CSE 312: Foundations of Computing II **Summer 2020**

Section 4: Approximating Distinct Elements in a Stream

Thursday, July 16th *Scribe: Pemi Nguyen*

4.1 Overview

This section describes streaming algorithm to approximate distinct elements, a probabilistic algorithm to approximate the number of unique elements in a stream of data.

4.2 Motivations

Imagine you want to process large amounts of data but your company has limited storage capacity. For example, you want to count the number of distinct views for videos on Youtube. Your streaming model might possibly have to process billions of views to count billions of unique views. We definitely don't want to store every single user ID, because it's a waste of memory.

What can you do? A naive approach would be to use a hash set to store each new user IDs as you receive it. However, this would require a very large hash set. The memory required would be at least linear in the number of distinct entries, which would already be too much. In worse-case scenario, if there are N unique elements in a stream of N data points, then the space complexity will be in $O(N)$.

Another approach might be to attempt to use some subset of the IDs as a representative of the entire stream, e.g. by counting the number k_m of distinct elements in some random selection of k elements of the n entries and estimating the actual number k of distinct elements as $\frac{n}{m}k_m$. It turns out this type of approach fails as well, roughly because you can't reliably gain information about things that appear extremely infrequently reliably. Views on Youtube don't follow any certain distribution.

4.3 Data Stream Model

Computers implementing query operations or computations on a massive data set nowadays (usually that data set doesn't fit on a single computer or polynomial runtime is not efficient) follows the Data Stream Model, in which a streaming algorithm can only access data in a single pass, and only one piece at a time. Therefore at any moment, the computer can only have access to its own memory and a single piece of the input data. Therefore, we would like to come up with an algorithm that only does one pass through the data stream while maintaining a space complexity of less than $O(logN)$, where N is the total number of elements in a stream.

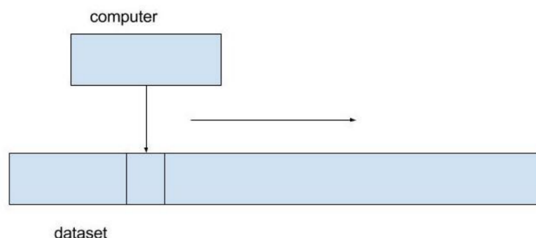

Figure 4.1: Data Stream Model

4.4 An estimator for approximating distinct elements in a stream

It turns out the biggest problem we have is how to save space without having to store all unique elements. With billions of data, we certainly don't need an exact count to every single digit, and a approximation within a reasonable margin of error is totally fine. This motivates us to think about using a randomized approach to approximate the number of distinct elements in a stream.

Idea: Suppose we have the following stream of integers:

$$S = [32, 5, 17, 32, 14, 5, 17, 5, 32, 17]$$

We can easily see that this stream has 4 distinct elements.

To get an estimate of k, the number of distinct elements, without having to store all elements in a hashing data structure (which means we want $k \approx 4$), we can use a hash function h [1] that maps each distinct integer x_i in the input stream to a uniformly random real number in [0, 1]. Define Y to be the smallest hash values of k distinct elements:

$$Y := \min_{i=1,\ldots,k} h(x_i)$$

We constantly update Y every time we see a smaller hash value $h(x_i)$ than the current minimum hash value. This is why this estimator is constant space.

We can estimate k by:

$$k := \frac{1}{Y} - 1$$

For example, if $h(32) = 0.43$, $h(5) = 0.19$, $h(17) = 0.85$ and $h(14) = 0.61$, then Y would be equal to 0.19 at the end of the process. Thus:

$$k = \frac{1}{0.19} = 4.26$$

We can confirm that k is approximately 4. This gives us an idea on how to approximate the number of distinct elements in a stream.

[1] It is hard to numerically define a hash function that can uniformly distribute the hash values of a set of inputs, no matter how big the set of inputs is. This requires lots of advanced knowledge about systems, which we don't expect you to understand all the details. For implementation of Bloom filter and Distinct elements, we both use **Murmurhash**, which is a consistent function that allows us to uniformly distribute a set of inputs over a range. You can check how we implement a hash function using **Murmurhash** in the coding portion for pset3.

Let h be a hash function that maps a set of values to a uniformly set of uniformly distributed random real numbers in [0, 1]. (We can write this as: $h : U \to [0, 1]$). Let Y be an estimator for the minimum of k distinct hash values: $Y = \min_{i=1,...,k} h(x_i)$ (roughly speaking, an estimator is a statistic that states some facts about a certain distribution).

If there are k distinct elements x_1, x_2,...,x_k, then:

$$\mathbb{E}[Y] = \frac{1}{k+1}$$

.
We will prove this property in order to show that we can estimate k by using $k = \frac{1}{Y} - 1$.

Intuition for the proof:

1. The larger the number of distinct elements in a stream are, the smaller the expected minimum-value hash will be (because we assume the all hash values mapped from a set of inputs are uniformly distributed, so we will have more chance at getting a really small value as the number of inputs are bigger).

2. If you space out k elements evenly and uniformly in $[0, 1]$, then $\frac{1}{k+1}$ is the exact value of the minimum-value hash.

Proof: $Y = \min(h(x_1), h(x_2), ...h(x_k))$. Then $P(h(x_i) \leq t) = t$. By independence, we have $P(Y > t) = (1 - t)^k$. Use the PDF formula, you will have (this is what you have have to derive for pset 3):

$$\mathbb{E}[Y] = \frac{1}{k+1}$$

Therefore, this estimate for k, the number of distinct elements in a stream of data, is correct by **expectation**. One of the advantages of using this estimator is that the space complexity will be in $O(1)$. Let's take a look at the algorithm based on this idea:

Algorithm 1 Distinct Elements algorithm

```
function ESTIMATE(arr)        ▷ Get an estimate of distinct elements from stream arr with length N
    Initialize a hash function h which hashes a elements uniformly across [0, 1].
    val ←∞
    for i = 1, 2, ..., N do
        val ← min(val, h(arr[i]))
    end for
    return ⌊1/val − 1⌋
```

Using this estimate alone could be very bad. It's not enough for an estimator to be correct in expectation. Let's calculate the variance of Y. The proof is left to you:

$$\text{Var}(Y) = \frac{k}{(k+1)^2(k+2)} \approx \frac{1}{(k+1)^2}$$

Thus, the standard deviation of Y is:

$$\text{std}(Y) = \sqrt{\text{Var}(Y)} = \frac{1}{k+1} = \mathbb{E}[Y]$$

The estimator Y can be considered a random variable with mean $\frac{1}{k+1}$ and standard deviation $\frac{1}{k+1}$. As we can see, the standard deviation is basically equal to the expectation (at large value of k). Because of this high variance, it is very likely that Y is more than one standard deviation from the mean, which will certainly yield a bad estimate. We want to find a better estimator that has a less variance so that our estimate can be more likely to be closer to the expected value.

Optional: We can prove that Y is not a good estimator using **Chebyshev's inquality** (which you will learn later in this class):

$$P(|Y - \mathbb{E}[Y]| > \epsilon\mathbb{E}[Y]) \leq \frac{\text{Var}(Y)}{(\epsilon\mathbb{E}[Y])^2} \approx \frac{1}{\epsilon^2}$$

Ideally we want to choose an $\epsilon < 1$ to get a better bound for our probability. However, in that case $P(|Y - \mathbb{E}[Y]| > \epsilon\mathbb{E}[Y]) \leq 1$, which is a useless bound.

4.5 Taking the mean of t estimators in 4.4

As we can see, Y is not a good estimator because of its large variance. Ideally, we want to have an estimator with a tighter variance so that when running the algorithm, we're more likely to get an estimated value very close to $\mathbb{E}[Y]$.

Idea: In problem 2.d of pset 2, we have derived the following lemma:

Lemma 4.5.1: Expectation and standard deviation of a sample mean

Let X_1, X_2,..., X_n be n independent and identically distributed variables from a distribution. The sample mean is $\overline{X} = \frac{1}{n}\sum_{i=1}^{n} X_i$. We have the following properties:

$$\mathbb{E}[\overline{X}] = \mathbb{E}[X]$$

$$\text{Var}(\overline{X}) = \frac{1}{n}\text{Var}(X)$$

Since Y can be considered a random variable that follows a certain distribution, we can use this idea to tighten the variance of Y by using a different estimator $Z = \overline{Y}$ instead, which is the mean of t independent versions of Y.

Construct t hash functions h_1, h_2,..., h_t, where each one maps a set of values to a uniformly set of uniformly distributed random real numbers in [0, 1]. Define:

$$Z = \frac{1}{t}\sum_{i=1}^{t} Y_i$$

We thus have:

$$\mathbb{E}[Z] = \mathbb{E}[Y] = \frac{1}{k+1}$$

$$\text{Var}(Z) = \frac{1}{t}\text{Var}(Y) = \frac{1}{t(k+1)^2}$$

In other words, by using t hash functions, we've reduced the variance by a factor of t.

We have an **improved algorithm**:

Algorithm 2 An improved Distinct Elements algorithm using a mean of estimators

```
function ESTIMATE(arr)        ▷ Get an estimate of distinct elements from stream arr with length N
    Initialize t independent hash functions h_j, each of which hashes elements uniformly across [0, 1].
    Initialize t values val_j to ∞
    for i = 1, 2, ..., N do
        for j = 1, 2, ..., t do
            val_j ← min(val_j, h_j(arr[i]))
        end for
    end for
    return mean_{1≤j≤t} ( ⌊ 1/val_j − 1 ⌋ )
```

Optional: Using Chebyshev's inequality, we have:

$$P\left(Z - \mathbb{E}[Z] \geq \epsilon\mathbb{E}[Z]\right) \leq \frac{\text{Var}(Z)}{\left(\epsilon\mathbb{E}[Z]\right)^2} \approx \frac{1}{\epsilon^2 t}$$

Therefore, by introducing and increasing t, we can reduce the probability of returning a value outside of the range $[(1-\epsilon)\mathbb{E}[Z], (1+\epsilon)\mathbb{E}[Z]]$ (by a factor of t).

4.6 Areas for improvement

Some lingering questions for you to think about (which might potentially appear in future problem sets):

- We know that we don't have to use a big t close to the size of a stream to get a good estimate. Still, how big t should be to get an estimate within a decent margin of error? In the coding portion of pset 3 for Distinct elements, you can try increasing t yourself to see how the approximation of distinct elements is improved.
- If we keep on increasing the value of t, the estimate will be better but it will take a large t to get a good approximation, especially when the exact number of distinct elements is very big (which still puts us under a space constraint). Is there a way to keep improving this estimator to get a better estimate? One idea is to use median of means. We can get s independent copies of the estimator Z and return median$(Z_1, Z_2, ..., Z_s)$ are our improved estimate of the number of distinct elements.

References

[1] http://web.stanford.edu/class/cs369g/files/lectures/lec1.pdf

[2] https://courses.cs.washington.edu/courses/cse312/16au/slides/DistinctElements.pdf