ℓ_0 Sampling and Sparse Recovery

Student Number: 871089
Student Name: Zhuohan Xie

1. Introduction

The aim of this report is to achieve ℓ_0 sampling with help of s-sparse recovery in the streaming dataset where the frequency of the item could be negative. The report separated this scheme into two parts, Sparse recovery and ℓ_0 sampling, discussed design choices and analysed experiment results separately, then applied a good choice in the code. Also, it analysed the application domain of this code and proposed future improvements.

2. Test Dataset

I generated multiple files with different features for various test domain. Here are details of some of them which give me interesting feedback.

Table 1: Test dataset detail

File Name	Test Domain	Expected Return	Stream Entries	Comments
Testdata1	One-Sparse Recovery	One sparse recovery	50000	
Testdata2		"zero"	50000	
Testdata3		"more"	87642	
Testdata4	S-Sparse Recovery	S sparse recovery	60000	Entries in random, 20 sparsity
Testdata5		S sparse recovery	60000	Entries in sequence, 3000 sparsity
Testdata6	Insert-Only ℓ_0 sampling	One sample item	60000	60 items, turnstile stream
Testdata7	Dynamic ℓ_0 sampling	One sample item	40210	Universe of 40, general stream with deletion
Testdata8		One sample item	502391	Universe of 50, general stream with deletion

3. Sparse Recovery

3.1 One-Sparse Recovery

I implemented what is proposed in [1] so as to deal with negative frequency, which can fool Ganguly's test. The only design choice is prime number p, I set p as 5 different prime number and run the same input files (Testdata1, Testdata2 and Testdata3) 1000 times to check the failure rate and total running time to figure out the influence of p regrading accuracy and running time. In one sparse, only some certain values need to be stored and changing p would only impact little space, therefore, consumed space is not considered here.

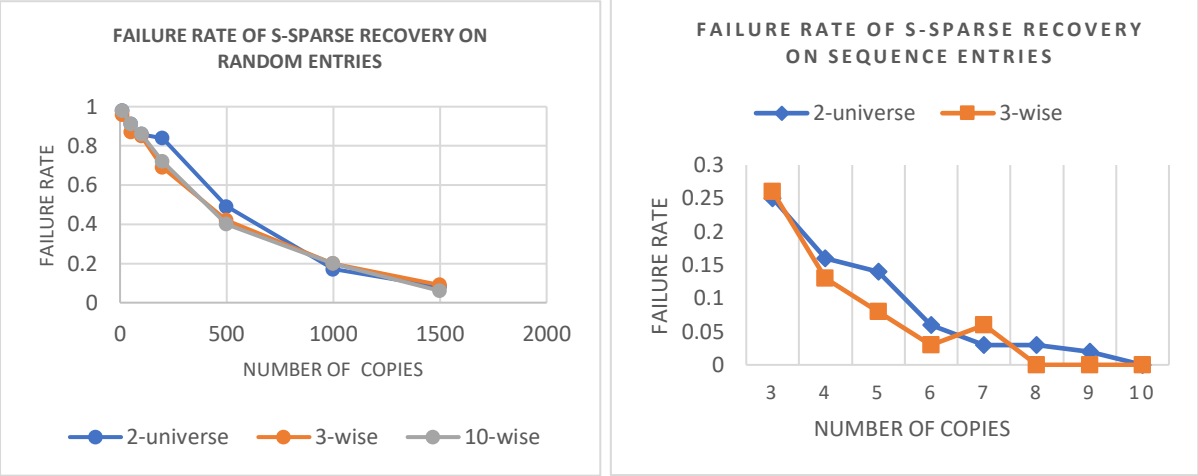
One interesting thing is that size of p has nothing to do with accuracy, we can save time by setting p as a smaller prime, however, I did get into trouble with BigInteger when setting p as a very small prime as 5. therefore, I believe it's reasonable to set p as 6619 to guarantee a relatively fast speed guarantee the performance.

Table 2: Exploration of One-Sparse Recovery

p	Failurerate1 (%)	Failurerate2 (%)	Failurerate3 (%)	Running Time (ms)
17	0	0	0	1400128
107	0	0	0	1567925
6619	0	0	0	1857263
811955629	0	0	0	5411927
1073741789	0	0	0	5523728

3.2 S-Sparse Recovery

For S-sparse recovery, I implemented in the approach was taught in the lecture, the design choices here could copy numbers and hash functions. Clearly, in my implementation, it's unlikely to make mistakes when result is "more" or "zero" because I only count number of vectors when no collision and return "more" otherwise. Therefore, the interesting part of my implementation is to test whether it can recover s-sparse successfully and I used Testdata4 and Testdata5 to explore trade-offs between copy numbers and failure rate. Also, as proposed in the paper, pair-wise hash function is required to reduce collision, therefore, I chose different k-wise hash functions to evaluate whether they could behave like pair-wise. As shown in Graph 1, k-wise hash functions would not behave differently by just changing p and increasing copy number can reduce the failing rate with the sacrifice of more time and space. It's good to choose 500 copies to guarantee success rate for one s-sparse recovery scheme. Besides, one interesting thing I noticed is that low level independent hash function behaves extremely well when the entries are in the sequence.

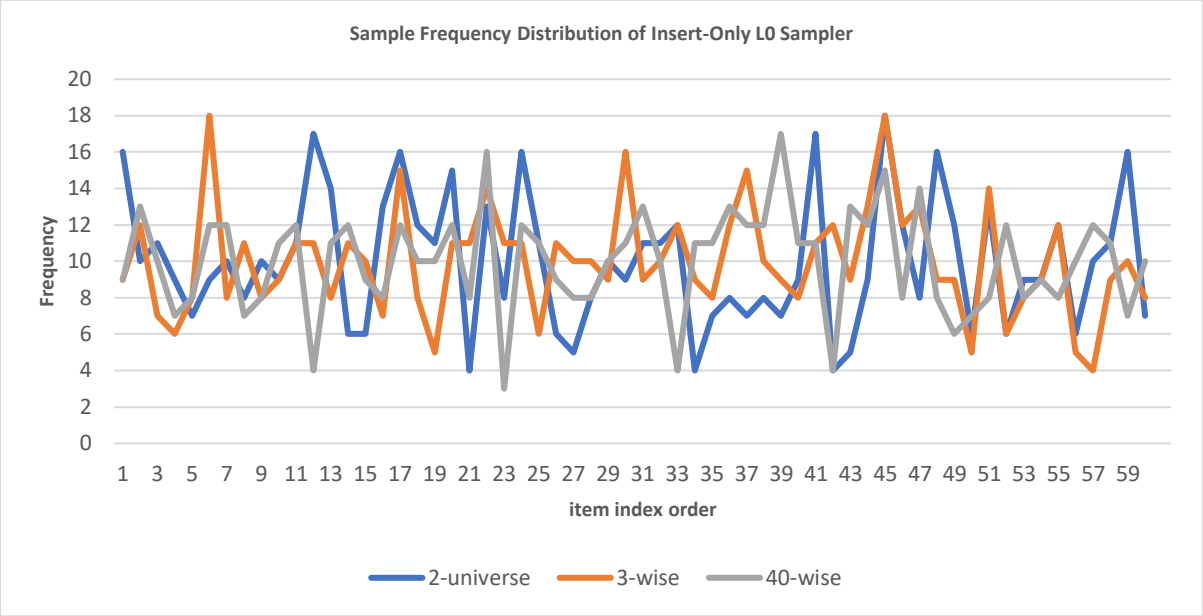


Graph 1: Exploration of S-Sparse Recovery

4. ℓ_0 Sampling

4.1 Insert-Only ℓ_0 Sampling

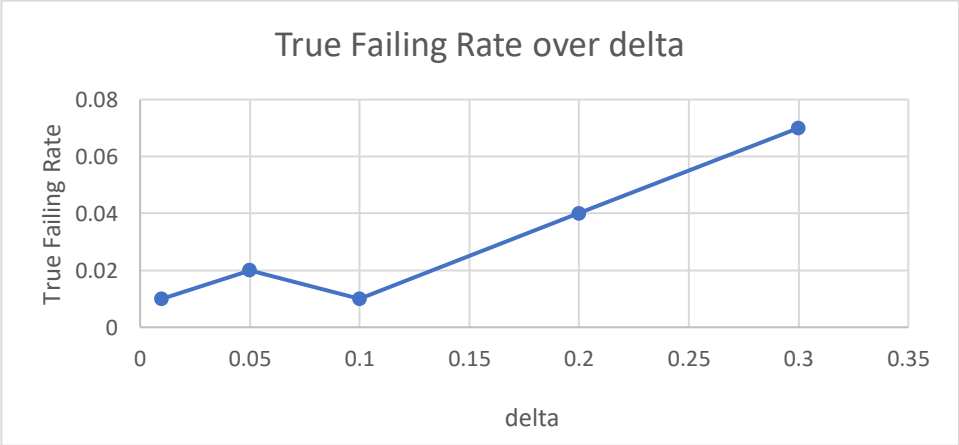
The way I implement Insert-Only ℓ_0 sampling is just taking the minimum hash values of all entries' items and store the corresponding <item, frequency> pair for returning result. In this case, each item has the same probability of being hashed to the smallest value regardless of its frequency. The important thing of this approach is to make sure the hash function can uniformly distribute all items and the collision is rare to happen, especially no more than one item should have same minimum hash value. By running Testdata6 which contains 60 coordinates whose indexes are generated randomly 600 times, I generated the sample frequency distribution as below. It seems that 40-wise hash function outperforms since it has more values closer around 10, which is the expected average frequency here.



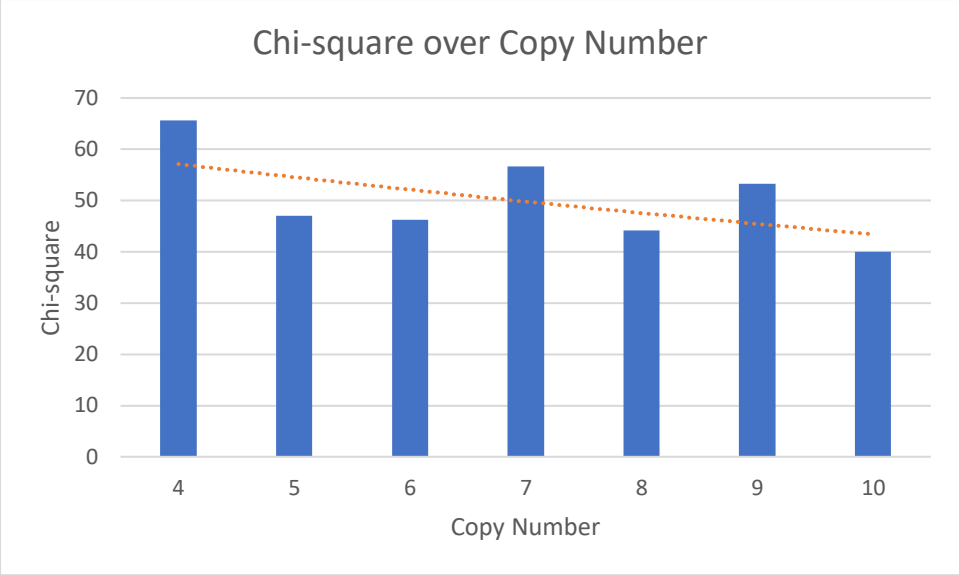
Graph 2: Exploration of Insert-Only ℓ_0 sampling

4.2 Dynamic ℓ_0 Sampling

Dynamic ℓ_0 sampling requires δ as failing rate and needs to perform sparse recovery with s = $\left\lceil 4\log(\frac{1}{\delta}) \right\rceil$. Also, a k-wise independent hash function is required for sampling process where $k \geq \frac{s}{2}$. Clearly, the true failing rate can be influenced by other parameters such as copies of S-sparse recovery. However, in my experiment, increasing copies will not lower failing rate much because once one subset is successfully recovered, the coordinate we want to sample can be returned from that subset. And in subsets with small number of coordinates, it's unlikely for hash function to cause collision. Combining Graph 3 and 4, true failing rate will increase with δ increasing and chi-square will drop when copy number grows, but not too much, therefore, it's reasonable to set δ as 0.01 and copy number as 4 to get a good performance and save time and space.



Graph 3: Exploration of true failing rate over δ



Graph 4: Exploration of true chi-square over copy number

5. Discussion

In this project, five java classes are implemented so as to achieve a nearly uniform dynamic ℓ_0 sampling, which can be applied in a variety of domains within computational geometry, graph algorithms and data analysis [1]. Since in the real life, the entries of data can be extremely large with deletion and we may want to sample from this regardless of frequency without consuming too much space. My code can be used as a basic sense of how these algorithms can be implemented and applied in simple datasets.

6. Conclusion

The report discussed design choices and analysed corresponding experiment results of dynamic ℓ_0 sampling in two major parts (Sparse Recovery and ℓ_0 sampling) separately and implemented one with reasonably good performance in the code. Potential future improvement would be to find a pair-wise independent hash function for s-sparse recovery, also, test datasets with more features can be applied to test robustness of this scheme.

Reference

[1] Cormode, G., & Firmani, D. (2014). A unifying framework for ℓ 0-sampling algorithms. Distributed and Parallel Databases, 32(3), 315-335.