Finding Frequent Items in Data Streams*†

Joshua Ramirez, joshur4@cs.washington.edu

Andrew Wei, nowei@cs.washington.edu

December 2019

1 Introduction

The goal is to find the most frequent items in a stream without using a lot of space. The problem setup is as follows: Suppose we have a stream that is too large to store in memory and we can only do a single pass on the data, how do we find the most frequently occurring items in the stream? A problem like this might come up in search engines or pretty much any application that deals with streaming data. The paper claims that the only algorithm with theoretical guarantee before this was the Sampling algorithm, which takes uniform random sample of the data.

The variables we will use are a stream $\mathcal{S} = q_1, q_2, ..., q_N$, where $q_i \in \{o_1, ..., o_m\}$ and each o_i occurs n_i items. We then order o_i such that $n_1 \geq n_2 \geq ... \geq n_m$ and then we finally have $f_i = n_i/N$.

The paper wanted to solve FindCandidateTop($\mathcal{S}, k, l$), which outputs a list of l elements that contained the k most common elements in $\mathcal{S}$, but if the count of the k-th most common number differed from the count of the $l+1$-th most common number by 1, then it would be hard to find l elements that would contain the top k elements.

So instead they wanted to find FindApproxTop($\mathcal{S}, k, \epsilon$), which outputs a list of k elements from $\mathcal{S}$ where each element, i, in the list has count n_i s.t. $n_i > (1-\epsilon)n_k$ with high probability. The algorithm the paper introduced utilizes a CountSketch, which estimates the frequencies of the most common items. A second algorithm unrelated

*based on Finding Frequent Items in Data Streams by Charikar et al.

†we also coded the main algorithm here: https://github.com/nowei/frequent_items

to the original problem is also proposed by taking the difference between two CountSketches (that use the same hash functions) for finding the maximum change in items between two streams.

As previously described, a solution to FindCandidateTop($\mathcal{S}, k, l$) is the Sampling algorithm, where we sample and keep a list and a counter of the things seen. Where each sample increments the counter for that sample and is added to the list if it isn't in the list already. Then there are two variants of the Sampling algorithm. One by Gibbons and Matias [GM98], where it tries to keep things with probability $\tau = 1$, and as it runs out of space, lowers τ until an element is kicked out from the sample, and continues adding things with the new probability. Then there is another one by Fang et al. [FSGM+96], which uses a similar multi-hashing scheme to the paper.

2 Main contributions

A CountSketch is a data structure that is essentially a table of hash tables. We will use this data structure to keep track of the approximate frequencies of each element in the stream.

We have t pairwise independent hash functions $h_1, ..., h_t$ that hash values to $\{1, ..., b\}$ and t pairwise independent hash functions $s_1, ..., s_t$ that hash values to $\{-1, +1\}$. Note that these hash functions are independent from each other. Then initialize the CountSketch, C, as a 2D array of size $t \times b$, i.e. t tables with b buckets each.

Let us consider a counter, c. As we go through the stream, we hash the values q using s and add it to c. We then have that the expectation of any given item o_i is $\mathbb{E}[c * s(o_i)] = n_i$.

Proof:

Note that: $\mathbb{E}[s(o_j)s(o_i)], i \neq j$ is $s(o_j)s(o_i) = 1$ if $s(o_j) = s(o_i)$ (collide) or -1 otherwise. The probability that they collide is 1/2 so we have that $\mathbb{E}[s(o_j)s(o_i)] = 1 * \Pr(\text{collision}) + -1 * \Pr(\text{no collision}) = 0$.

$$
\begin{aligned}
\mathbb{E}[c * s(o_i)] &= \mathbb{E}\left[\sum_{j=1}^{m} n_j s(o_j)s(q_i)\right] \\
&= \sum_{j=1}^{m} \mathbb{E}[n_j s(o_j)s(o_i)] && \text{Linearity of Expectation} \\
&= \sum_{\substack{j=1 \\ j \neq i}}^{m} \mathbb{E}[n_j s(o_j)s(o_i)] + n_i * \mathbb{E}[s(o_i)^2] \\
&= 0 + n_i * 1
\end{aligned}
$$

$$= n_i$$

So we conclude that $\mathbb{E}[c * s(o_i)] = n_i$.

The main problem with this is that if everything goes to the same counter, the values will become skewed easily. Then we could try using t hash functions and counters, we add $c_j += s_j(q_i)$ for each j. We then get $\mathbb{E}[c_j * s_j(o_i)] = n_i$. Since the process is repeated, the variance decreases if we take the mean or median. But this isn't robust to things with high frequencies, since they will skew the counts, so the next alteration is to replace each counter with a hash table of counters and we hash to a specific counter using h_i, so the elements only affect one bin in each hash table. Then we have $\mathbb{E}[C[i][h_i(q)] * s_i(q)] = n_q$, which can be derived with a similar proof to what we did above.

There are two main operations used in the algorithm: Add and Estimate. Given an item, q, and a CountSketch, C, the Add operation adds the value of $s_i(q)$ to where $h_i(q)$ hashes to on the i-th table in C for $i \in \{1, ..., t\}$. Given an item, q, and a CountSketch, C, the Estimate operation returns the median value of the values accessible using $h_i(q)$ on the i-th table in C for $i \in \{1, ..., t\}$. Succinctly,

- Add(C, q): For i in $\{1, ..., t\}$: $C[i][h_i(q)]$ += $s_i(q)$
- Estimate(C, q): return Median([$C[i][h_i(q)]$ * $s_i(q)$ for i in $\{1, ..., t\}$])

The algorithm is as follows:

Algorithm 1

```
1: procedure FindApproxTop(S, k, ε)
2:     Initialize heap
3:     for q ∈ S do
4:         Add(C,q)
5:         if q ∈ heap then
6:             Increment its count and repercolate if necessary
7:         else
8:             count ← Estimate(C, q)
9:             if size(heap) < k then
10:                Add (count, q) to the heap
11:            else if count < min(heap) then
12:                evict the minimum heap element
13:                Add (count, q) to the heap
14:    return heap
```

Then we present the algorithm for finding the items with the largest change in frequency. Note that it is necessary that the hash functions between the two functions remain the same, otherwise it would be impossible to compare the counts. We denote $n_q^{\mathcal{S}}$ as the number of times that item q appears in stream $\mathcal{S}$. We want to find the items q such that $|n_q^{\mathcal{S}_1} - n_q^{\mathcal{S}_2}|$ is maximized. We will take two passes over the data, first updating the counters and then second identifying the elements with the largest change in frequency.

Algorithm 2

procedure FINDMAXCHANGE($\mathcal{S}_1$, $\mathcal{S}_2$, k, l)

  Initialize CountSketch, C

  for $q \in \mathcal{S}_1$ **do**

    for $i \in [1, t]$, $C[i][h_i(q)] -= s_i(q)$

  for $q \in \mathcal{S}_2$ **do**

    for $i \in [1, t]$, $C[i][h_i(q)] += s_i(q)$

  Create set A that has capacity l that stores the largest values seen.

  for $q \in \mathcal{S}_1$ and $\mathcal{S}_2$ **do**

    $\hat{n}_q$ = Median([$C[i][h_i(q)]$ * $s_i(q)$ for i in $\{1, ..., t\}$])

    if $|A| < l$ **then**

      Add ($\hat{n}_q$, q) to A

    else if $\hat{n}_q > \min(A)$ **then**

      Add ($\hat{n}_q$,q) to A

      Remove $\min(A)$

    if $(\_, q) \in A$ **then**

      Keep a count of the number of times we've seen q in $\mathcal{S}_1$ and $\mathcal{S}_2$ in $n_q^{\mathcal{S}_1}$ and $n_q^{\mathcal{S}_2}$ respectively

    else if $n_q^{\mathcal{S}_1} > 0$ or $n_q^{\mathcal{S}_2} > 0$ **then**

      Clear the memory associated with $n_q^{\mathcal{S}_1}$ and $n_q^{\mathcal{S}_2}$

  return top k $q : (\_, q) \in A$ where $|n_q^{\mathcal{S}_1} - n_q^{\mathcal{S}_2}|$ is largest.

3 Overview of Analysis

For the sake of simplicity let us order the elements of $\{o_1, \cdots, o_m\}$ such that $n_1 \geq n_2, \cdots, \geq n_m$, where n_i is the number of times o_i shows up in the stream. Also we let $h[o_i]s[o_i]$ be the estimated count of an element o_i given to us by the algorithm. For the sake of the analysis, we will be working with an arbitrary element o_q and its count will be n_q. Also we will let n_k be the count of the k^{th} largest element. Finally, let δ be the probability

that Count Sketch fails. From the algorithm we saw that we need $O(t \times b)$ counters, and the goal of this proof is to find the optimal t and b (We actually need $O(tb + k)$ counters for our heap but we will assume that k is constant). To be more specific the goal of this section is to prove the following theorem:

Theorem 1 *The Count Sketch algorithm solves* FindApproxTop$(\mathcal{S}, k, \epsilon)$ *using*

$$O\left(k \log\left(\frac{N}{\delta}\right) + \frac{\sum_{q'=k+1}^{m} n_{q'}^2}{(\epsilon n_k)^2} \log\left(\frac{N}{\delta}\right)\right)$$

counters.

The paper says it uses this much space however, it does not take into account the amount of space it needs to store $O(t)$ hash functions so we reworded the Theorem to fit what we will prove. Before we begin we will label a few variables needed for our proof. Let h be an arbitrary hash function from $\{o_1, \cdots, o_m\}$ to $\{1, \cdots, b\}$ and let s be the hash function from $\{o_1, \cdots, o_m\}$ to $\{-1, +1\}$. Now let l be an arbitrary position in our stream and let $n_q[l]$ be the count of o_q up to the l^{th} position in our stream. Let $A[q]$ be a set of all the indices of elements that map to the same counter that o_q gets mapped to under h (not including q itself). Now let $A_k[q] = A[q]/K$, where K is the set of indices of the top k elements. Lastly we assign $v[q] = \sum_{q' \in A[q]} n_{q'}^2$, $v_k[q] = \sum_{q' \in A_k[q]} n_{q'}^2$. Now that we have set the stage we will find a bound for b. Notice that since h is pairwise independent we have:

$$\begin{aligned}
\mathbb{E}[v_k[q]] &= \sum_{i=k+1}^{m} Pr[h(o_q) = h(o_i)] n_i^2 \\
&= \sum_{i=k+1}^{m} \frac{1}{b} n_i^2 \\
&= \frac{\sum_{i=k+1}^{m} n_i^2}{b}.
\end{aligned}$$

Also from our analysis of $E[h[o_q]s[o_q]]$ from previous sections we can see that that

$$E[h[o_q]s[o_q]] \leq v_k[q].$$

From here we will use basic probabilistic techniques such as Markov's inequality and union bounds to achieve simple events on a few events. First we have,

$$Pr\left[v_k[q] \geq 8\frac{\sum_{i=k+1}^{m} n_i^2}{b}\right] = Pr\left[v_k[q] \geq 8\mathbb{E}[v_k[q]]\right] \tag{1}$$

$$\leq \frac{1}{8} \tag{2}$$

From Markov's inequality. Let us label the event $\{v_k[q] \leq 8\frac{\sum_{i=k+1}^{m} n_i^2}{b}\}$ as $SV[q]$. This inequality's purpose is to show that the variance of our estimated count will be small with high probability. Now we want to find the probability that $A[q]$ contains at least one of the top k elements or phrased in another way, $A[q] \cap K \neq \emptyset$. By the union bound we find that

$$\begin{aligned} Pr[A[q] \cap K \neq \emptyset] &\leq \sum_{i=1}^{k} Pr[o_i \in A[q]] \\ &= \sum_{i=1}^{k} Pr[h(o_i) = h(o_q)] \\ &= \frac{k}{b}. \end{aligned}$$

Thus, from here we can say that the probability that no top k element gets mapped to $A[q]$ happens with probability at least $1 - \frac{k}{b}$. Now from here on out we assume that $b \geq 8k$ thus, $Pr[A[q] \cap K = \emptyset] \geq 1 - \frac{1}{8}$. We will call the event that $\{A[q] \cap K = \emptyset\}$ as $NC[q]$. Finally from the bound of the expected estimated count we have

$$Pr\left[|h[o_q]s[o_q] - n_q[l]|^2 \leq 8Var(h[o_q]s[o_q])\right] \geq 1 - \frac{1}{8} \tag{3}$$

by Markov's inequality. We label the event as $SD[q]$. The purpose of this inequality is to show that we deviate from the actual count of o_q by a small amount with a decent probability. This finally leads us to

$$\begin{aligned} Pr\left[NC[q] \cap SV[q] \cap SD[q]\right] &= 1 - Pr\left[NC[q]^c \cup SV[q]^c \cup SD[q]^c\right] \\ &= 1 - \frac{3}{8} \\ &\geq \frac{5}{8} \end{aligned}$$

where the second equality comes from the union bound. Now also realize that if the events

$$\begin{aligned} Pr\left[NC[q] \cap SV[q] \cap SD[q]\right] &= Pr\left[\{A[q] \cap K \neq \emptyset\} \cap \{v_k[q] \leq 8\frac{\sum_{i=k+1}^{m} n_i^2}{b}\} \cap \{|h[o_q]s[o_q] - n_q[l]|^2 \leq 8Var(h[o_q]s[o_q])\}\right] \\ &\leq Pr\left[|h[o_q]s[o_q] - n_q[l]| \leq 8\sqrt{\frac{\sum_{i=k+1}^{m} n_i^2}{b}}\right] \end{aligned}$$

Let $\gamma = 8\sqrt{\frac{\sum_{i=k+1}^{m} n_i^2}{b}} = 8\sqrt{E[v_k[q]]}$. We can conclude that,

$$Pr[|h[o_q]s[o_q] - n_q[l]| \leq 8\gamma] \geq \frac{5}{8}.$$

So far we were to show that for a single element, its estimated count will deviate from its actual count at a position l by 8γ with probability greater than $\frac{5}{8}$. This is still a bad lower bound, but fortunately we still have not taken into account that we use the median count of t estimates. Thus, we will use the median trick which was heavily used throughout the course as well as Chernoff bounds to get a huge improvement. Now let X_i be a Bernoulli random variable that is 1 if $|h_i[o_q]s_i[o_q] - n_q[l]| \leq 8\gamma$ and 0 otherwise (here h_i and s_i corresponds to the hash functions for the i^{th} row of counters). Now let $X = \sum_{i=1}^{t} X_i$. Notice that $E[X] = \sum_{i=1}^{t} E[X_i] \geq \frac{5t}{8}$. Now using Chernoff bound we have that

$$\begin{aligned}
Pr\left[X \leq \frac{t}{2}\right] &= Pr[X - E[X] \leq \frac{t}{2} - E[X]] \\
&\leq Pr[X - E[X] \leq \frac{t}{2} - \frac{5t}{8}] \\
&= Pr[X - E[x] \leq -\frac{t}{8}] \\
&\leq Pr\left[|X - E[x]| \geq \frac{t}{8}\right] \\
&= exp(-O(t))
\end{aligned}$$

If we let $t = \Omega(log(\frac{N}{\delta}))$his gives us the following lemma

Lemma 2 *With probability* $1 - \frac{\delta}{N}$,

$$|median_{i \in \{1,\cdots,t\}}\{h_i[o_q]s_i[o_q]\} - n_q(l)| \leq 8\gamma$$

We can strengthen this lemma by using a union bound to get the following:

Lemma 3 *With probability* $1 - \delta$, *for all* $l \in \{1, \cdots, n\}$,

$$|median_{i \in \{1,\cdots,t\}}\{h_i[o_q]s_i[o_q]\} - n_q[l]| \leq 8\gamma$$

where o_q *is the element that occurs in position* l

The last two lemma's are incredibly important. We now know that at any point in the stream with probability $1 - \delta$ our algorithm will estimate the count of an element o_i to be in the range $[n_i - 8\gamma, n_i + 8\gamma]$. This implies that if o_i has an actual count in the range of $[n_k - 8\gamma, n_k + 8\gamma]$ then it could be mistaken for a top k element without actually being a top k element. In general if we have two elements o_i and o_j then we could end up

mistaking them for each other if $|n_i - n_j| \leq 16\gamma$. Using this, if we set $16\gamma \leq \epsilon n_k$ then, this implies that if $n_i < (1-\epsilon)n_k$ then it will never be in our estimated top k count because $(1-\epsilon)n_k \leq n_k - 16\gamma$. Furthermore this implies that if an element o_i has the property that $n_i \geq (1+\epsilon)n_k$, then it will always be in the top k for our algorithm with probability $1-\delta$. We will show this is true by letting o_i have the property that $n_i \geq (1+\epsilon)n_k$. Then we have that $o_i \geq n_k + 16\gamma$. o_i will not show up in our estimated top k if and only if some non top k element gets mistaken for a top k in our algorithm and o_i gets kicked out. Let e be the estimated count of any element that would not appear in the actual top k elements. Then with probability $1-\delta$ we have

$$e \leq n_k + 8\gamma \leq n_i - 8\gamma \leq h[o_i]s[o_i]$$

Thus, o_i will always be in our estimated top k. Finally since γ is a function of b, solving the inequality $16\gamma \leq \epsilon n_k$ for b we get the last piece to our puzzle:

Lemma 4 *if* $b \geq 8\max\{k, \frac{32\sum_{i=k+1}^{m} n_i^2}{(\epsilon n_k)^2}\}$*, then the estimated top* k *elements occur at least* $(1-\epsilon)n_k$ *times in the sequence; furthermore all elements with frequencies at least* $(1+\epsilon)n_k$ *occur amongst the top* k *elements.*

Thus, along with our previous assumptions of t and b if $b = O\left(k + \frac{\sum_{i=k+1}^{m} n_i^2}{(\epsilon n_k)^2}\right)$ and $t = O(log(\frac{N}{\delta}))$ we get Theorem 1.

4 Conclusion

The algorithm presented by the paper is more involved than other algorithms mentioned in this summary. However, it accomplishes the task it set out to conquer with benefits that make it useful. We were able to lower the variance of the estimates while, providing a bound on the space used in the algorithm. This is extremely vital because for some distributions, this algorithm requires much less space than the sampling algorithm proposed in the beginning. This also gave a way to analyze the change of frequency items between two different streams which has a plethora of uses which, had not previously been solved effectively.