Probabilistic counters

Rodrigo Ferreira

***Resumo* –Este artigo começa por introduzir o atual problema ou motivação para a necessidade de contar números grandes de eventos com números menores. Passa então a explicar o conceito teórico de dois tipos de contadores diferentes, contadores de probabilidade fixa (*fpc*) e contadores logarítmicos de probabilidade decrescente (*ldpc*). Podendo depois retirar algumas observações e conclusões, ao avaliar dados de simulações de ambos os tipos de contadores aqui descritos, fazendo comparações e avaliando o seu desempenho.**

***Abstract* –This paper starts out by introducing the current problem or motivation from which the need to count big numbers of events with smaller numbers arises. Then it explains the theoric concepts behind two types of counters, fixed probability counters (*fpc*) and logarithmic decreasing probability counters (*ldpc*). After which we can make some observations and draw some conclusions by evaluating data gathered from simulating both counters, drawing comparisons and evaluating their performances.**

I. INTRODUCTION

A. The issue

As time passes in an era of ever increasing technology use and development, the amount of data grows exponentially, for instance, in 2018, 2.5 quintillion bytes of data were created every day [1] and that number is only increasing. Many areas such as online social networks, search engines, product and consumer tracking, online content delivery, and large scale scientific experiments struggle with the amount of memory they require, as the approach of simply counting events the exact number of times they occur, for such large amounts of data, is no longer feasible.

There are two main ways of solving this problem, one solution is increasing the number of systems and thus increasing the capacity to accumulate more data (which is expensive both in equipment and energy spent), the other solution is to count or store the data differently, in a more efficient manner (spacewise).

B. The context

In this article, we are taking a look at the latter option, ways to count events (data) more efficiently, and thus saving space. For this, we will make use of probabilistic counters.

Probabilistic counters are counting tools that rely on some randomness to decide whether to count or not each event. At first this may seem like a ridiculous solution, but for really big amounts of data the randomness affects the counter's precision a lot less, and thus we end up counting around the same number of events, using less bits. [2]. The two types of counters considered in this article will now be explained.

II. TYPES OF PROBABILISTIC COUNTERS

A. Fixed probability

Fixed probability are the simplest type of probabilistic counter, every time an event occurs, the algorithm will decide whether to increment the counter or not, based on a randomly generated number and a fixed probability P that describes the counter.

An exact counter can be seen as a probabilistic counter with fixed probability $P = 1$ meaning that every event causes the counter to increment. Such a counter (exact) requires $\lceil \log_2 n \rceil$ bits to represent n events. Making it so that when the amount of data is very large, a big number of bits is required to count all the events, and thus, there is a need for more efficient counters.

Simply decreasing the incrementing chance to $P = \frac{1}{2^k}$ for $k \geq 1$ can go a long way. Such a counter, with $P = \frac{1}{2^k}$ chance, requires $\lceil \log_2(P{*}n) \rceil$ bits to count n events, because the value on the counter will be approximately $P * n$ and, because $0 < P < 1$ (because $P = \frac{1}{2^k}$ for $k \geq 1$), the amount of bits required will always be less or equal to the amount required for an exact counter ($P = 1$).

Although this is in general a decent solution, there are some problems, for instance, for bigger amounts of data, the counters will still get very big, we only decrease them to about $\frac{1}{2^k} * n$ where n is the number of events. And another big problem being that if an event occurs very few times, with a smaller P chance, it's very likely that the counter won't increment at all, staying at 0 as if it never occurred.

B. Decreasing probability

In order to fix the *fpc*'s main caveats, another approach was invented, where the chance for an event to trigger the counter to increment, does so with ever decreasing likelihood, meaning that as the counter gets larger, the chance of incrementing it decreases.

This is done through by choosing a base a which describes this type of counter, and for an event e, the chance of incrementing its counter is given by $\frac{1}{a^{counter[e]}}$ where $counter[e]$ is the current value of e's counter. This formula clearly demonstrates that the chances favour smaller counter numbers and decrease for bigger counter numbers, it also makes it so it's guaranteed to increment on the first occurrence of an event, because $\frac{1}{a^0} = 1$. This way, the first occurrences of an event are more important than following ones, to address the problem of having the counter at 0 even if events occurred.

Another consequence of this approach is that the counters contain very small numbers even for a huge amount of data, due to the decreasing probability, as we will see later. As n events result in a counter value of $\log_a n$, which in turn, takes $\lceil \log_2(\log_a n) \rceil$ bits to represent, where a is the counter's particular base. This is why these counters are considered logarithmic.

III. The algorithm

The algorithm is very simple in this case, using a class to represent the test chains (*char_chain*) with attributes such as its source string, its size, the chain itself, and an exact count, implemented in the generation of the chain for better efficiency. There are also classes for the *fpc*'s (*prob_counter*) and *ldpc*'s (*dec_prob_counter*), both very similar, containing the basic defining attributes, all the letter's counters themselves and some statistical data.

There is then a method to simulate a counter for a certain character chain for n times, which was used to get a better view of the results, it prints each result individually in /testdata/simulations/ as well as the collective statistics over all simulations to a file in /testdata/stats/. There are examples of these files on the *testdata* folder.

A. Results

A.1 Testing conditions

As it was previously stated, the original task was to build the test chains from the string of my full name, but we will use "rodrigomiguelmaiaferreirarrrrrrrrrroooooo" with additional r's and o's in the end to ensure that some characters occur dramatically more often than others.

Then, we simulate each counter 20 times for some random character chains of differing sizes of 100, 500, 1000, 5000, 10000, 50000, 100000, 500000 and 1000000 characters, to get some stats to draw conclusions from.

The counters used here are: a fixed probability counter with chance $P = 0.5$, equivalent to 50%, meaning that we're equally as likely to increment the counter or do nothing for each event found; And a logarithmic decreasing probability counter with $base = \sqrt{2}$, meaning that for each event found, the chance of incrementing the counter is given by $\frac{1}{\sqrt{2}^c}$, where c is the current counter value.

In order to calculate the statistics shown, across all simulations (for each chain size), for each letter the values were collected and then averaged for the mean, for the min and max the process was just picking the smallest and biggest value respectively. This was the process for all stats except for estimated values, those were calculated following the formulas once the min, mean and max counter values were found.

All the results are available in the *testdata* folder, all the counters simulated are in the *simulations* folder and most importantly, the statistics can be found in the *stats* folder.

A.2 Fixed Probability Counter

The fixed probability counter seemed to be a pretty consistent approach, although it falters for smaller character chains, especially in the cases where the characters show up very rarely, as the consistent 50% chance (applying equal importance to the first occurrence of a letter as to its n'th) allows for some mistakes where such letters can be ignored and have their counter represent 0 events.

But, as the character chain size grew, its average relative error decreased significantly to the low $0.x$%, its accuracy approached 100%, the relative order of the letters was correct on average and it required less bits on average than an exact counter, though it still scales rather poorly. The only downside is that the absolute error increased on average, but that is to be expected, as a counter value gets bigger, an increment means a big jump in the estimated value, which can cause big absolute errors.

A.3 Logarithmic Decreasing Probability Counter

The logarithmic decreasing probability counter performed very differently.

Where the *fpc* had problems, the *ldpc* performed great, because it gives substantially more importance to an events first occurrences, it was very rare that any letters showed 0 where they shouldn't. Another advantage is that due to that decreasing probability, the counters get to a point where it's very unlikely that they will be incremented, and thus, even for bigger sizes, the counters did take very few bits to represent as expected.

That's where the *ldpc*'s advantages end though, as it performed very inconsistently on all metrics compared to the *fpc*.

For instance, for the simulations of character chain size 1000000, the average relative errors were pretty high (the average of all of them being around 33.5%), with one counter (letter d) even getting a value as high as 43.5%. The average absolute error also increased substantially, but it's also expected since the increments get very rare, and due to the formula to calculate the estimated values from the counter values, an increment means a very big increase in estimated value, leading to huge errors). The average relative order of the letters wasn't as clear as with the *fpc* but it was still decent. The average accuracy ratio was rounding to about 99.4% for all letters, though there were some big differences on both sides so it's not a very reliable 99.4%.

A.4 Comparisons and Observations

The advantages and disadvantages of both counters are very clearly demonstrated when we plot the various metrics.

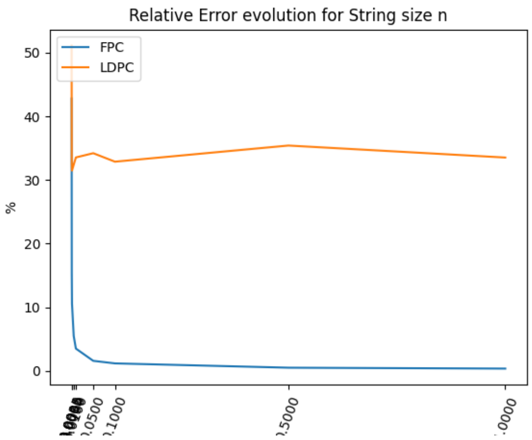

TABLE I

Average relative error for chains size [100,500,1000,5000,10000,50000,100000,500000,1000000]

Across all tests, fpc outperforms $ldpc$ when it comes to relative error, mostly due to its consistent probability, which can cause big relative errors for smaller tests, but gets evened out for bigger tests as the random factor means less.

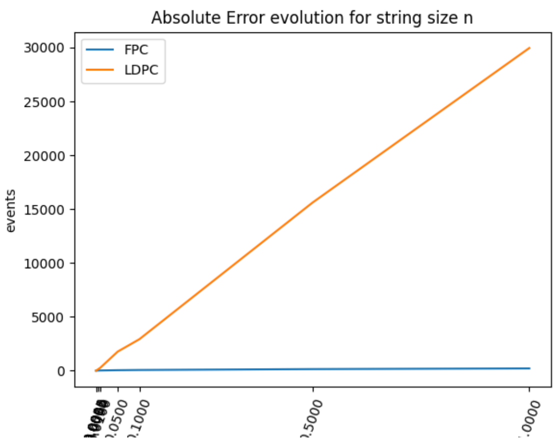

TABLE II

Average absolute error for chains size [100,500,1000,5000,10000,50000,100000,500000,1000000]

As expected both methods have an increase in absolute errors, though $ldpc$'s is so big that it dwarfs the fpc's in comparison.

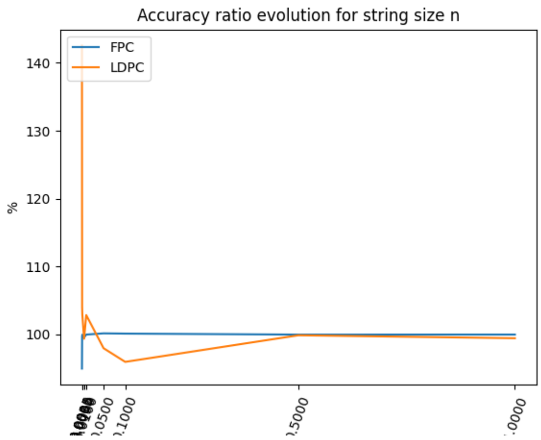

TABLE III

Average accuracy ratio for chains size [100,500,1000,5000,10000,50000,100000,500000,1000000]

Table III might be a bit misleading, it shows the $ldpc$'s average accuracy ratio leading to about 100%, but lots of big errors occur on both sides (100%+ and 100%-, which can be seen by its huge relative errors), making it average out on 100%. The fpc's value is much more reliable, as shown by its smaller relative errors.

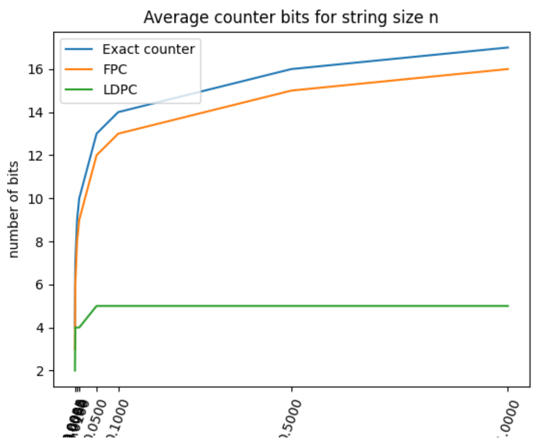

TABLE IV

Average counter bits needed for chains size [100,500,1000,5000,10000,50000,100000,500000,1000000]

As previously stated and also demonstrated on table IV, this is both the biggest advantage and disadvantage of the $ldpc$, as its small counters outperform both the fpc and exact counters, but also lead to its imprecision, causing huge errors as seen in table I and II. In order to better visualize this disparity, for the biggest chain size, the most frequent letter was r, occurring 365412 times, while the fpc had an average counter value of 182685, the $ldpc$ had an average counter value of 34 for the exact same letter.

IV. Conclusions

I think it's clear that both types of counters have their advantages and disadvantages, and I would be curious to see how they would perform for much bigger amounts of data though I am limited by my computer's processing power. Regardless, it's a very interesting and demanding area, where progress is still needed. I'm interested in learning more about the subject and seeing what kinds of new approaches might come along in the following years.

References

[1] Bernard Marr, "How much data do we create every day? the mind-blowing stats everyone should read", 2018. **URL:** https://www.forbes.com/sites/bernardmarr/2018/05/21/how-much-data-do-we-create-every-day-the-mind-blowing-stats-everyone-should-read/#6a005e9c60ba

[2] "Approximate counting algorithm". **URL:** https://en.wikipedia.org/wiki/Approximate_counting_algorithm