Solving The Correlation Measuring Problem Using A Cardinality Estimation Algorithm

John M. Singleton and Yalan Yin's CSCI 551 Project

due by 11:59 PM on M 12/9/2019

1 Introduction

The idea for our project came from a motivating example given in a new book that provides an accessible introduction to probabilistic data structures and algorithms [2]. The example discusses the problem of determining the number of distinct k-mers in a piece of DNA, or what is called in more general terms the cardinality problem. (A k-mer is simply a substring of length k.) As a Big Data problem, probabilistic algorithms offer a way to find savings for both processing time and memory requirements in applications where the exact cardinality is not required. We chose to implement the *LogLog* algorithm, which appeared in 2003 and was followed by *HyperLogLog* and *HyperLogLog++*. Other probabilistic algorithms exist to solve the cardinality problem (eg., MinCount); however, the LogLog family of algorithms include the most popular ones in practice [2].

A good application in biology area for LogLog algorithm is the correlation problem. It turns out that k-mer proportions of different pieces of DNA of the same organism tend to be very similar. When comparing pieces of DNA from different organisms, the proportions tend to be dissimilar. In other words, if one takes a piece of DNA from an unidentified organism, an analysis of its k-mer proportions can lead to an identification of the organism [3]. The alphabet of DNA is $\Sigma = \{A, C, G, T\}$, and therefore there are 4^k different k-mers. As we will see, in large sequences and for a k-mer with a small k such as 3 or 4, almost all of the possible k-mers will occur. Things get more interesting for higher, where a fraction of the possible k-mers do not occur.

2 LogLog Algorithm

2.1 Approach

The LogLog algorithm makes use of a single hash function, for which we chose the non-cryptographic hash function MurmurHash3 (https://anaconda.org/conda-forge/mmh3). It accepts an ASCII string (like a k-mer) as input, which is mapped to a value from 0 to 2^{32}-1. These values are uniformly distributed. This is seen in Line 6 of our pseudocode, which comes from Gakhov [2].

For our application, we start off with a genomic sequence of length n, which we store as a string. If we are doing a calculation for k-mers of length k, then our for loop submits (n-k+1) k-mers starting at positions 0 to (n-k+1) to the hash function. We then convert this hash value to the "LSB 0" numbering scheme, which is a binary representation where the least significant bits are on the left and we enforce that the leading '0's

will not be truncated. So, for example, in the "LSB 0" numbering scheme, a 10-bit representation of the number 21 would be '1010100000'.

Now, we introduce an important fact first noticed by Flajolet and Martin [1]. The probability that the resulting value will be like 0^k1 (have k leading zeros) is equal to $2^{-(k+1)}$. This can be converted into a cardinality estimate by inverting the value, 2^{k+1}. In the LogLog algorithm, the idea is to keep track of the largest value of (k+1) that occurs amongst all of the elements of our dataset. This is the basis of a whole class of probabilistic counting algorithms. The discussion below is the modifications of the above ideas.

Most of the rest of the algorithm is stochastic averaging. We create m COUNTERS, and we store a value indicating the rank, (k+1) value, of a given element in the COUNTER given by taking the p = lg(m) least significant bits of our value. The only requirement is to make sure that p is a small fraction of the total number of the 32 bits of our values. This is because the rank will be determined by looking at the remaining (32-p) bits.

At the end of the for loop, we average the values in the COUNTERS, and take 2 to this power, then multiply this by a correction factor α_m, and m.

2.2 Algorithm

Below is the pseudocode we implemented, which comes from Gakhov[2].

Algorithm 1 Estimating cardinality with LogLog

Input: Dataset D
Input: Array of m LogLog counter with hash function h
Output: Cardinality estimation
COUNTER[j] $\leftarrow 0$, $j = 0...m-1$
for x $\in$ D **do**
    $i \leftarrow$ h(x) := $(i_0 i_1 ... i_{M-1})_2$, $i_k \in \{0, 1\}$
    $j \leftarrow (i_0 i_1 ... i_{p-1})_2$
    $r \leftarrow rank((i_p i_{p+1} ... i_{M-1})_2)$
    $COUNTER[j] \leftarrow max(COUNTER[j], r)$
end for
$R \leftarrow 1/m \sum_{k=0}^{m-1} COUNTER[j]$
return $\alpha_m * m * 2^R$

3 Analysis

3.1 Dataset

For this project, we mainly used the following DNA sequences: Human Papillomarirus 4(HPV4), Human Papillomavirus 5(HPV5) and Human Herpesvirus 1(HHV1), which were supplied in the paper [3]. Besides that, we also used the large file "AL935263.fna" for the analysis of time complexity section (https://www.ncbi.nlm.nih.gov/nuccore/AL935263).

3.2 Accuracy

There is a formula for the approximate standard error in the cardinality estimation which depends inversely on the square root of the number of counters used: $\delta \approx \frac{1.3}{\sqrt{m}}$ [2, 1]. As an example of how to use this formula, we can consider one of the cardinality estimates from our own experiment (see Figure 2). We found, for m=64, k=6, and the first 1,000 nucleotides of Sequence 1, a cardinality estimate of 433. The standard error $\delta \approx \frac{1.3}{\sqrt{m}} = \frac{1.3}{\sqrt{64}} = 0.1625 \approx 16\%$, implying that the true cardinality is between 362 and 504.

3.3 Time Complexity

The running time for most commonly used non-cryptographic hash functions (eg., MurmurHash3) is O(1) time [2]. The LogLog algorithm processes each k-mer once, and there are (n-k+1) k-mers in a sequence of length n. Therefore, for k << n, the time complexity of LogLog algorithm is O(n)O(1) = O(n).

Although our primary reason for implementing a brute force algorithm was to find the exact cardinality, we also decided to analyze the algorithm's time complexity. For brute force algorithm, we can see that, like the LogLog algorithm, it also processes each k-mer once. Again, there are a total of (n-k+1) k-mers. For the processing of each k-mer, we call Python's keys() method, which in Python 3.x, takes O(1) time (https://wiki.python.org/moin/TimeComplexity). In most cases, we will also have an insertion of a new key/value pair into the dictionary, which takes linear time in the size of the dictionary. Since most of the keys are distinct, we think that in general this will be O(n), making our brute force algorithm $O(n^2)$. However, the results for our timing tests suggested that it is linear. See the sample output named Figure 1 below. This may be because we didn't consider large enough values of the sequence length for the quadratic nature to appear. Or, our understanding of the time complexity of the insertion method is incorrect.

3.4 Space Complexity

The real advantage of the LogLog algorithm may be its savings in terms of space. According to [2], an estimate of the number of bits of storage needed for the LogLog algorithm is $mlglgn$ and a more precise estimate which relies upon the number of counters as well as the length of the sequence is $mlglg\frac{n}{m}(1+O(1))$.

Supposed there are 2^{30} elements with m = 256, $\log_2 log_2 2^{30} \approx 4.91$. In another word, 5 bits per bucket is enough for 2^{30} elements.

4 Application

If you take a look at Figure 2 below, we have listed the cardinality estimates and proportions (the cardinality estimate / 4^k) for three different sequences and nine different values of k. Two of the sequences are from the same species of organism (herpesvirus) and one of them is from a different organism (papillomavirus). The hypothesis is that the proportions in the two sequences coming from the same species will be more similar to each other than to the third sequence which comes from a different species. Can you tell which sequence is the latter one? In our example, the answer is Sequence 3.

In our example, we can also see how, even though the LogLog algorithm is not an exact algorithm, it is still able to distinguish the piece of DNA which came from the different organism from the two that came from the same organism. This suggests that the correlation problem is a suitable application for probabilistic algorithms.

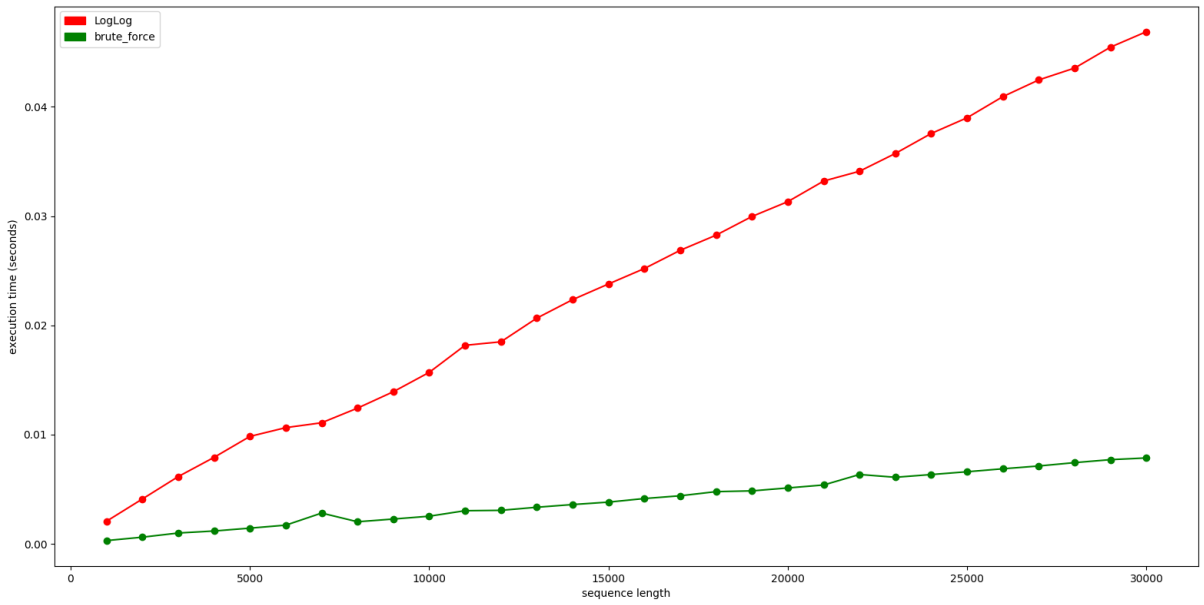

Figure 1
Sample output for LogLog and Brute Force Algorithms with k=25, m=64, p=6, and testing the execution time for pieces of sequence from 1000 to 30000 in increments of 1000 from dataset "AL935263.fna".

```
----------parameters used-------------------
m=64
p=6

-----------tested DNA sequences-------------
sequence1: NC_001457.1 Human papillomavirus type 4, complete genome
sequence2: NC_001531.1 Human papillomoavirus type 5, complete genome
sequence3: NC_001806.2 Human herpesvirus 1 strain 17, complete genome


-------------tested k values----------------
[3, 4, 5, 6, 7, 8, 9, 10, 11, 12]


-------LogLog Cardinality Estimates ---------

k:          3    4   5    6    7    8    9    10   11  12
sequence1: 39  102  260  433  372  633  613  443  453  453
sequence2: 39  115  278  380  457  415  448  510  581  510
sequence3: 37  78  178  234  364  372  376  389  406  372


-------------Cardinality/4^k----------------

k:             3           4           5           6           7           8           9           10          11          12
sequence1: 0.61894034  0.40133316  0.25465611  0.10591635  0.02275374  0.00967094  0.00234044  0.00042280  0.00010801  0.00002700
sequence2: 0.61894034  0.45211048  0.27175383  0.09300791  0.02795252  0.00633912  0.00170960  0.00048672  0.00013857  0.00003042
sequence3: 0.57999897  0.30613656  0.17431242  0.05712920  0.02226617  0.00568843  0.00143759  0.00037127  0.00009693  0.00002222
```

Figure 2
Sample output for our experiment in Application section

5 Conclusion

Based on the idea of probabilistic counting which was identified by Flajolet and Durand in 2003, a whole class of interesting probabilistic algorithms have been invented that solve a large number of problems. This class of algorithms is the only practical solution when the dataset being processed is larger than the amount of memory available in a system. The space required for LogLog algorithm is lglgn where n is the number of elements processed.

Based on LogLog, an improved algorithm - HyperLogLog was developed and is implemented in well-known databases including Amazon Redshift and Apacha CouchDB, etc. [2]. For our project, we sought to understand the idea behind LogLog and to implement it in Python. With regards to the time complexity, we were able to show that the LogLog algorithm runs in linear time. Besides that, we found an good application of using LogLog to identify if different DNA sequences came from the same species or not. We tested three different DNA sequences with different k values. Our results show that even with some loss of accuracy, LogLog is still a good option to solve the correlation problem especially without sacrifices of storage.

6 Attachment - Python Implementation code

kmer_loglog.py - This file contains our implementation of the LogLog algorithm, the brute force algorithm, and an input function for reading a FASTA file.

kmer_loglog_timings.py - This file adds to the first file our last code for taking timings.

kmer_loglog_application.py - This file adds to the first file a few lines which provide a demonstration of how these cardinality estimates can be applied to the correlation measurement problem.

References

[1] Marianne Durand and Philippe Flajolet. Loglog counting of large cardinalities. In *European Symposium on Algorithms*, pages 605–617. Springer, 2003.

[2] Andrii Gakhov. *Probabilistic Data Structures and Algorithms for Big Data Applications*. BoD–Books on Demand, 2019.

[3] Aaron Sievers, Katharina Bosiek, Marc Bisch, Chris Dreessen, Jascha Riedel, Patrick Froß, Michael Hausmann, and Georg Hildenbrand. K-mer content, correlation, and position analysis of genome dna sequences for the identification of function and evolutionary features. *Genes*, 8(4):122, 2017.