COMP 480/580 – Probabilistic Algorithms and Data Structure Oct 3, 2024

Lecture 11

Lecturer: Ansumali Shrivastava
Scribe By:
Michael Menezes (mnm5) Balavanthapu Sanjeev (sb200) Daniel Terrazas (jdt11)

1 Count-Sketches

1.1 Problem Definition

We are looking at a class problems where we have this idea of counting items in a stream arriving one after another. In addition, there are too many of them to store all at once. For example, we observe the objects $o_1, o_5, o_{10}, o_1, o_{2000}, o_{10}, o_1, ...$ We continue this for 2 billion items and at time t we want to know how many instances of O_1 were observed. We could have a very long vector of all the unique items.

$[c_1, c_2, ..., c_m]$	the count of the ith object
$[o_1, o_2, ..., o_m]$	the ith object

Now, we have a vector and counts can be initialized to 0 with the vector is evolving over time. At every time instance the counts are being updated. This is not a problem if we can store this. But in practical settings we have a small amount of memory. Can we do it reasonably in a small memory? As we have seen with bloom filters and general hashing functions, there's a tradeoff between accuracy and space. In this case, the array is giving an estimate where the length of the array will determine the error.

Although we have seen how power of k choices allows us to trade off a linear reduction in space for an exponential increase in accuracy, for now, we simplify to one array and one hash function. Let $h(o)$ be a hash function. Then, every time we observe o_1, $h(o_1)$ will hash it to the same spot. Maybe there are collisions. If item i collides with item j the whole count goes into the array. There are two ways to visualize, we can see this as streaming where we see one element at a time or as remapping an large vector to a small vector. Either way, the core problem is:

Given a stream of incoming data objects, how can we keep track of the data using limited memory?

1.2 Building an Estimator

We know that some problems are easier than others. For example, keeping track of the total count of objects is simple. We just have a counter and increment it for each observation. In contrast to this, getting the counts of a subset is much harder. In a similar vein, getting $\sum C_i^2$ is not obvious. But possible to approximate as discussed in section 3. We will also see the $\sum C_i^0$ variation in section 2.

When trying to get the count of an arbitrary element in the stream, there are only so many things we can do since we can't store everything. For each object we observe, we can hash it to a spot in a small array and we can decide the amount that is added or removed at time t denote this by Δ_i^t. We constrain each Δ_i^t to depend on the o_i being hashed. This allows us to have consistency which as we'll see allows us to recover an estimator for the count of that object.

We start with the intuitive formula for the size of the count at the spot where we plan on hashing o_i.

$$A[h(o_i)] = \sum_{j=1}^{h} I_{\{h(o_j)=h(o_i)\}} C_j f(j)$$

Note I is an indicator random variable and thus A is also a random variable. Essentially, for each object, o_j, hashed to the same spot as o_i, we add their counts multiplied by a constant that depends on j. We are

interested in grabbing the count of a particular object $C_i = \sum_{t=1}^{\infty} \Delta_i^t$. So, we look at the expectation of our formula from before.

$$E[A[h(o_i)]] = C_i f(i) + \frac{1}{R} \sum_{j \neq i} C_j E[f(j)]$$

where R is the size of the reservoir. But now what do we do with the random variable $f(j)$? Since we want the term to cancel, we use the deferred decision of probability to flip a coin for each Δ and attach a probability to our function $f(j)$. Now, to get the count we divide everything by $f(i)$.

$$\frac{E[A[h(o_i)]]}{f(i)} = \frac{C_i f(i)}{f(i)} + \frac{1}{R} \sum_{j \neq 1} C_j \frac{E[f(j)]}{f(i)}$$

Recall, we aim to get $\sum_{j \neq 1} C_j \frac{E[f(j)]}{f(i)}$ as 0. We change out our $f(i)$ for $S(i)$ where $S(i) : \mathbb{N} \to \{-1, 1\}$ determines the sign. The big idea compared to count-min sketch is that instead of just hashing and counting, we now hash, inject, and count. Of course, this algorithm struggles when two heavy hitters collide with opposite signs, but this can be made extremely rare using power of k choices. Although this algorithm produces an unbiased estimator, using $\Delta := 1$ is not better than count-min because of failure probability is a bit higher.

$$S(k)A[h(k)] := \sum_{j=1}^{h} I_j C_j S(j) S(k) = C_k + \sum_{j \neq k} E[I_{kj}] C_j E[S(j)] E[s(k)]$$

1.3 Expectation

From the previous subsection, we have

$$\frac{E[A[h(o_i)]]}{f(i)} = C_i \frac{f(i)}{f(i)} + \frac{1}{R} \sum_{j \neq i} C_j \frac{E[f(j)]}{f(i)}$$

We let $f(i) = S(i) = 1$ or -1

$$E[A[h(o_i)] \cdot S(i)] = C_i E[S(i)^2] + \frac{1}{R} \sum_{j \neq i} C_j E[S(j)] E[S(i)]$$

$$E[A[h(o_i)] \cdot S(i)] = C_i S(i)^2$$

$$E[A[h(o_i)] \cdot S(i)] = C_i$$

1.4 Error

Using Chebyshev's, we will find a bound on the error and estimate the probability of error. First, we find the variance. Let $\hat{C}_i = A[h(o_i)] \cdot S(i)$:

$$Var(\hat{C}_i) = E[(\hat{C}_i - E[\hat{C}_i])^2]$$

$$= E[(\hat{C}_i - C_i)^2]$$

$$= E\left[\left(\sum_{j \neq i} I_j C_j S(j) S(i)\right)^2\right]$$

$$= E\left[\sum_{j\neq i} I_j^2 C_j^2 S(j)^2 S(i)^2 + \sum_{j\neq k} I_{kj} C_j C_k S(j) S(k) S(i)^2\right]$$

Since $E[S(i)] = 0$, our right sum becomes zero. $E[I_j^2] = E[I_j] = \frac{1}{R}$. Then,

$$Var(\hat{C}_i) = \frac{1}{R}\sum_{j\neq i} C_j^2$$

$$= \frac{\|C\|_2^2 - c_i^2}{R}$$

$$= \frac{\|C_{-i}\|_2^2}{R}$$

Where C_{-i} is the set of all counts minus C_i. Now, we can use Chebyshev's inequality to get a bound on the error.

$$P(|\hat{c}_i - c_i| \geq k\,\|C_{-i}\|_2) \leq \frac{1}{k^2 R}$$

Now, given desired error bound $k\,\|C_{-i}\|_2$ with probability δ. We can choose R such that:

$$R = \frac{1}{k^2\delta}$$

2 Count-Min Sketch

2.1 Methodology

The *Count-Min Sketch* (CMS) is a solution to tackle the problem of efficiently estimating the frequencies of elements in a high-volume data stream where storing the counts for each unique item is not possible due to memory constraints. Here we can be dealing with thousands or even billions of unique data particles. Given a stream of items $o_1, o_2, \ldots, o_n$ from a large universe U, CMS utilizes a two-dimensional array of counters $A[d][w]$ along with d independent hash functions $h_1, h_2, \ldots, h_d$, each mapping items to one of w possible positions. For every incoming item o_i, CMS increments the counters at positions $A[j][h_j(o_i)]$ for each hash function $j = 1, 2, \ldots, d$. To estimate the count $\hat{C}_x$ of any specific item x, CMS retrieves the minimum value across the corresponding counters:

$$\hat{C}_x = \min_{1\leq j\leq d} A[j][h_j(x)]$$

This method ensures that $\hat{C}_x$ is an upper bound of the true count C_x, with the error $\hat{C}_x - C_x$ bounded by ϵN with high probability, where $N = \sum_{k=1}^{n} C_k$ is the total number of items in the stream. The parameters d and w are chosen based on the desired error tolerance ϵ and confidence level $1 - \delta$, typically setting:

$$w = \left\lceil \frac{e}{\epsilon} \right\rceil \quad \text{and} \quad d = \left\lceil \ln\left(\frac{1}{\delta}\right) \right\rceil$$

This allows CMS to provide space-efficient, real-time frequency estimations suitable for large-scale data processing applications.

2.2 Explanation with Example

2.2.1 Simple Example

Consider a stream of words:

$$\text{apple}, \text{banana}, \text{apple}, \text{cherry}, \text{apple}, \text{banana}$$

Assume:

- Number of hash functions (d) = 2
- Width of each hash table (w) = 5
- Hash functions defined as:

$$h_1(\text{apple}) = 1, \quad h_1(\text{banana}) = 2, \quad h_1(\text{cherry}) = 3$$
$$h_2(\text{apple}) = 2, \quad h_2(\text{banana}) = 1, \quad h_2(\text{cherry}) = 4$$

2.2.2 Updating the CMS

Hash Function	0	1	2	3	4
h_1	0	0	0	0	0
h_2	0	0	0	0	0

Processing each word:

2.3 Processing Stream Items

1. **apple**: $h_1(\text{apple}) = 1$, $h_2(\text{apple}) = 2$

$$A[h_1(\text{apple})][1] += 1 \Rightarrow A[h_1][1] = 1$$
$$A[h_2][2] += 1 \Rightarrow A[h_2][2] = 1$$

Hash Function	**0**	**1**	**2**	**3**	**4**
h_1	0	1	0	0	0
h_2	0	0	1	0	0

tableCMS Matrix after processing "apple"

2. **banana**: $h_1(\text{banana}) = 2$, $h_2(\text{banana}) = 1$

$$A[h_1(\text{banana})][2] += 1 \Rightarrow A[h_1][2] = 1$$
$$A[h_2][1] += 1 \Rightarrow A[h_2][1] = 1$$

Hash Function	**0**	**1**	**2**	**3**	**4**
h_1	0	1	1	0	0
h_2	0	1	1	0	0

tableCMS Matrix after processing "banana"

3. **apple**: $h_1(\text{apple}) = 1$, $h_2(\text{apple}) = 2$

$$A[h_1(\text{apple})][1] += 1 \Rightarrow A[h_1][1] = 2$$
$$A[h_2][2] += 1 \Rightarrow A[h_2][2] = 2$$

Hash Function	**0**	**1**	**2**	**3**	**4**
h_1	0	2	1	0	0
h_2	0	1	2	0	0

tableCMS Matrix after processing "apple" again

4. **cherry**: $h_1(\text{cherry}) = 3$, $h_2(\text{cherry}) = 4$

$$A[h_1(\text{cherry})][3] += 1 \Rightarrow A[h_1][3] = 1$$
$$A[h_2][4] += 1 \Rightarrow A[h_2][4] = 1$$

Hash Function	0	1	2	3	4
h_1	0	2	1	1	0
h_2	0	1	2	0	1

tableCMS Matrix after processing "cherry"

5. **apple**: $h_1(\text{apple}) = 1$, $h_2(\text{apple}) = 2$

$$A[h_1(\text{apple})][1] += 1 \Rightarrow A[h_1][1] = 3$$

$$A[h_2][2] += 1 \Rightarrow A[h_2][2] = 3$$

Hash Function	0	1	2	3	4
h_1	0	3	1	1	0
h_2	0	1	3	0	1

tableCMS Matrix after processing "apple" again

6. **banana**: $h_1(\text{banana}) = 2$, $h_2(\text{banana}) = 1$

$$A[h_1(\text{banana})][2] += 1 \Rightarrow A[h_1][2] = 2$$

$$A[h_2][1] += 1 \Rightarrow A[h_2][1] = 2$$

Hash Function	0	1	2	3	4
h_1	0	3	2	1	0
h_2	0	2	3	0	1

tableCMS Matrix after processing "banana"

2.3.1 Final Count Matrix

Hash Function	0	1	2	3	4
h_1	0	3	2	1	0
h_2	0	2	3	0	1

2.3.2 Estimating Counts

To estimate the count of an item, take the minimum value from the corresponding counters across all hash functions.

$$\hat{C}_{\text{apple}} = \min(A[h_1(\text{apple})][1], A[h_2(\text{apple})][2]) = \min(3, 3) = 3$$
$$\hat{C}_{\text{banana}} = \min(A[h_1(\text{banana})][2], A[h_2(\text{banana})][1]) = \min(2, 2) = 2$$
$$\hat{C}_{\text{cherry}} = \min(A[h_1(\text{cherry})][3], A[h_2(\text{cherry})][4]) = \min(1, 1) = 1$$

2.4 Theorem and Proof

Theorem 1 (Count-Min Sketch Error Bound)**.** *Let $A[d][w]$ be a Count-Min Sketch with d independent hash functions $h_1, h_2, \ldots, h_d$, each mapping items to one of w columns. For any item x with true count C_x, the CMS estimate $\hat{C}_x$ satisfies:*

$$C_x \leq \hat{C}_x \leq C_x + \epsilon N$$

with probability at least $1 - \delta$, provided that:

$$w = \left\lceil \frac{e}{\epsilon} \right\rceil \quad \textit{and} \quad d = \left\lceil \ln\left(\frac{1}{\delta}\right) \right\rceil$$

where $N = \sum_{k=1}^{n} C_k$ is the total number of items in the stream, ϵ is the error tolerance, and δ is the failure probability.

Proof. **Lower Bound ($C_x \leq \hat{C}_x$)**:

Each counter $A[j][h_j(x)]$ for $j = 1, 2, \ldots, d$ is incremented every time item x appears. Therefore, the minimum value across these counters is at least C_x:

$$\hat{C}_x = \min_{1 \leq j \leq d} A[j][h_j(x)] \geq C_x$$

Upper Bound ($\hat{C}_x \leq C_x + \epsilon N$):

The overestimation in each counter $A[j][h_j(x)]$ is caused by collisions with other items. The expected overestimation for a single hash function is:

$$\mathbb{E}[A[j][h_j(x)]] = C_x + \frac{N - C_x}{w} \leq C_x + \frac{N}{w}$$

Using Markov's Inequality, the probability that the overestimation exceeds ϵN for one hash function is:

$$\Pr\left[A[j][h_j(x)] - C_x \geq \epsilon N\right] \leq \frac{\mathbb{E}[A[j][h_j(x)] - C_x]}{\epsilon N} = \frac{N/w}{\epsilon N} = \frac{1}{w\epsilon}$$

To ensure that none of the d hash functions exceed the error bound simultaneously, we set:

$$\frac{1}{w\epsilon} \leq \frac{\delta}{d}$$

Choosing:

$$w = \left\lceil \frac{e}{\epsilon} \right\rceil \quad \text{and} \quad d = \left\lceil \ln\left(\frac{1}{\delta}\right) \right\rceil$$

ensures that:

$$\Pr\left[\hat{C}_x > C_x + \epsilon N\right] \leq d \cdot \frac{1}{w\epsilon} \leq \delta$$

Thus, with probability at least $1 - \delta$, the estimate $\hat{C}_x$ does not exceed $C_x + \epsilon N$. □