Streaming algorithms

Scribe: Muhan Li

Introduction

(Quoted from wiki)

Streaming algorithms are designed for processing data streams, in which input is presented as a sequence of items and can be examined in only a few passes (usually just one), and algorithms have access to limited memory, which means they cannot record the whole stream for re-inspection.

Counting most frequent elements in stream

A basic problem

we are observing a contiguously flowing stream S, we want to detect the most frequently occurring element X in this stream.

A basic solution

Lets define the following algorithm, which

```
def most_frequent_element(stream):
    cur_element = null
    counter = 0
    for e in stream:
        if e = cur_element or e = null:
            cur_element = e
            counter = counter + 1
        else:
            counter = counter - 1
            if counter = 0:
                cur_element = null
    return cur_element
```

Theorem

if some element x occurs more than $\frac{n}{2}$ times in a stream with n elements, then the algorithm will return x.

Theorem proof

Define a virtual counter for each distinct element in the stream, and define the following rules:

$$counter_v(y) = \begin{cases} counter & \text{if } cur\_element = y \\ -counter & \text{otherwise} \end{cases}$$

Our goal is proving:

$$counter_v(x) > 0 \quad \text{(at the end of the algorithm.)}$$

because only one element can have a positive virtual counter, if that element is x, then $cur\_element = x$ and x will be returned.

Since for all y, $counter_v(y)$ satisfies:

- If $e = y$, $counter_v(y) += 1$ (whether $cur\_element = y \; or \; \neq y$)
- if $e \neq y$, $counter_v(y)$ may decrease by 1 (only if previous $counter_v^{prev}(y) \geq 0$), therefore $\geq counter_v^{prev}(y) - 1$

Therefore when we observe x: its virtual counter is incremented, in other cases, its virtual counter may be decremented. And:

$$\begin{aligned}
&counter_v(x) \\
&\geq \{\#\text{ occur of x}\} - (n - \{\#\text{ occur of x}\}) \\
&= 2 * \{\#\text{ occur of x}\} - n \\
&> 0
\end{aligned}$$

The Misra-Gries algorithm

Goal of Misra-Gries: Find elements that occur more than $\frac{1}{k+1}$ percent, with the following restrictions:

- Keeps at most k active elements
- For each element, it has a counter counter[e] for e

```
def misra_gries(stream):
    for e in stream
        if e is in the set of active elements:
            counter[e] += 1
        elif num_active_elements < k:
            counter[e] = 1
        else: # num_active_elements = k
            for x in active_elements:
                counter[x] -= 1
            remove x with counter[x] = 0
```

Theorem

Every element that occurs more than $\frac{1}{k+1}$ percent in stream is output by this algorithm.

Theorem proof

Define a virtual counter for all elements as:

$$counter_v[e] = k * counter(e) - \sum_{\text{x is active}} counter[x]$$

$$counter(e) = \begin{cases} counter[e] & \text{if } e \text{ is active} \\ 0 & \text{otherwise} \end{cases}$$

Claim:

If we observe e, then virtual counter $counter_v[e]$ is incremented by k, since:

1. When e is active, $counter_v[e] += k * 1$.
2. When e is not active, and $num\_active\_elements < k$, $counter_v[e] += k * 1$.
3. When e is not active, and $num\_active\_elements = k$, the sum part will decrease by $k * 1$, and the minus operator makes it increase k.

otherwise, $counter_v[e]$ doesn't decrease by more than 1, since:

1. When e is active, first part $-= k$, sum part $-= k - 1$, and turned by minus operator to $+= k - 1$, therefore total decrease is 1.

2. When e is not active, and $num\_active\_elements < k$, added elements cause a decrease of 1.
3. When e is not active, and $num\_active\_elements = k$, $counter_v[e] += k * 1$

When e occurs > $\frac{n}{k+1}$ times, where n is the number of elements in the stream, the virtual counter of e satisfies:

$$counter_v[e] >= \frac{n}{k+1} * k - (n - \frac{n}{k+1}) = 0$$

Therefore every element that occurs more than $\frac{1}{k+1}$ percent in stream will be output by this algorithm.

Parallel?

The Misra-Gries algorithm is a serial algorithm. We cannot apply this to a parallel scenario without further modifications. This problem leads us to the next algorithm, called the "count-min sketch".

The count-min sketch

Basic idea

For every distinct element e, we can return $cm(e)$, an estimate on the number of occurrences of e, in stream,

Let $m(e)$ be the true number of occurrences.

And if we can guarantee:

$$cm(e) \geq m(e)$$
$$cm(e) \leq m(e) + \epsilon n, \epsilon \ll 1 \quad w.h.p.$$

Then $cm(e)$ could be a rather accurate upper-bound of e frequency.

Design

Let $h_1, \ldots, h_k$ be 2-universal hash functions

Define a counter table: $T^{\{k,l\}}$

```
def add(x):
    for i in {1, ..., k}:
        T[i, h_i(x)] += 1

def freq(x):
    return min_i (T[i, h_i(x)])
```

Here, `freq(x)` is the $cm(e)$ function we have defined for all distinct elements in stream.

Proof

It is clear that $cm(x) \geq m(x)$ since hash collision will only increase the count of $T[i, h_i(x)]$ for all h_i.

Need to prove $cm(e) \leq m(e) + \epsilon n, \epsilon \ll 1 \quad w.h.p.$

Let S be the stream,, $\Delta = cm(e) - m(e)$:

$$
\begin{aligned}
\Delta &= T[i, h_i(x)] - m(x) \\
&= \sum_{y \in S, y \neq x} \mathbb{I}\{h_i(x) = h_i(y)\}
\end{aligned}
$$

Let l be an unknown constant representing the ratio of $h_i(x) = h_i(y)$ in all $y \neq x$, the expectation of Δ satisfies:

$$
\begin{aligned}
\mathbb{E}[\Delta] &= \frac{\mathbb{I}\{y \in S, y \neq x\}}{l} \\
&\leq \frac{n - m(x)}{l}
\end{aligned}
$$

And with markov inequality we can bound Δ with:

$$
\begin{aligned}
P\{\Delta \geq \epsilon n\} \leq \frac{(n/l)}{\epsilon n} = \frac{1}{\epsilon l} \\
P\{\forall i, \Delta_i \geq \epsilon n\} \leq (\frac{1}{\epsilon l})^k
\end{aligned}
$$

Let $l = \lceil e/\epsilon \rceil$:

$$
P\{cm(x) \geq m(x) + \epsilon n\} \leq \frac{1}{e^k}
$$

Performance

Memory consumption of the count-min sketch:

The number of counters $= k * l \simeq (ek/\epsilon)$

Parallel:

The sketches (hash table) can be combined, and this is easily parallel-able.

Counting distinct elements in stream

In this problem, we would like to count the number of distinct elements in stream.

Algorithm 1 (one-half sparse sampling)

- D - dict / hash table
- h - hash function (2-universal / ROM)

  h: u -> {0, ..., 2^M-1}

```
def process(stream):
    for x in stream:
        s1(y) - is the last bit of y
        if s1(h(x)) = 0:
            add x to D

    return 2 * size(D)
```

Memory consumption of this algorithm: store $n/2$ elements for all n distinct elements

Analysis

The randomness of this algorithm comes from $h(x)$, which is a random binary string of length M.

Why not choose with a random variable but use the last bit of the hash string? Because if a random variable is used to sample elements, then for each distinct element e, the union bound probability being sampled at least once is:

$$\{\text{\# of occurence}\} * \frac{1}{2} / \{\text{\# of all elements}\} \neq \frac{1}{2}$$

However, using last bit of hash guarantees that the sampling probability is exactly $\frac{1}{2}$, since same element hash to the same value, and sampled probability does not depend on frequency..

Let the number of distinct elements be n, let ALG be the number of distinct elements output by our algorithm, using the Chernoff bound:

$$\begin{aligned}
&P\{|\frac{ALG}{2} - \frac{n}{2}| > \frac{\delta n}{2}\} \\
&= P\{|ALG - n| > \delta n\} \\
&\leq 2 * e^{-\frac{\delta^2 n}{6}} \\
&(\text{6 because we are using half of the elements, u} = \text{n}/2)
\end{aligned}$$

Algorithm 2 ($1/2^k$ sparse sampling)

Define k, a parameter

```
def process(stream):
    for x in stream:
        let sk(y) be the suffix of y of length k
        if sk(h(x)) = "0...0" (all 0 sequence of length k):
            Add h(x) to D
    return 2^k * len(D)
```

Let random variable $X_i = \mathbb{I}\{s_k(h(x)) = 0...0\}$. There are:

$$P\{X_i = 1\} = \frac{1}{2^k}$$
$$E[\sum X_i] = \frac{n}{2^k}$$

memory consumption: $m = \frac{n}{2^k}$ in expectation, which decreases exponentially as k increases (however, k couldn't be too large as we need to guarantee the error bound), using the Chernoff bound:

$$P\{|ALG - n| \geq \delta n\} \leq 2 * e^{-\frac{\delta^2 n}{3 * 2^k}}$$

Problem

if n is small, the hitting probability (with specified hash suffix) is low, and since n is not known in advance, we cannot fix parameter k. We want to design an algorithm which dynamically adapts k according to what it has seen from the stream.

Algorithm 3 (Adaptive k)

```
def process(stream):
    Get m*
    k = 1
    for x in stream:
        if z(x) >= k:
            Add (x, z(x)) to D
        if len(D) >= m*:
            Remove (x, z(x)) with z(x) <= k
            k += 1
    return 2^k * len(D)
```

Algorithm construction and analysis

Let function $z(y)$ return the longest contiguous 0 suffix length of binary sequence y, Eg: $z(1001000)$ return 3.

If we store $(x_1, z(x_1)), \ldots, (x_n, z(x_n))$, and for $k = 1, \ldots, log_2 n$, count the number of elements with $z(x) \geq k$,

two observations: (second is done with union bound):

1. $\forall k$, $P\{|ALG_k - n| \geq \delta n\} \leq 2e^{-c\frac{\delta^2 n}{2^k}}$ for some constant c
2. With union bound and observation 1:

$$P\{\forall k \leq k^*, |ALG_k - n| \geq \delta n\} \leq k^* * 2e^{-c\frac{\delta^2 n}{2^{k^*}}}$$

   (actually there is a better bound by using the properties of a geometric series, since it decrease exponentially fast from k* to k=0)

$$P\{\forall k \leq k^*, |ALG_k - n| \geq \delta n\} \leq \sum_{k=0}^{k^*} e^{-\frac{\delta^2 n}{3*2^k}} < 2e^{-\frac{\delta^2 n}{3*2^k}} = 2e^{-\frac{\delta^2 m}{3}}$$

Let $m* = n/2^{k*}$ (delta is a fixed value, by solving the error rate, we can compute m*)

Eg: let $\delta = 0.05$, maximum error probability $= 1\%$:

$$2e^{-\frac{0.05^2 m}{3}} \leq 0.01$$
$$m \geq \ldots$$