Notes about LotterySampling

Gonzalo Solera

October 2018

1 Introduction

Disclaimer: This notes are not formal enough and their goal is just to note down some ideas that may will be used in the theoretical part of the TFG. There are grammatical errors, things bad explained, etc. So again, this is just some random thoughts written fast.

We can combine different block pieces to create many distinct algorithms. And we have a name scheme to describe how the algorithm is by just enumerating which of these block pieces are used. However, the number of combinations is huge and most of them don't make sense or don't work well. So instead of keep elaborating this name scheme, I propose just to enumerate what the building blocks are. Now that we know some of these combinations that work specially well, we can call these specific instances with an arbitrary name. I mention this because I definitely have discarded many algorithms that were before included in the name scheme. For example, I totally discarded the least recently used (LRU) variants of the algorithms, also the non-ticket-update versions, etc. I want to get rid of that complicated name scheme since all those variants are not considered anymore.

So assuming that we already know what the building blocks are (most of them mentioned in Conrado's document) and on Slack and Trello, I'm going to proceed with a small theoretical introductory explanation of what properties we want in our algorithms and how we obtain them.

2 Introducing a threshold into SpaceSaving

SpaceSaving or SS, gives a deterministic guarantee that all the elements of frequency greater than $1/m$ are going to be in the sample. However I'm going to sketch a similar probabilistic bound on a very specific data stream: Consider a data stream in which one element z has frequency $f = f(z) = \frac{1}{m}$. All the other items on the stream are different (so all of them have frequency $\frac{1}{N}$).

The behaviour of SS with such stream will be the following: Assume that the apparitions of z follow exactly its frequency, ie. for each m elements, it appears once. Then the first m elements will enter into the sample of SS, all of them sharing one bucket (initially of count 1). I'm going to call that bucket (the bucket with lower count) the "frontier". Note that z is in that bucket. In the implementation proposed on the SS paper, we can sort the elements on one bucket by least recently used, having on the "left" the most recent element inserted into the bucket and on the "right", the oldest element.

The elements on the frontier are in danger, since they will be replaced by new elements if they don't "get far from it" going into buckets of higher count. And the element that will be replaced by the next element if the next element on the stream is not itself, is the one on the right of the frontier.

From the insertion of z, the following $m-1$ elements on the stream will be different elements. For each of those apparitions, the rightmost element on the frontier is evicted and the new one enters on the bucket of 1 unit increased count. The mth will be z again. Just before that apparition, z was on the "most dangerous position", at the right of the frontier. It saved itself from being evicted. If the frequency of z was a bit lower, it would have been evicted, even though its frequency is clearly much higher than any other element's.

Now consider the algorithm SST, which will only replace a sampled element by a non-sampled one with a fixed probability t. If z is in the sample, it will suffice to have a frequency higher than $\frac{1-t}{m}$ to remain in the sample, since only $1-t$ of the other elements will enter into the sample, so it will approach the "frontier" more slowly. That means that for high values of t the frequency that z "can" have to remain in the sample is much much lower than $\frac{1}{m}$ (the needed frequency for the original SS).

So by only defining such a threshold, we will be able to retain in the sample elements with much lower frequency than the original SS, since once they passed it, much more non-heavyhitters need to appear in order to be evicted again. Note that if there are other heavyhitters with higher frequency, they will "push" this less frequent element back into the frontier automatically, so it's not bad to have a high threshold in this sense. We just "can" capture elements with low frequency, but if that frequency is "too low" compared to the lowest frequency of the real heavyhitters, the element will not survive in the sample anyways (because it won't escape from the frontier).

But the problem is that if this threshold is too high, maybe the heavyhitters will never pass through it. So basically we want to find a threshold **low enough** to guarantee that every heavyhitter passes it some time and **high enough** to give it enough time to receive more hits and go away from the frontier. We need extra information like the frequency of the heavyhitters and the length of the stream to be able to adjust this threshold analytically, which makes SST not

very interesting.

Note that the very specific data stream used in the previous explanation was only useful for transmitting the intuition. The same idea applies for a general data stream. By fixing the threshold, we make the frontier bucket virtually much higher. But with the downside that maybe some heavyhitters never pass it.

We want an algorithm that adjusts the threshold automatically depending on the characteristics of the stream, such that the compromise between "low enough" and "high enough" is optimal.

3 Our algorithm

I'm going to name what we've being calling LotterySampling as LotterySamplingOriginal or LSO. LSO originally had this 'multilevel' feature that consisted on having 2 sets of tickets, one with increasing size. Since that didn't performed well and it wasn't 'fair' to compare with other algorithms, I discard it and when I refer to LSO I assume there's only one 'level'.

The original reasoning behind this algorithm was: The more times an element appears in the stream, the more probable will be to get a higher ticket. So we'll keep the m highest tickets and the elements that got them. We remove from the sample the element with lowest ticket every time a new element that wasn't being sampled appears in the stream with a higher token.

(Note: the tokens are discrete numbers from an implementation-dependent interval. In our implementation, the tokens are numbers between 1 and $\theta = 2^{64}$.)

One better way of explaining the previous concept would be the following: For an element z that appears $f(z)$ times in the stream, we define a random variable X_i for each apparition i that measures the token that z obtained in that apparition. So we have the random variables $X_1, X_2, ..., X_{f(z)}$, which are the tokens of z. Then we define the random variable $X = \max\{X_1, X_2, ..., X_{f(z)}\}$ (the ticket from z). The probability of X being less than t, thanks that X_i are independent, is:

$$\Pr[X \leq t] = \prod_{i=1}^{f(z)} \Pr[X_i \leq t] = (\frac{t}{\theta})^{f(z)}$$

We can also define the probability

$$\Pr[X = t] = \begin{cases} \frac{1}{\theta^{f(z)}} & \text{t = 1} \\ \Pr[X \leq t] - \Pr[X \leq t-1] & \text{t ¿ 1} \end{cases}$$

Then we can calculate the expectation of X, in function of the number it appears in the stream, $f(z)$:

$$\begin{aligned}
\mathbb{E}[X] &= \sum_{t=1}^{\theta} t \Pr[X = t] \\
&= Pr[X = 1] \\
&+ 2(\Pr[X \leq 2] - \Pr[X \leq 1]) \\
&+ 3(\Pr[X \leq 3] - \Pr[X \leq 2]) \\
&+ ... \\
&+ \theta(\Pr[X \leq \theta] - \Pr[X \leq \theta - 1] \\
&= \\
&- \Pr[X \leq 1] \\
&- \Pr[X \leq 2] \\
&- ... \\
&- \Pr[X \leq \theta - 1] \\
&+ \theta \Pr[X \leq \theta] \\
&= \theta - \left(\frac{\theta - 1}{\theta}\right)^{f(z)} - ... - \left(\frac{1}{\theta}\right)^{f(z)}
\end{aligned}$$

which asymptotically is equivalent to $\frac{f(z)}{f(z)+1}\theta$. This value increases as the frequency of z increases. If there was 0 variance, then by capturing the m highest tickets we would solve the problem exactly, and this is what LSO would do in that case.

However, the variance is not 0, and although it asymptotically converges to 0, the expectation also converges to θ at the same time. That means that having elements with high frequency in the stream (or long streams) will imply having high tickets, very close between them. So even if the variance decreases with such high frequencies (or high counts specifically), a very small variance will affect a lot, making that the ticket for some non-heavyhitter with much less frequency than the heavyhitters get a higher ticket.

Also, a very high ticket (respect to its expectation) for a non-heavyhitter element in LSO is very harmful, since it will occupy one of the m spots in the sample.

Another harmful behaviour is that non-heavyhitters could "join tokens". I'm a bit confused by this and I still need to think about it. Consider the very extreme case used to introduce the SST algorithm (a stream in which only one element is frequent with frequency f, all the others only appear once in the stream). LSO will keep the m highest tickets/tokens. In this case, we could

consider ALL the elements on the stream (except the heavyhitter one) are the same item. So we could define $X_1, X_2, X_{N(1-f)}$ similarly as before, pretending that all of the distinct elements are actually the same. Then LSO will keep X as defined before. And also $X^2, X^3, ..., X^m$ where each X^i is the ith highest token. X will definitely be much much higher than the expected ticket of z, X^z. But will $X^z > X^m$? (This is slightly solved with the next algorithm but I need to keep thinking about this) (I also would like to get the mathematical expression of $\mathbb{E}[X^i]$, maybe it decreases radically)

We can consider that minimum ticket as a threshold and to use a SST algorithm underneath that uses this threshold. And this is basically our most interesting algorithm, LotterySampling (LS, note the name change). In LS, for each element z that appears in the stream, it gets a new token t. If z was in the sample and its ticket is less than t, t becomes its ticket. If it wasn't in the sample and t is higher than the lowest ticket from the sample $t_{\min}$ (the threshold), z replaces z_{lef} (item with less estimated frequency, not necessary the one with $t_{\min}$) and keeps its new token as ticket. And $f_i(z) = f'(z_{\text{lef}})+1$ (like in SS).

This combination of LSO and SST addresses the problem with the variance, and chooses the value of the threshold depending on the stream, adapting to it progressively. Note that the threshold never decreases although it may increase very slowly, depending on the frequency of the lowest heavyhitter, which is great. LS gets better results in ALL the tests than any other algorithm. It even reached the performance of optimal SST with its threshold tuned manually.

The variance can affect in the following manners:

1. Non-heavyhitter gets a too high ticket (in the worst case, between the m highest tickets). Then it enters into the sample, replacing another element which could be a heavyhitter. But it's replacing an element from the frontier, so one of the less frequent heavyhitters or a non-heavyhitter. It may or may not increase the threshold, depending on whether the replaced element had the lowest ticket. Since it's not a heavyhitter, it won't get far from the frontier, and it will be replaced soon by another element.

2. Non-heavyhitter gets a too low ticket. It doesn't matter if it doesn't enter into the sample then.

3. Heavyhitter gets a low ticket respect to its expected value. If the element was from the most frequent heavyhitters, it will be w.h.p. inside the m tickets anyways. If it was one of the less frequent heavyhitters The variance will also affect to the other heavyhitters, and in particular, it affects to the less frequent heavyhitters. This variance will make some of those tickets to be higher than expected, but also some others will be less than expected. Since the threshold is the lowest ticket and may be affected by the variance, this element's ticket maybe will even pass the

threshold. If it doesn't pass it, then (probably) this element is much less frequent than the other heavyhitters so it's not that bad to miss it.

4. Heavyhitters get too high tickets (comparing to their expected tickets) due to the variance. It is not very hurtful unless that happens with many heavyhitters, making impossible for other less frequent heavyhitters to enter the threshold. Similarly as before, this needs to happen to a lot of elements at the same time to be bad, which is unlikely.

Regarding the mentioned problem called "ticket joining", I think this is addressed in this algorithm by the following reasoning. If there are already some heavyhitters on the sample, they will be far from the frontier. And the threshold won't increase because of the ticket joining, since the minimum ticket will be "locked" with a small value by one of these sampled heavyhitters. It will remain like this although the threshold increases by some of these sampled heavyhitters, but that would be much slowly and in a "healthy" and intended way. And during that time, if another heavyhitter gets into the sample, it will replace one of the non-heavyhitters (they are in the frontier) that were occasioning this ticket joining. In the case of LSO, this new heavyhitter would just replace the heavyhitter with lowest ticket that was preciselly maintaining the threshold low!!!

4 Conclusion

Really, really, really good results with LS, but logarithmic cost to keep the ticket order. The alternative: LotterySamplingMean (LSM). The only better properties of this variation respect to LS is regarding efficiency. It just needs constant time per element and it doesn't need to use memory to keep the heap (the ordering of tickets). It doesn't have the cool theoretical explanation as before, it's more intuition, and it doesn't work as good as LS in terms of accuracy (in examples, LS works better, and in some, LS works MUCH better). It basically fails in choosing the threshold, which many times is too high. The problem it has is that in expectation, the threshold is the mean of ticket expectations. This is good if all the heavyhitters have the same frequency, if not, that doesn't make much sense because it will miss the less frequent heavyhitters with ticket expectation under that mean. Also, even in that "optimal" case (in which all the heavyhitters have same frequency), the mean of tickets cancels off the variance, which is in fact not good, because it would be better to be conservative to catch heavyhitters with "negative" variance (that makes them to not get the expected ticket).

Regarding ticket aging, I propose to forget it for this project. Maybe in the future if all this ends up being succesful we can consider more complex ideas like this or other distributions apart from the uniform for the tokens, etc. But I think it would be good to stop thinking more algorithms since we already have a VERY promising one, which will probably be enough for this project.

And similarly with ticket stealing. However I can give it a try and if they show very promising results, keep working with it. Also, we can think in the future ways for generalizing this to the more general problem in which each item on the stream has a weight W (as used by Amazon). SS can deal with this by simply summing W to the frequency counter instead of just 1. And maybe we could generate W tickets and take the maximum? I think that would work but we would need to avoid doing W operations, maybe there's a prob. dist. that "simulates" this behaviour.

PS: Lo del cambio de nombres que he hecho, es decir, lo de llamar LotterySamplingOriginal a tu idea original y LotterySampling a esta variante nueva que se me ocurrió esta semana lo he hecho porque: Me parece un nombre muy chulo y más "comercial" (como mencionates el otro dia en la llamada) el de LotterySampling, y tiene más sentido que LotterySaving. Pero si no te gusta lo cambiamos, y llamo al nuevo LotterySpaceSaving y punto.