Distinct Sampling on Streaming Data with Near-Duplicates*

Jiecao Chen
Indiana University Bloomington
Bloomington, IN, USA
jiecchen@umail.iu.edu

Qin Zhang
Indiana University Bloomington
Bloomington, IN, USA
qzhangcs@indiana.edu

ABSTRACT

In this paper we study how to perform distinct sampling in the streaming model where data contain near-duplicates. The goal of distinct sampling is to return a distinct element uniformly at random from the universe of elements, given that all the near-duplicates are treated as the same element. We also extend the result to the sliding window cases in which we are only interested in the most recent items. We present algorithms with provable theoretical guarantees for datasets in the Euclidean space, and also verify their effectiveness via an extensive set of experiments.

1 INTRODUCTION

Real world datasets are always noisy; imprecise references to same real-world entities are ubiquitous in the business and scientific databases. For example, YouTube contains many videos of almost the same content; they appear to be slightly different due to cuts, compression and change of resolutions. A large number of webpages on the Internet are near-duplicates of each other. Numerous tweets and WhatsApp/WeChat messages are re-sent with small edits. This phenomenon makes data analytics more difficult. It is clear that direct statistical analysis on such noisy datasets will be erroneous. For instance, if we perform standard distinct sampling, then the sampling will be biased towards those elements that have a large number of near-duplicates.

On the other hand, due to the sheer size of the data it becomes infeasible to perform a comprehensive data cleaning step before the actual analytic phase. In this paper we study how to process datasets containing near-duplicates in the data stream model [4, 23], where we can only make a sequential scan of data items using a small memory space before the query-answering phase. When answering queries we need to treat all the near-duplicates as the same universe element.

This general problem has been recently proposed in [9], where the authors studied the estimation of the number of distinct elements of the data stream (also called F_0). In this paper we extend this line of research by studying another fundamental problem in the data stream literature: the *distinct sampling* (a.k.a. ℓ_0-sampling), where at the time of query we need to output a random sample among all the distinct elements of the dataset. ℓ_0-sampling has many applications that we shall mention shortly.

We remark, as also pointed out in [9], that we cannot place our hope on a *magic hash function* that can map all the near-duplicates into the same element and otherwise into different elements, simply because such a magic hash function, if exists, needs a lot of bits to describe.

The Noisy Data Model and Problems. Let us formally define the noisy data model and the problems we shall study. In this paper we will focus on points in the Euclidean space. More complicated data objects such as documents and images can be mapped to points in their feature spaces.

We first introduce a few concepts (first introduced in [9]) to facilitate our discussion. Let $d(\cdot,\cdot)$ be the distance function of the Euclidean space, and let α be a parameter (distance threshold) representing the maximum distance between any two points in the same group.

Definition 1.1 (data sparsity). We say a dataset S (α,β)-*sparse* in the Euclidean space for some $\beta \geq \alpha$ if for any $u, v \in S$ we have either $d(u,v) \leq \alpha$ or $d(u,v) > \beta$. We call $\max_\beta \beta/\alpha$ the *separation ratio*.

Definition 1.2 (well-separated dataset). We say a dataset S *well-separated* if the separation ratio of S is larger than 2.

Definition 1.3 (natural partition; F_0 of well-separated dataset). We can naturally partition a well-separated dataset S to a set of groups such that the intra-group distance is at most α, and the inter-group distance is more than 2α. We call this the unique *natural partition* of S. Define the number of distinct elements of a well-separated dataset w.r.t. α, denoted as $F_0(S,\alpha)$, to be the number of groups in the natural partition.

We will assume that α is given as a user-chosen input to our algorithms. In practice, α can be obtained for example by sampling a small number of items of the dataset and then comparing their labels.

For a general dataset, we need to define the number of distinct elements as an optimization problem as follows.

Definition 1.4 (F_0 of general dataset). The number of distinct elements of S given a distance threshold α, denoted by $F_0(S,\alpha)$, is defined to be the size of the *minimum* cardinality partition $\mathcal{G} = \{G_1, G_2, \ldots, G_n\}$ of S such that for any $i = 1, \ldots, n$, and for any pair of points $u, v \in G_i$, we have $d(u,v) \leq \alpha$.

Note that the definition for general datasets is consistent with the one for well-separated datasets.

We next define ℓ_0-sampling for noisy datasets. To differentiate with the standard ℓ_0-sampling we will call it *robust* ℓ_0-sampling; but

*Both authors are supported by NSF CCF-1525024 and IIS-1633215.

PODS'18, June 10–15, 2018, Houston, TX, USA

ACM ISBN 978-1-4503-4706-8/18/06…$15.00
https://doi.org/10.1145/3196959.3196978

we may omit the word "robust" in the rest of the paper when it is clear from the context. We start with well-separated datasets.

Definition 1.5 (robust ℓ_0-sampling on well-separated dataset). Let S be a well-separated dataset with natural partition $\mathcal{G} = \{G_1, G_2, \ldots, G_n\}$. The robust ℓ_0-sampling on S outputs a point $u \in S$ such that

$$\forall i \in [n], \mathbf{Pr}[u \in G_i] = 1/n. \tag{1}$$

That is, we output a point from each group with equal probability; we call the outputted point the *robust ℓ_0-sample*.

It is a little more subtle to define robust ℓ_0-sampling on general datasets, since there could be multiple minimum cardinality partitions, and without fixing a particular partition we cannot define ℓ_0-sampling. We will circumvent this issue by targeting a slightly weaker sampling goal.

Definition 1.6 (robust ℓ_0-sampling on general dataset). Let S be a dataset and let $n = F_0(S, \alpha)$. The robust ℓ_0-sampling on S outputs a point q such that,

$$\forall p \in S, \mathbf{Pr}[q \in \texttt{Ball}(p, \alpha) \cap S] = \Theta(1/n), \tag{2}$$

where $\texttt{Ball}(p, \alpha)$ is the ball centered at p with radius α.

Let us compare Equation (1) and (2). It is easy to see that when S is well-separated, letting $G(p)$ denote the group that p belongs to in the natural partition of S, we have

$$G(p) = \texttt{Ball}(p, \alpha) \cap S,$$

and thus we can rewrite (1) as

$$\forall p \in S, \mathbf{Pr}[q \in \texttt{Ball}(p, \alpha) \cap S] = 1/n. \tag{3}$$

Comparing (2) and (3), one can see that the definition of robust ℓ_0-sampling on general dataset is consistent with that on well-separated dataset, except that we have relaxed the sample probability by a constant factor.

Computational Models. We study robust ℓ_0-sampling in the standard streaming model, where the points $p_1, \ldots, p_m \in S$ comes one by one in order, and we need maintain a sketch of $S_t = \{p_1, \ldots, p_t\}$ (denoted by $sk(S_t)$) such that at any time t we can output an ℓ_0-sample of S_t using $sk(S_t)$. The goal is to minimize the size of sketch $sk(S_t)$ (or, the memory space usage) and the processing time per point under certain accuracy/approximation guarantees.

We also study the *sliding window* models. Let w be the window size. In the *sequence-based* sliding window model, at any time step t we should be able to output an ℓ_0-sample of $\{p_{\ell-w+1}, \ldots, p_\ell\}$ where p_ℓ is the latest point that we receive by the time t. In the *time-based* sliding window model, we should be able to output an ℓ_0-sample of $\{p_{\ell'}, \ldots, p_\ell\}$ where $p_{\ell'}, \ldots, p_\ell$ are points received in the last w time steps $t-w+1, \ldots, t$. The sliding window models are generalizations of the standard streaming model (which we call the *infinite window* model), and are very useful in the case that we are only interested in the most recent items. Our algorithms for sliding windows will work for both sequence-based and time-based cases. The only difference is that the definitions of the expiration of a point are different in the two cases.

Our Contributions. This paper makes the following theoretical contributions.

(1) We propose a robust ℓ_0-sampling algorithm for well-separated datasets in the streaming model in constant dimensional Euclidean spaces; the algorithm uses $O(\log m)$ words of space (m is the length of the stream) and $O(\log m)$ processing time per point, and successes with probability $(1 - 1/m)$ during the whole streaming process. This result matches the one in the corresponding noiseless data setting. See Section 2.1
(2) We next design an algorithm for sliding windows under the same setting. The algorithm works for both sequence-based and time-based sliding windows, using $O(\log n \log w)$ words of space and $O(\log n \log w)$ processing time per point with success probability $(1 - 1/m)$ during the whole streaming process. We comment that the sliding window algorithm is much more complicated than the one for the infinite window, and is our main technical contribution. See Section 2.2.
(3) For general datasets, we manage to show that the proposed ℓ_0-sampling algorithms for well-separated datasets still produce almost uniform samples on general datasets. More precisely, it achieves the guarantee (2). See Section 3.
(4) We further show that our algorithms can also handle datasets in high dimensional Euclidean spaces given sufficiently large separation ratios. See Section 4.
(5) Finally, we show that our ℓ_0-sampling algorithms can be used to efficiently estimate F_0 in both the standard streaming model and the sliding window models. See Section 5.

We have also implemented and tested our ℓ_0-sampling algorithm for the infinite window case, and verified its effectiveness on various datasets. See Section A.

Related Work. We now briefly survey related works on distinct sampling, and previous work dealing with datasets with near-duplicates.

The problem of ℓ_0-sampling is among the most well studied problems in the data stream literature. It was first investigated in [14, 24, 26], and the current best result is due to Jowhari et al. [28]. We refer readers to [13] for an overview of a number of ℓ_0-samplers under a unified framework. Besides being used in various statistical estimations [14], ℓ_0-sampling finds applications in dynamic geometric problems (e.g., ϵ-approximation, minimum spanning tree [24]), and dynamic graph streaming algorithms (e.g., connectivity [1], graph sparsifiers [2, 3], vertex cover [10, 11] maximum matching [1, 5, 10, 30], etc; see [32] for a survey). However, all the algorithms for ℓ_0-sampling proposed in the literature only work for noiseless streaming datasets.

ℓ_0-sampling in the sliding windows on noiseless datasets can be done by running the algorithm in [6] with the rank of each item being generated by a random hash function. As before, this approach cannot work for datasets with near-duplicates simply because the hash values assigned to near-duplicates will be different.

ℓ_0-sampling has also been studied in the distributed streaming setting [12] where there are multiple streams and we want to maintain a distinct sample over the union of the streams. The sampling algorithm in [12] is essentially an extension of the random sampling algorithms in [15, 34] by using a hash function to generate random ranks for items, and is thus again unsuitable for datasets with near-duplicates.

The list of works for F_0 estimation is even longer (e.g., [7, 19, 22, 23, 25, 29]; just mention a few). Estimating F_0 in the sliding window

model was studied in [37]. Again, all these works target noiseless data.

The general problem of processing noisy data streams *without* a comprehensive data cleaning step was only studied fairly recently [9] for the F_0 problem. A number of statistical problems (F_0, ℓ_0-sampling, heavy hitters, etc.) were studied in the distributed model under the same noisy data model [36]. Unfortunately the multi-round algorithms designed in the distributed model cannot be used in the data stream model because on data streams we can only scan the whole dataset once without looking back.

This line of research is closely related to *entity resolution* (also called *data deduplication*, *record linkage*, etc.); see, e.g., [17, 20, 27, 31]. However, all these works target finding and merging all the near-duplicates, and thus cannot be applied to the data stream model where we only have a small memory space and cannot store all the items.

Techniques Overview. The high level idea of our algorithm for the infinite window is very simple. Suppose we can modify the stream by only keeping one representative point (e.g., the first point according to the order of the data stream) of each group, then we can just perform a uniform random sampling on the representative points, which can be done for example by the following folklore algorithm: We assign each point with a random rank in $(0, 1)$, and maintain the point with the minimum rank as the sample during the streaming process. Now the question becomes:

> Can we identify (not necessarily store) the first point of each group space-efficiently?

Unfortunately, we will need to use $\Omega(n)$ space (n is the number of groups) to identify the first point of each group for a noisy streaming dataset, since we have to store at least 1 bit to "record" the first point of each group to avoid selecting other later-coming points of the same group. One way to deal with this challenge is to subsample a set of groups *in advance*, and then only focus on the first points of this set of groups. Two issues remain to be dealt with:

(1) How to sample a set of groups in advance?
(2) How to determine the sample rate?

Note that before we have seen all points in the group, the group itself is *not* well-defined, and thus it is difficult to assign an ID to a group at the beginning and perform the subsampling. Moreover, the number of groups will keep increasing as we see more points, we therefore have to decrease the sample rate along the way to keep the small space usage.

For the first question, the idea is to post a random grid of side length $\Theta(\alpha)$ (α is the group distance threshold) upon the point set, and then sample cells of the grid instead of groups using a hash function. We then say a group

(1) G is *sampled* if and only if G's *first* point falls into a sampled cell,
(2) G is *rejected* if G has a point in a sampled cell, however the G's first point is not in a sampled cell.
(3) G is *ignored* if G has no point in a sampled cell.

We note that the second item is critical since we want to judge a group only by its first point; even there is another point in the group that is sampled, if it is not the first point of the group, then we will still consider the group as rejected. On the other hand, we do not need to worry about those ignored groups since they are not considered at the very beginning.

To guarantee that our decision is consistent on each group we have to keep some *neighborhood* information on each rejected group as well to avoid "double-counting", which seems to be space-expensive at the first glance. Fortunately, for constant dimensional Euclidean space, we can show that if grid cells are randomly sampled, then the number of non-sampled groups is within a constant factor of that of sampled groups. We thus can control the space usage of the algorithm by dynamically decreasing the sample rate for grid cells. More precisely, we try to maintain a sample rate as low as possible while guarantee that there is at least one group that is sampled. This answers the second question.

The situation in the sliding window case becomes complicated because points will expire, and consequently we cannot keep decreasing the grid cell sample rate. In fact, we have to increase the cell sample rate when there are not enough groups being sampled. However, if we increase the cell sample rate in the middle of the process, then the neighborhood information of those previously ignored groups has already got lost. To handle this dilemma we choose to maintain a hierarchical sampling structure. We choose to describe the high level ideas as well as the actual algorithm in Section 2.2.2 after the some basic algorithms and concepts have been introduced.

For general datasets, we show that our algorithms for well-separated datasets can still return an almost uniform random distinct sample. We first relate our robust ℓ_0-sampling algorithm to a greedy partition process, and show that our algorithm will return a random group among the groups generated by that greedy partition. We then compare that particular greedy partition with the minimum cardinality partition, and show that the number of groups produced by the two partitions are within a constant factor of each other.

Comparison with [9]. We note that although this work follows the noisy data model of that in [9] and the roadmap of this paper is similar to that of [9] (which we think is the natural way for the presentation), the contents of this paper, namely, the ideas, proposed algorithms, and analysis techniques, are all very different from that in [9]. After all, the ℓ_0-sampling problem studied in this paper is different from the F_0 estimation studied in [9]. We note, however, that there are natural connections between distinct elements and distinct sampling, and thus would like to mention a few points.

(1) In the infinite window case, we can easily use our robust ℓ_0-sampling algorithm to get an algorithm for $(1+\epsilon)$-approximating robust F_0 using the same amount of space as that in [9] (see Section 5). We note that in the noiseless data setting, the problem of ℓ_0-sampling and F_0 estimation can be reduced to each other by easy reductions. However, it is not clear how to straightforwardly use F_0 estimation to perform ℓ_0-sampling in the noisy data setting using the same amount of space as we have achieved. We believe that since there is no magic hash function, similar procedure like finding the representative point of each group is necessary in any ℓ_0-sampling algorithm in the noisy data setting.
(2) Our sliding window ℓ_0-sampling algorithm can also be used to obtain a sliding window algorithm for $(1+\epsilon)$-approximating F_0 (also see Section 5). However, it is not clear how to extend

Notation	Definition
S	stream of points
m	length of the stream
w	length of the sliding window
$n = F_0(S)$	number of groups
$\mathcal{G}/G$	set of groups / a group
$G(p)$	group containing point p
α	threshold of group diameter
$\mathbb{G}/C$	grid / a grid cell
$\textsc{cell}(p)$	cell containing point p
$\textsc{adj}(p)$	set of cells adjacent to $\textsc{cell}(p)$
$\texttt{Ball}(p, \alpha)$	$\{q \mid d(p, q) \le \alpha\}$
ϵ	approximation ratio for F_0

Table 1: Notations

the F_0 algorithm in [9] to the sliding window case, which was not studied in [9].

(3) In order to deal with general datasets, in [9] the authors introduced a concept called F_0-ambiguity and used it as a parameter in the analysis. Intuitively, F_0-ambiguity measures the least fraction of points that we need to remove in order to make the dataset to be well-separated. This definition works for problems whose answer is a single number, which does *not* depend on the actual group partition. However, different group partitions do affect the result of ℓ_0-sampling, even that all those partitions have the minimum cardinality. In Section 3 we show that by introducing a relaxed version of random sampling we can bypass the issue of data ambiguity.

Preliminaries. In Table 1 we summarize the main notations used in this paper. We use $[n]$ to denote $\{1, 2, \ldots, n\}$.

We say x is $(1+\epsilon)$-approximation of y if $x \in [(1-\epsilon)y, (1+\epsilon)y]$.

We need the following versions of the Chernoff bound.

LEMMA 1.7 (STANDARD CHERNOFF BOUND). *Let $X_1, \ldots, X_n$ be independent Bernoulli random variables such that $\mathbf{Pr}[X_i = 1] = p_i$. Let $X = \sum_{i\in[n]} X_i$. Let $\mu = \mathbf{E}[X]$. It holds that $\mathbf{Pr}[X \ge (1+\delta)\mu] \le e^{-\delta^2\mu/3}$ and $\mathbf{Pr}[X \le (1-\delta)\mu] \le e^{-\delta^2\mu/2}$ for any $\delta \in (0, 1)$.*

LEMMA 1.8 (VARIANT OF CHERNOFF BOUND). *Let $Y_1, \ldots, Y_n$ be n independent random variables such that $Y_i \in [0, T]$ for some $T > 0$. Let $\mu = \mathbf{E}[\sum_i Y_i]$. Then for any $a > 0$, we have*

$$\mathbf{Pr}\left[\sum_{i\in[n]} Y_i > a\right] \le e^{-(a-2\mu)/T}.$$

2 WELL-SEPARATED DATASETS IN CONSTANT DIMENSIONS

We start with the discussion of ℓ_0-sampling on well-separated datasets in constant dimensional Euclidean space.

2.1 Infinite Window

We first consider the infinite window case. We present our algorithm for 2-dimensional Euclidean space, but it can be trivially extended to $O(1)$-dimensions by appropriately changing the constant parameters.

Let $\mathcal{G} = \{G_1, \ldots, G_n\}$ be the natural group partition of the well-separated stream of points S. We post a random grid $\mathbb{G}$ with side length $\frac{\alpha}{2}$ on $\mathbb{R}^2$, and call each grid square a *cell*. For a point p, define $\textsc{cell}(p)$ to be the cell $C \in \mathbb{G}$ containing p. Let

$$\textsc{adj}(p) = \{C \in \mathbb{G} \mid d(p, C) \le \alpha\},$$

where $d(p, C)$ is defined to be the minimum distance between p and a point in C. We say a group G *intersects* a cell C if $G \cap C \neq \emptyset$.

Assuming that all points have x and y coordinates in the range $[0, M]$ for a large enough value M. Let $\Delta = \frac{2M}{\alpha} + 1$. We assign the cell on the i-th row and the j-th column of the grid $\mathbb{G} \cap [0, M] \times [0, M]$ a numerical identification (ID) $((i-1)\cdot\Delta + j)$. For convenience we will use "cell" and its ID interchangeably throughout the paper when there is no confusion.

For ease of presentation, we will assume that we can use fully random hash functions for free. In fact, by Chernoff-Hoeffding bounds for limited independence [18, 33], all our analysis still holds when we adopt $\Theta(\log m)$-wise independent hash functions, using which will not affect the asymptotic space and time costs of our algorithms.

Let $h : [\Delta^2] \to \{0, 1, \ldots, 2^{\lceil 2\log\Delta\rceil} - 1\}$ be a fully random hash function, and define h_R for a given parameter $R = 2^k$ ($k \in \mathbb{N}$) to be $h_R(x) = h(x) \bmod R$. We will use h_R to perform sampling. In particular, given a set of IDs $Y = \{y_1, \ldots, y_t\}$, we call $\{y \in Y \mid h_R(y) = 0\}$ the set of sampled IDs of Y by h_R. We also call $1/R$ the *sample rate* of h_R.

As discussed in the techniques overview in the introduction, our main idea is to sample *cells* instead of groups in advance using a hash function.

Definition 2.1 (sampled cell). A cell C is *sampled* by h_R if and only if $h_R(C) = 0$.

By our choices of the grid cell size and the hash function we have:

FACT 1. *(a) Each cell will intersect at most one group, and each group will intersect at most $O(1)$ cells.*

(b) For any set of points $P = \{p_1, \ldots, p_t\}$,

$$\{p \in P \mid h_{2R}(\text{cell}(p)) = 0\} \subseteq \{p \in P \mid h_R(\text{cell}(p)) = 0\}.$$

In the infinite window case (this section) we choose the *representative* point of each group to be the *first* point of the group. We note that the representative points are fully determined by the input stream, and are independent of the hash function. We will define the representative point slightly differently in the sliding window case (next section).

We define a few sets which we will maintain in our algorithms.

Definition 2.2. Let S^{rep} be the set of representative points of all groups. Define the *accept set* to be

$$S^{\text{acc}} = \{p \in S^{\text{rep}} \mid h_R(\textsc{cell}(p)) = 0\},$$

and the *reject set* to be

$$S^{\text{rej}} = \{p \in S^{\text{rep}} \backslash S^{\text{acc}} \mid \exists C \in \textsc{adj}(p) \text{ s.t. } h_R(C) = 0\}.$$

For convenience we also introduce the following concepts.

Definition 2.3 (sampled, rejected, candidate group). We say a group G a *sampled* group if $G \cap S^{\text{acc}} \neq \emptyset$, a *rejected* group if $G \cap S^{\text{rej}} \neq \emptyset$, and a *candidate* group if $G \cap (S^{\text{acc}} \cup S^{\text{rej}}) \neq \emptyset$.

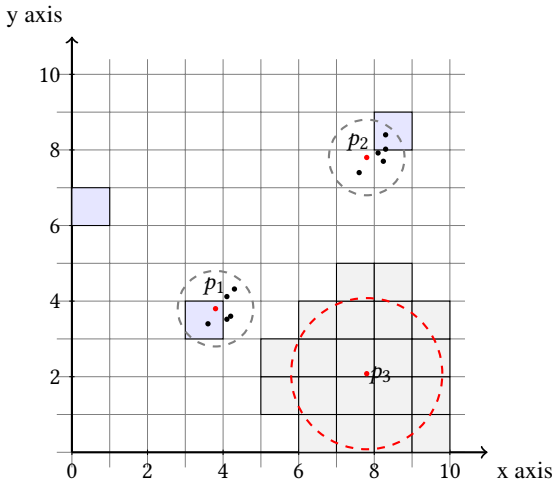

Figure 1: Each square is a cell; each light blue square is a sampled cell. Each gray dash circle stands for a group. Red points (p_1, p_2 and p_3) are representative points; p_1 is in the accept set and p_2 is in the reject set. Gray cells form $\text{ADJ}(p_3)$. $\alpha = 2$ in this illustration.

Figure 1 illustrates some of the concepts we have introduced so far.

Obviously, the set of sampled groups and the set of rejected groups are disjoint, and their union is the set of candidate groups. Also note that S^{acc} is the set of representative points of the sampled groups, and S^{rej} is the set of representative points of rejected groups.

We comment that it is important to keep the set S^{rej}, even that at the end we will only sample a point from S^{acc}. This is because otherwise we will have no information regarding the first points of those groups that may have points other than the first ones falling into a sampled cell, and consequently points in $S \backslash S^{\text{rep}}$ may also be included into S^{acc}, which will make the final sampling to be non-uniform among the groups. One may wonder whether this additional storage will cost too much space. Fortunately, since each group has diameter at most α, we only need to monitor groups that are at a distance of at most α away from sampled cells, whose cardinality can be shown to be small. More precisely, for a group G, letting p be its representative point, we monitor G if and only if there exists a sampled cell C such that $C \in \text{ADJ}(p)$. The set of representative points of such groups is precisely $S^{\text{acc}} \cup S^{\text{rej}}$.

Our algorithm for ℓ_0-sampling in the infinite window case is presented in Algorithm 1. We control the sample rate by doubling the range R of the hash function when the number of points of S^{acc} exceeds a threshold $\Theta(\log m)$ (Line 10 of Algorithm 1). We will also update S^{acc} and S^{rej} accordingly to maintain Definition 2.2.

When a new point p comes, if $\text{CELL}(p)$ is sampled and p is the first point in $G(p)$ (Line 6), we add p to S^{acc}; that is, we make p as the representative point of the sampled group $G(p)$. Otherwise if $\text{CELL}(p)$ is not sampled but there is at least one sampled cell in $\text{ADJ}(p)$, and p is the first point in $G(p)$ (Line 8), then we add p to S^{rej}; that is, we make p as the representative point of the rejected group $G(p)$.

Algorithm 1: ROBUST ℓ_0-SAMPLING-IW

```
1  R ← 1; S^acc ← ∅; S^rej ← ∅
2  κ_0 is chosen to be a large enough constant
   /* dataset is fed as a stream of points */
3  for each arriving point p do
       /* if p is not the first point of a
          candidate group, skip it          */
4      if ∃u ∈ S^acc ∪ S^rej s.t. d(u,p) ≤ α then
5          continue
       /* if p is the first point of a
          candidate group                   */
6      if h_R(cell(p)) = 0 then
7          S^acc ← S^acc ∪ {p}
8      else if ∃C ∈ adj(p) s.t. h_R(cell(C)) = 0 then
9          S^rej ← S^rej ∪ {p}
10     if |S^acc| > κ_0 log m then
11         R ← 2R
12         update S^acc and S^rej according to the updated hash
             function h_R
   /* at the time of query:               */
13 return a random point in S^acc
```

On the other hand, if there is at least one sampled cell in $\text{ADJ}(p)$ (i.e., $G(p)$ is a candidate group) and p is *not* the first point (Line 4), then we simply discard p. Note that we can test this since we have already stored the representation points of all candidate groups. In the remaining case in which $G(p)$ is not a candidate group, we also discard p.

At the time of query, we return a random point in S^{acc}.

Correctness and Complexity. We show the following theorem.

THEOREM 2.4. *In constant dimensional Euclidean space for a well-separated dataset, there exists a streaming algorithm (Algorithm 1) that with probability $1 - 1/m$, at any time step, it outputs a robust ℓ_0-sample. The algorithm uses $O(\log m)$ words of space and $O(\log m)$ processing time per point.*

The correctness of the algorithm is straightforward. First, S^{acc} is a random subset of S^{rep} since each point $p \in S^{\text{rep}}$ is included in S^{acc} if and only if $h_R(\text{CELL}(p)) = 0$. Second, the outputted point is a random point in S^{acc}. The only thing left to be shown is that we have $|S^{\text{acc}}| > 0$ at any time step.

LEMMA 2.5. *With probability $1 - 1/m$, we have $|S^{acc}| > 0$ throughout the execution of the algorithm.*

PROOF. At the first time step of the streaming process, p_1 is added into S^{acc} with certainty since R is initialized to be 1. Then S^{acc} keeps growing. At the moment when $|S^{\text{acc}}| > \kappa_0 \log m$, R is doubled so that each point in S^{acc} is resampled with probability $\frac{1}{2}$. After the resampling,

$$\Pr[|S^{\text{acc}}| = 0] \leq \left(\frac{1}{2}\right)^{\kappa_0 \log m} \leq \frac{1}{m^2}. \tag{4}$$

By a union bound over at most m resample steps, we conclude that with probability $1 - 1/m$, $|S^{\text{acc}}| > 0$ throughout the execution of the algorithm. □

We next analyze the space and time complexities of Algorithm 1.

LEMMA 2.6. *With probability* $(1-1/m)$ *we have* $S^{rej} = O(\log m)$ *throughout the execution of the algorithm.*

PROOF. Consider a fixed time step. Let $S = S^{\text{acc}} \cup S^{\text{rej}}$. For a fixed $p \in S^{\text{rep}}$, since $|\text{ADJ}(p)| \leq 25$ (we are in the 2-dimensional Euclidean space), and each cell is sampled randomly, we have

$$\mathbf{Pr}[p \in S^{\text{rej}}] \leq \frac{24}{25} \cdot \mathbf{Pr}[p \in S]. \tag{5}$$

We only need to prove the lemma for the case $\mathbf{Pr}[p \in S^{\text{rej}}] = \frac{24}{25} \cdot \mathbf{Pr}[p \in S]$; the case $\mathbf{Pr}[p \in S^{\text{rej}}] < \frac{24}{25} \cdot \mathbf{Pr}[p \in S]$ follows directly since p is less likely to be included in S^{rej}.

For each $p \in S$, define X_p to be a random variable such that $X_p = 1$ if $p \in S^{\text{rej}}$, and $X_p = 0$ otherwise. Let $X = \sum_{p \in S^{\text{rej}}} X_p$. We have $X = \left|S^{\text{rej}}\right|$ and $\mathbf{E}[X] = \frac{24}{25}|S|$. By a Chernoff bound (Lemma 1.7), we have

$$\mathbf{Pr}\left[X - \mathbf{E}[X] > 0.01\mathbf{E}[X]\right] \leq e^{-\frac{0.01^2 \cdot \mathbf{E}[X]}{3}}. \tag{6}$$

If $|S| \leq 80000 \log m$ then we immediately have $\left|S^{\text{rej}}\right| \leq |S| = O(\log m)$. Otherwise by (6) we have

$$\mathbf{Pr}[X > 1.01\mathbf{E}[X]] < 1/m^2.$$

We thus have

$$\begin{aligned} 1/m^2 &> \mathbf{Pr}[X > 1.01\mathbf{E}[X]] \\ &= \mathbf{Pr}[X > 1.01 \cdot \frac{24}{25}|S|] \\ &= \mathbf{Pr}[X > 0.9696(X + \left|S^{\text{acc}}\right|)] \\ &= \mathbf{Pr}[0.0304X > 0.9696\left|S^{\text{acc}}\right|]. \end{aligned}$$

According to Algorithm 1 it always holds that $|S^{\text{acc}}| = O(\log m)$. Therefore $\left|S^{\text{rej}}\right| = X = O(\log m)$ with probability at least $1 - 1/m^2$. The lemma follows by a union bound over m time steps of the streaming process. □

By Lemma 2.6 the space used by the algorithm can be bounded by $O(|S^{\text{acc}}| + \left|S^{\text{rej}}\right|) = O(\log m)$ words. The processing time per point is also bounded by $O(|S^{\text{acc}}| + \left|S^{\text{rej}}\right|)$.

2.2 Sliding Windows

We now consider the sliding window case. Let w be the window size. We first present an algorithm that maintains a set of sampled points in S^{acc} with a fixed sample rate $1/R$; it will be used as a subroutine in our final sliding window algorithm (Section 2.2.2).

2.2.1 A Sliding Window Algorithm with Fixed Sample Rate.

We describe the algorithm in Algorithm 2, and explain it in words below.

Besides maintaining the accept set and the reject set as that in the infinite window case, Algorithm 2 also maintains a set A consisting of key-value pairs (u, p), where u is the *representative* point of a candidate group (u can be a point outside the sliding window as long as the group has at least one point inside the sliding window), and p is the *latest* point of the same group (thus p must be in the sliding window). Define $A(S^{\text{acc}}) = \{p \mid \exists u \in S^{\text{acc}} \text{ s.t. } (u, p) \in A\}$.

Algorithm 2: SW WITH FIXED SAMPLE RATE $1/R$

```
1  for each expired point p do
2      if ∃(u, p) ∈ A then
3          delete (u, p) from A, delete u from S^acc ∪ S^rej
4  for each arriving point p do
       /* if there already exists a point of
          the same group in S^acc ∪ S^rej      */
5      if ∃u ∈ S^acc ∪ S^rej s.t. d(u, p) ≤ α then
6          A ← (u, p) ∪ A\(u, ·)
       /* otherwise we set p as a
          representative of its group          */
7      else if h_R(cell(p)) = 0 then
8          S^acc ← S^acc ∪ {p}, A ← A ∪ (p, p)
9      else if ∃C ∈ adj(p) s.t. h_R(C) = 0 then
10         S^rej ← S^rej ∪ {p}, A ← A ∪ (p, p)
```

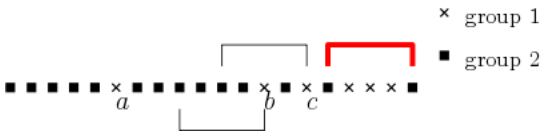

Figure 2: Representative points in sliding windows. There are two different groups, and the red window is the current sliding window (of size $w = 5$). Point c is *not* the representative point of Group 1 because the window right before it (inclusive) contains point b which is also in Group 1. Point b is the representative point because it is the *latest* point such that there is no other point of Group 1 in the window right before b.

For each sliding window, we guarantee that each candidate group G has exactly one representative point. This is achieved by the following process: for each candidate group G, if there is no maintained representative point, then we take the first point u as the representative point (Line 8 and Line 10). When the last point p of the group expires, we delete the maintained representative point u from $S^{\text{acc}} \cup S^{\text{rej}}$, and delete (u, p) from A (Line 3).

For a new arriving point p, if there already exists a point $u \in S^{\text{acc}} \cup S^{\text{rej}}$ in the same group G, then we simply update the last point in the pair $(u, \cdot)$ we maintained for G (Line 6). Otherwise p is the first point of $G(p)$ in the sliding window. If $G(p)$ is a sampled group, then we add p to S^{acc} and (p, p) to A (Line 8); else if $G(p)$ is a rejected group, then we add p to S^{rej} and (p, p) to A (Line 10).

The following observation is a direct consequence of Algorithm 2. It follows from the discussion above and the testing at Line 7 of Algorithm 2.

OBSERVATION 1. *In Algorithm 2, at any time for the current sliding window, we have*

(1) *Each group has exactly one representative point, which is fully determined by the stream and is* independent *of the hash function. More precisely, a point p becomes the representative*

point of group G in the current window if p is the latest point in G such that the window right before p (inclusive) has no point in G. See Figure 2 for an illustration.

(2) *The representative point of each group in the current window is included in S^{acc} with probability $1/R$.*

2.2.2 A Space-Efficient Algorithm for Sliding Windows.

We now present our space-efficient sliding window algorithm. Note that the algorithm presented in Section 2.2.1, though being able to produce a random sample in the sliding window setting, does not have a good space usage guarantee; it may use space up to w/R where w is the window size.

The sliding window algorithm presented in this section works simultaneously for both sequence-based sliding window and time-based sliding window.

High Level Ideas. As mentioned, the main challenge of the sliding window algorithm design is that points will expire, and thus we cannot keep decreasing the sample rate. On the contrary, if at some point there are too few non-expired sampled points, then we have to increase the sample rate to guarantee that there is at least one point in the sliding window belonging to S^{acc}. However, if we increase the sample rate in the middle of the streaming process, then the neighborhood information of a newly sampled group may already get lost. In other words, we cannot properly maintain S^{rej} when the sample rate increases.

To resolve this issue we have the "prepare" such a decrease of $|S^{\text{acc}}|$ in advance. To this end, we maintain a hierarchical set of instances of Algorithm 2, with sample rates $1/R$ being $1, 1/2, 1/4, \ldots$ respectively. We thus can guarantee that in the lowest level (the one with sample rate 1) we must have at least one sampled point.

Of course, to achieve a good space usage we cannot endlessly insert points to all the Algorithm 2 instances. We instead make sure that each level ℓ stores at most $\left|S_\ell^{\text{acc}} \cup S_\ell^{\text{rej}}\right| = O(\log m)$ points, where S_ℓ^{acc} and S_ℓ^{rej} are the accept set and reject set respectively in the run of an Algorithm 2 instance at level ℓ. We achieve this by maintaining a *dynamic* partition of the sliding window. In the ℓ-th subwindow we run an instance of Algorithm 2 with sample rate $1/2^\ell$. For each incoming point, we "accept" it at the highest level ℓ in which the point falls into S_ℓ^{acc}, and then delete *all* points in the accept and reject sets in all the lower levels. Whenever the number of points in S_ℓ^{acc} at some level ℓ exceeds the threshold $c \log m$ for some constant c, we "promote" most of its points to level $\ell + 1$. The process may cascade to the top level.

At the time of query we properly resample the points maintained at each S_ℓ^{acc} ($\ell = 0, 1, \ldots$) to unify their sample probabilities, and then merge them to S^{acc}. In order to guarantee that if the sliding window is not empty then we always have at least one sampled point in S^{acc}, during the algorithm (in particular, the promotion procedure) we make sure that the last point of each level ℓ is always in the accept set S_ℓ^{acc}.

REMARK 1. *The hierarchical set of windows reminisces the* exponential histogram *technique by Datar et al. [16] for* basic counting *in the sliding window model. However, by a careful look one will notice that our algorithm is very different from exponential histogram, and is (naturally) more complicated since we need to deal with both* distinct elements *and* near-duplicates*. For example, the exponential histogram algorithm in [16] partitions the sliding window* deterministically *to subwindows of size* $1, 2, 4, \ldots$*. Suppose we are only interesting in the representative point of each group, we basically need to delete all the other points in each group in the sliding window, which will change the sizes of the subwindows. Handling near duplicates adds another layer of difficulty to the algorithm design; we handle this by employing Algorithm 2 (which is a variant of the algorithm for the infinite window in Section 2.1) at each of the subwindows with different sample rates. The interplay between these components make the overall algorithm involved.*

Algorithm 3: Robust ℓ_0-SAMPLING-SW

```
1  R_ℓ ← 2^ℓ for all ℓ = 0, 1, ..., L.
2  for ℓ ← 0 to L do
       /* create an algorithm instance
          according to Algorithm 2 with fixed
          sample rate 1/R_ℓ                          */
3      ALG_ℓ ← (⊥, ⊥, ⊥, R_ℓ)
4  for each arriving point p do
5      for ℓ ← L downto 0 do
6          ALG_ℓ(p) /* feed p to the instance ALG_ℓ
               */
7          if ∃ (u, p) ∈ A_ℓ then
               /* prune all subsequent levels  */
8              for j ← ℓ − 1 downto 0 do
9                  ALG_j ← (⊥, ⊥, ⊥, R_j)
10             if |S_ℓ^acc| > κ0 log m then
11                 j ← ℓ
12                 create a temporary instance ALG
13                 while (|S_j^acc| > κ0 log m) do
14                     (ALG, ALG_j) ← SPLIT(ALG_j)
15                     ALG_{j+1} ← MERGE(ALG, ALG_{j+1})
16                     j ← j + 1
17                     if j > L then return "error"
18         break
   /* at the time of query:                         */
19 S ← ∅
20 Let c be the maximum index ℓ such that S_ℓ^acc ≠ ⊥
21 for ℓ ← 1 to c do
22     include each point in the set {p | ∃ (·, p) ∈ A_ℓ} to S with
         probability R_ℓ/R_c
23 return a random point from S
```

The Algorithm. We present our sliding window algorithm in Algorithm 3 using Algorithm 4 and Algorithm 5 as subroutines.

We use ALG to represent an instance of Algorithm 2. For convenience, we also use ALG to represent all the data structures that are maintained during the run of Algorithm 2, and write $\text{ALG} = (S^{\text{acc}}, S^{\text{rej}}, A, R)$, where $S^{\text{acc}}, S^{\text{rej}}$ are the accept and reject sets respectively, A is the key-value pair store, and R is the reciprocal of the sample rate.

Algorithm 4: SPLIT($\mathtt{ALG}_\ell$)

create instances $\mathtt{ALG}_a = (S_a^{\text{acc}}, S_a^{\text{rej}}, A_a, R_a)$ and $\mathtt{ALG}_b = (S_b^{\text{acc}}, S_b^{\text{rej}}, A_b, R_b)$

$t = \max\{t' \mid (p_{t'} \in S_\ell^{\text{acc}}) \wedge (h_{R_{\ell+1}}(\text{CELL}(p_{t'})) = 0)\}$

$S_a^{\text{acc}} \leftarrow \{p_k \in S_\ell^{\text{acc}} \mid (k \le t) \wedge (h_{R_{\ell+1}}(\text{CELL}(p_k)) = 0)\}$; $S_a^{\text{rej}} \leftarrow \{p_k \in S_\ell^{\text{rej}} \mid (k \le t) \wedge (h_{R_{\ell+1}}(\text{CELL}(p_k)) = 0)\}$; $A_a \leftarrow \{(u, \cdot) \in A_\ell \mid u \in S_\ell^{\text{acc}}\}$; $R_a \leftarrow R_{\ell+1}$

$S_b^{\text{acc}} \leftarrow \{p_k \in S_\ell^{\text{acc}} \mid k > t\}$; $S_b^{\text{rej}} \leftarrow \{p_k \in S_\ell^{\text{rej}} \mid k > t\}$; $A_b \leftarrow \{(u, \cdot) \in A_\ell \mid u \in S_\ell^{\text{acc}}\}$; $R_b \leftarrow R_\ell$

return $(\mathtt{ALG}_a, \mathtt{ALG}_b)$

Algorithm 5: MERGE($\mathtt{ALG}_a, \mathtt{ALG}_b$)

create a temporary instance $\mathtt{ALG} = (S^{\text{acc}}, S^{\text{rej}}, A, R)$

$S^{\text{acc}} \leftarrow S_a^{\text{acc}} \cup S_b^{\text{acc}}$; $S^{\text{rej}} \leftarrow S_a^{\text{rej}} \cup S_b^{\text{rej}}$; $A \leftarrow A_a \cup A_b$; $R \leftarrow R_a$

return ALG

Set $R_\ell = 2^\ell$ for $\ell = 0, 1, \ldots, L = \log w$. In Algorithm 3 we create L instances of Algorithm 2 with empty $S_\ell^{\text{acc}}, S_\ell^{\text{rej}}, A_\ell$ (denoted by '$\perp$'), and sample rates $1/R_\ell$ respectively. We call the instance with $R_\ell = 2^\ell$ the level ℓ instance.

When a new point p comes, we first find the highest level ℓ such that p is sampled by $\mathtt{ALG}_\ell$ (i.e., $p \in S_\ell^{\text{acc}}$), and then delete all the structures of $\mathtt{ALG}_j$ ($j < \ell$), except keep their sample rates $1/R_j$ (Line 5 to Line 9).

If after including p, the size of S_ℓ^{acc} becomes more than $\kappa_0 \log m$, we have to do a series of updates to restore the invariant that the accept set of each level contains at most $\kappa_0 \log m$ points at any time step (Line 10 to Line 16). To do this, we first split the instance of $\mathtt{ALG}_\ell$ into two instances (Algorithm 4). Let point p be the last point in S_ℓ^{acc} which is sampled by hash function $h_{R_{\ell+1}}$. We promote all the points in $S_\ell^{\text{acc}} \cup S_\ell^{\text{rej}}$ arriving before (and include) p to level $\ell + 1$ by resampling them using $h_{R_{\ell+1}}$, which gives a new level $\ell + 1$ instance ALG.

We next try to merge ALG with $\mathtt{ALG}_{\ell+1}$ who have the same sample rate by merging their accept/reject sets and the sets of key-value pairs (Algorithm 5). The merge may result $\left|S_{\ell+1}^{\text{acc}}\right| > \kappa_0 \log m$, in which case we have to perform the split and merge again. These operations may propagate to the upper levels until we research a level ℓ in which we have $\left|S_\ell^{\text{acc}}\right| \le \kappa_0 \log m$ after the merge.

At the time of query, we have to integrate the maintained samples in all L levels. Since at each level we sample points use different sample rates $1/R_\ell$, we have to resample each point in S_ℓ^{acc} with probability R_ℓ/R_c where c is the largest level that has a non-empty accept set (Line 20 to Line 22).

Correctness and Complexity. In this section we prove the following theorem.

THEOREM 2.7. *In constant dimensional Euclidean space for a well-separated dataset, there exists a sliding window algorithm (Algorithm 3) that with probability $1 - 1/m$, at any time step, it outputs a robust ℓ_0-sample. The algorithm uses $O(\log w \log m)$ words of space and $O(\log w \log m)$ amortized processing time per point.*

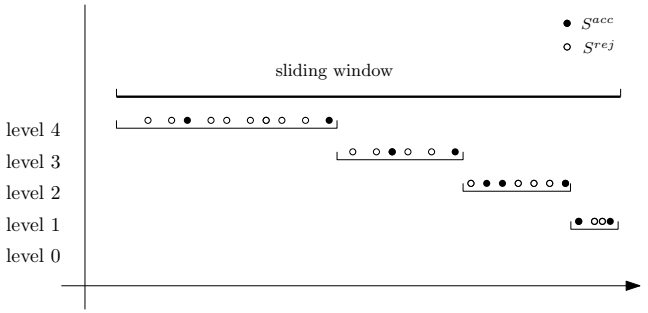

Figure 3: An illustration of subwindows of a sliding window; the subwindow at level 0 is an empty set.

First, it is easy to show the probability that Algorithm 3 outputs "error" is negligible.

LEMMA 2.8. *With probability $1 - 1/m^2$, Algorithm 3 will not output "error" at Line 17 during the whole streaming process.*

PROOF. Fix a sliding window W. Let $G_1, \ldots, G_k$ ($k \le w$) be the groups in W. The sample rate at level L is $1/R_L = 1/2^L = 1/w$. Let X_ℓ be a random variable such that $X_\ell = 1$ if the ℓ-th group is sampled, and $X_\ell = 0$ otherwise. Let $X = \sum_{\ell=1}^{k} X_\ell$. Thus $\mathbf{E}[X] = k \cdot 1/R_L \le w \cdot 1/w = 1$. By a Chernoff bound (Lemma 1.8) we have that with probability $1 - 1/m^3$, we have $X \le \kappa_0 \log m$ for a large enough constant κ_0. The lemma then follows by a union bound over at most m sampling steps. □

The following definition is useful for the analysis.

Definition 2.9 (subwindows). For a fixed sliding window W, we define a subwindow W_ℓ for each instance $\mathtt{ALG}_\ell$ ($\ell = 0, 1, \ldots, L$) as follows. W_L starts from the first point in the sliding window to the last point (denoted by p_{t_L}) in $A(S_L^{\text{acc}})$. For $\ell = L-1, \ldots, 1$, W_ℓ starts from $p_{t_{\ell+1}+1}$ to the last point (denoted by p_{t_ℓ}) in $A(S_\ell^{\text{acc}})$. W_0 starts from p_{t_1+1} to the last point in the window W.

See Figure 3 for an illustration of subwindows.

We note that a subwindow can be empty. We also note the following immediate facts by the definitions of subwindows.

FACT 2. *$W_0 \cup W_1 \cup \ldots \cup W_L$ covers the whole window W.*

FACT 3. *Each subwindow W_ℓ ($\ell = 1, \ldots, L$) ends up with a point in $A(S_\ell^{acc})$.*

For $\ell = 0, 1, \ldots, L$, let $\mathcal{G}_\ell$ be the set of groups whose last points lie in W_ℓ, and let S_ℓ^{rep} be the set of their representative points. From Algorithm 3, 4 and 5 it is easy to see that the following is maintained during the whole streaming process.

FACT 4. *During the run of Algorithm 3, at any time step, S_ℓ^{acc} is formed by sampling each point in S_ℓ^{rep} with probability $1/R_\ell$.*

The following lemma guarantees that at the time of query we can always output a sample.

LEMMA 2.10. *During the run of Algorithm 3, at any time step, if the sliding window contains at least one point, then when querying we can always return a sample, i.e.,* $|S| \geq 1$.

PROOF. The lemma follows from Fact 3, and the fact that $\texttt{ALG}_0$ includes every point in S_0^{rep} ($R_0 = 1$). □

Now we are ready to prove the theorem.

(FOR THEOREM 2.7). We have the following facts:

(1) $S_0^{\text{rep}}, S_1^{\text{rep}}, \ldots, S_L^{\text{rep}}$ are set of representatives of *disjoint* sets of groups $\mathcal{G}_0, \mathcal{G}_1, \ldots, \mathcal{G}_L$. And $\cup_{\ell=0}^{L} \mathcal{G}_\ell$ is the set of all groups who have the last points inside the sliding window.
(2) By Fact 4 each S_ℓ^{acc} is formed by sampling each point in S_ℓ^{rep} with probability $1/R_\ell$.
(3) Each point in S_ℓ^{rep} is included in S with probability R_ℓ/R_c (Line 22 of Algorithm 3).
(4) By Lemma 2.10, $|S| \geq 1$ if the sliding window is not empty.
(5) The final sample returned is a random sample of S.
(6) By Lemma 2.8, with probability $(1 - 1/m)$ the algorithm will not output "error".

By the first three items we know that S is a random sample of the last points of all groups within the sliding window, which, combined with Item 4, 5 and 6, give the correctness of the theorem.

We now analyze the space and time complexities. The space usage at each level can be bounded by $O(\log m)$ words. This is due to the fact that $\left|S_j^{\text{acc}}\right|$ is always no more $\kappa_0 \log m$, and consequently A_j has $O(\log m)$ key-value pairs. Using Lemma 2.6 we have that with probability $1 - 1/m^2$, $\left|S_j^{\text{rej}}\right| = O(\log m)$.[1] Thus by a union bound, with probability $(1 - O(\log w/m^2))$, the total space is bounded by $O(\log w \log m)$ words since we have $O(\log w)$ levels.

For the time cost, simply note that the time spent on each point at each level during the whole streaming process can be bounded by $O(\log m)$, and thus the amortized processing time per item can be bounded by $O(\log w \log m)$. □

2.3 Discussions

We conclude the section with some discussions and easy extensions.

Sampling k Points with/without Replacement. Sampling k groups *with* replacement can be trivially achieved by running k instances of the algorithm for sampling one group (Algorithm 1 or Algorithm 3) in parallel. For sampling k groups *without* replacement, we can increase the threshold at Line 10 of Algorithm 1 to $\kappa_0 k \log m$, by which we can show using exactly the same analysis in Section 2.1 that with probability $1 - 1/m$ we have $|S^{\text{acc}}| \geq k$. Similarly, for the sliding window case we can increase the threshold at Line 10 of Algorithm 3 to $\kappa_0 k \log m$.

Random Point As Group Representative. We can easily augment our algorithms such that instead of always returning the (fixed) representative point of a randomly sampled group, we can return a random point of the group. In other words, we want to return each point $p \in G$ with equal probability $\frac{1}{n \cdot |G|}$.

[1] We can reduce the failure probability $1/m$ to $1/m^2$ by appropriately changing the constants in the proof.

For the infinite window case we can simply plug-in the classical Reservoir sampling [35] in Algorithm 1. We can implement this as follows: For each group G that has a point stored in $S^{\text{acc}} \cup S^{\text{rej}}$, we maintain an $e_G = (v, ct)$ pair where ct is a counter counting the number of points of this group, and v is the random representative point. At the beginning (when the first point u of group G comes) we set $e_G = (u, 1)$. When a new point p is inserted, if there exists $u \in S^{\text{acc}}$ such that $d(u, p) \leq \alpha$ (i.e., u and p are in the same group), we increment the counter ct for group $G(u)$, and reset $e_G = (u, p)$ with probability $\frac{1}{ct}$. For the sliding window case, we can just replace Reservoir sampling with a random sampling algorithm for sliding windows (e.g., the one in [8]).

3 GENERAL DATASETS

In this section we consider general datasets which may *not* be well-separated, and consequently there is *no* natural partition of groups. However, we show that Algorithm 1 still gives the following guarantee.

THEOREM 3.1. *For a general dataset S in constant dimensional Euclidean space, there exists a streaming algorithm (Algorithm 1) that with probability $1 - 1/m$, at any time step, it outputs a point q satisfying Equality (2), that is,*

$$\forall p \in S, \mathbf{Pr}[q \in \texttt{Ball}(p, \alpha) \cap S] = \Theta(\tfrac{1}{F_0(S, \alpha)}),$$

where $\texttt{Ball}(p, \alpha)$ *is the ball centered at p with radius α.*

Before proving the theorem, we first study group partitions generated by a greedy process.

Definition 3.2 (Greedy Partition). Given a dataset S, a greedy partition is generated by the following process: pick an arbitrary point $p \in S$, create a new group $G(p) \leftarrow \texttt{Ball}(p, \alpha) \cap S$ and update $S \leftarrow S \backslash G(p)$; repeat this process until $S = \emptyset$.

LEMMA 3.3. *Given a dataset S, let $n_{\texttt{opt}}$ be the number of groups in the minimum cardinality partition of S, and $n_{\texttt{gdy}}$ be the number of groups in an* arbitrary *greedy partition. We always have $n_{\texttt{opt}} = \Theta(n_{\texttt{gdy}})$.*

PROOF. We first show $n_{\texttt{gdy}} \leq n_{\texttt{opt}}$. Let $G(p_1), \ldots, G(p_{n_{\texttt{gdy}}})$ be the groups in the greedy partition according to the orders they were created, and let $H_1, \ldots, H_{n_{\texttt{opt}}}$ be the minimum partition.

We prove by induction. First it is easy to see that $G(p_1)$ must cover the group containing p_1 in the minimum partition (w.l.o.g. denote that group by H_1). Suppose that $\bigcup_{j=1}^{i} G(p_j)$ covers i groups $H_1, \ldots, H_i$ in the minimum partition, that is, $\bigcup_{j=1}^{i} H_j \subseteq \bigcup_{j=1}^{i} G(p_j)$, we can show that there must be a new group H_{i+1} in the minimum partition such that $\bigcup_{j=1}^{i+1} H_j \subseteq \bigcup_{j=1}^{i+1} G(p_j)$, which gives $n_{\texttt{gdy}} \leq n_{\texttt{opt}}$. The induction step follows from the following facts.

(1) $p_{i+1} \notin \bigcup_{j=1}^{i} G(p_j)$.
(2) $\texttt{Ball}(p_{i+1}, \alpha) \subseteq \bigcup_{j=1}^{i+1} G(p_j)$.
(3) The diameter of each group in the minimum partition is at most α.

Indeed, by (1) and the induction hypothesis we have $p_{i+1} \notin \bigcup_{j=1}^{i} H_j$. Let H_{i+1} be the group containing p_{i+1} in the minimum partition. Then by (2) and (3) we must have $H_{i+1} \subseteq \texttt{Ball}(p_{i+1}, \alpha) \subseteq \bigcup_{j=1}^{i+1} G(p_j)$.

We next show $n_{\mathtt{opt}} \le O(n_{\mathtt{gdy}})$. This is not obvious since the diameter of a group in the greedy partition may be larger than α (but is at most 2α), while groups in the minimum partition have diameter at most α. However, in constant dimensional Euclidean space, each group in a greedy group partition can intersect at most $O(1)$ groups in the minimum cardinality partition. We thus still have $n_{\mathtt{opt}} \le O(n_{\mathtt{gdy}})$. □

Now we are ready to prove the theorem.

(FOR THEOREM 3.1). We can think the group partition in Algorithm 1 as a greedy process. Let $(q_1, \ldots, q_z)$ be the sequence of points that are included in S^{acc}, according to their arriving orders in the stream. We can generate a greedy group partition on $\bigcup_{i=1}^{z} \mathtt{Ball}(q_i, \alpha)$ as follows: for $i = 1, \ldots, z$, create a new group $G(q_i) \leftarrow \mathtt{Ball}(q_i, \alpha) \cap S$ and update $S \leftarrow S \backslash G(q_i)$. Let $\mathcal{G}_{\mathtt{sub}} = \{G(q_1), \ldots, G(q_z)\}$. We then apply the greedy partition process on the remaining points in S, again according to their arriving orders in the stream. Let $q_{z+1}, \ldots, q_{n_{\mathtt{gdy}}}$ be the representative points of the remaining groups. Let $\mathcal{G} = \{G(q_1), \ldots, G(q_{n_{\mathtt{gdy}}})\}$ be the final group partition of S. We have the following facts.

(1) Each group in $\mathcal{G}$ intersects $\Theta(1)$ grid cell in $\mathbb{G}$.
(2) Each grid cell in $\mathbb{G}$ is sampled by the hash function h_R with equal probability.
(3) $q_1, \ldots, q_z$ are the representative points of their groups in $\mathcal{G}_{\mathtt{sub}}$.
(4) Algorithm 1 returns a sample randomly from $q_1, \ldots, q_z$.

By items 1 and 2, we know that each group in $\mathcal{G}$ is included in $\mathcal{G}_{\mathtt{sub}}$ with probability $\Theta(|\mathcal{G}_{\mathtt{sub}}| / |\mathcal{G}|)$. By items 3 and 4, we know that Algorithm 1 returns a random group from $\mathcal{G}_{\mathtt{sub}}$. Therefore each group $G \in \mathcal{G}$ is sampled by Algorithm 1 with probability $\Theta(1/n_{\mathtt{gdy}}) = \Theta(1/n_{\mathtt{opt}})$, where the last equation is due to Lemma 3.3.

Now for any $p \in S$, according to the greedy process and Algorithm 1, there must be some $q \in S$ such that $G(p) \subseteq \mathtt{Ball}(q, \alpha)$, *and* if $G(p)$ is sampled then q is the sampled point. So the probability that q is sampled is at least the probability that $G(p)$ is sampled. Finally, note that if $p \in \mathtt{Ball}(q, \alpha)$ then we also have $q \in \mathtt{Ball}(p, \alpha)$. We thus have

$$\mathbf{Pr}[\exists\, q \in \mathtt{Ball}(p, \alpha) \text{ s.t. } q \text{ is sampled}] = \Omega(1/n_{\mathtt{opt}}). \tag{7}$$

On the other hand, in constant dimensional Euclidean space $\mathtt{Ball}(p, \alpha)$ can only intersect $O(1)$ groups in the greedy partition. We thus also have

$$\mathbf{Pr}[\exists\, q \in \mathtt{Ball}(p, \alpha) \text{ s.t. } q \text{ is sampled}] = O(1/n_{\mathtt{opt}}). \tag{8}$$

The theorem follows from (7) and (8). □

It is easy to see that the above arguments can also be applied to the sliding window case with respect to Algorithm 3.

COROLLARY 3.4. *For a general dataset in constant dimensional Euclidean space, there exists a sliding window algorithm (Algorithm 3) that with probability $1 - 1/m$, at any time step, it outputs a point q such that $\forall p \in S$, $\mathbf{Pr}[q \in \mathtt{Ball}(p, \alpha)] = \Theta(1/n_{\mathtt{opt}})$, where S is the set of all the points in the sliding window, and $n_{\mathtt{opt}}$ is the size of the minimum cardinality partition of S with group radius α.*

4 HIGH DIMENSIONS

In this section we consider datasets in d-dimensional Euclidean space for general d. We show that Algorithm 1, with some small modifications, can handle (α, β)-sparse dataset in d-dimensional Euclidean space with $\beta > d^{1.5}\alpha$ as well.

THEOREM 4.1. *In the d-dimensional Euclidean space, for an (α, β)-sparse dataset with $\beta > d^{1.5}\alpha$, there is a streaming algorithm such that with probability $1 - 1/m$, at any time step, it outputs a robust ℓ_0-sample. The algorithm uses $O(d \log m)$ words of space and $O(d \log m)$ processing time per item.*

REMARK 2. *We can use Johnson-Lindenstrauss dimension reduction to weaken the sparsity assumption to $\beta \ge c_\alpha \log^{1.5} m \cdot \alpha$ for some large enough constant c_α.*

We place a random grid $\mathbb{G}$ with side length $d\alpha$. Since the dataset is (α, β)-sparse with $\beta > d^{1.5}\alpha$, each grid cell can intersect at most one group. However, in the d-dimensional space a group can intersect 2^d grid cells in the worst case, which may cause difficulty to maintain S^{rej} in small space – in the worst case we would have $\left|S^{\mathrm{rej}}\right| = \Omega(2^d)$ while $|S^{\mathrm{acc}}|$ is still small. Fortunately, in the following lemma we show that for any $p \in S^{\mathrm{rep}}$, the probability that $p \in S^{\mathrm{rej}}$ will not be too large compared with the probability that $p \in S^{\mathrm{acc}}$.

LEMMA 4.2. *For any fixed $p \in S^{rep}$, we have*

$$\mathbf{Pr}[p \in S^{rej}] \le \kappa_1 \cdot \mathbf{Pr}[p \in S^{acc} \cup S^{rej}],$$

where $\kappa_1 \in (0, 1)$ is a constant.

PROOF. For a group G, let $\mathtt{Ball}(G, \alpha) = \{p \mid d(p, G) \le \alpha\}$ where $d(p, G) = \min_{q \in G} d(p, q)$. It is easy to see that $\mathtt{Ball}(G, \alpha)$ has a diameter of at most 3α because the diameter of G is at most α.

Since the random grid has side length $d\alpha$, the probability that $\mathtt{Ball}(G, \alpha)$ is cut by the boundaries of cells in each dimension is at most $\mu = \frac{3}{d}$. If $\mathtt{Ball}(G, \alpha)$ is cut by i dimensions, the number of cells it intersects is at most 2^i, and consequently $|\mathrm{ADJ}(p)| \le 2^i$ for each $p \in G$.

Recall that each cell is sampled with probability $\frac{1}{R}$, we thus have

$$\begin{aligned}
& \mathbf{Pr}[p \in S^{\mathrm{rej}} \cup S^{\mathrm{acc}}] \\
\le & \sum_{i \ge 1} \mathbf{Pr}[p \in S^{\mathrm{rej}} \cup S^{\mathrm{rej}} \mid |\mathrm{ADJ}(p)| = i] \cdot \mathbf{Pr}[|\mathrm{ADJ}(p)| = i] \\
\le & \sum_{i=0}^{d} \binom{d}{i} \mu^i (1-\mu)^{d-i} \frac{2^i}{R} \\
= & \frac{(2\mu + 1 - \mu)^d}{R} \\
\le & \frac{(1 + \frac{3}{d})^d}{R} = O\left(\frac{1}{R}\right).
\end{aligned}$$

Since $S^{\mathrm{acc}} \cap S^{\mathrm{rej}} = \emptyset$, we have

$$\begin{aligned}
\mathbf{Pr}[p \in S^{\mathrm{rej}}] &= \mathbf{Pr}[p \in S^{\mathrm{rej}} \cup S^{\mathrm{acc}}] - \mathbf{Pr}[p \in S^{\mathrm{acc}}] \\
&\le \kappa_1 \cdot \mathbf{Pr}[p \in S^{\mathrm{acc}} \cup S^{\mathrm{rej}}]
\end{aligned}$$

for some constant $\kappa_1 \in (0, 1)$. □

By Lemma 4.2, and basically the same analysis as that in Lemma 2.6, we can bound the space usage of Algorithm 1 by $O(d \log m)$ ($O(\log m)$

points in the d-dimensional space) throughout the execution of the algorithm with probability $(1 - 1/m)$, and consequently the running time.

We have a similar result for the sliding window case.

COROLLARY 4.3. *In the d-dimensional Euclidean space, for an (α, β)-sparse dataset with $\beta > d^{1.5}\alpha$, there is a sliding window algorithm such that with probability $1 - 1/m$, at any time step, it outputs a robust ℓ_0-sample. The algorithm uses $O(d \log w \log m)$ words of space and $O(d \log w \log m)$ processing time per item, where w is the size of the sliding window.*

5 DISTINCT ELEMENTS

In this section we show that our algorithms for robust ℓ_0-sampling can be used for approximating the number of robust distinct elements in both infinite window and sliding window settings.

Estimating F_0 in the infinite window. We first recall the algorithm for estimating the number of distinct elements on noisefree datasets by Bar-Yossef et al. [7]. We maintain an integer z initialized to be 0, and a set B consisting of at most κ_B/ϵ^2 items for a large enough constant κ_B. For each arriving point, we perform an ℓ_0-sampling with probability $1/2^z$ and add the point to B if sampled. At the time $B > \kappa_B/\epsilon^2$, we update $z \leftarrow z + 1$, and re-sample each point in B with probability $1/2$ so that the overall sample probability is again $1/2^z$. It was shown in [7] that with probability at least 0.9, $|B|\, 2^z$ approximates the robust F_0 up to a factor of $(1 + \epsilon)$. We can run $\Theta(\log m)$ independent copies of above procedure to boost the success probability to $1 - 1/m$.

Now we can directly plug our ℓ_0-sampling algorithm into the framework of [7]. We simply replace the threshold $\kappa_0 \log m$ at Line 10 of Algorithm 1 with κ_B/ϵ^2. At the time of query we return $|S^{\text{acc}}| \cdot R$ as the estimation of the number of distinct groups. Again, running $\Theta(\log m)$ independent copies of the algorithm and taking the median will boost the constant success probability to high probability.

Estimating F_0 in the sliding windows. We can use the ideas from Flajolet and Martin [23] (which is known as the FM sketch). We run $\Theta(1/\epsilon^2)$ independent copies of our sliding window algorithm. For each of the copies, we find the largest ℓ that S^{acc}_ℓ includes at least one non-expired sample. We average all those ℓ's as $\bar{\ell}$. It follows [23] that with probability 0.9, $\phi 2^{\bar{\ell}}$ gives $(1 + \epsilon)$-approximation to the robust F_0 in the sliding window, where ϕ is a universal constant to correct the bias (see, e.g., [23]). We can then run $\Theta(\log m)$ copies of above procedure and taking the median to boost the success probability. Similarly we can also plug-in our algorithm to HyperLogLog [21].

6 EXPERIMENTS

We have conducted an extensive set of experiments for our algorithm in the infinite window case. Due to the space constraints we delay the whole section to Appendix A.

7 CONCLUDING REMARKS

In this paper we study how to perform distinct sampling in the noisy data stream setting, where items may have near-duplicates and we would like to treat all the near-duplicates as the same element. We have proposed algorithms for both infinite window and sliding windows cases. The space and time usages of our algorithms only poly-logarithmically depend on the length of the stream. Our extensive experiments have demonstrated the effectiveness and the efficiency of our distinct sampling algorithms.

As observed in [9], the random grid we have used for dealing with data points in the Euclidean space is a particular locality-sensitive hash function,[2] and it is possible to generalize our algorithms to general metric spaces that are equipped with efficient locality-sensitive hash functions. We leave this generalization as a future work.

REFERENCES

[1] K. J. Ahn, S. Guha, and A. McGregor. Analyzing graph structure via linear measurements. In *SODA*, pages 459–467, 2012.
[2] K. J. Ahn, S. Guha, and A. McGregor. Graph sketches: sparsification, spanners, and subgraphs. In *PODS*, pages 5–14, 2012.
[3] K. J. Ahn, S. Guha, and A. McGregor. Spectral sparsification in dynamic graph streams. In *APPROX-RANDOM*, pages 1–10, 2013.
[4] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. *J. Comput. Syst. Sci.*, 58(1):137–147, 1999.
[5] S. Assadi, S. Khanna, Y. Li, and G. Yaroslavtsev. Maximum matchings in dynamic graph streams and the simultaneous communication model. In *SODA*, pages 1345–1364, 2016.
[6] B. Babcock, M. Datar, and R. Motwani. Sampling from a moving window over streaming data. In *SODA*, pages 633–634, 2002.
[7] Z. Bar-Yossef, T. S. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan. Counting distinct elements in a data stream. In *RANDOM*, pages 1–10, 2002.
[8] V. Braverman, R. Ostrovsky, and C. Zaniolo. Optimal sampling from sliding windows. In *PODS*, pages 147–156, 2009.
[9] D. Chen and Q. Zhang. Streaming algorithms for robust distinct elements. In *SIGMOD*, pages 1433–1447, 2016.
[10] R. Chitnis, G. Cormode, H. Esfandiari, M. Hajiaghayi, A. McGregor, M. Monemizadeh, and S. Vorotnikova. Kernelization via sampling with applications to finding matchings and related problems in dynamic graph streams. In *SODA*, pages 1326–1344, 2016.
[11] R. H. Chitnis, G. Cormode, M. T. Hajiaghayi, and M. Monemizadeh. Parameterized streaming: Maximal matching and vertex cover. In *SODA*, pages 1234–1251, 2015.
[12] Y.-Y. Chung and S. Tirthapura. Distinct random sampling from a distributed stream. In *IPDPS*, pages 532–541, 2015.
[13] G. Cormode and D. Firmani. A unifying framework for ℓ_0-sampling algorithms. *Distributed and Parallel Databases*, 32(3):315–335, 2014.
[14] G. Cormode, S. Muthukrishnan, and I. Rozenbaum. Summarizing and mining inverse distributions on data streams via dynamic inverse sampling. In *VLDB*, pages 25–36, 2005.
[15] G. Cormode, S. Muthukrishnan, K. Yi, and Q. Zhang. Continuous sampling from distributed streams. *J. ACM*, 59(2):10:1–10:25, 2012.
[16] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. *SIAM J. Comput.*, 31(6):1794–1813, 2002.
[17] X. L. Dong and F. Naumann. Data fusion: resolving data conflicts for integration. *Proceedings of the VLDB Endowment*, 2(2):1654–1655, 2009.
[18] D. Dubhashi and A. Panconesi. *Concentration of Measure for the Analysis of Randomized Algorithms*. Cambridge University Press, New York, NY, USA, 1st edition, 2009.
[19] M. Durand and P. Flajolet. Loglog counting of large cardinalities. In *Algorithms-ESA 2003*, pages 605–617. Springer, 2003.
[20] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios. Duplicate record detection: A survey. *IEEE Trans. Knowl. Data Eng.*, 19(1):1–16, 2007.
[21] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. In *AofA: Analysis of Algorithms*, pages 137–156, 2007.
[22] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. *DMTCS Proceedings*, (1), 2008.
[23] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. *J. Comput. Syst. Sci.*, 31(2):182–209, 1985.
[24] G. Frahling, P. Indyk, and C. Sohler. Sampling in dynamic data streams and applications. *Int. J. Comput. Geometry Appl.*, 18(1/2):3–28, 2008.
[25] S. Ganguly. Counting distinct items over update streams. *Theoretical Computer Science*, 378(3):211–222, 2007.
[26] P. B. Gibbons and S. Tirthapura. Estimating simple functions on the union of data streams. In *SPAA*, pages 281–291, 2001.

[2] See for example https://en.wikipedia.org/wiki/Locality-sensitive_hashing.

[27] T. N. Herzog, F. J. Scheuren, and W. E. Winkler. *Data quality and record linkage techniques*, volume 1. Springer, 2007.
[28] H. Jowhari, M. Saglam, and G. Tardos. Tight bounds for lp samplers, finding duplicates in streams, and related problems. In *PODS*, pages 49–58, 2011.
[29] D. M. Kane, J. Nelson, and D. P. Woodruff. An optimal algorithm for the distinct elements problem. In *PODS*, pages 41–52, 2010.
[30] C. Konrad. Maximum matching in turnstile streams. In *ESA*, pages 840–852, 2015.
[31] N. Koudas, S. Sarawagi, and D. Srivastava. Record linkage: similarity measures and algorithms. In *SIGMOD*, pages 802–803. ACM, 2006.
[32] A. McGregor. Graph stream algorithms: a survey. *SIGMOD Record*, 43(1):9–20, 2014.
[33] J. P. Schmidt, A. Siegel, and A. Srinivasan. Chernoff–hoeffding bounds for applications with limited independence. *SIAM Journal on Discrete Mathematics*, 8(2):223–250, 1995.
[34] S. Tirthapura and D. P. Woodruff. Optimal random sampling from distributed streams revisited. In *DISC*, pages 283–297, 2011.
[35] J. S. Vitter. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 11(1):37–57, 1985.
[36] Q. Zhang. Communication-efficient computation on distributed noisy datasets. In *SPAA*, pages 313–322, 2015.
[37] W. Zhang, Y. Zhang, M. A. Cheema, and X. Lin. Counting distinct objects over sliding windows. In *ADC*, pages 75–84, 2010.

A EXPERIMENTS

In this section, we present our experimental results for Algorithm 1 as well as some implementation details. The algorithm is easy to implement yet very efficient, and can be of interest to practitioners.

A.1 The Setup

Datasets. We verify our algorithms using the following real and synthetic datasets.

- `Rand5`: 500 randomly generated points in $\mathbb{R}^5$; each coordinate is a random number from $(0, 1)$.
- `Rand20`: 500 randomly generated points in $\mathbb{R}^{20}$; each coordinate is a random number from $(0, 1)$.
- `Yacht`: 308 points taken from the UCI repository yacht hydrodynamics data set.[3] Each point is in $\mathbb{R}^7$, and it measures the sailing yachts movement.
- `Seeds`: 210 points taken from the UCI repository seeds data set.[4] Each point is in $\mathbb{R}^8$, consisting of measurements of geometrical properties of kernels belonging to three different varieties of wheat.

We perform two types of near-duplicate generations on each dataset. In the first transformation we generate near-duplicates as follows: We first rescale the dataset such that the minimum pairwise distance is 1. Then for each point $\mathbf{x}_i$ $(i = 1, 2, \ldots, n)$, we pick a number k_i uniformly at random from $1, 2, \ldots, 100$, and add k_i near-duplicate points w.r.t. $\mathbf{x}_i$, each of which is generated as follows:

(1) Generate a vector $\mathbf{z} \in \mathbb{R}^d$ such that each coordinate of $\mathbf{z}$ is chosen randomly from $(0, 1)$.
(2) Randomly sample a number $\ell \in \left(0, \frac{1}{2d^{1.5}}\right)$ and rescale $\mathbf{z}$ to length ℓ. Let $\hat{\mathbf{z}}$ be the resulting vector.
(3) Create a near-duplicate point $\mathbf{y} = \mathbf{x}_i + \hat{\mathbf{z}}$.

Note that each point $\mathbf{x}_i$, together with the near-duplicate points around it, forms a group. We still name the resulting datasets as `Rand5`, `Rand20`, `Yacht` and `Seeds` respectively.

[3] https://archive.ics.uci.edu/ml/datasets/Yacht+Hydrodynamics
[4] https://archive.ics.uci.edu/ml/datasets/seeds

In the second transformation, the number of near-duplicates we generate for each data point follows the *power-law* distribution. More precisely, we randomly order the points as $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$, and for each point $\mathbf{x}_i$, we add $\lceil n \cdot i^{-1} \rceil$ noisy points in the same way as above. We denote the resulting datasets as `Rand5-pl`, `Rand20-pl`, `Yacht-pl` and `Seeds-pl` respectively.

Measurements. For each dataset, we run each of our proposed sampling algorithms a large number of times (denoted by #runs, which ranges from $200,000$ to $500,000$), and count the number of times each group being sampled. We report the following results measuring the performance of our proposed algorithm.

- `pTime`: Processing time per item; measured by millisecond. The running time is tested using single thread.
- `pSpace`: Peak space usage throughout the streaming process; measured by word.

We record `pTime` and `pSpace` by taking the average of 100 runs where in each run we scan the whole data stream.

The following two accuracy measurements for the ℓ_0-sampling algorithms follow from [13].

- `stdDevNm`: Let F_0 be the number of groups, and let $f^* = \frac{1}{F_0}$ be the target probability. We calculate the standard deviation of the empirical sampling distribution and normalize it by f^*.
- `maxDevNm`: We calculate the normalized maximum deviation of the empirical sampling distribution as

$$\max_i \left\{ \frac{|f_i - f^*|}{f^*} \right\},$$

where f_i is the empirical sampling probability of the i-th group.

We will also visualize the number of times each group being sampled.

All of the eight datasets are randomly shuffled before being fed into our algorithms as data streams. We return the sample at the end of a data stream.

Computational Environment. Our algorithms are implemented in C++ and the visualization code is implemented in python+matplotlib. We run our experiments in a PowerEdge R730 server equipped with 2 x Intel Xeon E5-2667 v3 3.2GHz. This server has 8-core/16-thread per CPU, 192GB Memeory and 1.6TB SSD.

A.2 Computing $\textsc{adj}(p)$ in $\mathbb{R}^d$

Before presenting the experimental results, we discuss some important details of our implementation.

Recall the definition of $\textsc{adj}(p)$ introduced in Section 2:

$$\textsc{adj}(p) = \{C \in \mathbb{G} \mid d(p, C) \leq \alpha\}.$$

In Section 2 we did not spell out how to compute $\textsc{adj}(p)$ in $\mathbb{R}^d$ because our discussion focused on the case $d = \Theta(1)$, where computing $\textsc{adj}(p)$ only takes $O(1)$ time. However, the naive implementation that we enumerate all the adjacent cells of $\textsc{cell}(p)$ and test for each cell C whether we have $d(p, C) \leq \alpha$ takes $\Theta(d \cdot 3^d)$ time: we have 3^d cells to exam, and each cell has d coordinates. This is expensive for large d. In the following we illustrate how to compute $\textsc{adj}(p)$ efficiently in practice.

According to Lemma 4.2, the expected size of $\textsc{adj}(p)$ is always bounded by a small constant given a sufficiently large separation

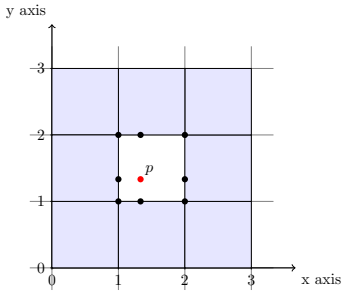

Figure 4: Nearest points of $p = (x_1, x_2) \in \mathbb{R}^2$ (red point). The eight light blue cells are cells adjacent to CELL(p). There are eight black points, each of which is the nearest point from p in the corresponding cell

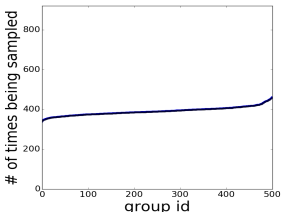

Figure 5: `Rand5` **dataset. #runs** $= 200,000$

ratio. Therefore it is possible to compute ADJ(p) more efficiently if we can avoid the exhaustive enumeration of the adjacent cells of CELL(p). We approach this by effectively pruning cells that are impossible to meet the requirement that $d(p, C) \le \alpha$.

To simplify the presentation, we set the side length of the grid $\mathbb{G}$ to be 1 and rescale the value α accordingly. We further identify the grid with the Cartesian coordinates with the origin $\mathbf{0} = (0, \ldots, 0)$. Consider a point $p = (x_1, x_2, \ldots, x_d)$. To calculate the distance from p to a cell C that is adjacent to CELL(p), we move p to the nearest point in C and record the distance being moved. The movement can be done sequentially: first in the direction of x_1, and then in the direction of x_2, ... until x_d.

Figure 4 gives an illustration of the nearest points of p in $\mathbb{R}^2$. To enumerate all those nearest points, we iterate the coordinates of p from x_1 to x_d. For x_i, we have three options: (1) move to $\lfloor x_i \rfloor$; (2) move to $\lceil x_i \rceil$; and (3) do not move. Consequently we have 3^d different nearest points, and thus 3^d different cells which are the cells adjacent to CELL(p) including CELL(p) itself. We then perform the enumeration using a DFS search and prune the search as long as the accumulated distance exceeds α. The details of this procedure are presented in Algorithm 6 and Algorithm 7.

A.3 Results and Discussions

We visualize our experimental results in Figure 5-15. Figure 13 and Figure 14 show the results for time and space respectively, and Figure 15 presents the deviations of the empirical sampling distributions. Figure 5-12 visualize the empirical sampling distribution of each dataset.

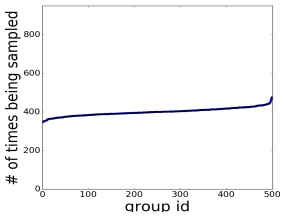

Figure 6: `Rand20` **dataset. #runs** $= 200,000$

Algorithm 6: SEARCHADJ($p, i, s, (y_1, \ldots, y_{i-1}, \perp, \ldots, \perp)$)

```
/* p = (x_1, ..., x_d) ∈ R^d; i = 1, 2, ..., d is the depth
   of the DFS search; s is the square of
   the distance of the movement                          */
if s > α^2 then
    /* the distance of the movement exceeds
       α; no need to continue the search                 */
    return
if i > d then
    q ← (y_1, ..., y_d)
    /* Since q is on the boundary, we add
       0.01 · (q − p) to make sure that it moves
       inside a cell so that CELL(q') is
       well defined                                      */
    q' ← q + 0.01 · (q − p)
    Emit CELL(q')
    return
/* move x_i to ⌊x_i⌋                                    */
SEARCHADJ(p, i + 1, s + (⌊x_i⌋ − x_i)^2,                 /* no
          (y_1, ..., y_{i−1}, ⌊x_i⌋, ⊥, ...))
    movement                                             */
SEARCHADJ(p, i + 1, s,
          (y_1, ..., y_{i−1}, x_i, ⊥, ...))
    /* move x_i to ⌈x_i⌉                                 */
SEARCHADJ(p, i + 1, s + (⌈x_i⌉ − x_i)^2,
          (y_1, ..., y_{i−1}, ⌈x_i⌉, ⊥, ...))
```

Algorithm 7: ADJ(p)

```
/* p = (x_1, ..., x_d) ∈ R^d                            */
q ← (⊥, ⊥, ..., ⊥) ∈ R^d
return all cells emitted by SEARCHADJ(p, 1, 0, q)
```

We now briefly discuss these results in words.

Accuracy. From Figure 5-12 we can see that the empirical sampling distributions of our algorithm are very close to the uniform distribution. This can be further supported by the results presented in Figure 15 where in all datasets, `stdDevNm` is no larger than 0.1 and `maxDevNm` is no larger than 0.2.

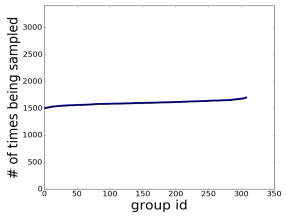

Figure 7: `Yacht` **dataset. #runs** = 500, 000

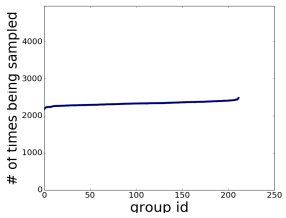

Figure 8: `Seeds` **dataset. #runs** = 500, 000

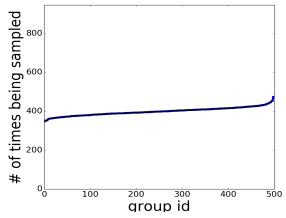

Figure 9: `Rand5-pl` **dataset. #runs** = 200, 000

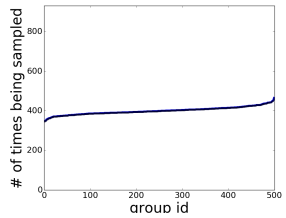

Figure 10: `Rand20-pl` **dataset. #runs** = 200, 000

Running Time. From Figure 13 we can observe that Algorithm 1 runs very fast. The processing time per item is only $1 \sim 3.5 \times 10^{-5}$ second using single thread.

By comparing the results for datasets `Rand5`, `Rand20`, `Rand5-pl` and `Rand20-pl`, we observe that the running time increases when the dimension d increases. This is due to the fact that manipulating vectors takes more time when d increases.

Space Usage. Figure 14 demonstrates the space usage of our algorithms on different datasets. We observe that our algorithm is very space-efficient and the dimension of the data points will typically affect the space usage.

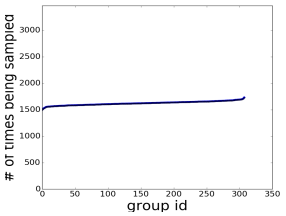

Figure 11: `Yacht-pl` **dataset. #runs** = 500, 000

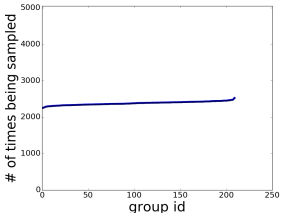

Figure 12: `Seeds-pl` **dataset. #runs** = 500. 000

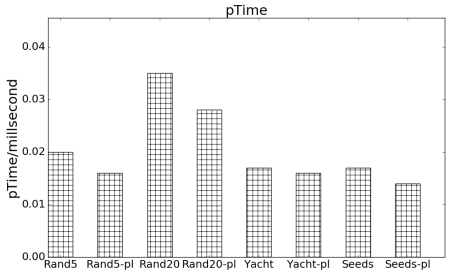

Figure 13: `pTime`

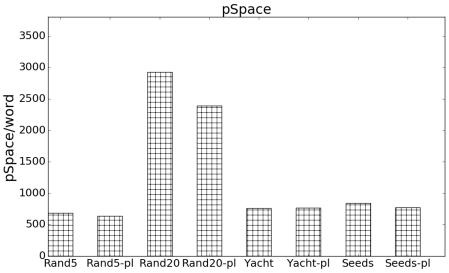

Figure 14: `pSpace`

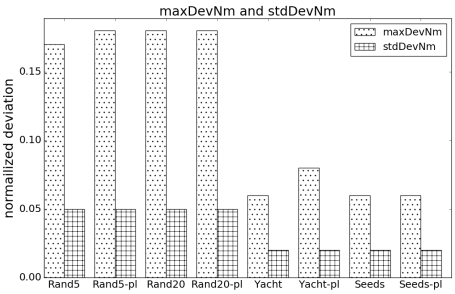

Figure 15: `maxDevNm` **and** `stdDevNm`