_1, P_2) = \frac{\|B_1 \textbf{ AND } B_2\|_1}{c_1 + c_2 - \|B_1 \textbf{ AND } B_2\|_1}, \tag{4}$$

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):

$$|P| \approx \|B_P\|_1 = c_p. \tag{5}$$

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:

$$B_{(P_1 \cap P_2)} \approx (B_1 \textbf{ AND } B_2). \tag{6}$$

Equation (4) 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 (4) is much faster than on explicit profiles, and is independent of the actual size of the explicit profiles. This is illustrated in Table 1 which shows the computation time of Eq. (4) on the same profiles as Figure 1 for SHFs of different lengths (as in Fig. 1, the values are averaged over 4.9 millions computations). For instance, estimating Jaccard's index between two SHFs of 1024 bits (the default in our experiments) takes 0.120 ms, a 23-fold speedup compared to two explicit profiles of 80 items.

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). Bloom filters often employ multiple hash functions to minimize false positives. By contrast, multiple hash functions increase single-bit collisions, and therefore degrade the approximation provided by SHFs.

2.4 Formal analysis of the Jaccard estimator $\widehat{J}(P_1, P_2)$

For readability in this section, we will generally treat bit arrays (i.e. belonging to $\{0,1\}^n$) as sets of bit positions (belonging to $\mathcal{P}([\![0, b-1]\!])$). We will also note B_X the set of bits set to `1` by the (sub)profile P_X: $B_X = h(P_X)$.

For two given profiles P_1 and P_2, the distribution of $\widehat{J}(P_1, P_2)$ is governed by how the random hash function h maps the items of P_1 and P_2 unto the bit positions $[\![0, b-1]\!]$. The analysis of this random mapping is made simpler if we distinguish between the items that are present in both P_1 and P_2 ($P_\cap = P_1 \cap P_2$) from those are unique to P_1 (resp. P_2), noted $P_{\Delta 1} = P_1 \setminus P_\cap$ (resp. $P_{\Delta 2} = P_2 \setminus P_\cap$). These three sets are represented by the three top rectangles of Figure 2.

$P_\cap$, $P_{\Delta 1}$, and $P_{\Delta 2}$ are disjoint by definition, but their bit images $B_\cap$, $B_{\Delta 1}$, and $B_{\Delta 2}$ by h (shown as circles in Figure 2) are typically not, because of collisions. To analyze these collisions we introduce the following three helping sets:

- $B_{\widehat{\eta}_1}$ are the bits of $B_{\Delta 1}$ (corresponding to items only present in P_1) that do not collide with those of $B_\cap$:

$$B_{\widehat{\eta}_1} = B_{\Delta 1} \setminus B_\cap;$$

- $B_{\widehat{\eta}_2}$ is the equivalent for P_2:

$$B_{\widehat{\eta}_2} = B_{\Delta 2} \setminus B_\cap;$$

- and $B_{\widehat{\beta}}$ contains the collisions between $B_{\widehat{\eta}_1}$ and $B_{\widehat{\eta}_2}$:

$$B_{\widehat{\beta}} = B_{\widehat{\eta}_1} \cap B_{\widehat{\eta}_2}.$$

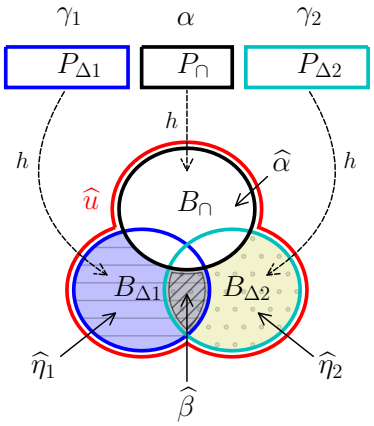


Figure 2: Illustration of the collisions caused by h on two profiles $P_1 = P_{\Delta 1} \cup P_\cap$ and $P_2 = P_{\Delta 2} \cup P_\cap$. Capital letters ($P$ and B) denote sets, Greek letters denote sizes, and the hat symbol ($\widehat{\ }$) represent random variables.

If we note $\widehat{\alpha}, \widehat{\eta}_1, \widehat{\eta}_2, \widehat{\beta}$ the sizes of the sets $B_\cap$, $B_{\widehat{\eta}_1}$, $B_{\widehat{\eta}_2}$, and $B_{\widehat{\beta}}$, respectively, we can express $\widehat{J}(P_1, P_2)$ as

$$\widehat{J}(P_1, P_2) = \frac{\widehat{\alpha} + \widehat{\beta}}{\widehat{\alpha} + \widehat{\eta}_1 + \widehat{\eta}_2 - \widehat{\beta}} = \frac{2\widehat{\alpha} + \widehat{\eta}_A + \widehat{\eta}_B}{\widehat{u}} - 1, \tag{7}$$

where $\widehat{u} = \widehat{\alpha} + \widehat{\eta}_1 + \widehat{\eta}_2 - \widehat{\beta}$ is the number of bits set to 1 by either P_1 or P_2, i.e. $\widehat{u} = |B_1 \cup B_2| = \|B_1 \textbf{ OR } B_2\|_1$.

The distribution of $\widehat{J}(P_1, P_2)$ is determined by the joint distribution of the random values $(\widehat{u}, \widehat{\alpha}, \widehat{\eta}_1, \widehat{\eta}_2)$ when h (shown with dashed arrows in Figure 2) is chosen uniformly randomly among all functions h that map $P_{\Delta 1} \cup P_\cap \cup P_{\Delta 2}$ onto $[\![0, b-1]\!]$. (Note that $\widehat{\beta} = \widehat{\alpha} + \widehat{\eta}_1 + \widehat{\eta}_2 - \widehat{u}$ is determined by the four other values, and therefore does not appear in the quadruplet.)

Theorem 1. *The probability distribution of $(\widehat{u}, \widehat{\alpha}, \widehat{\eta}_1, \widehat{\eta}_2)$ is given by*

$$\mathbb{P}(\widehat{u}, \widehat{\alpha}, \widehat{\eta}_1, \widehat{\eta}_2 | \alpha, \gamma_1, \gamma_2) = \frac{Card\_h(\widehat{u}, \widehat{\alpha}, \widehat{\eta}_1, \widehat{\eta}_2, \alpha, \gamma_1, \gamma_2)}{b^{(\alpha+\gamma_1+\gamma_2)}}, \tag{8}$$

where

- *α, γ_1, and γ_2 are resp. the sizes of the sets $P_\cap$, $P_{\Delta 1}$, and $P_{\Delta 2}$ introduced earlier;*
- *$Card\_h()$ is the number of hashing functions from $P_{\Delta 1} \cup P_\cap \cup P_{\Delta 2}$ unto $[\![0, b-1]\!]$ that produce the quadruplet $(\widehat{u}, \widehat{\alpha}, \widehat{\eta}_1, \widehat{\eta}_2)$, defined by*

$$Card\_h(\widehat{u}, \widehat{\alpha}, \widehat{\eta}_1, \widehat{\eta}_2, \alpha, \gamma_1, \gamma_2) = \binom{b}{\widehat{u}} \binom{\widehat{u}}{\widehat{\alpha}} \binom{\widehat{u} - \widehat{\alpha}}{\widehat{\beta}} \binom{\widehat{u} - \widehat{\alpha} - \widehat{\beta}}{\widehat{\eta}_1 - \widehat{\beta}} \widehat{\alpha}! \left\{ {\alpha \atop \widehat{\alpha}} \right\} \times \xi(\gamma_1, \widehat{\eta}_1 + \widehat{\alpha}, \widehat{\eta}_1) \times \xi(\gamma_2, \widehat{\eta}_2 + \widehat{\alpha}, \widehat{\eta}_2);$$

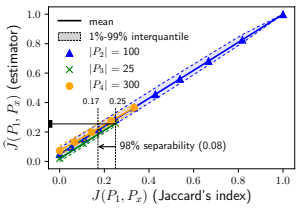


Figure 3: Mean and 1%-99% interquantile of $\widehat{J}$ between a profile P_1 of 100 items, and 3 profiles P_2, P_3, P_4 of varying sizes, using SHFs of $b = 1024$ bits. $\widehat{J}$ is biased, but shows a tight spread of values. As a result the risk of misordering neighbors drops fast with decreasing similarity.

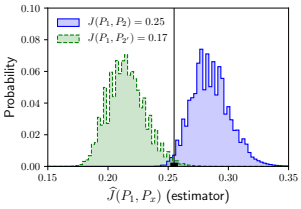


Figure 4: Distribution of estimation when the real Jaccard indices with P_1 equal 0.25 and 0.17 (represented in bins of 0.0025). $|P_1| = |P_2| = |P_{2'}| = 100$ items, $b = 1024$ bits. When $P_{2'}$'s real similarity with P_1 drops below 0.17, the risk of misordering P_2 and P'_2 drops below 2%.

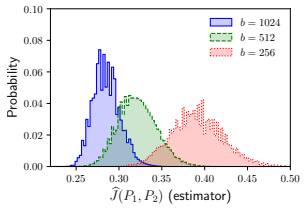


Figure 5: Distribution of estimations $\widehat{J}(P_1, P_2)$, when $\widehat{J}(P_1, P_2) = 0.25$, $|P_1| = |P_2| = 100$ items, for different values of b. As the size of the SHFs decreases, the spread of estimations grows, leading to more frequent misordering of profiles over shorter ranges.

- $\left\{ {\alpha \atop \widehat{\alpha}} \right\}$ *denotes Stirling's number of the second type;*
- *and* $\xi(x, y, z)$ *is the number of functions* $f : X \mapsto Y$ *from a finite set* X *onto a finite set* Y*, that are surjective on a subset* $Z \subseteq Y$ *of* Y*, with* $x = |X|$*,* $y = |Y|$*,* $z = |Z|$*, defined by*

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

Proof. The formula for $\mathbb{P}(\widehat{u}, \widehat{\alpha}, \widehat{\eta}_1, \widehat{\eta}_2 | \alpha, \gamma_1, \gamma_2)$ derives from a counting strategy on the functions from $P_\cup = P_{\Delta 1} \cup P_\cap \cup P_{\Delta 2}$ unto $[\![0, b-1]\!]$: the denominator $b^{(\alpha+\gamma_1+\gamma_2)}$ is the total number of such functions.

The numerator, $Card\_h()$, is the number of functions h that yield precisely the quadruplet $(\widehat{u}, \widehat{\alpha}, \widehat{\eta}_1, \widehat{\eta}_2)$. We obtain $Card\_h()$ with a constructive argument. Because $P_\cap$, $P_{\Delta 1}$ and $P_{\Delta 2}$ are disjoint, h can be seen as the piece-wise combination of three independent random functions $h_{|P_\cap}$, $h_{|P_{\Delta 1}}$, and $h_{|P_{\Delta 2}}$, where $h_{|X}$ denotes the restriction of h to a set X.

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

Similar arguments for $B_\cap$, $B_{\widehat{\beta}}$, and $B_{\widehat{\eta}_1} \setminus B_\cap$ yield that overall the total number of choices for $B_\cup$, $B_\cap$, $B_{\widehat{\beta}}$, $B_{\widehat{\eta}_1} \setminus B_\cap$ (and $B_{\widehat{\eta}_2} \setminus B_\cap$) is $\binom{b}{\widehat{u}}\binom{\widehat{u}}{\widehat{\alpha}}\binom{\widehat{u}-\widehat{\alpha}}{\widehat{\beta}}\binom{\widehat{u}-\widehat{\alpha}-\widehat{\beta}}{\widehat{\eta}_1-\widehat{\beta}}$.

Once these supporting sets have been chosen, we pick $h_{|P_\cap}$, $h_{|P_{\Delta 1}}$, and $h_{|P_{\Delta 2}}$. $h_{|P_\cap}$ is a surjection from $P_\cap$ to $B_\cap$. 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.

$h_{|P_{\Delta 1}}$ maps $P_{\Delta 1}$ onto $B_\cup$, but only needs to be surjective on $B_{\Delta 1} \setminus B_\cap = B_{\widehat{\eta}_1}$. In addition, $h_{|P_{\Delta 1}}$ only maps elements unto $B_{\Delta 1} \subseteq B_{\widehat{\eta}_1} \cup B_\cap$, whose cardinal is $\widehat{\eta}_1 + \widehat{\alpha}$.

Let us note $\xi(x, y, z)$ the number of functions $f : X \mapsto Y$ from a finite set X onto a finite set Y, that are 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 $\xi(\gamma_1, \widehat{\eta}_1 + \widehat{\alpha}, \widehat{\eta}_1)$ functions $h_{|P_{\Delta_1}}$. Similarly there are $\xi(\gamma_2, \widehat{\eta}_2 + \widehat{\alpha}, \widehat{\eta}_2)$ functions $h_{|P_{\Delta_2}}$. The product of the above quantities yields *Card_h*(), and as a result (8).

□

Combining the formula for $\mathbb{P}(\widehat{u}, \widehat{\alpha}, \widehat{\eta}_1, \widehat{\eta}_2 | \alpha, \gamma_1, \gamma_2)$ provided by Theorem 1 and Eq. (7), we can compute the distribution and moments of the estimator $\widehat{J}$ for any $\alpha, \gamma_1, \gamma_2$. For instance, $\widehat{J}$'s mean is equal to

$$\mathbb{E}\big(\widehat{J}(P_1, P_2)\big) = \sum_{(\widehat{u}, \widehat{\alpha}, \widehat{\eta}_1, \widehat{\eta}_2)} \mathbb{P}(\widehat{u}, \widehat{\alpha}, \widehat{\eta}_1, \widehat{\eta}_2 | \alpha, \gamma_1, \gamma_2) \times \left(\frac{2\widehat{\alpha} + \widehat{\eta}_A + \widehat{\eta}_B}{\widehat{u}} - 1 \right).$$

Figure 3 uses this strategy to plot the behavior of the estimator $\widehat{J}$ against the real Jaccard index when comparing a profile, P_1 of 100 items with other profiles P_2, P_3, P_4 of varying sizes.

$\widehat{J}$ is biased. For instance when $J(P_1, P_2) = 0.25$—the right vertical dashed line in the figure—$\widehat{J}$ returns an average value of 0.286. However, this absolute bias has little impact on KNN algorithms, which only need to *order* nodes correctly, rather than to predict exact similarities. Instead, the spread and overlap of the values returned by $\widehat{J}$ drive its impact on a KNN approximation. This effect is thankfully limited: for instance, if we assume that P_2 belongs to P_1's exact KNN neighborhood with a similarity of $J(P_1, P_2) = 0.25$, an algorithm using $\widehat{J}$ might exclude P_2 if it finds another profile $P_{2'}$ that $\widehat{J}$ wrongly considers as more similar:

$$J(P_1, P_{2'}) < J(P_1, P_2) \text{ and } \widehat{J}(P_1, P_{2'}) > \widehat{J}(P_1, P_2).$$

Figure 3 shows this misordering has a very low probability of occurring (less than 2%) when $J(P_1, P_{2'})$ is lower than 0.17 (left vertical dashed line). This is because $\widehat{J}(P_1, P_2)$ has a 99% probability of being *higher* than 0.254 (1%-percentile value, shown as a solid horizontal line with a rectangle mark on the y-axis), while $P_{2'}$ profiles with a real Jaccard's index lower than 0.17 have a 99% probability of being *lower* than this cut-off value (and this is roughly independent of $P_{2'}$'s size). This phenomenon is shown in more detail in Figure 4 when $P_{2'}$ contains 100 items: the error in similarity caused by the use of SHFs to compute $\widehat{J}$ is bounded with high probability by a quantity proportional to the spread of estimated values. When SHFs are large enough compared to the size of the profiles being compared (here with $b = 1024$ bits), this spread is limited, but it increases as b gets smaller (Figure 5), highlighting a natural trade-off between compactness and accuracy that we will revisit in Section 4.

2.5 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 profile from its compacted SHF. This obfuscation can allow 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* [49], and *ℓ-diversity* [41]. 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?

2.5.1 k-anonymity

Definition 1. *Consider an obfuscation mechanism $obf : \mathcal{X} \mapsto \mathcal{Y}$ that maps a clear-text 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*

$$\left| obf^{-1}\left(obf\left(x\right)\right)\right| \geq k. \tag{9}$$

Theorem 2. *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 profile 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 SHFs ensure at least $(2^{\frac{m}{b}})$-anonymity for the whole dataset.

The anonymity granted by GoldFinger increases with the size of the item set $m = |I|$. For instance, one of the datasets we consider, AmazonMovies, has 171,356 items. With 1024-bit SHFs (the typical size we use), GoldFinger provides 2^{167}-anonymity, i.e. each compacted profile is indistinguishable from at least $2^{167} \approx 1.87 \times 10^{50}$ possible profiles.

2.5.2 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

Table 2: Description of the datasets used in our experiments

Dataset	Users	Items	Scale	Ratings > 3	$\lvert P_u \rvert$	$\lvert P_i \rvert$	Density
movielens1M (ml1M) [25]	6,038	3,533	1-5	575,281	95.28	162.83	2.697%
movielens10M (ml10M) [25]	69,816	10,472	0.5-5	5,885,448	84.30	562.02	0.805%
movielens20M (ml20M) [25]	138,362	22,884	0.5-5	12,195,566	88.14	532.93	0.385%
AmazonMovies (AM) [43]	57,430	171,356	1-5	3,263,050	56,82	19.04	0.033%
DBLP [53]	18,889	203,030	5	692,752	36.67	3.41	0.018%
Gowalla (GW) [17]	20,270	135,540	5	1,107,467	54.64	8.17	0.040%

captured by a second metric, ℓ-*diversity* [41]. 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 possible profiles of a given SHF include science-fiction movies, you can infer that the user enjoys science fiction. ℓ-diversity measures the difficulty of such inferences.

Definition 2. *Consider an obfuscation mechanism* $obf : \mathcal{P}(I) \mapsto \mathcal{Y}$ *that maps a set of items* $P \subseteq I$ *to an obfuscated value in* $\mathcal{Y}$. $obf()$ *is* ℓ-anonymous *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*[4], $obf()$ *is* ℓ*-anonymous for* $P \subseteq I$ *iff*

$$\max_{\substack{\mathcal{Q} \subseteq obf^{-1}(obf(P)) \setminus \{P\}: \\ \forall P_1, P_2 \in \mathcal{Q}: P_1 \cap P_2 = \emptyset}} |\mathcal{Q}| \geq \ell - 1. \tag{10}$$

Theorem 3. *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. Assuming an arbitrary order on 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 [23]. It can be easily obtained by inserting random noise to the SHF [2].

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

3 Experimental Setup

3.1 Datasets

We use six publicly available datasets (Table 2). We binarize each dataset by only keeping in a user profile P_u those items that user u has rated higher than 3.

3.1.1 Movielens

Movielens [25] is a group of anonymous datasets containing movie ratings collected on-line between 1995 and 2015 by GroupLens Research [47]. The datasets (before binarization) contain movie ratings on a 0.5-5 scale by users who have at least performed 20 ratings. We use 3 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).

3.1.2 AmazonMovies

AmazonMovies [43] (AM) is a dataset of movies reviews from Amazon collected between 1997 and 2012. We restrain our study to users with at least 20 ratings (before binarization) to avoid users with not enough data (this problem, the *cold start problem*, is generally treated separately [32]). After binarization, the dataset contains 57,430 users; 171,356 items; and 3,263,050 ratings.

3.1.3 DBLP

DBLP [53] 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 of 5. As with AM, we only consider users with at least 20 ratings. The resulting dataset contains 18,889 users, 203,030 items; and 692,752 ratings.

3.1.4 Gowalla

Gowalla [17] (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.

3.2 Baseline algorithms and competitors

We apply GoldFinger to four existing KNN algorithms: Brute Force (as a reference point), NNDescent [22], Hyrec [8] and LSH [27]. 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* [37], a binary sketching technique proposed to estimate Jaccard's index between sets, albeit in a different context than ours.

Table 3: Preparation time of each dataset for the native approach, b-bit minwise hashing (MinHash) & GoldFinger.

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

GoldFinger is orders of magnitude faster than MinHash, whose overhead is prohibitive.

3.2.1 b-bit minwise hashing (MinHash)

A standard technique to approximate Jaccard's index values between sets is the *MinHash* [10] algorithm. MinHash creates multiple independent permutations on the IDs of a item universe, and keeps for each profile the item with the smallest ID, after each permutation. The Jaccard's index between two profiles can be estimated by counting the proportion of minimal IDs that are equal in the compacted representation of the two profiles.

Although *MinHash* does not produce a compact binary representation, it was recently extended to keep only the lowest b bits of each minimal element [37]. This approach, called *b-bit minwise hashing* (we call it MinHash for short, even though it is an improvement of the original algorithm), creates very compact binary summaries of profiles, comparable to our SHFs, from which a Jaccard's index can be estimated.

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 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 [28]), 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 3 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.

3.2.2 Brute force

The Brute Force algorithm simply computes the similarities between every pair of profiles, performing a constant number of similarity computations equal to $\frac{n\times(n-1)}{2}$. While this is computationally intensive, this algorithm produces an exact KNN graph.

3.2.3 NNDescent

NNDescent [22] constructs an approximate KNN graph (or ANN) by relying on a local search and by limiting the number of similarities computations.

NNDescent starts from an initial random graph, which is then iteratively refined to converge to an ANN graph. 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: it exploits the order on user IDs, and maintains update flags to avoid computing several times the same similarities. It also reverses the current KNN approximation to increase the space search among neighbors. The algorithm stops either when the number of updates during one iteration is below the value $\delta \times k \times n$, with a fixed δ, or after a fixed number of iterations.

3.2.4 Hyrec

Hyrec [8] uses a strategy similar to that of NNDescent, exploiting the fact that a neighbor of a neighbor is likely to be a neighbor. As NNDescent, Hyrec starts with a random graph which is then refined. Hyrec primarily differs from NNDescent in its iteration strategy. At each iteration, for each user u, Hyrec compares all the neighbors' neighbors of u with u, rather than comparing u's neighbors between themselves. Hyrec also does not reverse the current KNN graph. As NNDescent, it stops when the number of changes is below the value $\delta \times k \times n$, with a fixed δ, or after a fixed number of iterations.

3.2.5 LSH

Locality-Sensitive-Hashing (LSH) [27] reduces the number of similarity computations by hashing each user into several buckets. Neighbors are then selected among users found in the same buckets. To insure that similar users tend to be hashed into the same buckets, LSH uses min-wise independent permutations of the item set as its hash functions, similarly to the MinHash algorithm [10].

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 [28].

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 has been prepared), and *(ii)* the *quality* of the resulting KNN (Sec. 2.1). When applying GoldFinger to recommendation, we also measure the *recall* obtained by the recommender. Throughout our experiments, we use a 5-fold cross-validation, and average results on the 5 resulting runs.

3.5 Implementation details and hardware

We have implemented Brute Force, Hyrec, NNDescent and LSH (with and without GoldFinger) in Java 1.8. Our experiments run on a 64-bit Linux server with two Intel Xeon E5420@2.50GHz, totaling 8 hardware threads, 32GB of

Table 4: Computation time and KNN quality with native algorithms (*nat.*) and GoldFinger (GolFi).

datasets		*comp. time (s)*			*KNN quality*		
	algo	**nat.**	**GolFi**	*gain%*	**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*

GoldFinger yields the shortest computation times across all datasets (in bold), yielding gains (*gain*) of up to 78.9% against native algorithms. The loss in quality at worst moderate, ranging from 0.22 to an improvement of 0.11.

memory, and a HHD of 750GB. Unless stated otherwise, we use all 8 threads. Our code is available online[5].

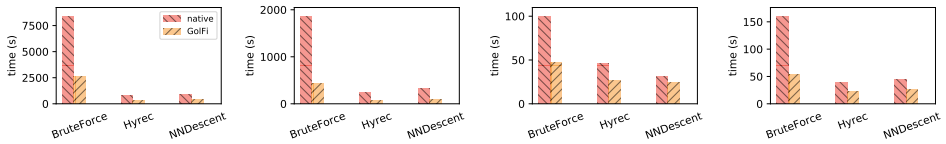
4 Evaluation Results

4.1 Computation time and KNN quality

The performance of GoldFinger (*GolFi*) in terms of execution time and KNN quality is summarized in Table 4. 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 6 (time) and 7 (quality).

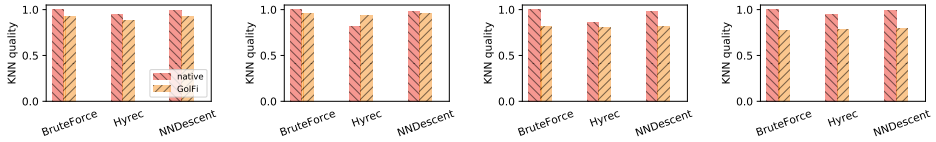
Overall, GoldFinger delivers the fastest computation times across all datasets, for a small loss in quality ranging from 0.22 (with Brute Force on Gowalla) to

[5]https://gitlab.inria.fr/oruas/SamplingKNN



(a) movielens20M (b) AmazonMovies (c) DBLP (d) Gowalla

Figure 6: 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.



(a) movielens20M (b) AmazonMovies (c) DBLP (d) Gowalla

Figure 7: KNN quality using a 1024 bits SHF (higher is better). GoldFinger (GolFi) only experiences a small decrease in quality.

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. Because AmazonMovies, DBLP and Gowalla are comparatively very sparse (Table 2), the buckets created by LSH tend to contain few users. As a result, the overall computation time is dominated by the bucket creation, 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.

4.2 Memory and cache accesses

By compacting profiles, GoldFinger reduces the amount of memory needed to process of dataset. To gauge this effect, we use `perf`[6] to profile the memory accesses of GoldFinger. `perf` uses hardware counters to measure 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.

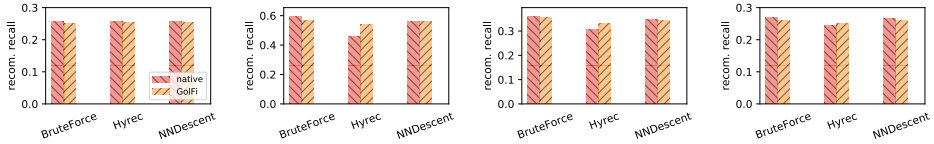
Table 5 summarizes the measures obtained on Brute Force, Hyrec, NNDescent and LSH on ml10M, both without (*native*) and with GoldFinger (*GolFi*).

[6] https://perf.wiki.kernel.org/index.php/Main_Page

Table 5: L1 stores and L1 loads with the native algorithms (*nat.*) and GoldFinger (GolFi) on ml10M.

	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*

GoldFinger drastically reduces the number of L1 accesses, yielding reductions (*gain*) ranging from 67.2% to 87.7%.



(a) movielens20M (b) AmazonMovies (c) DBLP (d) Gowalla

Figure 8: Recommendation quality using a 1024 bits SHF (higher is better). GoldFinger's (GolFi) recall loss is negligible.

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. 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.

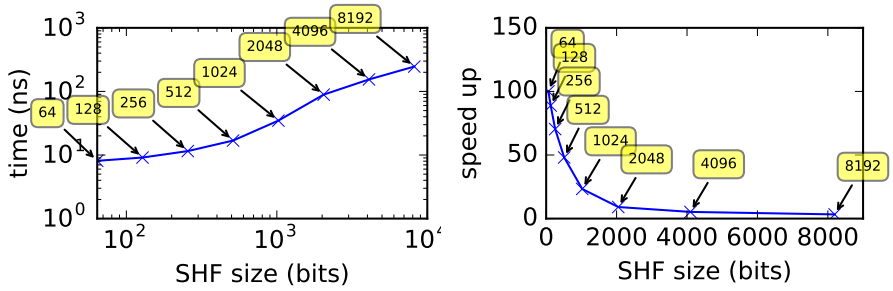
4.3 GoldFinger in action: recommendations

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 8 shows the recall of the recommendation made with the native algorithms and with their GoldFinger counterparts on all datasets. These results



(a) Time of one similarity computation

(b) Speed-up

Figure 9: 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.

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.

5 Sensitivity Analysis

As explained in Section 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 deliver higher speedups, resulting in an inherent trade-off between execution time and quality. In this section we focus on ml10M.

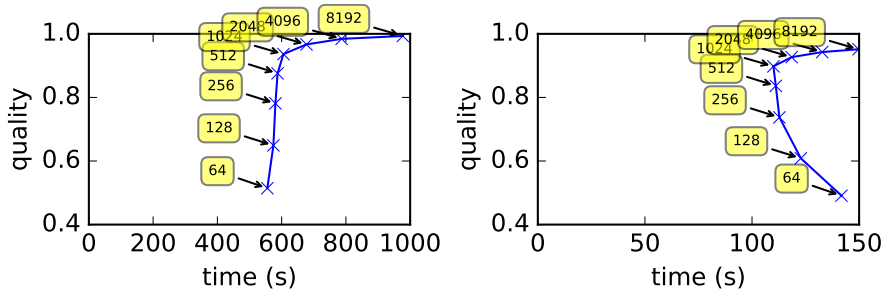
5.1 Impact on the similarity computation time

SHFs aim at drastically decreasing the cost of individual similarity computations. To assess this effect, Figure 9 shows the average computation time of one similarity computation when using SHFs (Eq. 4), and its corresponding 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 time required is divided by the total number of similarities computed, 2.5×10^9, and then averaged over 4 runs. 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.

5.2 Impact on the execution of the algorithm

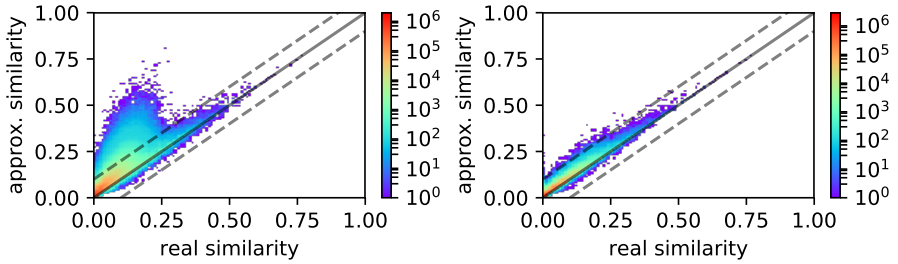
Figure 10 shows how the overall execution time and the quality of Brute Force and Hyrec evolve when we increase the size of the SHFs. (LSH presents a similar behavior to that of Brute Force, and NNDescent to that of Hyrec.)

As expected, larger SHFs cause Brute Force to take longer to compute, while delivering a better KNN quality (Fig. 10a). The overall computation time does not exactly follow that of individual similarity computations (Figure 9a), as the



(a) Brute Force (b) Hyrec

Figure 10: Relation between the execution time and the quality in function of the size of SHF.



(a) 1024 bits (b) 4096 bits

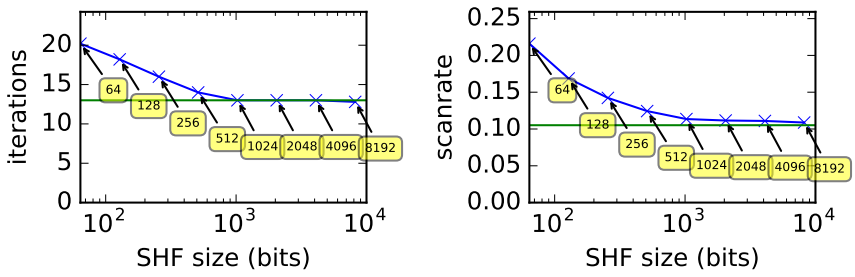
Figure 11: Heatmaps similarities on ml10M. The distortion of the similarity decreases when the size of SHF augments.

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 10b). 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.

5.3 Impact on estimated similarity values

Figure 11 shows how SHFs tend to distort estimated similarity values between pairs of users in ml10M, when using 1024 (Figure 11a) and 4096 bits (Figure 11b). The x-axis represents the real similarity of a pair of users $(u, v) \in U^2$, the y-axis represents its similarity obtained with GoldFinger, and the z-axis represents the number of pairs whose real similarity is x and estimated similar-



(a) Number of iterations (b) Scanrate

Figure 12: Effect of compression on the convergence of Hyrec on ml10M. GoldFinger converges to the native approach when the size of SHF augments.

ity y. (Note the log scale for z values.) The solid diagonal line is the identity function ($x = y$), while the two dashed lines 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 over $x = y$ indicate that the SHFs are over-approximating Jaccard's index, while points below correspond to an under-approximation. Figure 11 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 11a), the use of 1024 bits generate collisions that lead GoldFinger to overestimate low similarities (Figure 11b).

To understand why GoldFinger performs well despite this distortion, we analyze more carefully the distribution of pairs in Figure 11a. 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 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 user pairs can be found, 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 of the diagonal lower than Δ. With SHFs of size 1024, 52% of the pairs are at a distance lower than $\Delta = 0.01$, 75% for $\Delta = 0.02$, 94% for $\Delta = 0.05$ and 99% for $\Delta = 0.1$, which confirms the theoretical analysis of Section 2.4. This means that a large majority 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.

This explains why the Brute Force algorithm experiments a decrease in execution time along with a small drop in quality with GoldFinger. 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 NNDe-

scent's execution time first decreases as SHFs grow in size (as mentioned earlier, in Fig. 10).

To shed more light on this effect, Figure 12 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 (Fig. 12a), leading to a higher scanrate (Fig. 12b), and hence more similarity computations. When this occurs, the performance gain on individual similarity computations (Figure 9) does not compensate this higher scanrate, explaining the counter-intuitive pattern observed in Figure 10b.

6 Related work

For very small datasets, KNNs can be solved efficiently using specialized data structures [7,40,45]. For larger datasets, these solutions are more expensive than a brute force approach, and computing an exact KNN efficiently remains an open problem. Most practical approaches therefore compute an approximation of the KNN graph (ANN), as we do.

A first way to accelerate the computation time is to decrease the number of comparisons between users, taking the risk to miss some neighbors. *Recursive Lanczos Bisection* [16] for instance computes an ANN graph using a divide-and-conquer method. *NNDescent* [22] and *Hyrec* [8] rely on local search, i.e. they assume that a neighbor of a neighbor is likely to be a neighbor, and thus drastically decrease the scan rate, delivering substantial speedups. *KIFF* [9] uses the bipartite nature of the dataset to compute similarities only when users share an item. This approach works particularly well on sparse datasets but seems to have more difficulties with denser datasets such as the ones we studied. *Locality Sensitive Hashing* (LSH) [27] allows fast ANN graph computations by hashing users into buckets. The neighbors are selected only between the users of the same buckets. Several hash functions have been introduced, for different metrics [10,11,14]. All of the above works can be combined with our approach—as we have demonstrated in the case of *NNDescent*, *Hyrec*, and *LSH*—and are thus complementary to our contribution.

A second strategy to accelerate a KNN graph's construction consists in compacting users' profiles, in order to obtain a fast approximation of the similarity metric. For instance, a simple way the similarities computation is to limit the size of each user's profile by keeping the least popular items of each profiles [30]. The speed-up is interestering but lower than the one produced by GoldFinger. Instead of sampling based on popularity, *minwise hashing* [37] relies on the same principle as LSH to approximate Jaccard's index by only keeping a small subset of items for each user. It is space efficient but at the expense of a high preprocessing time. An extension of this work consists in hashing each item to 0 or 1 rather than storing its lowest bits [4]. This further reduces the memory footprint, but unfortunately maintains a prohibitive preprocessing time.

Similarly to our work, Bin, Heng *et al.* [19] use a bit array to represent profiles. Each feature, which has a continuous score comprised between 0 and 1, is rounded to either 0 or 1, and stored in one bit. This strategy unfortunately does not scale to large item sets (where items play the role of features), which are typical of the datasets we target. Closer to our work, Gorai *et al.* [24] use Bloom filters to encode the profiles of the nodes and then estimates Jaccard's index by using a bitwise `AND`. The use of Bloom filters preserves privacy, but the resulting loss in precision is unfortunately prohibitive. *Sketches* [13, 18, 48] are other compacted datastructures, which have been used for instance to find frequent items in data streams [3]. Unfortunately sketches are not optimized for set intersection.

To adapt the KNN graph construction to dynamic data, several metrics and recommendation systems have been proposed that take into account the timestamp of ratings [12, 15, 20, 21, 26, 33, 39, 54]. All these approaches remain however computationally intensive.

Privacy has today become a major concern for many information systems which aim to answer queries as accurately as possible while avoiding disclosing any sensible information. Multiple metrics with different semantics have been proposed to characterized privacy protection such as *k-anonymity* [50], *l-diversity* [42], *t-closeness* [36] or *differential privacy* [23]. An increasing number of works currently seek to add those properties to existing machine learning techniques [44, 52]. Blip [2] for instance provides differential privacy to users' profiles encoded into Bloom filters by injecting additional noise. By contrast, our approach naturally provides k-anonymity and l-diversity without any alteration of the bit array.

7 Conclusion

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 this idea on 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 randomized binary summary 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.

References

[1] Snips: The ai platform for voice-enabled devices. `https://snips.ai/`. last accessed April 19, 2017.

[2] M. Alaggan, S. Gambs, and A.-M. Kermarrec. Blip: Non-interactive differentially-private similarity computation on bloom filters. In *SSS*, 2012.

[3] E. Anceaume, Y. Busnel, N. Rivetti, and B. Sericola. Identifying global icebergs in distributed streams. In *SRDS*, 2015.

[4] Y. Bachrach and E. Porat. Sketching for big data recommender systems using fast pseudo-random fingerprints. In *ICALP*, 2013.

[5] X. Bai, M. Bertier, R. Guerraoui, A.-M. Kermarrec, and V. Leroy. Gossiping personalized queries. In *EDBT*, 2010.

[6] M. Bertier, D. Frey, R. Guerraoui, A.-M. Kermarrec, and V. Leroy. The gossple anonymous social network. In *Middleware*, 2010.

[7] A. Beygelzimer, S. Kakade, and J. Langford. Cover trees for nearest neighbor. In *ICML*, 2006.

[8] A. Boutet, D. Frey, R. Guerraoui, A.-M. Kermarrec, and R. Patra. Hyrec: leveraging browsers for scalable recommenders. In *Middleware*, 2014.

[9] A. Boutet, A.-M. Kermarrec, N. Mittal, and F. Taïani. Being prepared in a sparse world: the case of knn graph construction. In *ICDE*, 2016.

[10] A. Z. Broder. On the resemblance and containment of documents. In *Compression and Complexity of Sequences*, 1997.

[11] A. Z. Broder, S. C. Glassman, M. S. Manasse, and G. Zweig. Syntactic clustering of the web. *Comp. Networks and ISDN Sys.*, 1997.

[12] P. G. Campos, A. Bellogín, F. Díez, and J. E. Chavarriaga. Simple time-biased knn-based recommendations. In *CAMRa*, 2010.

[13] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. *Automata, languages and programming*, 2002.

[14] M. S. Charikar. Similarity estimation techniques from rounding algorithms. In *STOC*, 2002.

[15] C. Chen, H. Yin, J. Yao, and B. Cui. Terec: A temporal recommender system over tweet stream. *PVLDB*, 2013.

[16] J. Chen, H.-r. Fang, and Y. Saad. Fast approximate knn graph construction for high dimensional data via recursive lanczos bisection. *J. of ML Research*, 2009.

[17] E. Cho, S. A. Myers, and J. Leskovec. Friendship and mobility: User movement in location-based social networks. In *KDD*, 2011.

[18] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *J. of Algorithms*, 55(1), 2005.

[19] B. Cui, H. T. Shen, J. Shen, and K.-L. Tan. Exploring bit-difference for approximate knn search in high-dimensional databases. In *AusDM*, 2005.

[20] G. Damaskinos, R. Guerraoui, and R. Patra. Capturing the moment: Lightweight similarity computations. In *ICDE*, 2017.

[21] Y. Ding and X. Li. Time weight collaborative filtering. In *CIKM*, 2005.

[22] W. Dong, C. Moses, and K. Li. Efficient k-nearest neighbor graph construction for generic similarity measures. In *WWW*, 2011.

[23] C. Dwork. Differential privacy: A survey of results. In *TAMC*, 2008.

[24] M. Gorai, K. Sridharan, T. Aditya, R. Mukkamala, and S. Nukavarapu. Employing bloom filters for privacy preserving distributed collaborative knn classification. In *WICT*, 2011.

[25] F. M. Harper and J. A. Konstan. The movielens datasets: History and context. *ACM Trans. Interact. Intell. Syst.*, 2015.

[26] Y. Huang, B. Cui, W. Zhang, J. Jiang, and Y. Xu. Tencentrec: Real-time stream recommendation in practice. In *SIGMOD*, 2015.

[27] P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *STOC*, 1998.

[28] B. Jenkins. Hash functions. *Dr Dobbs Journal*, 1997.

[29] K. Keeton, C. B. Morrey III, C. A. Soules, and A. Veitch. Lazybase: freshness vs. performance in information management. *SIGOPS Op. Sys. Review*, 2010.

[30] A. Kermarrec, O. Ruas, and F. Taïani. Nobody cares if you liked star wars: KNN graph construction on the cheap. In *Euro-Par*, 2018.

[31] A. Labrinidis and N. Roussopoulos. Exploring the tradeoff between performance and data freshness in database-driven web servers. *The VLDB Journal*, 2004.

[32] X. N. Lam, T. Vu, T. D. Le, and A. D. Duong. Addressing cold-start problem in recommendation systems. In *IMCOM*, 2008.

[33] N. Lathia, S. Hailes, and L. Capra. Temporal collaborative filtering with adaptive neighbourhoods. In *SIGIR*, 2009.

[34] J. J. Levandoski, M. Sarwat, A. Eldawy, and M. F. Mokbel. Lars: A location-aware recommender system. In *ICDE*, 2012.

[35] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling distributed machine learning with the parameter server. In *OSDI*, 2014.

[36] N. Li, T. Li, and S. Venkatasubramanian. t-closeness: Privacy beyond k-anonymity and l-diversity. In *ICDE*, 2007.

[37] P. Li and A. C. König. Theory and applications of b-bit minwise hashing. *CACM*, 2011.

[38] G. Linden, B. Smith, and J. York. Amazon. com recommendations: Item-to-item collaborative filtering. *Internet Comp.*, 2003.

[39] N. N. Liu, M. Zhao, E. Xiang, and Q. Yang. Online evolutionary collaborative filtering. In *RecSys*, 2010.

[40] T. Liu, A. W. Moore, K. Yang, and A. G. Gray. An investigation of practical approximate nearest neighbor algorithms. In *Advances in neural information processing systems*, 2004.

[41] A. Machanavajjhala, J. Gehrke, D. Kifer, and M. Venkitasubramaniam. L-diversity: privacy beyond k-anonymity. In *ICDE*, 2006.

[42] A. Machanavajjhala, J. Gehrke, D. Kifer, and M. Venkitasubramaniam. l-diversity: Privacy beyond k-anonymity. In *ICDE*, 2006.

[43] J. J. McAuley and J. Leskovec. From amateurs to connoisseurs: modeling the evolution of user expertise through online reviews. In *WWW*, 2013.

[44] F. McSherry and I. Mironov. Differentially private recommender systems: building privacy into the net. In *KDD*, 2009.

[45] A. W. Moore. The anchors hierarchy: Using the triangle inequality to survive high dimensional data. In *UAI*, 2000.

[46] 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*, abs/1402.7063, 2014.

[47] P. Resnick, N. Iacovou, M. Suchak, P. Bergstrom, and J. Riedl. Grouplens: an open architecture for collaborative filtering of netnews. In *CSCW*, 1994.

[48] P. Roy, A. Khan, and G. Alonso. Augmented sketch: Faster and more accurate stream processing. In *SIGMOD*, 2016.

[49] P. Samarati. Protecting respondents identities in microdata release. *IEEE Trans. on Knowledge and Data Eng.*, 2001.

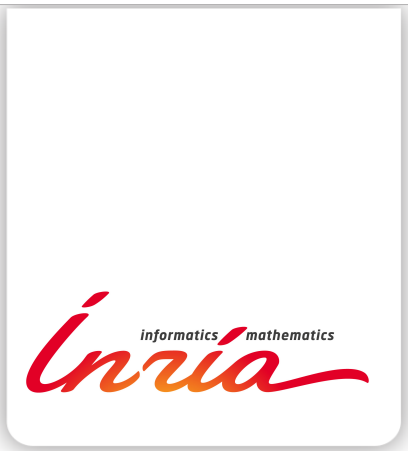
[50] L. Sweeney. k-anonymity: A model for protecting privacy. *International J. of Uncertainty, Fuzziness and Knowledge-Based Sys.*, 2002.

[51] C. J. van Rijsbergen. *Information retrieval*. Butterworth, 1979.

[52] K. Vu, R. Zheng, and J. Gao. Efficient algorithms for k-anonymous location privacy in participatory sensing. In *INFOCOM*, 2012.

[53] J. Yang and J. Leskovec. Defining and evaluating network communities based on ground-truth. *CoRR*, abs/1205.6233, 2012.

[54] X. Yang, Z. Zhang, and K. Wang. Scalable collaborative filtering using incremental update and local link prediction. In *CIKM*, 2012.



RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399