14 Distinct Counting and Mergability

This lecture will cover two important topics in Data Mining very large data: distinct item counting, and mergeability for generalization to large data.

The *Distinct Counting* problem is to determine the number of distinct items seen in a stream. The challenge is that we do not want to store a unique identifier for each item we observe, since that may take too much space. But if we see an items, and it comes again, we do not want to count that twice! This is an important statistic for websites who want to know how many *unique* visitors came to their site – and do not want to over-count if the same person just comes over and over again.

The second challenge is the *Mergeability* of data sketches. This property holds if we can create data summaries of two separate data sets, and then can combine the two summaries (without looking back at the original data) so that the resulting summary is as good as if it were created over the full data set. As good means that the trade-off between the space used and accuracy is maintained. This is useful when the data stream is too big, fast, or distributed for one compute to handle (so it is distributed to multiple machines), but we want to be able to have a single summary of all of the data in the end. It also allows for a wide variety of other big data settings where we can *arbitrarily* divide the data, process in parallel or somehow separately, and then combine together the results separately. The fact that the split can be arbitrary makes the methods very flexible, and often saves rounds of communication.

14.1 Reminder on 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.

Today we focus on F_0. Computing it exactly requires $\Omega(\min\{m, n\})$ space. We start with a warm-up exercise of a Bloom Filter.

14.2 Bloom Filters

The goal of a Bloom Filter [Bloom 1970] is to maintain a subset $S \subset [n]$ in small space. It must maintain all items, but may allow some items to appear in the set even if they are not. That is, it allows *false positives*, but not *false negatives*.

It maintains an array B of k bits. Each is initialized to 0. It uses k (random) hash functions $\{h_1, h_2, \ldots, h_t\} \in \mathcal{H}$. It then runs streaming Algorithm 14.2.1 on S.

Then on a query to see if $y \in S$, it returns YES only if for *all* $j \in [t]$ that $B[h_j(y)] = 1$. Otherwise it returns NO.

Algorithm 14.2.1 Bloom(S)

for $x \in S$ **do**
  for $j = 1$ **to** t **do**
    Set $B[h_j(x)] = 1$

So any item which was put in B it sets all associated hash values as 1, so on a query it will always return YES (no false negatives).

However, it may return YES for an item even if it does not appear in the set. It does not even need to have the exact same hash values from another item in the set, each hash collision could occur because a different item.

Analysis: We consider the case with $|S| = n$ items, where we maintain k bits in B, and use t hash functions.

$$
\begin{aligned}
1 - 1/k &\text{ Probability a bit not set to 1 by a single hash function} \\
(1 - 1/k)^t &\text{ Probability a bit not set to 1 by } k \text{ hash functions} \\
(1 - 1/k)^{tn} &\text{ On inserting } n \text{ elements, probability a bit is 0 } (\star) \\
1 - (1 - 1/k)^{tn} &\text{ On inserting } n \text{ elements, probability a bit is 1} \\
(1 - (1 - 1/k)^{tn})^t &\text{ Probability a false positive (using } k \text{ hash functions)} \\
\approx (1 - e^{-tn/k})^t &
\end{aligned}
$$

The "right" value of t is about $t \approx (k/n) \ln 2$.

($\star$) *This step is not quite right, although it is not too far off. It assumes independence of which bits are being set. Unfortunately, in most presentations of Bloom filters (and some peer-reviewed research papers) this is presented as the right analysis, but it is not. But it gives a pretty good illustration of how it works. The correct analysis is much more complicated.*

14.3 Distinct Item Counting

Can we use a Bloom Filter to estimate the number of distinct items?
No, at least not with poly$(\log m, \log n)$ space. After seeing $\approx \sqrt{k}$ distinct items, by the Birthday paradox sort of analysis, we expect every bit to be 1. So if there roughly $m/100$ distinct items, with less than $\approx \sqrt{m}/10$ bits of space, we do not have any information.

Hashing to continuous values. What if we stored continuous values, e.g., we had $h \sim \mathcal{H}$ so that

$$h : [n] \to [0, 1]$$

it produced a continuous value like we did for weighted random sampling? How does this help, there are no collisions (except for repetitions of the same items)?: Compute the min value!

$$\gamma_h = \min_{a_i \in A} h(a_i).$$

First note that since we pass everything through the hash function, then this gives the same result if we insert another item a_{m+1} in A that has the same values a previous item a_i. They hash to the same values. So the minimum has the same values.

Second, since the hash function is random, and what $h(q)$ returns is uniform over $[0,1]$ given that we fix q and choose $h \sim \mathcal{H}$ at random. Thus, without knowing h, the value γ_h only depends on the number of distinct items, not their values.

Third, we can analyze $\hat{\gamma} = \mathbf{E}_{h\sim\mathcal{H}}[\gamma_h]$. Since γ_h is a random variable with randomness depending on the choice of $h \sim \mathcal{H}$, given a fixed set of distinct items in the stream. Again because of the second observation, if the choice of h is random, it only depends on the *number* of distinct items in A. In fact

$$\hat{F}_0 = 1/\gamma_h - 1 \qquad \text{satisfies} \qquad \mathbf{E}_{h\sim\mathcal{H}}[\hat{F}_0] = F_0$$

Amplification. We can improve this by doing repeated trials. Instead of one hash function $h \sim \mathcal{H}$ we can consider t iid hash functions $h_1, h_2, \ldots, h_t \sim_{iid} \mathcal{H}$. Now let

$$\gamma = \frac{1}{t}\sum_{i=1}^{t} \gamma_{h_i}$$

and then our improved estimate as

$$\hat{F}_0 = 1/\gamma - 1.$$

We still can claim that $\mathbf{E}_{h\sim\mathcal{H}}[\hat{F}_0] = F_0$, but more importantly if we set $t \approx \frac{1}{\varepsilon^2}\frac{1}{\delta}$ then with probability at least $1-\delta$ we have

$$|\hat{F}_0 - F_0| \leq \varepsilon F_0.$$

Limitations. There are a couple reasons this method is still far from the method of choice.

First, the tail bound, and in particular, the dependence on δ (the probability of exceeding an ε-error bound) is too large. We should be able to get a size the depends at most logarithmically like $\log(1/\delta)$. This leads to the accuracy not being that trustworthy.

This can be improved with the so-called *median trick*. For this, we run the above algorithm with our choice of ε and $\delta' = 1/2$ (a constant), so $t = \frac{1}{\varepsilon^2}\frac{1}{1/\delta'} = O(1/\varepsilon^2)$. For each run we get an estimate $\hat{F}_0$. Now we repeat this $T = O(\log 1/\delta)$ times, and take the *median* of these T estimates. The total space is now $O(1/\varepsilon^2)T = O(\frac{1}{\varepsilon^2}\log\frac{1}{\delta})$, and it can be shown to achieve the same asymptotic error.

Second, the value γ can get quite small for very large sets. And then one needs accuracy at the very small values part of $[0,1]$. Using a float or double may or may not work well. We would rather consider a mechanism that considers bit representations to be very space efficient.

14.3.1 HyperLogLog

We now overview an algorithm analysis developed by Flajolet (and Martin) over several papers. Now imagine the random number the hash functions map to is represented as bits (e.g., 0101011). In this way 0000000 represents 0 and 1111111 represents (almost) 1. In general the j bit counts as $1/2^j$ towards the total values in $[0,1)$ so 0101011 represents $1/2^2 + 1/2^4 + 1/2^6 + 1/2^7 = 0.3359375$. As the number of bits get larger, then this gets closer and closer to approximate the continuous values in interval $[0,1)$.

So how do we store the smallest number we have seen so far (i.e. $\min_{a_i\in A} h(a_i)$)? Well, it is still just the sequence of bits, and probably is something like 0000101 where it starts with many 0s, since it is small. We can rule out most larger numbers if they have the first 1 bit occur earlier (e.g., 0010000) without looking at all bits, just where the first one bit occurred.

Flajolet and Martin's key idea is what if we approximated the smallest number (up to a factor 2), but just storing the location of the *first 1 bit*. This is great, since if we have F_0 distinct items, we expect to only need about $\log_2 F_0$ bits to store the first 1 bit for the smallest item. And how much space is needed to store the *location* of the first 1 bit?

... only $\log_2(\log_2 F_0)$ space, since we can represent that *location* in binary.

Let this location be s, then we can estimate F_0 as

$$\hat{F}_0 = 2^s - 1.$$

It is off by a constant factor (about a factor 2 with constant probability), and so we can use the same amplification trick to average $t = O(1/\varepsilon^2)$ copies and produce a $(1 \pm \varepsilon)$ approximation with constant probability, and then create $T = \log(1/\delta)$ estimates of that, and return their median value to get at estimate that with probability $1 - \delta$ satisfies

$$|\hat{F}_0 - F_0| \leq \varepsilon F_0.$$

Harmonic mean. The full HyperLogLog algorithm applies one additional trick. Combining discrete items with average and median (while often the best thing to do), turns out not to be the best method here. This is because the estimate γ is transformed by $1/\gamma - 1$ to get the base estimate. So instead if you create estimates $s_1, s_2, \ldots, s_t$ and instead of averaging them, take their harmonic mean this is more effective. That is, we first convert each s_i to a γ_i and construct the average

$$\hat{\gamma} = \frac{1}{t} \sum_{i=1}^{t} \gamma_i$$

This analysis turns out to require a lot more details, but gives superior performance.

This method is used widely in large tech companies to estimate unique users. There is also more extensions and follow-on work that can show some subtle empirical and theoretical improvements. But this is the classic approach, and achieves at least like 95% of the benefit in most cases.

14.4 Mergability

The standard streaming model is restricted in its operations. It can:

- insert 1 item into the summary
- report an estimate

But the only-insert-1-item restriction means it cannot be parallelized, and could be hard to update.

For very large scale processing, some parallelization is essential. We want multiple sites (e.g., routers) to individually summarize data, and then combine it later. Or to take a large data set (perhaps stored in the cloud), have local computation process and summarize each part, and the only aggregate the parts.

Or we could have data arrive sequentially, but want to be able to query recent subsets. We can aggregate old data into a single summary, and for smaller and smaller more recent chunks only aggregate parts of it. Then on a query we can combine recent subsets to produce overall results.

Or store data hierarchically (e.g., a balanced binary tree), with some repetition at multiple levels. If we store data at the leaf, and also in each node which has the leaf as a subtree; then we have an overhead of $O(\log n)$ space. But if we summarize with size $s(1/\varepsilon)$ at each node with depth $\log(n/s(1/\varepsilon))$, then this requires only $O(n)$ space: $O(n)$ is required at lowest layer, and each next layer has half as many summaries, so half as much space ($1 + 1/2 + 1/4 + ... \leq 2$). Now if we update (e.g., delete and perhaps reinsert) one item, we do not need to rebuild the full summary of all data. We can just re-combine data on the path from the updated leaf to the root. That only takes $O(\log n)$ time (times the cost of the merge, usually $O(s(1/\varepsilon))$.

Mergable summary. The key abstraction we need to make this work is the *mergeability* property. We need to add one additional step to the streaming model called a *merge*.

A *merge* takes in two summarizes $S_\varepsilon(A)$ and $S_\varepsilon(B)$ of two disjoint subsets of data A, B. Each summary is parameterized by an error parameter ε, and has a size $s(1/\varepsilon)$ that grows with $1/\varepsilon$. The step uses only $S_\varepsilon(A)$ and $S_\varepsilon(B)$ and create a new summary $S_\varepsilon(A \cup B)$. It should maintain the same error parameter ε *and* the same size relation $s(1/\varepsilon)$.

If a sketch permits this abstraction, then it can be run in parallel, handle queries on recent items (with logarithmic overhead in error, or space), and be used in a near-linear size data structure that allows for logarithmic time updates.

Many of the summaries we discussed are easily mergeable!

14.4.1 Example Mergeable Summaries

We discuss next some exemplar cases of mergeable summaries. These are more general, and also include quantile summaries (not discussed here, since more complicated).

Mergable Sampling. The goal here is to maintain a with-replacement or without-replacement random sample from the stream of size k. We may want this sample proportional to the weight w_i of an item either provided with the stream item a_i, or derived from it.

Reservoir sampling approaches could sometimes be made mergable. But there is a cleaner solution; using the bottom-k sampling approaches.

These assign each item a_i a uniform random value $u_i \sim \mathsf{Unif}[0, 1)$. Then the summary just maintains the smallest k values. When we merge two summaries, we can combine the lists, and take the bottom k, and have lost no information.

In fact, any *top-k* based summary is mergeable in this way.

With weighted samples, we can use the bottom-k of $\rho_i = -\frac{1}{w_i}\ln(u_i)$. And for with-replacement, we just run k summaries with 1 sample each in parallel.

Mergable CountMin Sketch. The CountMin Sketch is also mergable. Recall it maintains a $k \times t$ array of counters, and has a set of t hash functions – each indexing into one of the rows. On the merge, we make sure the same hash functions are used, and then we can just add up the counters from the two sketches.

Mergable HyperLogLog Sketch. At its core, it has a set of t hash functions, and each a representation of the smallest item seen represented by how many bits until the first 1 bit: as s_i for each hash function h_i. We again need to make sure both sketches use the same hash functions. Then we take the larger of the two values s_i for each sketch and keep the larger one.

Mergable Misra-Gries Sketch. Merging the Misra-Gries sketch is slightly trickier to see. Recall it maintains a set of k counters and k labels. The algorithm is simple, for any pair of labels that is maintained by both sketches, then we add together the counters. Afterwards we have at most $2k$ counters, and want to get down to only k counters. So we identify the $(k+1)$th smallest counter after the merge, let its value be δ. Then we subtract the value δ from each counter, and only retain the top k counters and their associated labels.

It is a little more intricate to see why this works. One reason for the challenge is that the contents of the sketch can change if we observe the data in different orders; this is not true for the other sketches discussed above. But it can be shown to have no more error than if we had maintained a single sketch on all of the data.