PA4 Report

Streaming Algorithms

Alex Huynh

Com S 435
Iowa State University

1 Implementation Details

There are two small details that need to be pointed out that are not trivial. For Count Sketch and AMS Sketch, they are similar in that taking $k = O(\log \frac{1}{\delta})$ hash functions for Count Sketch or the median of $k = O(\log \frac{1}{\delta})$ copies of algorithm B for Count Sketch yields the probability of a bad calculation to be upper bounded by δ.

We specifically solve for that here. Let X be 1 for the probability of the Chebyshev inequality success, and 0 otherwise. We proved in class this failure probability $\leq \frac{1}{3}$ given $L = \frac{3}{\epsilon^2}$ for Count Sketch and $k = \frac{15}{\epsilon^2}$ for AMS. Thus $\sum X \leq \frac{b}{3}$, where b is the is number of copies We would like $P(\sum X > \frac{b}{2}) \geq P(\frac{\sum X}{b} - \frac{2}{3} > -\frac{1}{6}) \leq P(|\frac{\sum X}{b} - \frac{1}{3}| > \frac{1}{6}) \leq \delta \implies b = 24 \log \frac{2}{\delta}$. We use this many hashes in Count Sketch and take the median of this many copies of algorithm B in AMS.

A second point is that I used random hash functions for AMS because it requires a lot of hash functions, and salting a high number of hash functions takes a lot of time, especially if they need to be truly random hash functions. For the random hash functions, I took $M = 100001$ as it is decently high but not too high that the computation of the random prime p would be lengthy.

2 Testing Framework

This is the main idea for testing the streaming algorithms Firstly, I defined the number of iterations each test should run. On each iteration, a random stream would be generated. The number of distinct elements and the maximum frequency of each element (thus each element appears randomly at least once but at most a certain amount) are determined by the constructor of the test class, TestSuite.java. On generation of the stream, statistics like f_1 and f_2 and N are stored for later use.

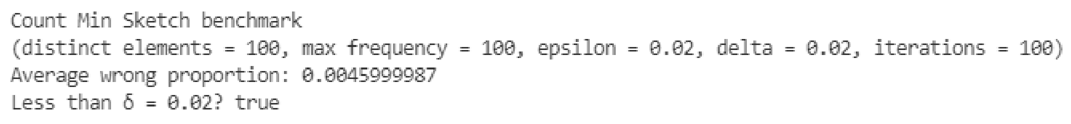
Testing heavy hitters is a bit different. After the stream is generated, I added a "max" amount of heavy hitters, where each added element is added to the stream $2 \times q \times N$ times. We multiply by 2 to ensure that the heavy hitter has enough frequency. However, upon adding too many heavy hitters, it is possible that earlier heavy hitters will no longer be a heavy hitter, and that is why I put max in quotations. After the Count Min Sketch is computed, I compare the size of the actual heavy hitter set to the approximate one. This is also done i iterations and a probability is calculated to determine performance.

For testing Count Min Sketch, I would call construct a CMS instance on the stream and then for every element x, I would get the approximate frequency $\hat{f}_1(x)$. If it was more than ϵN away from the true frequency $f_1(x)$, then I would increment a counter c. For each ith iteration, I would take the average wrong probability $\frac{c}{N}$ and add it to the total wrong probability variable, call it c^*. After all iterations, I divide c^* by the number of iterations to obtain the average wrong probability across all iterations which were across all elements of the stream. Our results are good if $c^* < \delta$. For testing Count Sketch, a similar approach is followed except our condition for incrementing c is if $|f_1(x) - \hat{f}_1(x)| \geq \epsilon\sqrt{F_2}$.

For comparing the two, I added extra an extra method in the classes, and that method set the number of rows and hash functions. From there, it was simple to compare the performance of the two. Note that to compare the performance, we needed epsilon, but that was calculated based on the number of columns that were set.

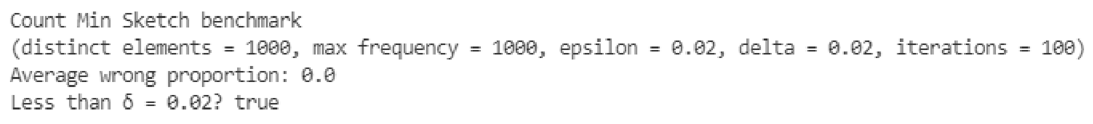
For testing AMS Sketch, there is only one counter c to maintain, and thus the average wrong probability is $\frac{c}{i}$ where i is the number of iterations. The condition for c to be incremented is if $|F_2 - \hat{F_2}| \geq \epsilon F_2$. Note that we do not need to report any actual tests for estimating F_2. For dimensionality reduction, there is again only one counter c which is incremented if $|L_2^2(AMS(X)) - L_2^2(X)| \geq \epsilon L_2^2(X)$. Again, our results are good if these average probabilities are $< \delta$.

3 Count Min Sketch



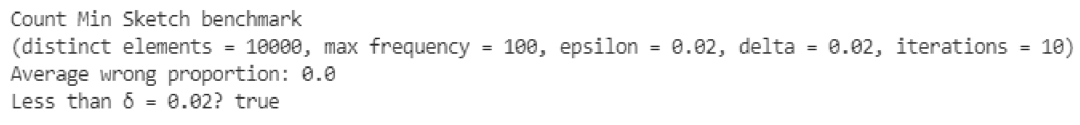
```
Count Min Sketch benchmark
(distinct elements = 100, max frequency = 100, epsilon = 0.02, delta = 0.02, iterations = 100)
Average wrong proportion: 0.0045999987
Less than δ = 0.02? true
```

Figure 1: CMS on small stream



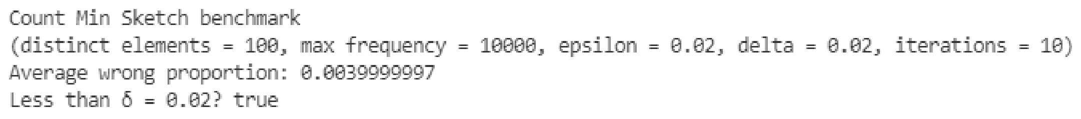
```
Count Min Sketch benchmark
(distinct elements = 1000, max frequency = 1000, epsilon = 0.02, delta = 0.02, iterations = 100)
Average wrong proportion: 0.0
Less than δ = 0.02? true
```

Figure 2: CMS on large stream



```
Count Min Sketch benchmark
(distinct elements = 10000, max frequency = 100, epsilon = 0.02, delta = 0.02, iterations = 10)
Average wrong proportion: 0.0
Less than δ = 0.02? true
```

Figure 3: CMS on low frequency stream

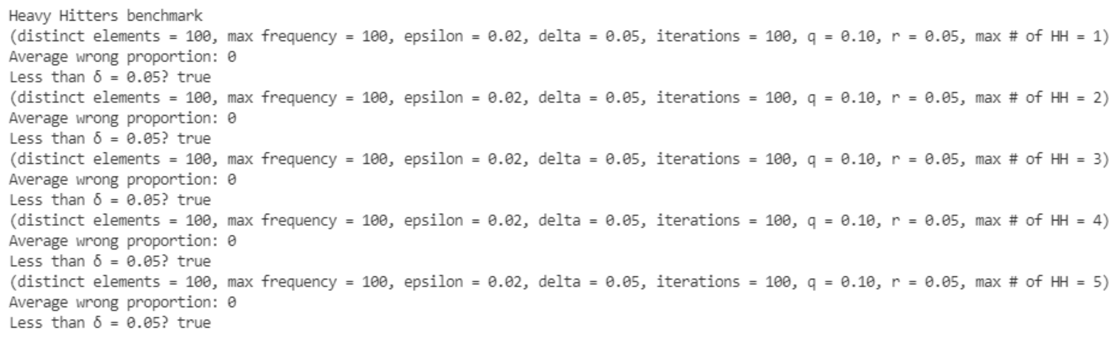


```
Count Min Sketch benchmark
(distinct elements = 100, max frequency = 10000, epsilon = 0.02, delta = 0.02, iterations = 10)
Average wrong proportion: 0.0039999997
Less than δ = 0.02? true
```

Figure 4: CMS on high frequency stream

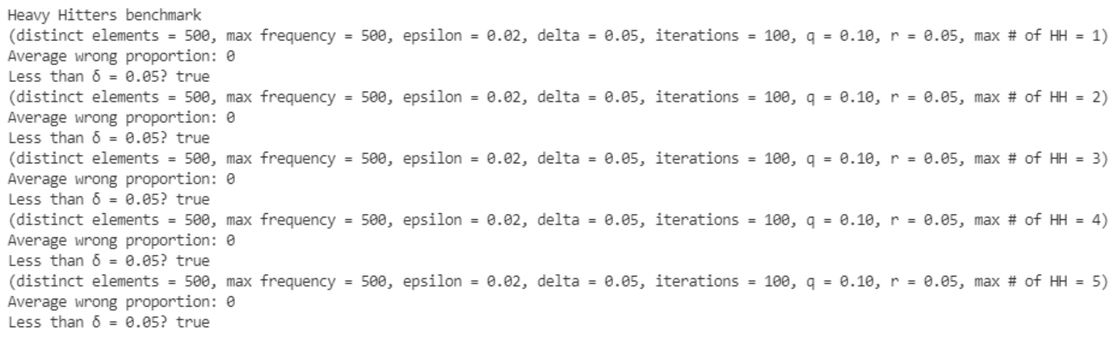
CMS works well for small and large streams as seen in the above figures. As we can also see, CMS works better on low frequency streams.

4 Heavy Hitters



```
Heavy Hitters benchmark
(distinct elements = 100, max frequency = 100, epsilon = 0.02, delta = 0.05, iterations = 100, q = 0.10, r = 0.05, max # of HH = 1)
Average wrong proportion: 0
Less than δ = 0.05? true
(distinct elements = 100, max frequency = 100, epsilon = 0.02, delta = 0.05, iterations = 100, q = 0.10, r = 0.05, max # of HH = 2)
Average wrong proportion: 0
Less than δ = 0.05? true
(distinct elements = 100, max frequency = 100, epsilon = 0.02, delta = 0.05, iterations = 100, q = 0.10, r = 0.05, max # of HH = 3)
Average wrong proportion: 0
Less than δ = 0.05? true
(distinct elements = 100, max frequency = 100, epsilon = 0.02, delta = 0.05, iterations = 100, q = 0.10, r = 0.05, max # of HH = 4)
Average wrong proportion: 0
Less than δ = 0.05? true
(distinct elements = 100, max frequency = 100, epsilon = 0.02, delta = 0.05, iterations = 100, q = 0.10, r = 0.05, max # of HH = 5)
Average wrong proportion: 0
Less than δ = 0.05? true
```

Figure 5: Heavy hitters on small stream

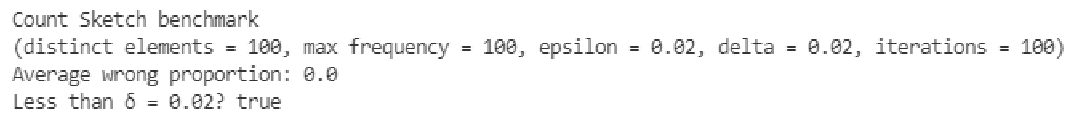


```
Heavy Hitters benchmark
(distinct elements = 500, max frequency = 500, epsilon = 0.02, delta = 0.05, iterations = 100, q = 0.10, r = 0.05, max # of HH = 1)
Average wrong proportion: 0
Less than δ = 0.05? true
(distinct elements = 500, max frequency = 500, epsilon = 0.02, delta = 0.05, iterations = 100, q = 0.10, r = 0.05, max # of HH = 2)
Average wrong proportion: 0
Less than δ = 0.05? true
(distinct elements = 500, max frequency = 500, epsilon = 0.02, delta = 0.05, iterations = 100, q = 0.10, r = 0.05, max # of HH = 3)
Average wrong proportion: 0
Less than δ = 0.05? true
(distinct elements = 500, max frequency = 500, epsilon = 0.02, delta = 0.05, iterations = 100, q = 0.10, r = 0.05, max # of HH = 4)
Average wrong proportion: 0
Less than δ = 0.05? true
(distinct elements = 500, max frequency = 500, epsilon = 0.02, delta = 0.05, iterations = 100, q = 0.10, r = 0.05, max # of HH = 5)
Average wrong proportion: 0
Less than δ = 0.05? true
```

Figure 6: Heavy hitters on large stream

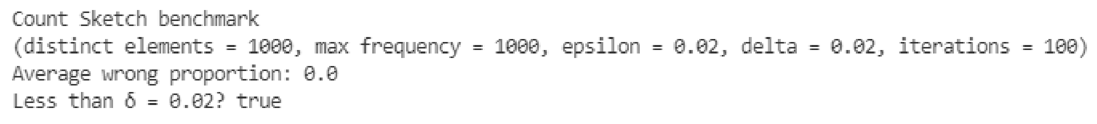
Heavy hitter set approximation works well for small and large streams as seen in the above figures.

5 Count Sketch



```
Count Sketch benchmark
(distinct elements = 100, max frequency = 100, epsilon = 0.02, delta = 0.02, iterations = 100)
Average wrong proportion: 0.0
Less than δ = 0.02? true
```

Figure 7: Count Sketch on small stream

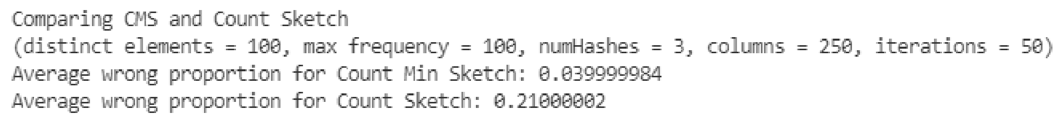


```
Count Sketch benchmark
(distinct elements = 1000, max frequency = 1000, epsilon = 0.02, delta = 0.02, iterations = 100)
Average wrong proportion: 0.0
Less than δ = 0.02? true
```

Figure 8: Count Sketch on large stream

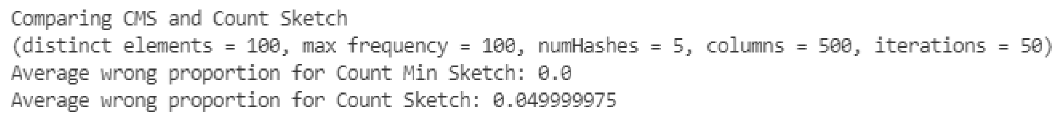
Count Sketch also works well as seen in the above figures.

6 Count Min Sketch vs. Count Sketch



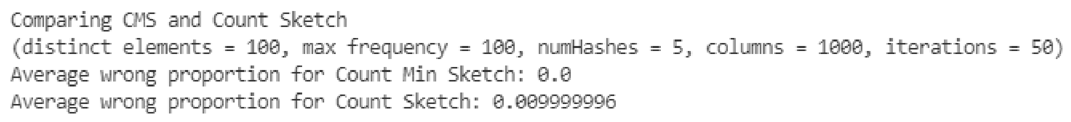
```
Comparing CMS and Count Sketch
(distinct elements = 100, max frequency = 100, numHashes = 3, columns = 250, iterations = 50)
Average wrong proportion for Count Min Sketch: 0.039999984
Average wrong proportion for Count Sketch: 0.21000002
```

Figure 9: 3 hash functions, 250 columns



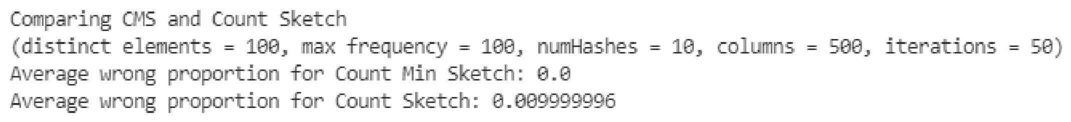
```
Comparing CMS and Count Sketch
(distinct elements = 100, max frequency = 100, numHashes = 5, columns = 500, iterations = 50)
Average wrong proportion for Count Min Sketch: 0.0
Average wrong proportion for Count Sketch: 0.049999975
```

Figure 10: 5 hash functions, 500 columns



```
Comparing CMS and Count Sketch
(distinct elements = 100, max frequency = 100, numHashes = 5, columns = 1000, iterations = 50)
Average wrong proportion for Count Min Sketch: 0.0
Average wrong proportion for Count Sketch: 0.009999996
```

Figure 11: Adding 5 more hash functions

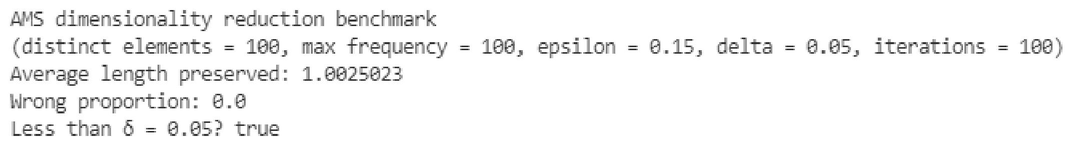


```
Comparing CMS and Count Sketch
(distinct elements = 100, max frequency = 100, numHashes = 10, columns = 500, iterations = 50)
Average wrong proportion for Count Min Sketch: 0.0
Average wrong proportion for Count Sketch: 0.009999996
```

Figure 12: Adding 500 more columns instead

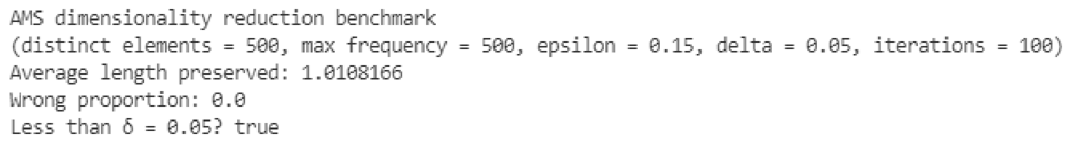
As we can see in the figure above, given a certain amount of hash functions and columns, Count Min Sketch performs better.

7 Dimensionality Reduction



```
AMS dimensionality reduction benchmark
(distinct elements = 100, max frequency = 100, epsilon = 0.15, delta = 0.05, iterations = 100)
Average length preserved: 1.0025023
Wrong proportion: 0.0
Less than δ = 0.05? true
```

Figure 13: Dimensionality reduction on small stream



```
AMS dimensionality reduction benchmark
(distinct elements = 500, max frequency = 500, epsilon = 0.15, delta = 0.05, iterations = 100)
Average length preserved: 1.0108166
Wrong proportion: 0.0
Less than δ = 0.05? true
```

Figure 14: Dimensionality reduction on large stream

The length of the vectors are preserved well. From the experiment, it seems every single of the 100 vectors were preserved to a good extent ($\epsilon = 0.15$), which is pretty amazing.