Neo Han Wei A0086731B E0267380

1. Introduction

The main goal of this experiment is to compare two different streaming algorithms for counting the number of times an item appears in a stream. We want to approximate the frequency of each distinct element in the stream given a limited amount of space.

We will be testing the two algorithms using 2 input streams, randomly generated from uniform and exponential distribution. As for accuracy, I will be using a simple Mean Absolute Error (MAE) metric for our purpose. To reduce the noise generated by any randomness, I getting the average of the MAE by running 100 iterations.

We will compare the 2 algorithms' performance under different conditions which will be described in the tests later.

Recommendations (assuming everything else is constant unless stated):

1. **Algorithm 1 performs better for exponentially distributed samples.**
2. **Algorithm 2 performs better for uniformly distributed samples.**
3. **Increasing the sample size N can improve the accuracy for algorithm 2 for uniformly distributed samples. (converges to uniform distribution)**
4. **Increasing space usage (A * B) can improve the accuracy for both algorithms for exponentially distributed sample. However, space and accuracy trade-off is better for algorithm 2 (more room for improvement).**
5. **Increasing space usage (A * B) is not helpful for algorithm 2 for uniformly distributed sample.**
6. **However, increasing space usage (A * B) will be helpful for algorithm 1 for uniformly distributed sample.**

2. Implementation

I will be using Python as my programming language since I am more familiar with it. I am also not that concerned about the speed but more particular about the accuracy of my test results. I have written a class called *Sampler* which will store some input parameters to be elaborated below. Inside *Sampler*, there are also several methods such as *query1*, *query2* and *result*. Full code that I wrote by myself can be found on my github if needed.

2.1 Hash Functions

I have implemented a prime field hash function which is defined as:

$$f(x) = ((ax + b) \bmod p) \bmod M$$

Where *a* and *b* are two random integer *a∈[1, p-1], b∈[0, p-1*]. *p* is predetermined prime number which has to be bigger than *M* which I interpret as the number of counters.

I am using a fixed *p*. However for *a* and *b*, I store these two parameters in two lists of length *A* which is the number of hash functions. They are stored in *self.a* and *self.b*. So for each hash function in *A*, they contain the same *p* but may not have the same *a* and *b*.

2.2 Counters

The counter is a 2-D matrix, labelled as *C*, of dimension *A* x *B*. I initialise this matrix by using *np.zeros((self.A, self.B))*. Depending on which distribution we are using, I will first randomly generate data from that distribution and *populate* the matrix *C* by incrementing based on the instructions.

2.3 Two Algorithms

I implemented these two algorithms under two methods called *query1* and *query2*.

- *Sampler.query1(x)* will return the median value of the counters that x is mapped to.
- *Sampler.query2(x)* will return the median value of the estimates which is defined as the counter minus its neighbour for each row of *C*.

2.4 Multiple Iterations

As mentioned in the introduction, I will be running multiple iterations (100) to find the average MAE to reduce the noise of the result. It will be implemented as such:

- Initialise two lists, *keep_mae_query1* and *keep_mae_query2*.
- For 100 iterations:
  - Randomly populate a new *C* based on the stated distribution.
  - Run *query1* and append result to *keep_mae_query1*.
  - Run *query2* and append result to *keep_mae_query2*.
- Return the two averages of *keep_mae_query1* and *keep_mae_query2.*

3. Test Design

3.1 Parameters

To run the experiment, we will be changing some parameters and then compare the accuracy.

Parameters (to be adjusted):

1. *A*, number of hash functions. *A ∈ {10, 15, 25}*
2. *B*, number of counters per hash function. *B ∈ {14, 18, 30}*
3. *M*, number of distinct integers we will randomly generate. *M ∈ {10, 15, 25}*
4. *N*, size of data stream. *N ∈ {1000, 2000, 5000}*

3.3 Accuracy

As mentioned, I will be using Mean Absolute Error or *MAE* for short. It is implemented in the following way:

MAE = np.mean(np.absolute(actual count - expected count))

The rationale for using MAE is that I just want a simple metric to tell me how well the algorithms estimate the count of the distinct integers. A metric like MSE will over-penalise counts that are far from the expected count with the squared term.

3.3 Distributions

As mentioned we are using two distributions for our experiments.

- Uniform distribution: I used numpy implementation, np.random.uniform.
- Exponential distribution: I used numpy implementation, np.random.uniform. The scaling factor is set at 1/-ln(0.5) to give us a distribution set in the instructions.

4. Test Results

4.1 Comparing Distributions

For this section, I will fix *A* and *B* and examine the accuracy of the two algorithms for different distributions and for different *M* and *N*. We also want to see which algorithm does better with more/less samples (*N*) and more/less distinct numbers (*M*).

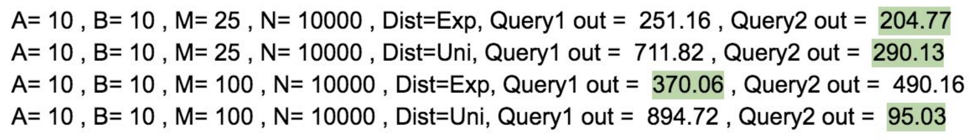
```
A= 10 , B= 10 , M= 10 , N= 1000 , Dist=Exp, Query1 out =  5.82 , Query2 out =  26.7
A= 10 , B= 10 , M= 10 , N= 1000 , Dist=Uni, Query1 out =  32.4 , Query2 out =  68.23
A= 10 , B= 10 , M= 10 , N= 2000 , Dist=Exp, Query1 out =  67.03 , Query2 out =  28.16
A= 10 , B= 10 , M= 10 , N= 2000 , Dist=Uni, Query1 out =  107.97 , Query2 out =  85.24
A= 10 , B= 10 , M= 10 , N= 5000 , Dist=Exp, Query1 out =  30.41 , Query2 out =  66.06
A= 10 , B= 10 , M= 10 , N= 5000 , Dist=Uni, Query1 out =  105.82 , Query2 out =  279.08
A= 10 , B= 10 , M= 15 , N= 1000 , Dist=Exp, Query1 out =  8.88 , Query2 out =  14.91
A= 10 , B= 10 , M= 15 , N= 1000 , Dist=Uni, Query1 out =  62.59 , Query2 out =  44.06
A= 10 , B= 10 , M= 15 , N= 2000 , Dist=Exp, Query1 out =  18.68 , Query2 out =  28.41
A= 10 , B= 10 , M= 15 , N= 2000 , Dist=Uni, Query1 out =  131.99 , Query2 out =  52.94
A= 10 , B= 10 , M= 15 , N= 5000 , Dist=Exp, Query1 out =  42.06 , Query2 out =  63.88
A= 10 , B= 10 , M= 15 , N= 5000 , Dist=Uni, Query1 out =  334.79 , Query2 out =  160.53
A= 10 , B= 10 , M= 25 , N= 1000 , Dist=Exp, Query1 out =  7.44 , Query2 out =  35.46
A= 10 , B= 10 , M= 25 , N= 1000 , Dist=Uni, Query1 out =  67.8 , Query2 out =  29.18
A= 10 , B= 10 , M= 25 , N= 2000 , Dist=Exp, Query1 out =  21.07 , Query2 out =  55.36
A= 10 , B= 10 , M= 25 , N= 2000 , Dist=Uni, Query1 out =  149.77 , Query2 out =  46.96
A= 10 , B= 10 , M= 25 , N= 5000 , Dist=Exp, Query1 out =  68.03 , Query2 out =  67.32
A= 10 , B= 10 , M= 25 , N= 5000 , Dist=Uni, Query1 out =  390.44 , Query2 out =  115.76
```

- Algorithm 1: We can also see that Query1 (Algorithm 1) mostly performs better for exponential distribution with 7 out of 9 experiments. For uniform distribution, the performance is not very good with only 2 out of 9 experiments.

    - Since the exponential distribution gives us a skewed distribution with a few elements that appear very frequently, there is very little chance for collision that results in over or under counting.
    - But for uniform distributed sample, all the elements have a M/N chance of appearing and the collision chance is definitely higher than the exponential distributed sample.

- Algorithm 2: Here we can see that Query2 (Algorithm 2) mostly performs better for uniform distribution with 7 out of 9 experiments. However for exponential distribution, Query2 is only better for 2 out of 9 experiments. This suggests that algorithm 2 increases the noise for exponential distribution.
    - Logically it makes sense. For exponential distribution, we have a few integers that appear very frequently like 0 and 1. If we use the estimate formula and subtract it's neighbour in the AB counter matrix, most likely it will subtract with zeros which does nothing to help reduce the noise.
    - BUT for uniform distribution, we expect a lot of the counts in AB counter matrix to be at least non-zeros. This will help to moderate some of the counts and reduce the noise.

- M and N: We study how M and N affects our results. I ran another two experiments with even higher M and N.



```
A= 10 , B= 10 , M= 25 , N= 10000 , Dist=Exp, Query1 out =  251.16 , Query2 out =  204.77
A= 10 , B= 10 , M= 25 , N= 10000 , Dist=Uni, Query1 out =  711.82 , Query2 out =  290.13
A= 10 , B= 10 , M= 100 , N= 10000 , Dist=Exp, Query1 out =  370.06 , Query2 out =  490.16
A= 10 , B= 10 , M= 100 , N= 10000 , Dist=Uni, Query1 out =  894.72 , Query2 out =  95.03
```

It seems that when *M* and *N* are a lot higher, algorithm 2 performs consistently better for both distributions. Sometimes the performance for exponential distributed sample is still not very good but we have already explained that above.

- This result is not surprising. As we populate the AB counter matrix with more and more samples, the chances of the AB counter matrix converging to its true underlying distribution is higher => better for uniform distribution.
- The same thing for exponential distribution doesn't really apply since most of the elements will still be the first few 0, 1, 2, 3. There is not much difference whether we use N = 1000 or N = 10000.

4.2 Comparing Space

In this section, I will be changing *A* and *B* which determines the size or space of the AB counter matrix that stores the counters. We know that with more space, both algorithm gives more accuracy. However we are interested to see what is the trade-off between space usage and accuracy. Here we will be fixing M and N.

```
A= 10 , B= 14 , M= 40 , N= 5000 , Dist=Exp, Query1 out =  20.53 , Query2 out =  89.11
A= 10 , B= 18 , M= 40 , N= 5000 , Dist=Exp, Query1 out =  21.39 , Query2 out =  38.44
A= 10 , B= 30 , M= 40 , N= 5000 , Dist=Exp, Query1 out =  3.08 , Query2 out =  12.3
A= 15 , B= 14 , M= 40 , N= 5000 , Dist=Exp, Query1 out =  6.54 , Query2 out =  12.68
```

```
A= 15 , B= 18 , M= 40 , N= 5000 , Dist=Exp, Query1 out =  4.02 , Query2 out =  12.01
A= 15 , B= 30 , M= 40 , N= 5000 , Dist=Exp, Query1 out =  2.98 , Query2 out =  3.27
A= 25 , B= 14 , M= 40 , N= 5000 , Dist=Exp, Query1 out =  4.49 , Query2 out =  11.92
A= 25 , B= 18 , M= 40 , N= 5000 , Dist=Exp, Query1 out =  3.09 , Query2 out =  3.86
A= 25 , B= 30 , M= 40 , N= 5000 , Dist=Exp, Query1 out =  3.04 , Query2 out =  3.35
A= 10 , B= 14 , M= 40 , N= 5000 , Dist=Uni, Query1 out =  247.5 , Query2 out =  121.83
A= 10 , B= 18 , M= 40 , N= 5000 , Dist=Uni, Query1 out =  153.6 , Query2 out =  122.92
A= 10 , B= 30 , M= 40 , N= 5000 , Dist=Uni, Query1 out =  64.7 , Query2 out =  123.85
A= 15 , B= 14 , M= 40 , N= 5000 , Dist=Uni, Query1 out =  247.31 , Query2 out =  122.85
A= 15 , B= 18 , M= 40 , N= 5000 , Dist=Uni, Query1 out =  145.07 , Query2 out =  123.57
A= 15 , B= 30 , M= 40 , N= 5000 , Dist=Uni, Query1 out =  62.11 , Query2 out =  124.13
A= 25 , B= 14 , M= 40 , N= 5000 , Dist=Uni, Query1 out =  247.7 , Query2 out =  122.78
A= 25 , B= 18 , M= 40 , N= 5000 , Dist=Uni, Query1 out =  138.04 , Query2 out =  123.26
A= 25 , B= 30 , M= 40 , N= 5000 , Dist=Uni, Query1 out =  68.55 , Query2 out =  123.93
```

- Exponential distributed sample: You can easily see that as space increase (A * B), accuracy improves. However the accuracy only improves until a certain point and it no longer improve anymore. Therefore, there is definitely a space-accuracy trade-off with diminishing returns for accuracy with very big space.
    - Algorithm 2: The improvement for algorithm 2 with exponentially distributed sample is much more significant, suggesting that we could allocate more space for this algorithm for better accuracy.
    - Algorithm 1: There is no point in increasing too much space for algorithm 1 because the improvement is very small after a certain point.
    - The understanding is that the exponentially distributed sample has only a few elements that appear very frequently. Even if you increase the amount of space (e.g. more B), it will not make much difference.
- Uniform distributed sample: You can also easily see that as the improvement is different for the two algorithms
    - Algorithm 1: The improvement in accuracy correlates positively with the space usage. This is expected because you lower the chance of having collisions by increasing the number of counters and hash functions.
    - Algorithm 2: There is totally no improvement in accuracy even if you increase the space.

5. Conclusion, Next Steps

From our findings, we can conclude that we cannot use a one size fit all approach to different distributions. Some algorithm is only good for certain distribution. In our case, algorithm 1 is much better suited for exponentially distributed samples. Algorithm 2 is better for uniformly distributed samples.

More experiments can be done to find out more. E.g. Trying out different distributions. We can also try different hash functions to see which hash function works the best for the distribution. There are definitely many dimension we can adjust to experiment.