Fingerprinting Big Data: The Case of KNN Graph Construction

Rachid Guerraoui[1], Anne-Marie Kermarrec[1,2], Olivier Ruas[3], François Taïani[3] *
[1]*EPFL (Switzerland),* [2]*Mediego (France),* [3]*Inria, Univ Rennes, CNRS, IRISA (France)*
Rachid.Guerraoui@EPFL.ch, amk@mediego.com, olivier.ruas@inria.fr, francois.taiani@irisa.fr

***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.**

***Index Terms*—KNN graphs, fingerprint, similarity**

I. Introduction

a) K-Nearest-Neighbor (KNN) graphs: K-Nearest-Neighbor (KNN) graphs[1] play a fundamental role in many big data applications, including search [2], [3], recommendation [5], [19], [21] and classification [25]. 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 [12].

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.

Computing an *exact* KNN graph rapidly becomes intractable on large datasets: under a brute force strategy, a dataset with a few thousands of nodes requires tens of billions of similarity computations. Many applications, however, only require a good approximation of the KNN graph [15], [17]. Recent KNN construction algorithms [5], [10] have therefore sought to reduce the number of similarity computations by exploiting a *greedy strategy*. These techniques, among the most efficient to date, seem, however, to have reached their limits.

b) Fingerprinting Big Data for space and speed: In this paper, rather than reducing an algorithm's complexity we propose to avoid the *extensive* 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.

More precisely, 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, 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's index, at close to no overhead.
In the following, we first present the context of our work and our approach (Sec. II). We then present our evaluation procedure (Sec. III) and our results (Sec. III-D) before discussing related work (Sec. IV), and concluding (Sec. V).

II. 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.

A. 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 is generally much smaller than I (the universe of all items).

Our aim is to approximate a KNN graph G_{KNN} over U relying on some function 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. We use Jaccard's index in the rest of the paper [27].

*Authors are listed in alphabetical order.
[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.

2375-026X/19/$31.00 ©2019 IEEE
DOI 10.1109/ICDE.2019.00186

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 $sim(u, -)$:

$$knn(u) \in \underset{S \subseteq U \setminus \{u\}: |S|=k}{\text{argmax}} \sum_{v \in S} sim(u, v), \tag{1}$$

Computing an exact KNN graph is particularly expensive: an 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 [5], [10].

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

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

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.

B. 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 [6]). This is because computing explicit similarity values on even medium-size profiles can be relatively expensive while computing a similarity such as Jaccard's index: $J(P_1, P_2) = \frac{|P_1 \cap P_2|}{|P_1 \cup P_2|}$. 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.

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

C. 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. 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 I 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.

The computation incurred by (4) is much faster than on explicit profiles, and is independent of the actual size of the explicit profiles. 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, while Bloom filters are designed to test whether individual elements belong to a set, SHFs are designed to approximate set similarities. Bloom filters often employ multiple hash functions to minimize false positives. Those increase single-bit collisions, and degrade the approximation provided by SHFs.

TABLE I
DESCRIPTION OF THE DATASETS USED IN OUR EXPERIMENTS

Dataset	Users	Items	Ratings > 3	$\lvert P_u \rvert$
movielens10M [12]	69,816	10,472	5,885,448	84.30
movielens20M [12]	138,362	22,884	12,195,566	88.14
AmazonMovies [23]	57,430	171,356	3,263,050	56,82

III. Experimental Setup

A. Datasets

We evaluate GoldFinger using two publicly available datasets (Table I). We binarize each dataset by only keeping in a user profile P_u those items that user u has rated higher than 3.

a) Movielens: Movielens [12] is a group of anonymous datasets containing movie ratings collected on-line between 1995 and 2015 by GroupLens Research [26]. The datasets (before binarization) contain movie ratings on a 0.5-5 scale by users who have at least performed 20 ratings. We use 2 versions of the dataset, movielens10M (ml10M) and movielens20M (ml20M).

b) AmazonMovies: AmazonMovies [23] (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 [18]).

B. Baseline algorithms and competitors

We apply GoldFinger to four existing KNN algorithms: Brute Force (as a reference point), NNDescent [10], Hyrec [5] and LSH [13]. We compare the performance and results of each of these algorithms in their native form (*native* for short) and when accelerated with GoldFinger.

a) Brute force: The Brute Force algorithm simply computes the similarities between every pair of profiles. While this is computationally intensive, this algorithm produces an exact KNN graph.

b) NNDescent: NNDescent [10] 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 and in particular it 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.

c) Hyrec: Hyrec [5] uses a strategy similar to that of NNDescent, exploiting the fact that a neighbor of a neighbor is likely to be a neighbor. 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.

d) LSH: Locality-Sensitive-Hashing (LSH) [13] 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 [7].

C. Experimental settings

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 [14].

TABLE II
COMPUTATION TIME AND KNN QUALITY WITH NATIVE ALGORITHMS (*nat.*) AND GOLDFINGER (GOLFI).

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

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.

a) 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. II-A).

b) Implementation details and hardware: Our experiments use Java 1.8. and 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. Unless stated otherwise, we use all 8 threads. Our code is available online[3].

D. Evaluation Results

The performance of GoldFinger (*GolFi*) in terms of execution time and KNN quality is summarized in Table II. The columns marked *nat.* denote the native algorithms. The columns in italics show the gain in computation time brought by GoldFinger (*gain %*), and the loss in quality (*loss*). The fastest times are shown in bold.

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, GoldFinger is able to reduce computation time substantially, from 58.3% to 78.9%, corresponding to speedups of 2.39 and 4.74 respectively.

GoldFinger only has a limited effect on the execution time of LSH on AmazonMovies because LSH must first create user buckets using permutations on the item universe, an operation that is proportional to the number of items. Because AmazonMovies is comparatively very sparse the overall computation time that is dominated by the bucket creation time.

IV. RELATED WORK

For small datasets, KNNs can be solved efficiently using specialized data structures [4], [22], [24]. These solutions do

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

not scale, 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* [8] computes an ANN graph using a divide-and-conquer method, while *NNDescent* [10] and *Hyrec* [5] rely on local search, and thus drastically decrease the scan rate. *KIFF* [6] computes similarities only when users share an item. KIFF works particularly well on sparse datasets but has more difficulties with denser datasets such as the ones we studied. *Locality Sensitive Hashing* (LSH) [13] allows fast ANN graph computations by hashing users into buckets. The neighbors are selected only between the users of the same buckets. All of the above works can be combined with our approach 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. Keeping only a fraction of the profiles speeds up Jaccard computation [16] but the resulting approach is not as fast as GoldFinger. *Minwise hashing* [1], [20] approximates Jaccard's index by only keeping a small subset of items for each user. It is space efficient but has a prohibitive preprocessing time.

Bin, Heng *et al.* [9] use a bit array to represent profiles: each feature has its value rounded to either 0 or 1, and stored in one bit. Unfortunately, the approach is not scalable for the datasets we study. Closer to our work, Gorai *et al.* [11] use Bloom filters to encode the profiles and then estimate Jaccard's index by using a bitwise `AND`. Despite providing privacy, the resulting loss in precision is prohibitive.

V. 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.

Our preliminary 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.

Acknowledgments

This work was partially funded by the PAMELA project of the French National Research Agency (ANR-16-CE23-0016) and the Web-Alter-Ego Google Focused Award.

References

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

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

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

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

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

[6] 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.

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

[8] 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.

[9] 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.

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

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

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

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

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

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

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

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

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

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

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

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

[22] 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.

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

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

[25] 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.

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

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