COMP90056 Assignment2 Report

Experimental Evaluation on L0 Sampler and Its Variants

Xiuge Chen 961392

1. Introduction

Many real-world applications produce massive data streams, such that an enormous number of item-frequency pairs $\langle c_i, f_i \rangle$ arrive rapidly. Traditional deterministic storing methods for such streams are extremely time-consuming and space-hungry, makes important properties very hard to maintain and compute. Therefore, lots of streaming algorithms are developed to use only a small amount of space to support fast but accurate updates/queries.

ℓ0 sampler is one such data structure designed to uniformly select non-zero frequency items in the stream regardless of their distribution. It has several applications, for example, calculate the "inverse distribution" in network or and database applications [1], or generate ε-nets and ε-approximations to approximate the weight of the minimum spanning trees [2].

Different ℓ0 samplers have been developed over the years. In this experiment, some of them are implemented and tested against different streams and data structure settings, to demonstrate their ability of uniform selection, as well as the differences among them.

2. Theoretical Background

2.1 k-wise and ε-min-wise independent hash family

According to lecture notes [3], ε-min-wise independent hash family is the family of hash functions H: {h: [n] → [n]}, such that for any S ⊂ [n] and x ∈ [n] \ S:

$$Pr_{h\sim H}[h(x) < min\ h(S)] = \frac{1}{|S|+1}(1\pm\varepsilon)$$

Moreover, the hash function family H is k-wise independent if and only if:

$$Pr_{h\sim H}[h(i_1) \wedge h(i_2) \wedge \cdots \wedge h(i_k)] = (\frac{1}{v})^k \ \forall (i_1, \cdots, i_k) \in U^k \ such\ that\ the\ keys\ are\ pairwise\ distinct.$$

Polynomial hashing is k-wise independent on the domain {1, ⋯ ,p}, where p is a prime number.

$$\{a_{k-1}x^{k-1} + a_{k-2}x^{k-2} + \cdots a_1 x + a_0 \mid a_{k-1}, \cdots, a_0 \in [p]\ \}$$

Each hash function constructed by k-polynomial hashing requires O(k·log(n)) bits to store, where n is the domain size. And calculating hash value could be done in O(k) time.

2.1 Insertion-only ℓ0 Sampler

ℓ0 sampling on insertion-only stream (contains only positive frequencies) could be achieved by selecting the item with the smallest hashed value, where the hash function is randomly chosen from the ε-min-wise independent hash family [3]. Indyk [4] showed that there exists a constant c>1, such that c·log(1/ε)-wise hash family is also ε-min-wise. Therefore insertion-only ℓ0 sampler could be constructed via the polynomial hashing with k=c·log(1/ε).

Since insertion-only ℓ0 sampler needs to store a hash function and the item with the minimum hash value, the space needed is O(log(1/ε)·log(n)), where ε is the error rate and n is the domain size. Update function involves calculating the hashed value and one comparison, so it could be done in O(log(1/ε)) time. Output function, however, will directly output stored item, thus it requires just O(1) time.

2.2 One and K Sparse Recovery

A stream could be represented as a vector, where each coordinate stores the total frequency of one item, a vector is k-sparse if it has k or fewer non-zero coordinates.

1-sparse recovery algorithm is proposed to determine whether a given vector is 1-sparse, it outputs either a message indicates there is 0 or more than 2 items, or the single item and its frequency. For turnstile stream (the total frequencies of each item is greater than 0), 1-sparse recovery could be built by keeping 3 variables w_1, w_2 and w_3, such that $w_1 = \Sigma f_i$, $w_2 = \Sigma f_i \cdot c_i$, and $w_3 = \Sigma f_i \cdot c_i^2$. During the output stage, it will calculate whether $w_1 \cdot w_3 = w_2^2$. This approach could not handle negative frequencies, Cormode [5] proposed another approach that replaces w_3 with $w_3 = \Sigma q^i \cdot f_i \bmod p$, for large prime p and random $q \in [p]$. One sparsity could be checked by calculating whether $w_3 = w_1 \cdot q^{w2/w1} \bmod p$.

As constructed above, 1-sparse recovery algorithms require $O(\log(n)+\log(F_1))$ bits (Turnstile) or $O(\log(n)+\log(F_1)+\log(p))$ bits (Cormode's) to store, where n is the domain size, F_1 is the total frequency of all items. Update/output could all be done in O(1) time since they both require only constant times of calculation. Cormode's one sparse recovery might output false-positive results if w_3 is a root, so it has a false rate probability of n/p.

Cormode [5] gave a k-sparse recovery algorithm with successful probability 1-δ based on 1-sparse recovery and pairwise hash family H: {h: [n] → [2k]}. It uses log(k/δ) randomly chosen hash functions and a 2-dimensional array with dimension (log(k/δ), 2k), each cell maintains a 1-sparse recovery structure. For each item-frequency pair $\langle c,a \rangle$ and all of the rows r_i, cells at $\langle r_i, h_i(c) \rangle$ will be updated. Lastly, it will go through all rows, extract unique items stored (if exist) there and obtain an estimation of total items. If all estimations are no greater than k and k or fewer items are extracted successfully in some rows, return them.

The space required by k-sparse recovery algorithm depends on the type of one sparse recovery used. If turnstile version of one sparse recovery is used, it needs $O(\log(k/\delta) \cdot 2k \cdot (\log(n)+\log(F_1)) + 2\log(n)) = O(k \cdot \log(k/\delta) \cdot (\log(n)+\log(F_1)))$ bits. If the general (Cormode) version is used, it needs $O(k \cdot \log(k/\delta) \cdot (\log(n)+\log(F_1)+\log(p)))$ bits. Each update requires $O(\log(k/\delta))$ time, while each output needs $O(2k \cdot \log(k/\delta)) = O(k \cdot \log(k/\delta))$ time.

2.3 General Stream ℓ0 Sampler

Based on the k-sparse recovery algorithm above, Cormode [5] proposed a general ℓ0 sampler that could deal with arbitrary stream using t-wise independent hash functions H: {h: [n] → [n^3]}. It will initialize appropriate k and t values, and log(n) levels of k-sparse recovery structure. When updating, each item is mapped from level 1 to level j with probability $1/2^j$, so that the stream will become k-sparse at some level. Then, it will go through each level, find the level that is k-sparse and items stored in this level, output the item with the smallest hashed. If unfortunately none of the levels gives k-sparse output, this algorithm might fail.

In order to achieve uniform selection (output item with probability $(1\pm\varepsilon)\frac{1}{N}\pm\delta$) and succeed probability of 1-δ, k and t should be initialized as $O(\log(1/\delta)+\log(1/\varepsilon))$ and O(k). So the space required is $O(\log(n) \cdot k \cdot \log(k/\delta) \cdot (\log(n)+\log(F_1)+\log(p)) + k \cdot \log(n)) = O(\log(n) \cdot k \cdot \log(k/\delta) \cdot (\log(n) + \log(F_1) + \log(p)))$, or $O(\log(n) \cdot k \cdot \log(k/\delta) \cdot (\log(n) + \log(F_1)))$ if using turnstile k-sparse recovery. Each update and output will go over at most log(n) levels of k-sparse recovery, thus, update needs $O(\log(n) \cdot \log(k/\delta))$ time, output needs $O(\log(n) \cdot k \cdot \log(k/\delta))$ time.

2.4 Distinct Element

Flajolet-Martin (FM) algorithm [6] is one of the distinct element algorithms that estimate the number of distinct items. It uses a bitmap to memorize the least-significant bit in the binary representation of items, and take the smallest index to estimate the size of distinct elements.

2.5 ℓ0 Sampler using Pairwise Independent Hash Function

Frahling [6] used pairwise independent hash functions and an extra distinct element estimator DE to construct ℓ0 sampler. Similarly, items are mapped to level j out of log(n) levels with probability 2^j, at each level j, O(log(1/δ)/ε) 1-sparse recoveries and pairwise hash functions H: { h: [n] → [T] } are maintained, where T = [2^{j+1} / ε]. For each update, this sampler will firstly update each cell's one sparse recovery if the hashed value of this item is equal to 0, then it will update DE. When outputting, it uses DE to locate the appropriate level, then outputs the first item obtained from one sparse recovery. If none of the items is recovered, the algorithm will fail.

As described above, this algorithm will use O(log(n) · log(1/δ) / ε · (log(n)+log(F_1)+log(p))) bits, O(log(n)· log(1/δ) / ε) time to update, and O(log(1/δ) / ε) to output. It takes more space and update time than the general ℓ0 sampler, but achieves simpler hash functions and faster output.

2.6 Baseline

To measure the benefits of ℓ0 samplers over the real uniform selections, an O(n·log(F1)) bits "baseline" approach is introduced, which simply uses an array to memorize all updates, then uniformly output one non-zero frequency item. Each update only requires O(1) time because the array could be accessed through the index. The output might need O(n) time since it needs to determine the set of non-zero frequency items before selection.

	ℓ0 Insertion-only	ℓ0 Turnstile	ℓ0 General	ℓ0 Pairwise	Baseline
Memory/ Space	O(log(1/ε) · log(n))	O(log(n) · k · log(k/δ) · (log(n) + log(F_1))) k=O(log(1/δ)+log(1/ε))	O(log(n) · k · log(k/δ) · (log(n)+log(F_1)+log(p))) k=O(log(1/δ)+log(1/ε))	O(log(n) · log(1/δ) / ε · (log(n)+log(F_1)+log(p)))	O(n·log(F1))
Update Time	O(log(1/ε))	O(log(n) · log(k/δ)) k=O(log(1/δ)+log(1/ε))	O(log(n) · log(k/δ)) k=O(log(1/δ)+log(1/ε))	O(log(n)· log(1/δ) / ε)	O(1)
Output Time	O(1)	O(log(n)·k·log(k/δ)) k=O(log(1/δ)+log(1/ε))	O(log(n)·k·log(k/δ)) k=O(log(1/δ)+log(1/ε))	O(log(1/δ) / ε)	O(n)
Fail rate	0	δ	δ	δ	0

Table 2-1: Theoretical comparison among L0 sampler, its variants and baseline sketches

3. Implementation

Java implementation is chosen because there are many useful java libraries to generate artificial stream data, such as *Commons Math* [8]. Moreover, to monitor the actual space usage, software like *VisualVM* [9] could be used to monitor the heap allocation.

All data structures are implemented as described, variables like w_3 in the 1-sparse recovery are stored as Long primitive to avoid overflow. Also, the memory-efficient method [10] is implemented to address the overflow problem of modular exponentiation calculation.

4. Evaluation Metrics

4.1 Space and Time Usage Comparison

VisualVM [9] is used to monitor the actual memory usages. And the start/end time (in nanoseconds) of different methods is recorded, so that their average time difference could be used as a rough estimation of time complexity.

4.2 Output Uniformity

If there are S non-zero frequency items and *N* experiments, uniformity could be measured as the difference between expected and actual selection distribution, Cormode [5] used standard deviation and maximum deviation as his metric. For the selected times of item i, its deviation d_i from expected value could be calculated as follows:

$$d_i = \frac{|f_i - f_i^*|}{f_i^*},\ f_i^* = \frac{N}{|S|}$$

The average deviation of all items *D* and the 1-δ percentile max deviation are chosen to measure the uniformity, since they could help estimate not only the general performance of different algorithms, but also the worst performance within the guarantee bound.

4.3 False and Error Rate

ℓ0 samplers like insertion-only ℓ0 sampler might mistakenly output deleted items, and most ℓ0 samplers have a small probability of fail. These two terms, error rate E_i and fail rate F_i, are also very critical aspects of the ℓ0 samplers, and could be calculated as follows:

$$E_i = \frac{|error\ output|}{N},\ F_i = \frac{|fail\ output|}{N},\ N\ is\ the\ experiments\ are\ performed$$

5. Experiment Preparation

5.1 Stream Data Generation

In order to test ℓ0 samplers on all kinds of streams, different artificial data is generated as below.

Name	Distribution	Stream type
UNI-I	Uniform	Insertion-only with f_i = [1,10]
Z03-I	Zipf, s=0.3	Insertion-only with f_i = [1,10]
Z06-I	Zipf, s=0.6	Insertion-only with f_i = [1,10]
Z09-I	Zipf, s=0.9	Insertion-only with f_i = [1,10]
Z09-T	Zipf, s=0.9	turnstile with f_i = [-10,10]
Z09-G	Zipf, s=0.9	general with f_i = [-10,10]
Z12-I	Zipf, s=1.2	Insertion-only with f_i = [1,10]
Z15-I	Zipf, s=1.5	Insertion-only with f_i = [1,10]

Table 5-1: Generated stream data

Zipf distribution produces skewed data, therefore is very suitable to test whether ℓ0 samplers could uniformly select items. Some modifications are also made to cover the corner/tricky cases, for example, in turnstile stream, frequencies of a few items are reduced to 0 (deletion) after increasing, in general stream, some special frequencies are included to make turnstile 1-sparse recovery output false-positive results, such as the sub-vector 〈5, 5, -1〉.

6. Result and Discussion

Since domain size does not impact the evaluation of uniformity and smaller domain usually require less time to test, all items are drawn from a relatively small universe U = [10^2]. The error rate *ε* the bad probability *δ* are fixed as 0.01, 10^4 experiments are performed.

6.1 Space and Time Usage

To better observe the space consumption differences among samplers, all space/time results are recorded from experiments with the universe of size 10^6 and stream length 10^7, the memory usage of data-irrelevant filed such as ClassLoader is ignored. The output time of the baseline algorithm is not tested because it takes too long to complete.

	ℓ0 Insertion-only	ℓ0 Turnstile	ℓ0 General	ℓ0 Pairwise	Baseline
Memory / bits	3,264	7,297,280	8,510,720	3,836,400	32,000,000

Table 6-1: Space usage of different ℓ0 samplers

Time / Nanoseconds	ℓ0 Insertion-only	ℓ0 Turnstile	ℓ0 General	ℓ0 Pairwise	Baseline
Update	16672.7588	3537.3246	5488.2549	117307.1947	104.7324
Output	42.8069	3992777.5270	5052732.2861	2364.3272	Too long

Table 6-2: Average time cost of different ℓ0 samplers

As shown above, most of the actual space/time usage is aligned with the theoretical analysis, turnstile ℓ0 sampler is slightly quicker than the general one because w_3 in the general 1-sparse recovery is slower to calculate. The only exception is that space/time required for general/ turnstile ℓ0 samplers are larger than pairwise ℓ0 sampler, it is because they assigned k as 12 * log(1/δ), which is much larger than 1/ε. Section 6.4 gives a further discussion about the possibility of reducing their space usages.

6.2 Stream Length and Distribution

This test evaluates whether stream length and distribution affects the overall performances. Theoretically, as long as the domain size is consistent, the sampling results should not be affected by stream length and distribution. UNI-I, Z03-I, Z06-I, Z09-I, Z12-I, and Z15-I with fixed stream length of 10^4 are selected to perform tests on stream distribution. And to test the impact of stream length, Z09-I dataset is chosen with different stream length L = [10^3, $5{\cdot}10^3$, 10^4, $5{\cdot}10^4$, 10^5, $5{\cdot}10^5$, 10^6].

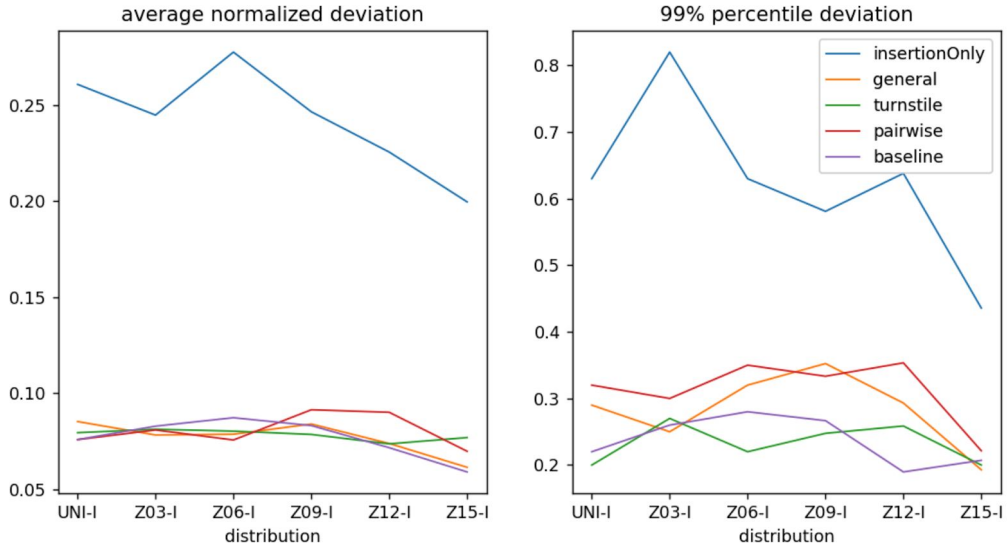


Figure 6-1: ℓ0 sampler evaluation on stream distribution

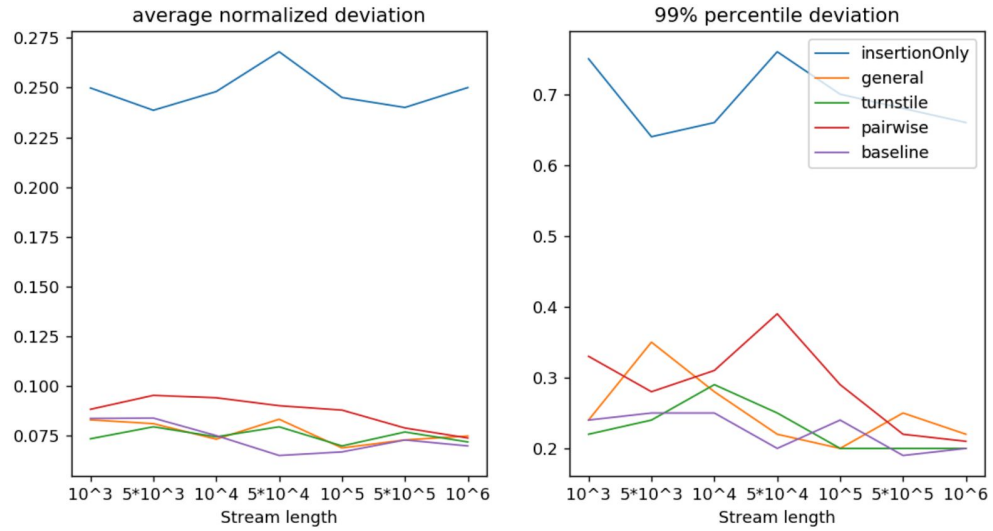
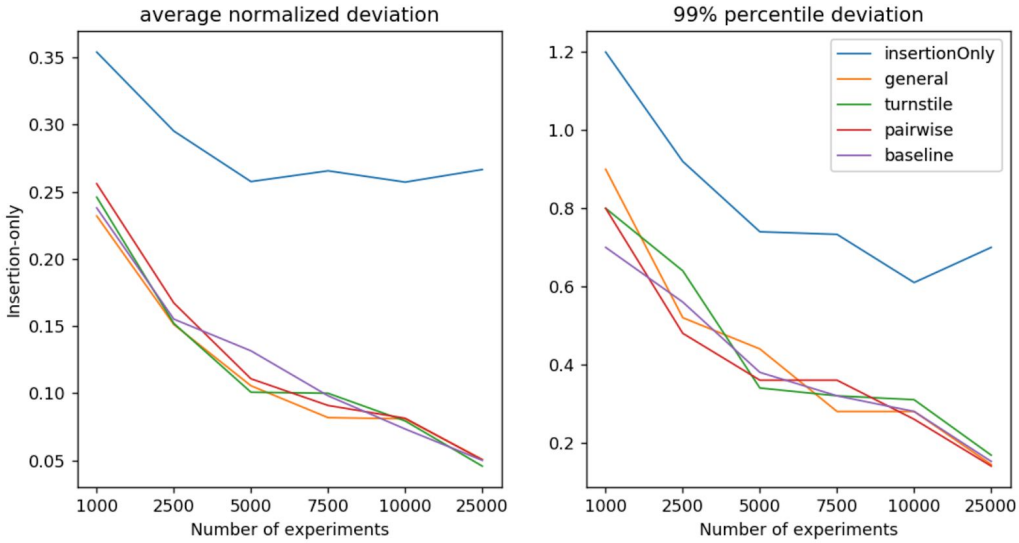


Figure 6-2: ℓ0 sampler evaluation on stream length

The results of different ℓ0 samplers are very consistent across different distribution, even for every skewed sampled data like Z15-I, its results are very close to uniformly sampled data UNI-I. Therefore, as expected, stream length and distribution will not affect ℓ0 samplers, thus in the following experiments, they will be fixed to some values.

6.3 Stream Type and Number of Experiments

The second test focuses on the influence of the stream type and the number of experiments performed. Ideally, if items are uniformly selected, as the number of experiments increases, the result distribution should become closer to expected uniform distribution, so that the deviation of each item should be smaller. Moreover, although different ℓ0 samplers are designed to handle different types of stream, they could also be applied in other kinds of stream with maybe some extra error outputs. Therefore, it is very important to understand the performance difference among different samplers, as well as the cost of using samplers on streams that they are not designed for. Z09-I, Z09-T and Z09-G dataset are selected with length 10^4.



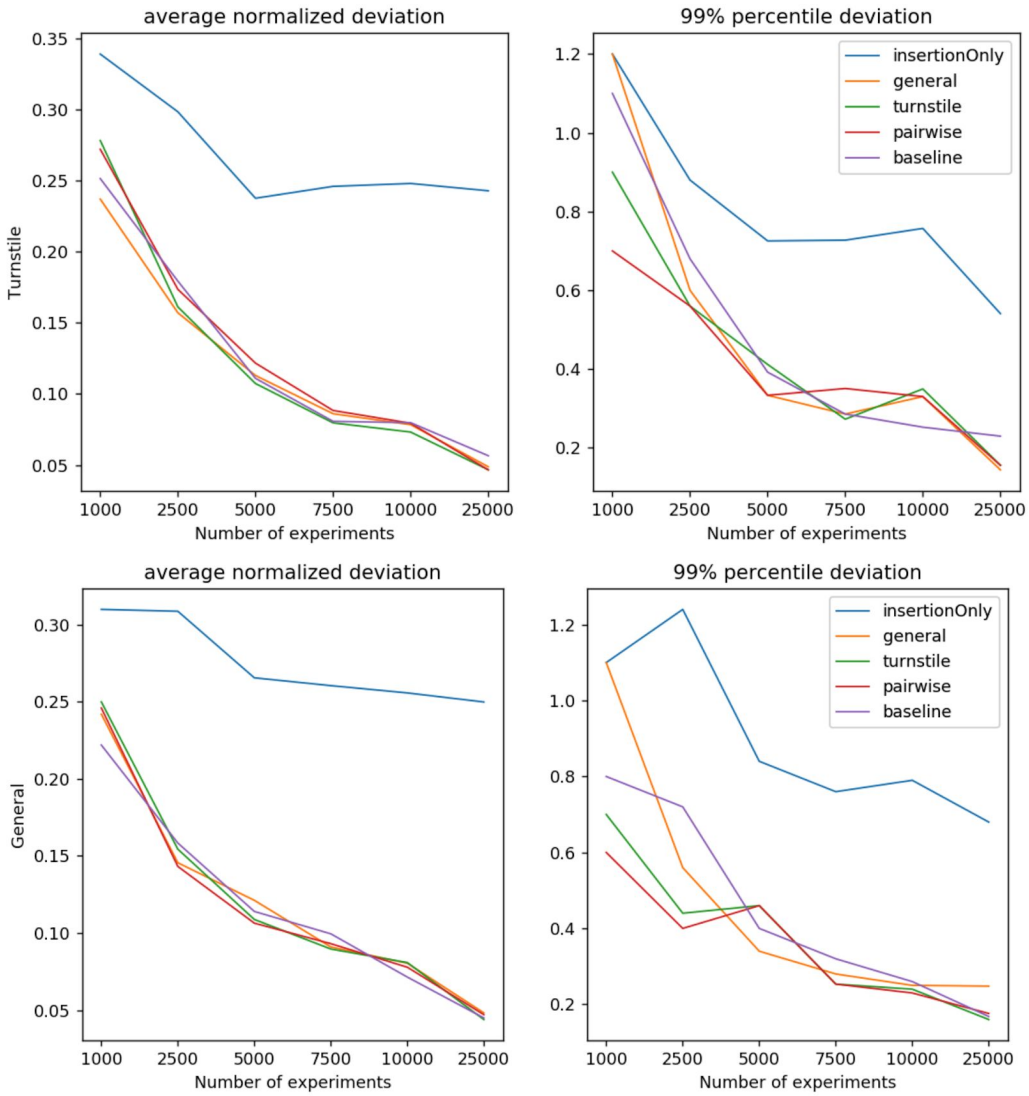


Figure 6-3: ℓ0 sampler evaluation on stream type and number of experiments

From the figures above, it is not surprising that as the number of experiments increases, both average and 99% deviation reduces significantly, the results are closer to uniformity. Moreover, the results of most samplers are very close to the baseline approach, which means their ability to uniformly select items is as good as the true uniform sampler. Insertion-only ℓ0 sampler, however, does not perform that good, potentially because the c parameter of c·log(1/ε)-wise hash function is too small, so that it could not fully replace the ε-min-wise hash function.

The results of turnstile ℓ0 samplers on different streams do not vary a lot, since the probability that all elements of special designed sub-vector ⟨5, 5, -1⟩ fall into one 1-sparse recovery is fairly small. However, turnstile/general stream will make insertion-only ℓ0 sampler output deleted item with probability around 0.03, because as mentioned in section 2, insertion-only ℓ0 sampler could not handle deletion and the generated data has approximately 3 items being deleted. All of the samplers' fail rate are way below 0.0001, because the associated parameter is chosen fairly large here, section 6.3 will give a detailed analysis of parameter choices.

6.4 ℓ0 Sampler Parameter Test (Space vs Performance)

Although theoretical analysis provides some guidance of the parameter choices (k of k sparse recovery, t of t-wise hash functions, number of repetitions...) of different ℓ0 samplers, sometimes it could result in a very large data structure. Therefore, this test evaluates the influence of different parameter settings, could these ℓ0 samplers achieve similar performance with smaller space (also shorter time)? Parameter c from c·log(1/ε)-wise hash function has a great influence on the space usage of insertion-only ℓ0 sampler, so it is tested against c=[1, 1.5, 2, 2.5, 3, 3.5, 4] on Z09-I dataset with length 10^4. For turnstile/general ℓ0 sampler, k from k-sparse recovery influences the most of space usage, so it is tested against k=[6, 12, 18, 24, 30, 36, 42] on Z09-G dataset with length 10^4. Lastly, the number of repetitions that pairwise ℓ0 sampler keeps contributes a lot in space, therefore, pairwise ℓ0 sampler is tested against number of repetitions r=[100, 200, 300, 400, 500, 600, 700] on Z09-G dataset with length 10^4.

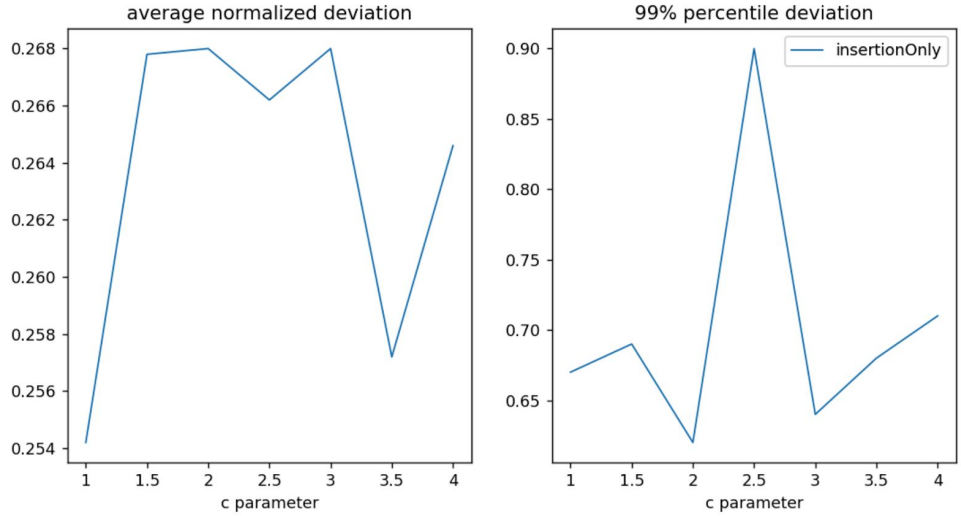


Figure 6-4: insertion-only ℓ0 sampler evaluation on parameter settings

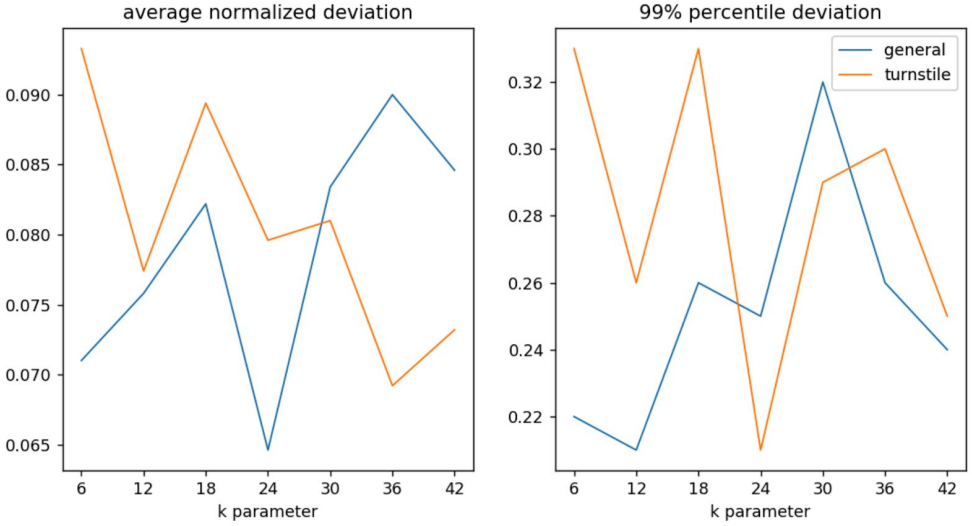


Figure 6-5: turnstile/general ℓ0 sampler evaluation on parameter settings

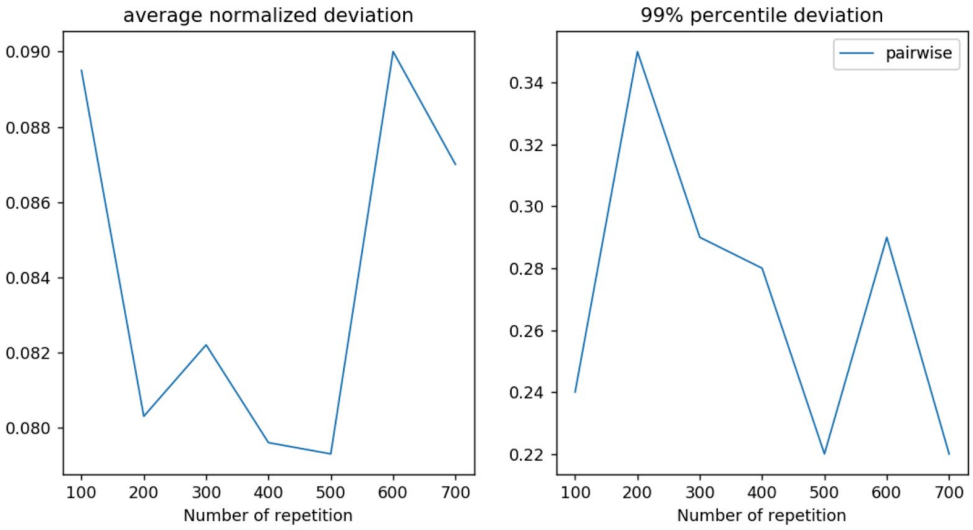


Figure 6-6: pairwise ℓ0 sampler evaluation on parameter settings

According to the figures above, the performances do not decrease a lot when the size of ℓ0 samplers reduced. However, the failure rates are influenced by the size of samplers because they break the guarantee bound. For example, fail rate of pairwise ℓ0 sampler is being reduced from 0.05 to 0 when increasing r from 100 to 600, fail rate of turnstile/general ℓ0 samplers are being reduced from 0.01 to 0 when increasing k from 6 to 12.

To achieve similar performance with nearly zero fail rates, pairwise ℓ0 sampler need to use 600 repetitions and turnstile/general ℓ0 samplers only need to set k as 6. Turnstile/general ℓ0 samplers with k=6 uses at most 173,568 bits, 2279.6319 nanoseconds to update and 327144.5639 nanoseconds to output, while pairwise ℓ0 sampler with r=600 requires 1,213,079 bits to store, 106552.7148 nanoseconds to update and 2654.8924 nanoseconds to output. In this sense, the space/time usage of these three samplers is aligned with the theoretical analysis, pairwise ℓ0 sampler uses more space than turnstile/general ℓ0 samplers.

6. Conclusion

To sum up, all of the ℓ0 samplers are very effective and could successfully select stream items based on their appearance rather than distribution, most of them are even able to select items as uniformly as the true uniform sampler, but only use a small amount of space and time. This result is consistent regardless of the stream length and distribution.

Insertion-only ℓ0 sampler uses the smallest space and time, also it is the simplest to implement, but its output distribution is slightly bias (nearly but not fully uniform). Therefore, it is very suitable for applications that have strict requirements on space/time usage, but loose demand on the uniformity of results. However, it might mistakenly output deleted items (treated deleted items as normal), thus should not be used if there are a lot of deletions in the stream.

Turnstile/general ℓ0 samplers are almost the same except their choice about 1-sparse recovery. They both could achieve better output uniformity than insertion-only ℓ0 sampler, but with the price of more space, slower time and implementation of k-wise independent hash family. Moreover, experiment indicates that appropriate reduction from the suggested size will not impact their overall performances, the fail rate will increase only after decreasing too much.

Pairwise ℓ0 sampler replaces the k-wise hash functions with the simpler pairwise hash functions and still could achieve similar performance. But it usually requires more space (more repetition) to execute. Unlike turnstile/general ℓ0 samplers, the update time of pairwise ℓ0 sampler is much

slower than the output time. Therefore, besides simpler hash functions, the quicker output is another advantage of pairwise ℓ0 sampler. Similarly, appropriate size reduction could be applied to pairwise ℓ0 sampler as long as the fail rate does not increase.

7. References

[1] Cormode, G., Muthukrishnan, S. and Rozenbaum, I., 2005, August. Summarizing and mining inverse distributions on data streams via dynamic inverse sampling. In *Proceedings of the 31st international conference on Very large data bases* (pp. 25-36). VLDB Endowment.

[2] Frahling, G., Indyk, P. and Sohler, C., 2008. Sampling in dynamic data streams and applications. *International Journal of Computational Geometry & Applications*, *18*(01n02), pp.3-28.

[3] Holland, W. and Wirth, T. (n.d.). *Lecture Notes Weeks 1-9*. [online] Melbourne: University of Melbourne. Available at: https://app.lms.unimelb.edu.au/bbcswebdav/pid-7758713-dt-content-rid -67485349_2/courses/COMP90056_2019_SM2/90056_lnotes_w1-w9.pdf [Accessed 14 Sep. 2019].

[4] Indyk, P., 2001. A small approximately min-wise independent family of hash functions. *Journal of Algorithms*, *38*(1), pp.84-90.

[5] Cormode, G. and Firmani, D., 2014. A unifying framework for ℓ0-sampling algorithms. *Distributed and Parallel Databases*, *32*(3), pp.315-335.

[6] Flajolet, P. and Martin, G.N., 1985. Probabilistic counting algorithms for data base applications. *Journal of computer and system sciences*, *31*(2), pp.182-209.

[7] Frahling, G., Indyk, P., Sohler, C., 2005. Sampling in dynamic data streams and applications. Symposium on Computational Geometry, pp. 142–149

[8] Commons Math. (2016). Apache Commons.

[9] Sedlacek, J. and Hurka, T. (2019). *VisualVM*. Oracle.

[10] En.wikipedia.org. (2019). *Modular exponentiation*. [online] Available at: https://en.wikipedia.org/wiki/Modular_ exponentiation [Accessed 20 Oct. 2019].