DOI:10.1145/3543667

To view the accompanying Technical Perspective, visit doi.acm.org/10.1145/3543843

Sampling Near Neighbors in Search for Fairness

By Martin Aumüller, Sariel Har-Peled, Sepideh Mahabadi, Rasmus Pagh, and Francesco Silvestri

Abstract

Similarity search is a fundamental algorithmic primitive, widely used in many computer science disciplines. Given a set of points S and a radius parameter $r > 0$, the r-near neighbor (r-NN) problem asks for a data structure that, given any query point q, returns a point p within distance at most r from q. In this paper, we study the r-NN problem in the light of individual fairness and providing equal opportunities: all points that are within distance r from the query should have the same probability to be returned. The problem is of special interest in high dimensions, where *Locality Sensitive Hashing* (LSH), the theoretically leading approach to similarity search, does not provide any fairness guarantee. In this work, we show that LSH-based algorithms can be made fair, without a significant loss in efficiency. We propose several efficient data structures for the exact and approximate variants of the fair NN problem. Our approach works more generally for sampling uniformly from a subcollection of sets of a given collection and can be used in a few other applications. We also carried out an experimental evaluation that highlights the inherent unfairness of existing NN data structures.

1. INTRODUCTION

In recent years, following a growing concern about the fairness of algorithms and their bias toward a specific population or feature, there has been an increasing interest in building algorithms that achieve (appropriately defined) *fairness.*[14] The goal is to remove, or at least minimize, unethical behavior such as discrimination and bias in algorithmic decision making, as nowadays, many important decisions, such as college admissions, offering home loans, or estimating the likelihood of recidivism, rely on machine learning algorithms. While algorithms are not inherently biased, nevertheless, they may create it by careless design, or by amplifying the already existing biases in the data.

There is no unique definition of fairness (see Hardt et al.[18] and references therein), but different formulations that depend on the computational problem at hand, and on the ethical goals we aim for. Fairness goals are often defined in the political context of socio-technical systems and have to be seen in an interdisciplinary spectrum covering many fields outside computer science. In particular, researchers have studied both *group fairness* (also known as statistical fairness), where demographics of the population are preserved in the outcome,[12] and *individual fairness*, where the goal is to treat individuals with similar conditions similarly.[14] The latter concept of "equal opportunity" requires that people who can achieve a certain advantaged outcome, such as finishing a university degree, or paying back a loan, have an equal opportunity of being able to get access to it in the first place.

Bias in the data used for training machine learning algorithms is a monumental challenge in creating fair algorithms. Here, we are interested in a somewhat different problem of handling the bias introduced by the data structures used by such algorithms. Specifically, data structures may introduce bias in the data stored in them and the way they answer queries, because of the way a data is stored and how it is being accessed. It is also possible that some techniques for boosting performance, like randomization and approximation that result in nondeterministic behavior, add to the overall algorithmic bias. For instance, some database indexes for fast search might give an (unexpected) advantage to some portions of the input data. Such a defect leads to selection bias by the algorithms using such data structures. It is thus natural to want data structures that do not introduce a selection bias into the data when handling queries. To this end, imagine a data structure that can return, as an answer to a query, an item out of a set of acceptable answers. The purpose is then to return uniformly a random item out of the set of acceptable outcomes, without explicitly computing the whole set of acceptable answers (which might be prohibitively expensive).

The Near Neighbor Problem

In this work, we study similarity search and in particular the near neighbor problem from the perspective of individual fairness. Similarity search is an important primitive in many applications in computer science such as machine learning, recommender systems, data mining, computer vision, and many others (see e.g., Andoni and Indyk[5] for an overview). One of the most common formulations of similarity search is the r-near neighbor (r-NN) problem, formally defined as follows. Let $(\mathcal{X}, \mathcal{D})$ be a metric space where the distance function $\mathcal{D}(\cdot, \cdot)$ reflects the (dis)similarity between two data points. Given a set $S \subseteq \mathcal{X}$ of n points and a radius parameter r, the goal of the r-NN problem is to preprocess S and construct a data structure, such that for a query point $\mathbf{q} \in \mathcal{X}$, one can report a point $\mathbf{p} \in S$, such that $\mathcal{D}(\mathbf{p}, \mathbf{q}) \leq r$ if such a point exists. As all the existing algorithms for the *exact* variant of the problem have either space or query time that depends exponentially on the ambient dimension of $\mathcal{X}$, people have

A version of this paper, entitled "Fair Near Neighbor Search via Sampling," was published in *SIGMOD Record 50*, 1 (Mar. 2021).

considered the approximate variant of the problem. In the *c-approximate near neighbor* (ANN) problem, the algorithm is allowed to report a point $\mathbf{p}$ whose distance to the query is at most cr if a point within distance r of the query exists, for some prespecified constant $c > 1$.

Fair Near Neighbor

As detailed below, common existing data structures for similarity search have a behavior that introduces bias in the output. Our goal is to capture and algorithmically remove this bias from these data structures. Our goal is to develop a data structure for the r-near neighbor problem that provides fairness among "all the points" in the neighborhood. That is all the points within distance r from the given query have the same probability to be returned. We introduce and study the *fair near neighbor* problem: If $B_S(\mathbf{q}, r)$ is the ball of input points at distance at most r from a query $\mathbf{q}$, we would like that each point in $B_S(\mathbf{q}, r)$ is returned as near neighbor of $\mathbf{q}$ with the uniform probability of $1/n(\mathbf{q}, r)$ where $n(\mathbf{q}, r) = |B_S(\mathbf{q}, r)|$.

Locality Sensitive Hashing

Perhaps the most prominent approach to get an ANN data structure is *via* Locality Sensitive Hashing (LSH) as proposed by Indyk and Motwani,[20] which leads to sublinear query time and sub-quadratic space. In particular, for $\mathcal{X} = \mathbb{R}^d$, by using LSH one can get a query time of $n^{\rho+o(1)}$ and space $n^{1+\rho+o(1)}$ where for the L_1 distance metric $\rho = 1/c$,[16] and for the L_2 distance metric $\rho = 1/c^2+o_c(1)$.[5] In the LSH framework, the idea is to hash all points using several hash functions that are chosen randomly, with the property that the collision probability between two points is a decreasing function of their distance. Therefore, closer points to a query have a higher probability of falling into a bucket being probed than far points. Thus, reporting a random point from a random bucket computed for the query produces a distribution that is biased by the distance to the query: closer points to the query have a higher probability of being chosen. On the other hand, the uniformity property required in fair NN can be trivially achieved by finding *all* r-near neighbors of a query and then randomly selecting one of them. However, this is computationally inefficient since the query time is a function of the size of the neighborhood. One contribution in this paper is the description of much more efficient data structures that still use LSH in a black-box way.

When Random Nearby Is Better than Nearest

The bias mentioned above toward nearer points is usually a good property, but is not always desirable. Indeed, consider the following scenarios:

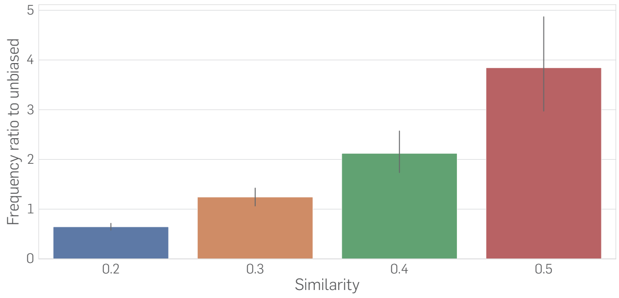
- The nearest neighbor might not be the best if the input is noisy, and the closest point might be viewed as an unrepresentative outlier. Any point in the neighborhood might be then considered to be equivalently beneficial. This is to some extent why k-NN classification[15] is so effective in reducing the effect of noise. Furthermore, k-NN works better in many cases if k is large, but computing the k nearest neighbors is quite expensive if k is large. Fortunately, quickly computing a random nearby neighbor can significantly speed up such classification.
- If one wants to estimate the number of items with a desired property within the neighborhood, then the easiest way to do it is *via* uniform random sampling from the neighborhood, for instance for density estimation[23] or discrimination discovery in existing databases.[27] This can be seen as a special case of query sampling in databases,[24] where the goal is to return a random sample of the output of a given query, for efficiently providing statistics on the query.
- We are interested in anonymizing the query: returning a random near-neighbor might serve as the first line of defense in trying to make it harder to recover the query. Similarly, one might want to anonymize the nearest neighbor,[25] for applications where we are interested in a "typical" data item close to the query, without identifying the nearest item.
- Popular recommender systems based on matrix factorization give recommendations by computing the inner product similarity of a user feature vector with all item feature vectors using some efficient similarity search algorithm. It is common practice to recommend those items that have the largest inner product with the user's vector. However, in general it is not clear that it is desirable to recommend the "closest" articles. Indeed, it might be desirable to recommend articles that are on the same topic but are not *too* aligned with the user feature vector and may provide a different perspective. As described in Adomavicius and Kwon,[1] recommendations can be made more diverse by sampling k items from a larger top-ℓ list of recommendations at random. Our data structures could replace the final near neighbor search routine employed in such systems.

To the best of our knowledge, previous results focused on exact near neighbor sampling in the Euclidean space up to three dimensions.[2, 19, 24] Although these results might be extended to $\mathbb{R}^d$ for any $d > 1$, they suffer from the *curse of dimensionality* as the query time increases exponentially with the dimension, making the data structures too expensive in moderately high dimensions. These bounds are unlikely to be significantly improved since several conditional lower bounds show that an exponential dependency on d in query time or space is unavoidable for *exact* near neighbor search.[4]

1.1. An example

Is a standard LSH approach really biased? As an example, we used the MinHash LSH scheme[10] to sample similar users from the Last.FM dataset used in the HetRec challenge (http://ir.ii.uam.es/hetrec2011). We associate each user with their top-20 artists and use Jaccard Similarity as similarity measure. We select one user at random as query, and repeatedly sample a random point from a random bucket and keep it if its similarity is above 0.2. Figure 1 reports on the ratio between the frequencies observed *via* this sampling approach from LSH buckets against an unbiased sample.

Figure 1. Bias introduced by uniform sampling from LSH buckets on the Last.FM dataset. The task is to (repeatedly) retrieve a uniform user among all users with similarity at least 0.2 to a fixed user. The result is split up into four buckets by rounding down the similarity to the first decimal. Error bars show the standard deviation. Compared to an unbiased sample, user vectors with small similarity are underrepresented, and users with high similarity are, by a factor of approximately 4 on average, overrepresented.



We see a large discrepancy: the higher the similarity, the more biased the LSH is in reporting these points as near neighbors. This would strongly affect statistics such as estimating the average similarity of a neighbor.

1.2. Problem formulations

Here, we formally define the variants of the fair NN problem that we consider. For all the constructions presented in this article, these guarantees fail with probability at most $\boldsymbol{\delta}$ for some prespecified small $\boldsymbol{\delta} > 0$.

Definition 1 (r-NNIS or Fair NN). *Let $S \subseteq \mathcal{X}$ be a set of n points in a metric space $(\mathcal{X}, \mathcal{D})$. The* r-near neighbor independent sampling *(r-NNIS), or simply the* Fair NN *problem, asks to construct a data structure for S that for any sequence of up to n queries $\mathbf{q}_1, \mathbf{q}_2, \ldots, \mathbf{q}_n$ satisfies the following properties with probability at least $1 - \boldsymbol{\delta}$:*

(I) For each query $\mathbf{q}_i$, it returns a point $\mathrm{OUT}_{i,\mathbf{q}_i}$ uniformly sampled from $B_S(\mathbf{q}_i, r)$.

(II) The point returned for query $\mathbf{q}_i$, with $i > 1$, is independent of previous query results. That is, for any $\mathbf{p} \in B_S(\mathbf{q}_i, r)$ and any sequence $\mathbf{p}_1, \ldots, \mathbf{p}_{i-1}$, we have $\Pr[\mathrm{OUT}_{i,\mathbf{q}_i} = \mathbf{p} \mid \forall_j \in [i-1]: \mathrm{OUT}_{j,\mathbf{q}_j} = \mathbf{p}_j] = 1/n(\mathbf{q}_i, r)$.

In the low-dimensional setting,[2,19] the r-near neighbor independent sampling problem is usually known as *independent range sampling* (IRS) problem. Next, motivated by applications, we define two approximate variants of the problem that we study in this work. More precisely, we slightly relax the fairness constraint, allowing the probabilities of reporting a neighbor to be an "almost uniform" distribution.

Definition 2 (Approximately Fair NN). *Consider a set $S \subseteq \mathcal{X}$ of n points in a metric space $(\mathcal{X}, \mathcal{D})$. The* Approximately Fair NN *problem asks to construct a data structure for S that for any query $\mathbf{q}$, returns each point $\mathbf{p} \in B_S(\mathbf{q}, r)$ with probability $\mu_\mathbf{p}$ where μ is an approximately uniform probability distribution:*

$$\mathbb{P}(\mathbf{q}, r)/(1+\varepsilon) \leq \mu_\mathbf{p} \leq (1+\varepsilon)\mathbb{P}(\mathbf{q}, r),$$

where $\mathbb{P}(\mathbf{q}, r) = 1/n(\mathbf{q}, r)$. We require the same independence guarantee as in Definition 1, that is, the result for query $\mathbf{q}_i$ must be independent of the results for $\mathbf{q}_1, \ldots, \mathbf{q}_{i-1}$, with $i \in \{2, \ldots, n\}$.

Furthermore, similar in spirit to the behavior of ANN, we allow the algorithm to report an almost uniform distribution from an *approximate* neighborhood of the query.

Definition 3 (Approximately Fair ANN). *Consider a set $S \subseteq \mathcal{X}$ of n points in a metric space $(\mathcal{X}, \mathcal{D})$. The* Approximately Fair ANN *problem asks to construct a data structure for S that for any query $\mathbf{q}$, returns each point $\mathbf{p} \in S'$ with probability $\mu_\mathbf{p}$ where $\varphi/(1+\varepsilon) \leq \mu_\mathbf{p} \leq (1+\varepsilon)\varphi$, where S' is a point set such that $B_S(\mathbf{q}, r) \subseteq S' \subseteq B_S(\mathbf{q}, cr)$, and $\varphi = 1/|S'|$. As before, the same independence guarantee as in Definition 1 is needed, that is, the result for query $\mathbf{q}_i$ must be independent of the results for $\mathbf{q}_1, \ldots, \mathbf{q}_{i-1}$, with $i \in \{2, \ldots, n\}$.*

1.3. Our results

We propose several solutions to the different variants of the Fair NN problem. Our solutions build upon the LSH data structure.[16] Let $\mathcal{S}(n, c)$ and $\mathcal{Q}(n, c)$ denote space and query time, respectively, of an LSH data structure that solves the c-ANN problem in the space $(\mathcal{X}, \mathcal{D})$.

- In Section 4.2, we provide a data structure for Approximately Fair ANN that uses space $\mathcal{S}(n, c)$ and whose query time is $\tilde{O}(\mathcal{Q}(n, c))$ *in expectation*. See Lemma 8 for the exact statement.
- Section 4.3 shows how to solve the Fair NN problem in expected query time $\tilde{O}(\mathcal{Q}(n,c) + \frac{n(\mathbf{q}, cr)}{n(\mathbf{q}, r)})$ and space usage $O(\mathcal{S}(n, c))$. See Lemma 9 for the exact statement.

The dependence of our algorithms on ε in the approximate variant is only $O(\log(1/\varepsilon))$. While we omitted the exact polylogarithmic factors in the list above, they are generally lower for the approximate versions. Furthermore, these methods can be embedded into existing LSH methods to achieve unbiased query results in a straightforward way. On the other hand, the exact methods will have higher logarithmic factors and use additional data structures.

A more exhaustive presentation of our results and further solutions for the Fair NN problem can be found in the full version of the paper.[8] Preliminary versions of our results were published independently in Har-Peled and Mahabadi,[17] Aumüller et al.[9] and then jointly in Aumüller et al.[7]

1.4. Sampling from a sub-collection of sets

In order to obtain our results, we first study a more generic problem in Section 2: Given a collection $\mathcal{F}$ of sets from a universe of n elements, a query is a sub-collection $\mathcal{G} \subseteq \mathcal{F}$ of these sets and the goal is to sample (almost) uniformly from the union of the sets in this sub-collection. We also show how to modify the data structure to handle outliers in Section 3. This is useful for LSH, as the sampling algorithm needs to ignore such points once they are reported as a sample. This setup allows us to derive most of the results concerning variants of Fair NN in Section 4 as corollaries from these more abstract data structures.

Some examples of applications of a data structure that provides uniform samples from a union of sets are as follows:

(A) Given a subset A of vertices in the graph, randomly pick (with uniform distribution) a neighbor to one of the vertices of A. This can be used in simulating disease spread.[22]

(B) As shown in this work, we use variants of the data structure to implement Fair NN.

(C) Uniform sampling for range searching.[19, 2] Indeed, consider a set of points, stored in a data structure for range queries. Using the above, we can support sampling from the points reported by several queries, even if the reported answers are not disjoint.

Being unaware of any previous work on this problem, we believe this data structure is of independent interest.

2. SAMPLING FROM A UNION OF SETS

The problem. Assume you are given a data structure that contains a large collection $\mathcal{F}$ of sets of objects. In total, there are $n = |\cup \mathcal{F}|$ objects. The sets in $\mathcal{F}$ are not necessarily disjoint. The task is to preprocess the data structure, such that given a sub-collection $\mathcal{G} \subseteq \mathcal{F}$ of the sets, one can quickly pick uniformly at random an object from the set $\cup \mathcal{G} := \cup_{A \in \mathcal{G}} A$.

Naive solution. The naive solution is to take the sets under consideration (in $\mathcal{G}$), compute their union, and sample directly from the union set $\cup \mathcal{G}$. Our purpose is to do (much) better—in particular, the goal is to get a query time that depends logarithmically on the total size of all the sets in $\mathcal{G}$.

Parameters. The query is a family $\mathcal{G} \subseteq \mathcal{F}$, and define $m = ||\mathcal{G}|| := \sum_{A \in \mathcal{G}} |A|$ (which should be distinguished from $g = |\mathcal{G}|$ and from $N = |\cup \mathcal{G}|$).

Preprocessing. For each set $A \in \mathcal{F}$, we build a set representation such that for a given element, we can decide if the element is in A in constant time. In addition, we assume that each set is stored in a data structure that enables easy random access or uniform sampling on this set (for example, store each set in its own array).

Variants. As in Section 1.2, we consider problem variants where sample probabilities are either *exact* or *approximate*.

2.1. Almost uniform sampling

The query is a family $\mathcal{G} \subseteq \mathcal{F}$. The ***degree*** of an element $x \in \cup \mathcal{G}$, is the number of sets of $\mathcal{G}$ that contain it—that is, $\mathsf{d}_{\mathcal{G}}(x) = |\mathsf{D}_{\mathcal{G}}(x)|$, where $\mathsf{D}_{\mathcal{G}}(x) = \{A \in \mathcal{G} \mid x \in A\}$. We start with an algorithm (similar to the algorithm of Section 4 in Karp and Luby[21]) that repeatedly does the following:

(I) Picks one set from $\mathcal{G}$ with probabilities proportional to their sizes. That is, a set $A \in \mathcal{G}$ is picked with probability $|A|/m$.

(II) It picks an element $x \in A$ uniformly at random.

(III) Outputs x and stops with probability $1/\mathsf{d}_{\mathcal{G}}(x)$. Otherwise, continues to the next iteration.

Since computing $\mathsf{d}_{\mathcal{G}}(x)$ exactly to be used in Step (III) is costly, our goal is instead to simulate a process that accepts x with probability *approximately* $1/\mathsf{d}_{\mathcal{G}}(x)$. We start with the process described in the following lemma.

LEMMA 1. *Assume we have g urns, and exactly $\mathsf{d} > 0$ of them, are non-empty. Furthermore, assume that we can check if a specific urn is empty in constant time. Then, there is a randomized algorithm, that outputs a number $Y \geq 0$, such that $\mathbb{E}[Y] = 1/\mathsf{d}$. The expected running time of the algorithm is $O(g/\mathsf{d})$.*

PROOF. The algorithm repeatedly probes urns (uniformly at random), until it finds a non-empty urn. Assume it found a non-empty urn in the i^{th} probe. The algorithm outputs the value i/g and stops.

Setting $p = \mathsf{d}/g$, and let Y be the output of the algorithm. We have that

$$\mathbb{E}[Y] = \sum_{i=1}^{\infty} \frac{i}{g}(1-p)^{i-1} p = \frac{p}{g(1-p)} \cdot \frac{1-p}{p^2} = \frac{1}{pg} = \frac{1}{\mathsf{d}},$$

using the formula $\sum_{i=1}^{\infty} i x^i = x/(1-x)^2$. The expected number of probes performed by the algorithm until it finds a non-empty urn is $1/p = g/\mathsf{d}$, which implies that the expected running time of the algorithm is $O(g/\mathsf{d})$. □

The natural way to deploy Lemma 1 is to run its algorithm to get a number y and then return 1 with probability y. The problem is that y can be strictly larger than 1, which is meaningless for probabilities. Instead, we back-off by using the value y/Δ, for some parameter Δ. If the returned value is larger than 1, we just treat it at zero. If the zeroing never happened, the algorithm would return one with probability $1/(\mathsf{d}_{\mathcal{G}}(x)\Delta)$. The probability of success is going to be slightly smaller, but fortunately, the loss can be made arbitrarily small by taking Δ to be sufficiently large.

LEMMA 2. *There are g urns, and exactly $\mathsf{d} > 0$ of them are not empty. Furthermore, assume one can check if a specific urn is empty in constant time. Let $\gamma \in (0, 1)$ be a parameter. Then one can output a number $Z \geq 0$, such that $Z \in [0, 1]$, and $\mathbb{E}[Z] \in I = \left[\frac{1}{\mathsf{d}\Delta} - \gamma, \frac{1}{\mathsf{d}\Delta}\right]$, where $\Delta = \lceil \ln \gamma^{-1} \rceil + 4 = \Theta(\log \gamma^{-1})$. The expected running time of the algorithm is $O(g/\mathsf{d})$. Alternatively, the algorithm can output a bit X, such that $\mathbb{P}[X = 1] \in I$.*

PROOF. We modify the algorithm of Lemma 1, so that it outputs $i/(g\Delta)$ instead of i/g. If the algorithm does not stop in the first $g\Delta + 1$ iterations, then the algorithm stops and outputs 0. Observe that the probability that the algorithm fails to stop in the first $g\Delta$ iterations, for $p = \mathsf{d}/g$, is $(1-p)^{g\Delta} \leq e^{-\frac{\mathsf{d}}{g} g\Delta} \leq e^{-\mathsf{d}\Delta} \leq e^{-\Delta} \ll \gamma$.

Let Z be the random variable that is the number output by the algorithm. Arguing as in Lemma 1, we have that $\mathbb{E}[Z] \leq 1/(\mathsf{d}\Delta)$. More precisely, we have

$$\mathbb{E}[Z] = \frac{1}{\mathsf{d}\Delta} - \sum_{i=g\Delta+1}^{\infty} \frac{i}{g\Delta}(1-p)^{i-1} p.$$

Easy calculations shows that

$$\sum_{i=gj+1}^{g(j+1)} \frac{i}{g}(1-p)^{i-1} p \leq (j+1)e^{-\mathsf{d}j}.$$

Let $g(j) = \frac{j+1}{\Delta} e^{-\mathsf{d}j}$. We have that $\mathbb{E}[Z] \geq \frac{1}{\mathsf{d}\Delta} - \beta$, where $\beta = \sum_{j=\Delta}^{\infty} g(j)$. Furthermore, for $j \geq \Delta$, we have

$$\frac{g(j+1)}{g(j)} = \frac{(j+2)e^{-\mathrm{d}(j+1)}}{(j+1)e^{-\mathrm{d}j}} \le \frac{5/4}{e^{\mathrm{d}}} \le \frac{1}{2}.$$

As such, we have that

$$\beta = \sum_{j=\Delta}^{\infty} g(j) \le 2g(\Delta) \le 2\frac{\Delta+1}{\Delta e^{\mathrm{d}\Delta}} \le 4e^{-\Delta} \le \gamma,$$

by the choice of value for Δ. This implies that $\mathbb{E}[Z] \geq 1/(\mathrm{d}\Delta) - \beta \geq 1/(\mathrm{d}\Delta) - \gamma$, as desired.

The alternative algorithm takes the output Z, and returns 1 with probability Z, and zero otherwise. □

Lemma 3. *The input is a family of sets $\mathcal{F}$ that one preprocesses in linear time. Let $\mathcal{G} \subseteq \mathcal{F}$ be a sub-family and let $N = |\cup\mathcal{G}|$, $g = |\mathcal{G}|$, and let $\varepsilon \in (0, 1)$ be a parameter. One can sample an element $x \in \cup\mathcal{G}$ with almost uniform probability distribution. Specifically, the probability p of an element to be output is $(1/N)/(1+\varepsilon) \le p \le (1+\varepsilon)(1/N)$. After linear time preprocessing, the query time is $O(g \log(g/\varepsilon))$, in expectation, and the query succeeds, with high probability (in g).*

Proof. The algorithm repeatedly samples an element x using steps (I) and (II). The algorithm returns x if the algorithm of Lemma 2, invoked with $\gamma = (\varepsilon/g)^{O(1)}$ returns 1. We have that $\Delta = \Theta(\log(g/\varepsilon))$. Let $\alpha = 1/(\mathrm{d}_{\mathcal{G}}(x)\Delta)$. The algorithm returns x in this iteration with probability p, where $p \in [\alpha - \gamma, \alpha]$. Observe that $\alpha \geq 1/(g\Delta)$, which implies that $\gamma \ll (\varepsilon/4)\alpha$, it follows that $(1/(\mathrm{d}_{\mathcal{G}}(x)\Delta))/(1 + \varepsilon) \le p \le (1 + \varepsilon)(1/(\mathrm{d}_{\mathcal{G}}(x)\Delta))$, as desired. The expected running time of each round is $O(g/\mathrm{d}_{\mathcal{G}}(x))$.

We prove the runtime analysis of the algorithm in the full version of the paper. In short, the above argument implies that each round, in expectation takes $O(Ng/m)$ time, where $m = ||\mathcal{G}||$. Further, the expected number of rounds, in expectation, will be $O(\Delta m/N)$. Finally, this implies that the expected running time of the algorithm is $O(g\Delta) = O(g \log(g/\varepsilon))$. □

Remark 1. *The query time of Lemma 3 can be made to work with high probability with an additional logarithmic factor. Specifically, with high probability, the query time is $O(g \log(g/\varepsilon) \log N)$.*

2.2. Uniform sampling

In this section, we present a data structure that samples an element uniformly at random from $\cup\mathcal{G}$. The data structure uses rejection sampling as seen before but splits up all data points using random ranks. Instead of picking an element from a weighted sample of the sets, it will pick a random segment among these ranks and consider only elements whose rank is in the selected range. Let Λ be the sequence of the $n = |\cup\mathcal{F}|$ input elements after a random permutation; the rank of an element is its position in Λ. We first highlight the main idea of the query procedure.

Let $k \geq 1$ be a suitable value that depends on the collection $\mathcal{G}$ and assume that Λ is split into k segments Λ_i, with $i \in \{0, ..., k - 1\}$. (We assume for simplicity that n and k are powers of two.) Each segment Λ_i contains the n/k elements in Λ with rank in $[i \cdot n/k, (i + 1) \cdot n/k)$. We denote with $\lambda_{\mathcal{G},i}$ the number of elements from $\cup\mathcal{G}$ in Λ_i, and with $\lambda \geq \max_i \lambda_{\mathcal{G},i}$ being an upper bound on the number of these elements in each segment. By the initial random permutation, we have that each segment contains at most $\lambda = \Theta((N/k) \log n)$ elements from $\cup\mathcal{G}$ with probability at least $1 - 1/n^2$. (Of course, N is *not* known at query time.)

The query algorithm works in the following steps in which all random choices are independent.

- (A) Set $k = n$, and let $\lambda = \Theta(\log n)$, $\sigma_{\text{fail}} = 0$ and $\sum = \Theta(\log^2 n)$.
- (B) Repeat the following steps until successful or $k < 2$:
    - (I) Assume the input sequence Λ to be split into k segments Λ_i of size n/k, where Λ_i contains the points in $\cup\mathcal{F}$ with ranks in $[i \cdot n/k, (i+1) \cdot n/k)$.
    - (II) Select an integer h in $\{0, ..., k - 1\}$ uniformly at random (i.e., select a segment Λ_h);
    - (III) Increment σ_{fail}. If $\sigma_{\text{fail}} = \sum$, then set $k = k/2$ and $\sigma_{\text{fail}} = 0$.
    - (IV) Compute $\lambda_{\mathcal{G},h}$ and with probability $\lambda_{\mathcal{G},h}/\lambda$, declare success.
- (C) If the previous loop ended with success, return an element uniformly sampled among the elements in $\cup\mathcal{G}$ in Λ_h, otherwise return $\perp$.

Since each object in $\cup\mathcal{G}$ has probability $1/(k\lambda)$ of being returned in Step (C), the result is a uniform sample of $\cup\mathcal{G}$. Note that the main iteration in Step (B) works for all values k, but a good choice has to depend on $\mathcal{G}$ for the following reasons. On the one hand, the segments should be small, because otherwise Step (IV) will take too long. On the other hand, they have to contain at least one element from $\cup\mathcal{G}$, otherwise we sample many "empty" segments in Step (II). We will see that the number k of segments should be roughly set to N to balance the trade-off. However, the number N of distinct elements in $\cup\mathcal{G}$ is not known. Thus, we use the naive upper bound of $k = n$. To compute $\lambda_{\mathcal{G},h}$ efficiently, we assume that, at construction time, the elements in each set in $\mathcal{F}$ are sorted by their rank.

Lemma 4. *Let $N = |\cup\mathcal{G}|$, $g = |\mathcal{G}|$, $m = \sum_{X\in\mathcal{G}} |X|$, and $n = |\cup\mathcal{F}|$. With probability at least $1 - 1/n^2$, the algorithm described above returns an element $x \in \cup\mathcal{G}$ according to the uniform distribution. With high probability, the algorithm has a running time of $O(g \log^4 n)$.*

Proof. We start by bounding the initial failure probability of the data structure. By a union bound, we have that the following two events hold simultaneously with probability at least $1 - 1/n^2$:

1. Every segment of size n/k contains no more than $\lambda = \Theta(\log n)$ elements from $\cup\mathcal{G}$ for all $k = 2^i$ where $i \in \{1, ..., \log n\}$. Since elements are initially randomly permuted, the claim holds with probability at least $1 - 1/(2n^2)$ by suitably setting the constant in $\lambda = \Theta(\log n)$.
2. It does not happen that the algorithm reports $\perp$. The probability of this event is upper bounded by the probability p' that no element is returned in the $\sum$ iterations where $k = 2^{\lceil \log N \rceil}$ (the actual probability is even lower, since an element can be returned in an iteration where $k > 2^{\lceil \log N \rceil}$). By suitably setting constants in

$\lambda = \Theta(\log n)$ and $\sum = \Theta(\log^2 n)$, we get:

$$p' = \left(1 - \frac{N}{k\lambda}\right)^{\Sigma} \leq e^{-\Sigma N/(k\lambda)} \leq e^{\Theta(-\Sigma/\log n)} \leq \frac{1}{2n^2}.$$

From now on assume that these events are true.

As noted earlier, each element has probability of $1/(k\lambda)$ of being returned as output, and thus, elements are equally likely to be sampled. Note also that the guarantees are independent of the initial random permutation as soon as the two events above hold. This means that the data structure returns a uniform sample from a union-of-sets.

For the running time, first focus on the round where $k = 2^{\lceil \log N \rceil}$. In this round, we carry out $\Theta(\log^2 n)$ iterations. In Step (IV), $\lambda_{\mathcal{G},h}$ is computed by iterating through the g sets and collecting points using a range query on segment Λ_h. Since elements in each set are sorted by their rank, the range query can be carried out by searching for rank hn/k using a binary search in $O(\log n)$ time and then enumerating all elements with rank smaller than $(h+1)n/k$. This takes time $O(\log n + o)$ for each set, where o is the output size. Since each segment contains $O(\log n)$ elements from $\cup\mathcal{G}$ with high probability, one iteration of Step (IV) takes time $O(g \log n)$.

The time to carry out all $\sum = \Theta(\log^2 n)$ iterations is thus bounded by $O(g \log^3 n)$. Observe that for all the rounds carried out before, k is only larger and thus, the segments are smaller. This means that we may multiply our upper bound with $\log n$, which completes the proof. □

Using count distinct sketches to find a good choice for the number of segments k, the running time can be decreased to $O(g \log^3 n)$; we refer to the full version[8] for more details.

3. HANDLING OUTLIERS

Imagine a situation where we have a marked set of outliers $\mathcal{O}$. We are interested in sampling from $\cup\mathcal{G}\backslash\mathcal{O}$. We assume that the total degree of the outliers in the query is at most $m_{\mathcal{O}}$ for some prespecified parameter $m_{\mathcal{O}}$. More precisely, we have $\mathsf{d}_{\mathcal{G}}(\mathcal{O}) = \sum_{x \in \mathcal{O}} \mathsf{d}_{\mathcal{G}}(x) \leq m_{\mathcal{O}}$. We get the following results by running the original algorithms from the previous section and removing outliers once we encounter them. If we encounter more than $m_{\mathcal{O}}$ outliers, we report that the number of outliers exceeds $m_{\mathcal{O}}$.

Running the algorithm described in Section 2.1 provides the guarantees summarized in the following lemma.

LEMMA 5. *The input is a family of sets $\mathcal{F}$ that one can preprocess in linear time. A query is a sub-family $\mathcal{G} \subseteq \mathcal{F}$, a set of outliers $\mathcal{O}$, a parameter $m_{\mathcal{O}}$, and a parameter $\varepsilon \in (0, 1)$. One can either:*

(A) Sample an element $x \in \cup\mathcal{G}\backslash\mathcal{O}$ with an ε-approximate uniform distribution: specifically, the probability μ_x of x to be output is $\varphi/(1+\epsilon) \leq \mu_x \leq (1+\epsilon)\varphi$, with $\varphi = 1/|\cup\mathcal{G}\backslash\mathcal{O}|$.

(B) Alternatively, report that $\mathsf{d}_{\mathcal{G}}(\mathcal{O}) > m_{\mathcal{O}}$.

The expected query time is $O(m_{\mathcal{O}} + g \log(n/\varepsilon))$, and the query succeeds with high probability, where $g = |\mathcal{G}|$, and $n = ||\mathcal{F}||$.

Running the algorithm described in Section 2.2 and keeping track of outliers has the following guarantees.

LEMMA 6. *The input is a family of sets $\mathcal{F}$ that one can preprocess in linear time. A query is a sub-family $\mathcal{G} \subseteq \mathcal{F}$, a set of outliers $\mathcal{O}$, and a parameter $m_{\mathcal{O}}$. With high probability, one can either:*

(A) Sample a uniform element $x \in \cup\mathcal{G}\backslash\mathcal{O}$, or

(B) Report that $\mathsf{d}_{\mathcal{G}}(\mathcal{O}) > m_{\mathcal{O}}$.

The expected query time is $O((g + m_{\mathcal{O}}) \log^4 n)$.

4. FINDING A FAIR NEAR NEIGHBOR

In this section, we employ the data structures developed in the previous sections to show the results on fair near neighbor search listed in Section 1.3.

First, let us briefly give some preliminaries on LSH. We refer the reader to Har-Peled et al.[16] for further details. Throughout the section, we assume our metric space $(\mathcal{X}, \mathcal{D})$ admits an LSH data structure.

4.1. Background on LSH

Locality Sensitive Hashing (LSH) is a common tool for solving the ANN problem and was introduced in Har-Peled et al.[16]

DEFINITION 4. *A distribution $\mathcal{H}$ over maps $h: \mathcal{X} \to U$, for a suitable set U, is $(r, c{\cdot}r, p_1, p_2)$*-sensitive *if the following holds for any $\mathbf{x}, \mathbf{y} \in \mathcal{X}$:*

- *if $\mathcal{D}(\mathbf{x}, \mathbf{y}) \leq r$, then $\Pr_h[h(\mathbf{x}) = h(\mathbf{y})] \geq p_1$;*
- *if $\mathcal{D}(\mathbf{x}, \mathbf{y}) > c \cdot r$, then $\Pr_h[h(\mathbf{x}) = h(\mathbf{y})] \leq p_2$.*

The distribution $\mathcal{H}$ is an LSH family, *and has quality $\rho = \rho(\mathcal{H}) = \frac{\log p_1}{\log p_2}$.*

For the sake of simplicity, we assume that $p_2 \leq 1/n$: if $p_2 > 1/n$, then it suffices to create a new LSH family $\mathcal{H}_K$ obtained by concatenating $K = \Theta(\log_{p_2}(1/n))$ independent and identically distributed hashing functions from $\mathcal{H}$. The new family $\mathcal{H}_K$ is (r, cr, p_1^K, p_2^K)-sensitive and ρ does not change.

The standard approach to (c, r)-ANN using LSH functions is the following. Let $\mathcal{D}$ denote the data structure constructed by LSH, and let c denote the approximation parameter of LSH. Each $\mathcal{D}$ consists of $\mathsf{L} = n^{\rho}$ hash functions $\ell_1, \ldots, \ell_{\mathsf{L}}$ randomly and uniformly selected from $\mathcal{H}$. $\mathcal{D}$ contains L hash tables $H_1, \ldots H_{\mathsf{L}}$: each hash table H_i contains the input set S and uses the hash function ℓ_i to split the point set into buckets. For each query $\mathbf{q}$, we iterate over the L hash tables: For any hash function, compute $\ell_i(\mathbf{q})$ and compute, using H_i, the set

$$H_i(\mathbf{p}) = \{\mathbf{p} : \mathbf{p} \in S, \ell_i(\mathbf{p}) = \ell_i(\mathbf{q})\} \tag{1}$$

of points in S with the same hash value; then, compute the distance $\mathcal{D}(\mathbf{q}, \mathbf{p})$ for each point $\mathbf{p} \in H_i(\mathbf{q})$. The procedure stops as soon as a (c, r)-near point is found. It stops and returns $\perp$ if there are no remaining points to check or if it found more than 3L far points. We summarize the guarantees in the following lemma.[16]

LEMMA 7. *For a given query point $\mathbf{q}$, let $S_{\mathrm{q}} = \cup_i H_i(\mathbf{q})$. Then for any point $\mathbf{p} \in B_S(\mathbf{q}, r)$, we have that with a probability of least $1 - 1/e - 1/3$, we have (i) $\mathbf{p} \in S_{\mathrm{q}}$ and (ii) $|S_{\mathrm{q}} \setminus B_S(\mathbf{q}, cr)| \leq 3\mathsf{L}$, that is, the number of outliers is at most 3L. Moreover, the expected number of outliers in any single bucket $S_{i,\ell i(\mathbf{q})}$ is at most 1.*

By repeating the construction $O(\log n)$ times, we guarantee that with high probability $B(\mathbf{q}, r) \subseteq S_{\mathbf{q}}$.

4.2. Approximately Fair ANN

For $t = O(\log n)$, let $\mathcal{D}_1, \ldots, \mathcal{D}_t$ be data structures constructed by LSH. Let $\mathcal{F}$ be the set of all buckets in all data structures, that is, $\mathcal{F} = \{H_i^j(\mathbf{p}) \mid i \leq \mathsf{L}, j \leq t, \mathbf{p} \in S\}$. For a query point $\mathbf{q}$, consider the family $\mathcal{G}$ of all buckets containing the query, that is, $\mathcal{G} = \{H_i^j(q) \mid i \leq L, j \leq t\}$, and thus $|\mathcal{G}| = O(\mathsf{L} \log n)$. Moreover, we let $\mathcal{O}$ to be the set of outliers, that is, the points that are farther than cr from q. Note that as mentioned in Lemma 7, the expected number of outliers in each bucket of LSH is at most 1. Therefore, by Lemma 5, we immediately get the following result.

LEMMA 8. *Given a set S of n points and a parameter r, we can preprocess it such that given query* $\mathbf{q}$, *one can report a point* $\mathbf{p} \in S$ *with probability* μ_p *where* $\varphi/(1 + \varepsilon) \leq \mu_{\mathbf{p}} \leq (1+\varepsilon)\varphi$, *S is a point set such that* $B_S(\mathbf{q}, r) \subseteq S \subseteq B_S(\mathbf{q}, cr)$, *and* $\varphi = 1/|S|$. *The algorithm uses space* $O(\mathsf{L} \log n)$ *and its expected query time is* $O(\mathsf{L} \log n \log(n/\varepsilon))$.

REMARK 2. *By repeatedly calling the query procedure and disregarding points at distance larger than r, the algorithm described above solves the Approximately Fair NN Problem (Definition 2). The probability that the algorithm succeeds in a round is* $\rho = n(\mathbf{q}, r)/n(\mathbf{q}, cr)$, *and as such the expected number of rounds is* $1/\rho$. *Thus, this approach has expected query time* $O\left(\mathsf{L}\frac{n(\mathbf{q},cr)}{n(\mathbf{q},r)}\log n \log(n/\varepsilon)\right)$.

4.3. Fair NN

We use the same setup as in the previous section and build $t = O(\log n)$ data structures $\mathcal{D}_1, \ldots, \mathcal{D}_t$ using LSH. We use the algorithm described in Section 2.2 with all points at distance more than r from the query marked as outliers. By the properties of the LSH and the random ranks, we expect to see $O\left(\left(\frac{n(\mathbf{q},cr)}{n(\mathbf{q},r)} + \mathsf{L}\right)\log n\right)$ points at distance at least r. This allows us to obtain the following results.

LEMMA 9. *Given a set S of n points and a parameter r, we can preprocess it such that given a query* $\mathbf{q}$, *one can report a point* $\mathbf{p} \in S$ *with probability* $1/n(\mathbf{q}, r)$. *The algorithm uses space* $O(\mathsf{L} \log n)$ *and has expected query time* $O\left(\left(\mathsf{L} + \frac{n(\mathbf{q},cr)}{n(\mathbf{q},r)}\right)\log^5 n\right)$.

5. EXPERIMENTAL EVALUATION

The example provided in Section 1.1 already showed the bias of sampling naively from the LSH buckets. In this section, we want to consider the influence of the approximative variants discussed here and provide a brief overview of the running time differences. A detailed experimental evaluation can be found in the full paper.[8]

For concreteness, we take the MNIST dataset of handwritten digits available at http://yann.lecun.com/exdb/mnist/. We use the Euclidean space LSH from Datar et al.,[13] set a distance threshold of 1250, and initialize the LSH with $\mathsf{L} = 100$ repetitions, $k = 15$, and $w = 3750$. These parameter settings provide a false negative rate of around 10%. We take 50 points as queries and test the following four different sampling strategies on the LSH buckets:

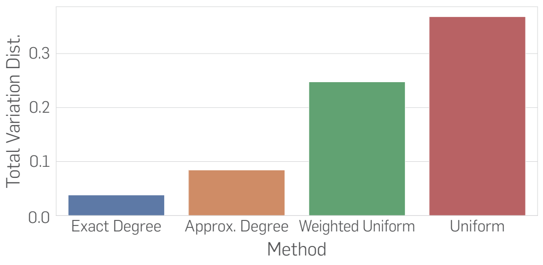
- **Uniform/uniform**: Picks bucket uniformly at random and picks a random point in bucket.
- **Weighted/uniform**: Picks bucket according to its size, and picks uniformly random point inside bucket.
- **Degree approximation**: Picks bucket according to size, and picks uniformly random point $\mathbf{p}$ inside bucket. It approximates $\mathbf{p}$'s degree (using Lemma 1) and rejects $\mathbf{p}$ with probability $1 - 1/\deg'(p)$. This is the approach discussed in Remark 2.
- **Optimal**: Picks bucket according to size, and picks uniformly random point $\mathbf{p}$ inside bucket. Then, it computes $\mathbf{p}$'s degree *exactly* and rejects $\mathbf{p}$ with probability $1 - 1/\deg(\mathbf{p})$. This is the approach discussed in Remark 2, but with exact degree approximation, solving Fair NN.

Each method removes non-close points that might be selected from the bucket. We remark that the variant Uniform/uniform most closely resembles a standard LSH approach. Weighted/Uniform takes the different bucket sizes into account, but disregards the individual frequency of a point. Thus, the output is *not expected* to be uniform, but might be closer in distribution to the uniform distribution.

Output Distribution. For each query $\mathbf{q}$, we compute the set of near neighbors $M(\mathbf{q})$ of $\mathbf{q}$ in the LSH buckets. For each sampling strategy, we carry out the query $100|M(\mathbf{q})|$ times. The sampling results give rise to a distribution μ on $M(\mathbf{q})$, and we compare this distribution to the uniform distribution in which each point is sampled with probability $1/|M(\mathbf{q})|$. Figure 2 reports on the total variation distance between the uniform distribution and the observed distribution, that is, $\frac{1}{2}\sum_{\mathbf{p}\in M(\mathbf{q})}\left|\mu(\mathbf{p}) - 1/|M(\mathbf{q})|\right|$. As in our introductory example, we see that uniformly picking an LSH bucket results in a heavily biased distribution. Taking the size of the buckets into account in the weighted case helps a bit, but still results in a heavily biased distribution. Even with the easiest approximation strategy for the degree, we see an improvement and achieve a total variation distance of around 0.08, with the optimal algorithm achieving around 0.04.

Differences in Running Time. Compared to a naïve approach of collecting all colliding points in the buckets and selecting a near neighbor at random, the methods presented here provide a speed-up of more than two orders of magnitude in our experiments. The fair methods based on rejection

Figure 2. Total variation distance of different approaches on the MNIST dataset.



sampling are approximately a factor of 10 slower than their biased counterparts that just pick a (weighted) point at random. Finally, the approximate degree sampling provides running times that are approximately two times faster than an exact computation of the degree.

6. CONCLUSION AND FUTURE WORK

In this paper, we have investigated a possible definition of fairness in similarity search by connecting the notion of "equal opportunity" to independent range sampling. An interesting open question is to investigate the applicability of our data structures for problems such as discrimination discovery,[27] diversity in recommender systems,[1] privacy preserving similarity search,[26] and estimation of kernel density.[11] Moreover, it would be interesting to investigate techniques for providing incentives (that is, reverse discrimination[27]) to prevent discrimination: An idea could be to merge the data structures in this paper with distance-sensitive hashing functions in Aumüller et al.,[6] which allow to implement hashing schemes where the collision probability is an (almost) arbitrary function of the distance. Finally, the techniques presented here require a manual trade-off between the performance of the LSH part and the additional running time contribution from finding the near points among the non-far points. From a user point of view, we would much rather prefer a parameterless version of our data structure that finds the best trade-off with small overhead, as discussed in Ahle et al.[3] in another setting.

Acknowledgments

S. Har-Peled was partially supported by a NSF AF award CCF-1907400. R. Pagh is part of BARC, supported by the VILLUM Foundation grant 16582. F. Silvestri was partially supported by PRIN Project n. 20174LF3T8 AHeAD.

References

1. Adomavicius, G., Kwon, Y. Optimization-based approaches for maximizing aggregate recommendation diversity. *INFORMS J. Comput 26*, 2 (2014), 351–369.
2. Afshani, P., Phillips, J.M. Independent range sampling, revisited again. In G. Barequet, and Y. Wang, eds. *Proc. 35th Int. Symposium on Computational Geometry (SoCG)*, volume 129 of *LIPIcs* (2019), 4:1–4:13.
3. Ahle, T.D., Aumüller, M., Pagh, R. Parameter-free locality sensitive hashing for spherical range reporting. In *Proc. 28th ACM-SIAM Symposium on Discrete Algorithms (SODA)* (2017), 239–256.
4. Alman, J., Williams, R. Probabilistic polynomials and hamming nearest neighbors. In *Proc. IEEE 56th Annual Symposium on Foundations of Computer Science (FOCS)* (2015), 136–150.
5. Andoni, A., Indyk, P. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM 51*, 1 (2008), 117–122.
6. Aumüller, M., Christiani, T., Pagh, R., Silvestri, F. Distance-sensitive hashing. In *Proc. 37th ACM Symposium on Principles of Database Systems (PODS)* (2018).
7. Aumüller, M., Har-Peled, S., Mahabadi, S., Pagh, R., Silvestri, F. Fair near neighbor search via sampling. *SIGMOD Rec 50*, 1 (2021), 42–49.
8. Aumüller, M., Har-Peled, S., Mahabadi, S., Pagh, R., Silvestri, F. Sampling a near neighbor in high dimensions—Who is the fairest of them all? to appear in ACM Transaction of Database Systems (2022.).
9. Aumüller, M., Pagh, R., Silvestri, F. Fair near neighbor search: Independent range sampling in high dimensions. In *Proc. 39th ACM Symposium on Principles of Database Systems (PODS)* (2020).
10. Broder, A.Z. On the resemblance and containment of documents. In *Proc. Compression and Complexity of Sequences* (1997), 21–29.
11. Charikar, M., Siminelakis, P. Hashing-based-estimators for kernel density in high dimensions. In C. Umans, ed. *Proc. 58th IEEE Symposium on Foundations of Computer Science (FOCS)* (2017), 1032–1043.
12. Chouldechova, A. Fair prediction with disparate impact: A study of bias in recidivism prediction instruments. *Big Data 5*, 2 (2017), 153–163.
13. Datar, M., Immorlica, N., Indyk, P., Mirrokni, V.S. Locality-sensitive hashing scheme based on p-stable distributions. In *Proc. 20th Symposium on Computational Geometry (SoCG)* (2004), 253–262.
14. Dwork, C., Hardt, M., Pitassi, T., Reingold, O., Zemel, R. Fairness through awareness. In *Proc. 3rd Innovations in Theoretical Computer Science Conference (ITCS)* (2012), 214–226.
15. Everitt, B.S., Landau, S., Leese, M. *Cluster Analysis*. Wiley Publishing, 2009.
16. Har-Peled, S., Indyk, P., Motwani, R. Approximate nearest neighbors: Towards removing the curse of dimensionality. *Theory Comput 8* (2012), 321–350. Special issue in honor of Rajeev Motwani.
17. Har-Peled, S., Mahabadi, S. Near neighbor: Who is the fairest of them all? In *Proc. 32nd Neural Info. Proc. Sys.* (NeurIPS) (2019), 13176–13187.
18. Hardt, M., Price, E., Srebro, N. Equality of opportunity in supervised learning. In *Neural Info. Proc. Sys.* (NIPS) (2016), 3315–3323.
19. Hu, X., Qiao, M., Tao, Y. Independent range sampling. In *Proc. 33rd ACM Symposium on Principles of Database Systems (PODS)* (2014), 246–255.
20. Indyk, P., Motwani, R. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proc. 30th Annu. ACM Sympos. Theory Comput*. (STOC) (1998), 604–613.
21. Karp, R.M., Luby, M. Monte-Carlo algorithms for enumeration and reliability problems. In *24th Symposium on Foundations of Computer Science (SFCS)*, IEEE Computer Society, 1983, 56–64.
22. Keeling, M.J., Eames, K.T. Networks and epidemic models. *J. R. Soc. Interface 2*, 4 (Sep. 2005), 295–307.
23. Kung Y-H, Lin, P.-S., Kao, C.-H. An optimal *k*-nearest neighbor for density estimation. *Stat. Probab. Lett 82*, 10 (2012), 1786–1791.
24. Olken, F., Rotem, D. Sampling from spatial databases. *Stat. Comput 5*, 1 (Mar 1995), 43–57.
25. Qi, Y., Atallah, M.J. Efficient privacy-preserving k-nearest neighbor search. In *Proc. 28th International Conference on Distributed Computing Systems (ICDCS)* (2008), 311–319.
26. Riazi, M.S., Chen, B., Shrivastava, A., Wallach, D.S., Koushanfar, F. Sub-linear privacy-preserving near-neighbor search with untrusted server on large-scale datasets. arXiv:1612.01835 (2016).
27. Thanh, B.L., Ruggieri, S., Turini, F. k-nn as an implementation of situation testing for discrimination discovery and prevention. In *Proc. 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)* (2011), 502–510.

Martin Aumüller (maau@itu.dk), IT University of Copenhagen, Denmark.

Sariel Har-Peled (sariel@illinois.edu), University of Illinois at Urbana-Champaign, IL, USA.

Sepideh Mahabadi (mahabadi@ttic.edu), Toyota Technological Institute at Chicago, IL, USA.

Rasmus Pagh (pagh@di.ku.dk), BARC and University of Copenhagen, Denmark.

Francesco Silvestri (silvestri@dei.unipd.it), University of Padova, Italy.