13 Frequency Approximation

A core mining problem is to find items that occur more than one would expect. These may be called outliers, anomalies, or other terms. Statistical models can be layered on top of or underneath these notions.

We begin with a very simple problem. There are m elements and they come from a domain $[n]$ (but both m and n might be very large, and we don't want to use $\Omega(m)$ or $\Omega(n)$ space). Some items in the domain occur more than once, and we want to find the items which occur the most frequently.

If we can keep a counter for each item in the domain, this is easy. But we will assume n is huge (like all possible IP addresses), and m is also huge, the number of packets passing through a router in a day.

13.1 Streaming

Recall that *streaming* is a model of computation that emphasizes *space* over all else. The goal is to compute something using as little storage space as possible. So much so that we cannot even store the input. Typically, you get to read the data once, you can then store something about the data, and then let it go forever! Or sometimes, less dramatically, you can make 2 or more passes on the data.

Formally, there is a stream $A = \langle a_1, a_2, \ldots, a_m \rangle$ of m items where (for this lecture) each $a_i \in [n]$. This means, the size of each a_i is about $\log n$ (to represent which element), and just to count how many items you have seen requires space $\log m$ (although if you allow approximations you can reduce this). Unless otherwise specified, $\log$ is used to represent $\log_2$ that is the base-2 logarithm. The goal is to compute a summary S_A using space that is only $\mathsf{poly}(\log n, \log m)$.

Let $f_j = |\{a_i \in A \mid a_i = j\}|$ represent the number of items in the stream that have value j. Let $F_1 = \sum_j f_j = m$ be the total number of elements seen. Let $F_2 = \sqrt{\sum_j f_j^2}$ be the sum of squares of elements counts, squarerooted. Let $F_0 = \sum_j f_j^0$ be the number of distinct elements.

13.1.1 Heavy Hitters

Consider the motivation of a streaming algorithm running on an internet router. It seems packets, which each include a destination IP address, a source IP address, and some data. There are a huge number $n = 2^{16}$ IP addresses, and also a huge number of packets m every hour or day.

A distributed denial of service (DDS) attach happens when many machines send packets to the same IP address, overwhelming the machine at that destination address, and effectively shutting it down. If we can detect this at the router level, we can block traffic before it gets to the destination machine. To do this, we need to identify an IP address that is getting a large fraction of all internet traffic (say more than 5%; $\phi = 0.05$); these are often called the *heavy-hitters*.

But it requires too much space to store a counter for each IP address, and we cannot store all packets. How do we keep track of the very frequent items without too much space?

13.2 Majority

One of the most basic streaming problems is as follows:

MAJORITY: if some $f_j > m/2$, output j. Otherwise, output anything.

How can we do this with $\log n + \log m$ space (one counter c, and one location ℓ)?

Answer: Maintaining that single label and counter, do the only thing feasible. If you see a new item with same label, increment the counter. If the label is different, decrement the counter. If the counter reaches

zero and you see a new element, replace the label, and set the counter to 1. The pseudocode is in Algorithm 13.2.1.

Algorithm 13.2.1 Majority(A)

```
Set c = 0 and ℓ = ∅
for i = 1 to m do
  if (a_i = ℓ) then
    c = c + 1
  else
    c = c − 1
  if (c < 0) then
    c = 1, ℓ = a_i
return ℓ
```

Why is Algorithm 13.2.1 correct? Consider the case where for some $j \in [n]$ we have $f_j > m/2$, the only relevant case. Since then $f_j > \sum_{j' \neq j} f_j$ we can match each stream element with $a_i \neq j$ (a "bad element") to another element $a_{i'} = j$ (a "good element"). If we chose the correct pairing (lets assume we did) then either the good element decremented the counter when the label was not j, or the bad element decremented the counter when the label was j. This results in a net 0 change in the counter for each pair. Its also possible that a bad element decremented the counter when the label was not equal to j, but this will only help. After this cancelation, there must still be unpaired good elements, and since then the label would need to be $\ell = j$ or the counter $c = 0$, they always end their turn with $\ell = j$ and the counter incremented. Thus after seeing all stream elements, we must terminate with $\ell = j$ and $c > 0$.

13.3 Misra-Gries Algorithm for Heavy Hitters

Now we generalize the MAJORITY problem to something much more useful.

k-FREQUENCY-ESTIMATION: Build a data structure S. For any $j \in [n]$ we can return $S(j) = \hat{f}_j$ such that

$$f_j - m/k \leq \hat{f}_j \leq f_j.$$

From another view, a *ϕ-heavy hitter* is an element $j \in [n]$ such that $f_j > \phi m$. We want to build a data structure for ε-approximate ϕ-heavy hitters so that it returns

- all f_j such that $f_j > \phi m$
- no f_j such that $f_j < \phi m - \varepsilon m$
- (any f_j such that $\phi m - \varepsilon m \leq f_j < \phi m$ can be returned, but might not be).

13.3.1 Misra-Gries Algorithm

[Misra+Gries 1982] Solves k-FREQUENCY-ESTIMATION in $k(\log m + \log n)$ space.

The trick is to run the MAJORITY algorithm, but with $(k-1)$ counters instead of 1. Let C be an array of $(k-1)$ counters $C[1], C[2], \ldots, C[k-1]$. Let L be an array of $(k-1)$ locations $L[1], L[2], \ldots, L[k-1]$.

- If we see a stream element that matches a label, we increment the associated counter.
- If not, and a counter is 0, we can reassign the associated label, and increment the counter.
- Finally, if all counters are non-zero, and no labels match, then we decrement all counters.

Psuedocode is provided in Algorithm 13.3.1.

Then on a query $q \in [n]$ to C, L, if $q \in L$ (specifically $L[j] = q$), then return $\hat{f}_q = C[j]$. Otherwise return $\hat{f}_q = 0$.

Algorithm 13.3.1 Misra-Gries(A)

Set all $C[i] = 0$ and all $L[i] = \emptyset$
for $i = 1$ **to** m **do**
  if $(a_i = L[j])$ **then**
    $C[j] = C[j] + 1$
  else
    if (some $C[j] = 0$) **then**
      Set $L[j] = a_i$ & $C[j] = 1$
    else
      for $j \in [k-1]$ **do** $C[j] = C[j] - 1$
return C, L

Analysis: Why is Algorithm 13.3.1 correct?

- A counter $C[j]$ representing $L[j] = q$ is only incremented if $a_i = q$, so we always have

$$\hat{f}_q \leq f_q.$$

- If a counter $C[j]$ representing $L[j] = q$ is decremented, then $k-2$ other counters are also decremented, and the current item's count is not recorded. This happens at most m/k times: since each decrement destroys the record of k objects, and since there are m objects total. Thus a counter $C[j]$ representing $L[j] = q$ is decremented at most m/k times. Thus

$$f_q - m/k \leq \hat{f}_q.$$

We can now apply this to get an additive ε-approximate FREQUENCY-ESTIMATION by setting $k = 1/\varepsilon$. We return $\hat{f}_q$ such that

$$|f_q - \hat{f}_q| \leq \varepsilon m.$$

Or we can set $k = 2/\varepsilon$ and return $C[j] + (m/k)/2$ to make error on both sides.

Space is $(1/\varepsilon)(\log m + \log n)$, since there are $(1/\varepsilon)$ counters and locations.

13.4 Count-Min Sketch

In contrast to the Misra-Gries algorithm, we describe a completely different way to solve the HEAVY-HITTER problem. It is called the Count-Min Sketch [Cormode + Muthukrishnan 2005].

Start with t independent (random) hash functions $\{h_1, \ldots, h_t\}$ where each $h_h : [n] \to [k]$.

Now we store a 2d array of counters for $t = \log(1/\delta)$ and $k = 2/\varepsilon$:

h_1	$C_{1,1}$	$C_{1,2}$	$\ldots$	$C_{1,k}$
h_2	$C_{2,1}$	$C_{2,2}$	$\ldots$	$C_{2,k}$
$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$
h_t	$C_{t,1}$	$C_{t,2}$	$\ldots$	$C_{t,k}$

After running Algorithm 13.4.1 on a stream A, then on a query $q \in [n]$ we can return

$$\hat{f}_q = \min_{j \in [t]} C_{j,h_j(q)}.$$

This is why it is called a *count-min sketch*.

Algorithm 13.4.1 Count-Min(A)

Set all $C_{i,j} = 0$
for $i = 1$ **to** m **do**
  for $j = 1$ **to** t **do**
    $C_{j,h_j(a_i)} = C_{j,h_j(a_i)} + 1$

Analysis: Clearly $f_q \leq \hat{f}_q$ since each counter has everything for q, but may also have other stuff (on hash collisions).

Next we claim that $\hat{f}_q \leq f_q + W$ for some over count value W. So how large is W?

Consider just one hash function h_i. It adds to W when there is a collision $h_i(q) = h_i(j)$. This happens with probability $1/k$.

So we can create a random variable $Y_{i,j}$ that represents the overcount caused on h_i for q because of element $j \in [n]$. That is, for each instance of j, it increments W by 1 with probability $1/k$, and 0 otherwise. Each instance of j has the same value $h_i(j)$, so we need to sum up all these counts. Thus

- $Y_{i,j} = \begin{cases} f_j & \text{with probability } 1/k \\ 0 & \text{otherwise.} \end{cases}$
- $\mathbf{E}[Y_{i,j}] = f_j/k$.

Then let X_i be another random variable defined

- $X_i = \sum_{j \in [n], j \neq q} Y_{i,j}$, and
- $\mathbf{E}[X_i] = \mathbf{E}[\sum_{j \neq q} Y_{i,j}] = \sum_{j \neq q} f_j/k = F_1/k = \varepsilon F_1/2$.

Now we recall the Markov Inequality. For a random variable X and a value $\alpha > 0$, then $\mathbf{Pr}[|X| \geq \alpha] \leq \mathbf{E}[|X|]/\alpha$. Since $X_i > 0$, then $|X_i| = X_i$, and set $\alpha = \varepsilon F_1$. And note $\mathbf{E}[|X|]/\alpha = (\varepsilon F_1/2)/(\varepsilon F_1) = 1/2$. It follows that

$$\mathbf{Pr}[X_i \geq \varepsilon F_1] \leq 1/2.$$

But this was for just one hash function h_i. Now we extend this to t *independent* hash functions:

$$\begin{aligned} \mathbf{Pr}[\hat{f}_q - f_q \geq \varepsilon F_1] = \mathbf{Pr}[\min_i X_i \geq \varepsilon F_1] &= \mathbf{Pr}[\forall_{i \in [t]} (X_i \geq \varepsilon F_1)] \\ &= \prod_{i \in [t]} \mathbf{Pr}[X_i \geq \varepsilon F_1] \leq 1/2^t = \delta, \end{aligned}$$

since $t = \log(1/\delta)$.

So that gives us a PAC bound. The Count-Min Sketch for any q has

$$f_q \leq \hat{f}_q \leq f_q + \varepsilon F_1$$

where the first inequality always holds, and the second holds with probability at least $1 - \delta$.

Space. Since there are kt counters, and each require $\log m$ space, then the total counter space is $kt \log m$. But we also need to store t hash functions, these can be made to take $\log n$ space each. Then since $t = \log(1/\delta)$ and $k = 2/\varepsilon$ it follows the overall total space is $t(k \log m + \log n) = ((2/\varepsilon) \log m + \log n) \log(1/\delta)$.

Turnstile Model: There is a variation of streaming algorithms where each element $a_i \in A$ can either add one or subtract one from corpus (like a turnstile at the entrance of a football game), but each count must remain positive. This Count-Min has the same guarantees in the turnstile model, but Misra-Gries does not.

13.5 Count Sketch

A predecessor of the Count-Min Sketch is the so-called Count Sketch [Charikar+Chen+Farach-Colton 02]. Its structure is very similar to the Count-Min Sketch, it again maintains a 2d array of counters, but now with for $t = \log(2/\delta)$ and $k = 4/\varepsilon^2$:

h_1	$C_{1,1}$	$C_{1,2}$	$\dots$	$C_{1,k}$
h_2	$C_{2,1}$	$C_{2,2}$	$\dots$	$C_{2,k}$
$\dots$	$\dots$	$\dots$	$\dots$	$\dots$
h_t	$C_{t,1}$	$C_{t,2}$	$\dots$	$C_{t,k}$

In addition to the t hash functions $h_j : [n] \to [k]$ it maintains t sign hash functions $s_j : [n] \to \{-1, +1\}$. Then each hashed-to counter is incremented by $s_j(a_i)$. So it might add 1 or subtract 1.

Algorithm 13.5.1 Count-Min Sketch(A)

Set all $C_{i,j} = 0$
for $i = 1$ **to** m **do**
  for $j = 1$ **to** t **do**
    $C_{j,h_j(a_i)} = C_{j,h_j(a_i)} + s_j(a_i)$

To query this sketch, it takes the median of all values, instead of the minimum.

$$\hat{f}_q = \mathsf{median}_{j \in [t]} \{C_{j,h_j}(q) \cdot s_j(q)\}.$$

Unlike the biased Count-Min Sketch, the other items hashed to the same counter as the query are unbiased. Half the time the values are added, and half the time they are subtracted. So then the median of all rows provides a better estimate. This insures the following bound with probability at least $1 - \delta$ for all $q \in [n]$:

$$|f_q - \hat{f}_q| \leq \varepsilon F_2.$$

Note this required $k = O(1/\varepsilon^2)$ instead of $O(1/\varepsilon)$, but usually the bound based on $F_2 = \sqrt{\sum_j f_j^2}$ is much smaller than $F_1 = \sum_j f_j$, especially for skewed distributions. We will discuss so-called *heavy-tailed* distributions later in the class.

13.6 A-Priori Algorithm

We now describe the *A-Priori Algorithm* for finding frequent item sets [Agrawal + Srikant 94]. The key idea is that any itemset that occurs frequently together must have each item (or any subset) occur at least as frequently.

First Pass. We first make one pass on all tuples, and keep a count for all n items. A hash table can be used. We set a threshold ε and only keep items that occur at least εm times (that is in at least ε percent of the tuples). For any frequent itemset that occurs in at least $100\varepsilon\%$ of the tuples, must have each item also occur in at least $100\varepsilon\%$ of the tuples.

A reasonable choice of ε might be 0.01, so we only care about itemsets that occur in 1% of the tuples. Consider that there are only n_1 items above this threshold. For instance if the maximum tuple size is $k_{\max}$ then we know $n_1 \leq k_{\max}/\varepsilon$. For $k_{\max} = 80$, and $\varepsilon = 0.01$, then $n_1 \leq 8000$, easily small enough to fit in memory. Note this is independent of the n or m.

Second Pass. We now make a second pass over all tuples. On this pass we search for frequent pairs of items, specifically, those items which occur in at least an ε-fraction of all tuples. Both items must have been found in the first pass. So we need to consider only $\binom{n_1}{2} \approx n_1^2/2$ pairs of counters for these pairs of elements.

After the pass, again we can then discard all pairs which occur less than an ε-fraction of all tuples. This remaining set is likely far less than $n_1^2/2$.

These remaining pairs are already quite interesting! They record all pairs that co-occur in more than an ε-fraction of purchases, and of course include those pairs which occur together even more frequently.

Further Passes. On the ith pass we can find sets of i items that occur together frequently (above an ε-threshold). For instance, on the third pass we only need to consider triples were all sub-pairs occur at least an ε-fraction of times themselves. These triples can be found as follows:

Sort all pairs (p, q) by their smaller indexed item (w.l.o.g. let this be p). Then for each smaller indexed item p, consider all completions of this pair q (e.g. a triple (p, q, r)). We only need to consider triples with (p, q, r) where $p < q < r$. Now for each pair (q, r), check if the pair (p, r) also remains. Only triples (p, q, r) which pass all of these tests are given counters in the third pass.

This can be generalized to checking only k conditions in the kth pass, and the remaining triples form a lattice.

13.6.1 Example

Consider the following dataset where I want to find all itemsets that occur in at least $1/3$ of all tuples (at least 4 times):

$$T_1 = \{1, 2, 3, 4, 5\}$$
$$T_2 = \{2, 6, 7, 9\}$$
$$T_3 = \{1, 3, 5, 6\}$$
$$T_4 = \{2, 6, 9\}$$
$$T_5 = \{7, 8\}$$
$$T_6 = \{1, 2, 6\}$$
$$T_7 = \{0, 3, 5, 6\}$$
$$T_8 = \{0, 2, 4\}$$
$$T_9 = \{2, 4\}$$
$$T_{10} = \{6, 7, 9\}$$
$$T_{11} = \{3, 6, 9\}$$
$$T_{12} = \{6, 7, 8\}$$

After the first pass 1 have the following counters:

0	1	**2**	**3**	4	5	**6**	**7**	8	**9**
2	3	**5**	**4**	3	3	**8**	**4**	2	**4**

So only $n_1 = 5$ items survive $\{2, 3, 6, 7, 9\}$.

In pass 2 we consider $\binom{n_1}{2} = 10$ pairs: $\{(2,3), (2,6), (2,7), (2,9), (3,6), (3,7), (3,9), (6,7), (6,9), (7,9)\}$. And we find the following counts:

$(2,3)$	$(2,6)$	$(2,7)$	$(2,9)$	$(3,6)$	$(3,7)$	$(3,9)$	$(6,7)$	$(\mathbf{6},\mathbf{9})$	$(7,9)$
1	3	1	2	3	0	1	3	**4**	2

We find that the only itemset pair that occurs in at least $1/3$ of all baskets is $(6, 9)$.

Thus there can be no itemset triple (or larger grouping) which occurs in all $1/3$ of all tuples since then all of its pairs would need to be in $1/3$ of all baskets, but there is only one such pair that satisfies that property.

We can now examine the association rules. And see that the count of item 6 is quite large 8, and is much bigger than that of item 9 which is only 4. Since there are 4 pairs $(6, 9)$, then every time 9 occurs, 6 also occurs.