Probabilistic counters

Rodrigo Ferreira

***Resumo* –Este artigo começa por contextualizar o assunto principal por detrás das ferramentas que são os contadores probabilisticos, porque e quando são necessários, qual é o seu propósito, e as suas vantagens e desvantagens. Compara também diferentes tipos de contadores probabilisticos, nomeadamente de probabilidade fixa e de probabilidade decrescente e avalia as suas diferenças em conceito e em desempenho.**

***Abstract* –This paper starts out by contextualizing the main subject behind the tools that are probabilistic counters, when and why they are necessary, what their purpose is and their advantages and disadvantages. It also compares different types of probabilistic counters, namely fixed probability and decreasing probability, and evaluates their differences in concept and in performance.**

I. Introduction

A. The issue

As time passes in an era of ever increasing technology use and development, the amount of data grows a lot faster than our systems can handle. There are two main approaches to solving this problem, either increasing the number of systems and thus increasing the capacity to accumulate more data, or to store the data differently, in a more compressed manner.

B. The context

In this article, we are taking a look at the latter option, ways to store data more efficiently. We will be using strings, of various sizes, generated by randomly picking characters from the source string composed of my full name "$rodrigomiguelmaiaferreira$", and we will be using different probabilistic counters to illustrate what is meant by "storing data differently".

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 is an example of a probabilistic counter with fixed probability $P = 1$ meaning that every event causes the counter to increment. Such a counter can represent 2^n events with n bits. 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 $k >= 1$ can go a long way, Such a counter, with $P = \frac{1}{2^k}$ chance can count up to 2^{nk} events using only n bits, because the value on the counter will be approximately $P * n$ and thus, it will take $log_2 P*n$ bits to represent, less than that of an exact counter. //CLARIFY N BITS OR N EVENTS N FIRST FORMULA 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 solve the fixed probability counter's previously mentioned main caveat, 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 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 occured. 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 $\log_2(\log_a n)$ bits to represent, where a is the counter's particular base. This is why they're also considered logarithmic counters.

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 fixed probability counters ($prob_counter$) and decreasing chance logarithmic counters ($dec_prob_counter$), both very similar, containig the basic defining attributes, the 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 as well as some collective statistics over all simulations to a file. 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 "rodrigomiguelmaiaferreirarrrrrrrrrrooooooo" 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. 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 $0.xx\%$, its accuracy approached 100%, the relative order of the letters was correct on average and it requires 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 expected 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, with one letter (r) even getting a value as high as 48.85, that same letter had an average absolute error of 178698 which is pretty significant (it's also expected since the increments get very rare, and due to the formula to calculate the expected values from the counter values, an increment means a very big increase in expected 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. And the average accuracy ratio was rounding to about 101% for all letters, though there were some big differences on both sides so it's not a very reliable 101%.

A.4 Comparisons and Observations

The advantages and disadvantages of both counters are very clearly demonstrated when we plot the various metrics. Across all tests, *fpc* outperforms *ldpc*

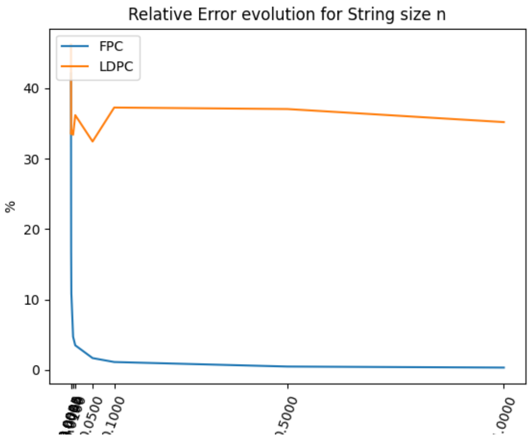

TABLE I
AVERAGE RELATIVE ERROR FOR CHAINS SIZE [100,500,1000,5000,10000,50000,100000,500000,1000000]

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. As expected both methods have an increase in absolute errors, though *ldpc*'s is so big that it dwarfs *fpc*'s in comparison. Table III might be a bit misleading, it shows the *ldpc*'s average accuracy ratio leading to 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. 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.

The most astounding difference above all else has to be the counter values of the *ldpc*, the counter of the most frequent character, r, increased from values around 8 with test chain size 100 to values surrounding the low 30's for chain size 1000000. This is the main advantage

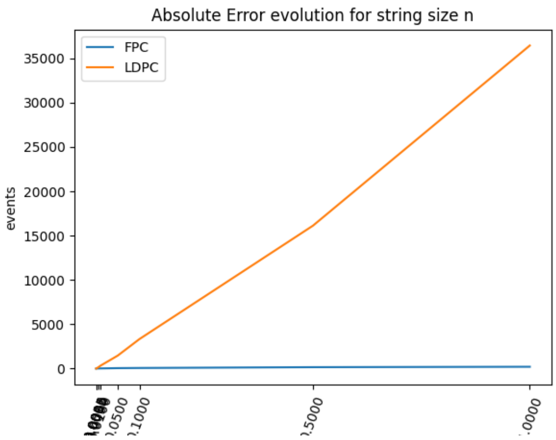

TABLE II

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

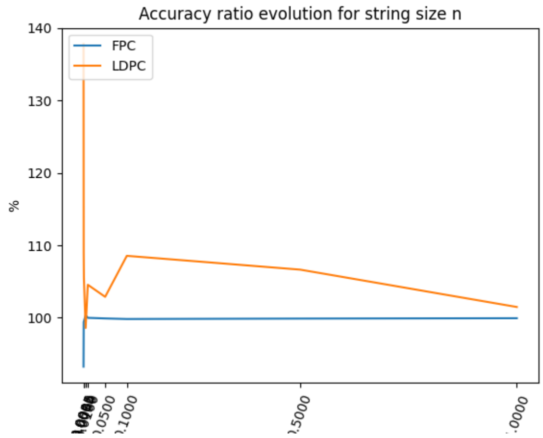

TABLE III

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

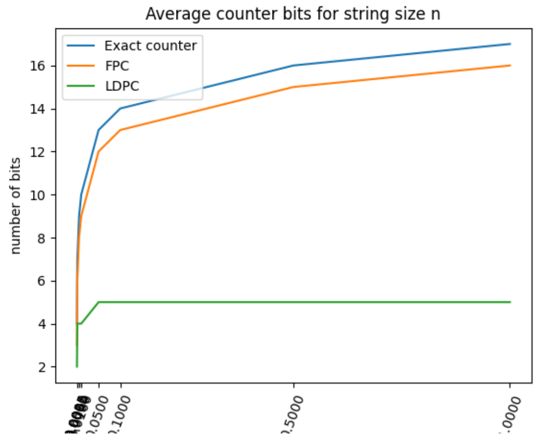

TABLE IV

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

of this approach. It may lose precision but it can keep its counters extremely low.

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