Advances in Data Mining 2018

Assignment 2

October 23, 2018

1 Introduction

When processing large files or data streams one often wishes there were some way to count the number of distinct elements. In data science this number is called cardinality, which is also the term used in the rest of this report. Counting the cardinality of certain data streams can provide very valuable information. For example, artificial web pages generated by human-like generators tend to either have a limited dictionary or use an unlikely amount of distinct words to disguise themselves. Quickly counting the number of distinct words on a web page can help detect these and exclude them from search results by the likes of Google. On the other hand, many large companies with an enormous amount of internet access points and critical files in storage are vulnerable to large scale DDOS attacks in which many computers try to gain access at once, saturating the service to the point where it suddenly fails or breaks. Detecting the number of distinct (source, destination) pairs could quickly recognize these and divert the attack.

Both examples mentioned suffer from the same defect, the number of elements to scan through is enormous. Simply making a dictionary with each distinct element and checking against it for every new element will not only be terribly inefficient it is also extremely time consuming and not fit for the purpose where relatively quick results have to be achieved. Often it is not actually necessary to know the precise number of distinct elements, a decent approximation is normally sufficient. As such, smarter methods which produce relatively accurate approximations of the count can be devised and have been devised in the past. In this report we will take a closer look at two of those methods and test them for time and memory efficiency and relative accuracy. Through extensive testing we achieve a good overview of the most beneficial values for several parameters and will present a concise Practical Guide outlining the best settings for different expected outcomes.

We will first provide a theoretical outline of the two different methods, then a brief overview of the experimental set up, after which an overview of the results will be given, followed by the conclusion in which we extensively explain the merits and limitations of the current available methods. In the appendix an easily readable 'Users Guide' is provided.

2 Methods

Both algorithms that we will implement in this report use the power of hash functions. The idea is that when more different elements are appearing in a data stream, more different hash values will result from these elements. If we see many different hash values, the chance increases that we see very unlikely hash values as well, such as one where the value ends in many zeroes. When hashing the data stream to a bit-string we can exploit this property, by only storing the length of the longest tail of zeroes that we have come across while hashing our data stream. This is the core idea of the two algorithms that will be explained in the following subsections. Counting the number of trailing zeroes in a bit-string (i.e., integer value) is implemented in the following way:

- For each bit string estimate the number of trailing zeroes by:

1 If the number is equal zero, return 32 (We implement a 32 bit version).

2 Otherwise check the last element of the bit string of the number.

3 Add one to a counter R if this element is a 0 and shift the bit string once to the right.

3 While the last element of the (shifted) bit string is a 0, go back to step 2.

4 Return the counter R, this is the number of trailing zeroes in this particular bit string.

2.1 The Flajolet-Martin Algorithm

The Flajolet-Martin algorithm is coined in Flajolet and Martin (1983), and the estimation of the cardinality of a data stream works as follows. We call the hash function h and the stream element a. $h(a)$ might then end in some amount of zeroes, which we will call the tail length of $h(a)$. If we denote the maximum length of any $h(a)$ that we have seen so far by R, then we can estimate the number of distinct elements in the data stream by 2^R.

This estimate makes sense because the probability that a given $h(a)$ ends in at least r zeroes is given by

2^{-r}. Therefore, if we have seen m distinct elements in the stream, the probability that none of these elements has a tail length of at least r is given by

$$\left(1 - 2^{-r}\right)^m \tag{1}$$

This can be rewritten to

$$\left((1 - 2^{-r})^{2^r}\right)^{m2^{-r}} \tag{2}$$

which is approximately equal to

$$\exp\left(-m2^{-r}\right) \tag{3}$$

for large r. Two conclusions can be drawn from Equation 3. If m is much larger than 2^r, then the probability that we find a tail of at least length r goes to one. Conversely, if m is much smaller than 2^r, the probability of finding a tail of length r goes to zero. This crudely shows us that this method of estimating the number of distinct elements is unlikely to be much too high or much too low.

We can improve upon this crude estimate by combining the results of different hash functions. However, this should be done with care for the following reasons. One might naively think that the best method to combine the results of different hash functions would be to take the mean or the median of all the values of 2^R that we get from each hash function. The problem with taking the mean is that the value 2^R doubles as R increases by just one, therefore the expected value of 2^R when taking the mean of many hash functions will grow infinitely. When taking the median value we are not sensitive to outliers in the value of 2^R, but the result will always be an integer power of two, and thus it will be impossible to obtain a close estimate with this method.

Chapter 4 of Rajaraman and Ullman (2011) discusses a solution to the problem, which is a combination of the mean and the median. The method is to group hash functions into buckets, and compute the average of the results of these buckets. The median of the averages is then taken as the resulting estimate for m. This reduces the effect that any outlier may have on the final result, and provides in principle any number as the resulting estimate for m, as long as enough hash functions are used. Group sizes are recommended to be of at least a small multiple of $\log_2 m$. However, before running the experiments that will be discussed later, we have found that taking the median of the results of the buckets, and then taking the mean of those medians gives a much better estimate for m, and thus this implementation will be used throughout this report.

2.2 The Log-Log Algorithm

A major disadvantage of the Flajolet-Martin algorithm is the high variance. A method to combat this variance was invented by Durand and Flajolet (2003), who proposed the Log-Log algorithm. This algorithm scans a multiset M of hashed elements, observing the patterns of the form 0*1 that might occur at the beginning of the hashed elements. This algorithm uses the position of the first 1-bit, $p(x)$, in element x to estimate the cardinality n. Similarly to the counting of the number of trailing zeroes in Section 2.1, We would expect about $n/2^k$ of the distinct elements of M to have a p value of k. Therefore, the quantity

$$R(M) = \underset{x \in M}{max}\ p(x) \tag{4}$$

can be used as a rough estimate of the value of $\log_2 n$. With this in mind, we can explain the next idea. This idea consists of separating elements into m buckets, where $m = 2^k$ and k is the number of bits of x that is used to represent the index of a bucket. We use the first k bits to determine which of the 2^k buckets are used and the remaining $32-k$ bits as the actual hash function. In this way, one 32 bit hash function is split into k $(32-k)$-bit hash functions. In this way, we keep 2^k separate cardinality estimates $R(M)$. These estimates are then combined by averaging the results of each bucket and the estimate for the cardinality is then given by

$$E = \alpha_m * m * 2^{1/m\sigma_j R_j} \tag{5}$$

where $\alpha_m = 0.79402$ (as found by Durand and Flajolet, 2003) and R_j is the parameter R computed for every bucket $j \in [0, m-1]$.

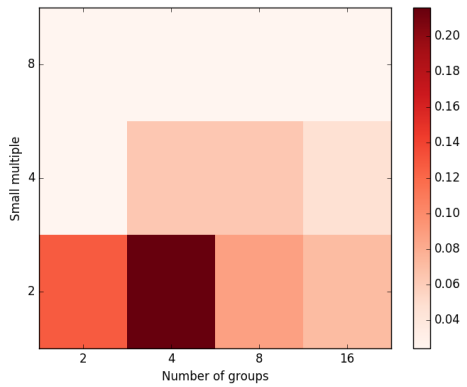
When implementing this algorithm we choose to use not the position of the first 1-bit but the number of trailing zeroes, as this function is also used for the Flajolet Martin algorithm. This approach is equivalent to using the position of the first 1-bit, since the probability of the p-th bit being the first 1-bit is the same as the probability of p being the number of trailing zeroes.

3 Experimental Set-up

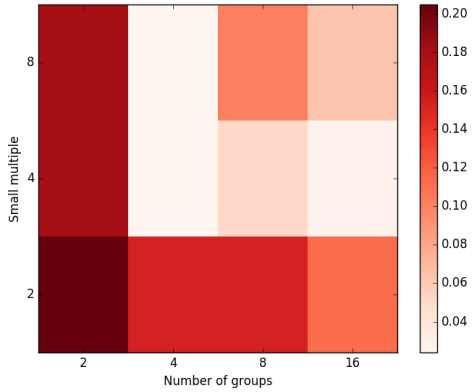
To test the two approaches set out in Section 2 we adopt an iterative process in which we loop over parameter combinations. In this way we hope to achieve a fill factor for our parameter space which allows us to draw sensible conclusions on preferred parameter values. One of our objectives is to formulate a recommendation to future users as to what parameter configuration they should use for a given order of magnitude of expected cardinality. We hope to be able to make such a recommendation by testing several parameter configurations for lists of elements with different numbers of unique elements.

Since we only want to test the performance of the two algorithms for different numbers of unique elements, we do not have to create text files, hash those and feed the hashes to our algorithm. Instead it is sufficient to simply generate a set of unique random integers (bit-strings of length 32) of a given length. Doing this we know precisely how many unique elements to expect, the length of the list, and can easily compare the approximation of the algorithm with the true number to calculate the Relative Approximation Error (RAE):

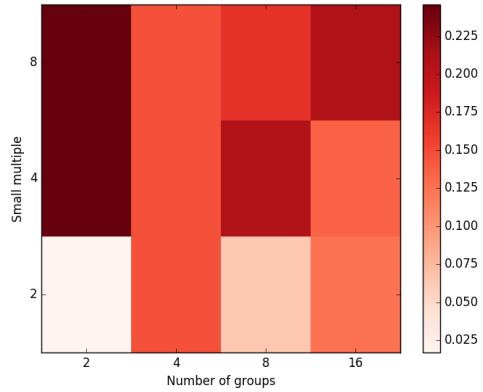
$$RAE = |\frac{C - E}{C}| \tag{6}$$



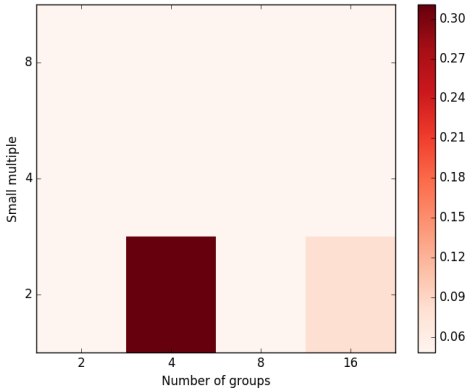
(a) Unique elements = 10^3



(b) Unique elements = 10^4



(c) Unique elements = 10^5



(d) Unique elements = 10^6

Figure 1: Relative error as a function of both the small integer and the number of groups for the Flajolet-Martin algorithm for different numbers of unique elements. As can be seen, there is no clear trend indicating what combination of parameters to choose.

where C is the actual cardinality and E is the estimated cardinality.

The parameters we test for the Flajolet-Martin algorithm are the number of hash functions, the number of groups these hash functions are partitioned in and the cardinality or size of the test set. We test all possible combinations of the parameters values given in Table 1. The only difference in this algorithm from the bare theory as presented earlier is that we take the average RAE over 10 repeats for every combination of parameters. This was implemented because of the large spread in results we obtained when running the same parameter setup multiple times as will be shown in the results.

Table 1: Parameter values used in Flajolet-Martin Algorithm

Parameter	Values
small multiple	2, 4, 8
number of groups	2, 4, 8, 16
size list	10^3, 10^4, 10^5, 10^6

The parameters we test for the LogLog algorithm are only the size of the list of unique elements (the cardinality) and the number of elements of the bit-string that are masked k, since the number of groups is determined uniquely by k to be 2^k. The tested sizes of the list of distinct elements are the same values as given in Table 1, and the parameter k is tested in integer steps between 2 and 22. Again, we average the RAE for multiple consecutive runs of the algorithm with the same parameters to get a more reliable estimate for the RAE.

4 Results

In this section we will describe the results obtained while probing our parameter space and try to interpret the results. To this end we employ several plotting mechanisms and show several singular plots which are drawn from a specific position in our parameter space to provide further elucidation, which in some cases is desperately needed. Again we will divide this section in two parts, first describing the Flajolet-Martin algorithm followed by the LogLog algorithm. A final comparison between the two will be made using the best possible results we achieved.

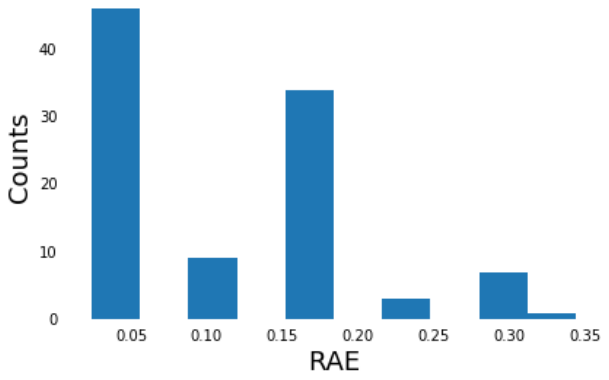


Figure 2: Histogram displaying the different RAE values for repeating the cardiancity estimate 100 times with the same parameter configuration (size=10^3, 8 groups and $4 * \log_2$ hash functions). This histogram displays a highly non-normal distribution.

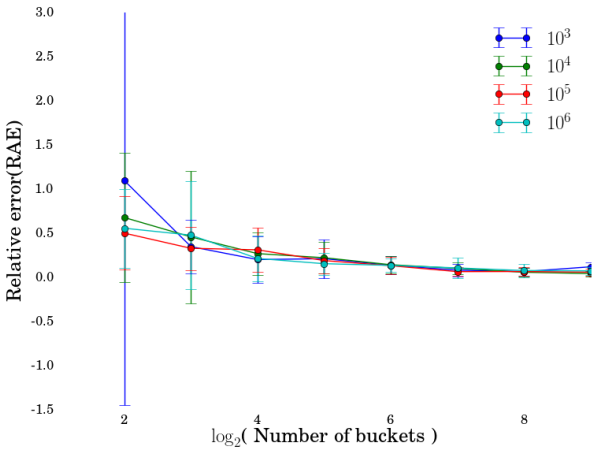


Figure 3: Relative error on the cardinality estimate for different numbers of unique elements and their 1σ bounds. These results were obtained by averaging over 50 iterations for each parameter configuration. Clearly, all cardinality estimates increase in accuracy for a higher number of buckets, all seem to follow them same trend.

4.1 Flajolet-Martin

The results for this algorithm are not simple to interpret. As a result we took several steps to gain additional understanding of the working of the algorithm and the results it produces. We started our experimentation by the method described before, looping through different combination of values of the relevant parameters (Table 1). As stated before, the RAE of the algorithm was determined by averaging the RAE over ten loops for every parameter combination and is shown in Fig. 1. It can clearly be observed that there is no trend whatsoever in these plots. Where we would expect a higher number of groups to always be beneficial for the accuracy, this does not appear from these results.

To investigate further what might be causing these results we searched for possible explanations, one clear clue is obtained from repeating a specific parameter configuration 100 times and studying the histogram of RAE for these runs. This histogram is displayed in Figure 2, it can be observed that the distribution of the RAE for repeating the cardinality estimate 100 times using the same parameters is far from a Gaussian. This makes it particularly difficult to use the RAE in any practical way. From the results displayed here it is not clear what value we should pick, not even after 100 runs. The initial guess might be to select a RAE of < 0.05, but 20% of the time, the RAE is actually four times as big. Taking the median in this histogram will result in a value close to 0.05, but in the next parameter configuration we might, by bad luck, have a different distribution and select a much bigger value. This explains the fact that there seem to be no trends in Figure 1.

4.2 LogLog

Since the algorithm employed only creates a two dimensional parameter space in which one of the dimensions is controlled rather than tuneable, we can display the results in more easily understandable 2D line plots. The plot in Fig. 3 clearly illustrates that increasing the number of buckets is crucial for improving the cardinality estimate. In this figure the number of unique elements doesn't seem to have a large impact on the relative error. The size of the error bars, although big for a small number of buckets, seems to decrease quite uniformly as well. A close look at the decreasing trend in Figure 4 reveals that the relative error of different numbers of unique elements decreases similarly. For larger numbers of unique elements this decreases continues for a larger range of number of buckets, but is not substantially different in character than for small numbers of buckets.

As can be seen in Fig. 4 the relative accuracy improves up to a certain point, after which it drastically increases. This peculiar behaviour can easily be explained by examining the LogLog algorithm. The algorithm creates a certain number of buckets, set by 2^k and crucially creates an empty list with that length. The list is to be filled with the number of zeroes in each bucket, however when there are more buckets than the total number of elements we supply the function with, many of the buckets will be empty. This means that the exponential in the cardinality computation will tend to unity, since the average of a lot of empty buckets and a few filled buckets is close to zero. If the exponential is effectively set to unity, the cardinality estimate will be very close to $2^k * \alpha$. Thus, when creating ever more buckets than elements in the stream, the cardinality estimate will simply scale with 2^k, which is what we see in this figure. This still holds true if not all elements in the stream are unique, as is realistic in real life applications, since the bucket ID's are hash functions of the first k bits of an element, these are fairly unique for all streams with a total number of elements $< 2^{32}$, and when we encounter an element for the second time its number of trailing zeroes will be compared with the number of trailing

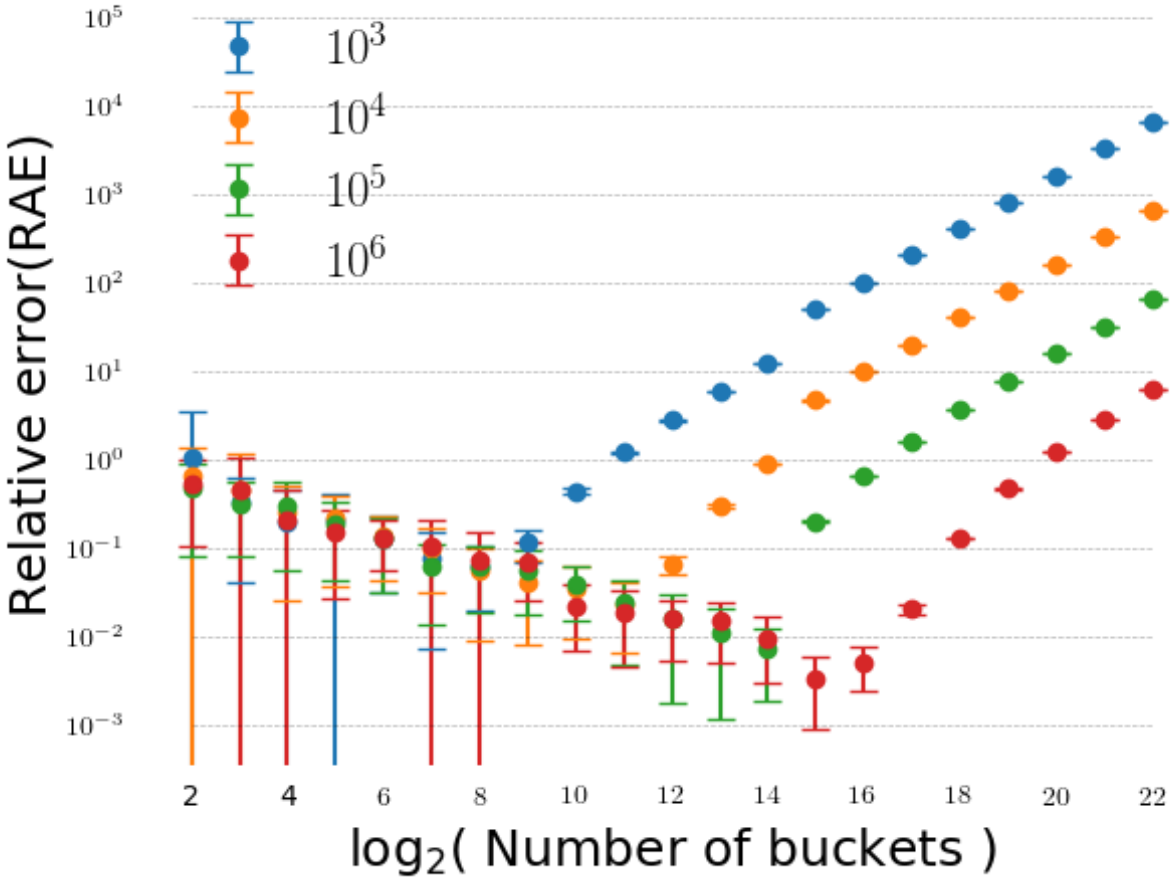


Figure 4: Relative error on the cardinality estimate for different numbers of unique elements. These results were obtained by averaging over 20 iterations for each parameter configuration. There is a clear turn off point from the otherwise uniformly decreasing lines for all sizes. This turn off point corresponds with the number of elements fed to the cardinality estimator which is here equal to the number of unique elements.

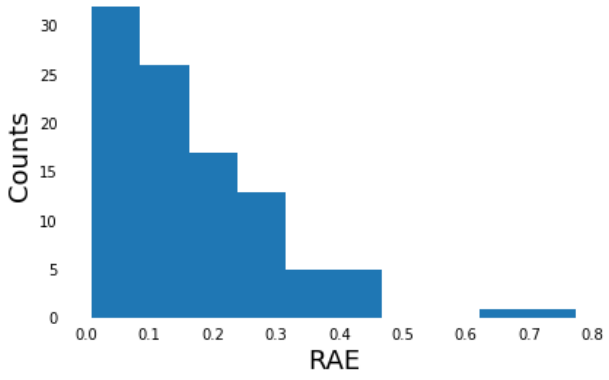


Figure 5: Histogram displaying the different RAE values for repeating the cardinality estimate 100 times with the same parameter configuration. The parameters used are: size of 10^3, $k = 5$. The histogram displays a near-normal distribution.

zeroes in the same bucket as we did the first time we encountered this particular element, so no change in the turn-off is expected. This is also verified by experiments not shown in this report. This gives us crucial information for the configuration of the number of buckets for a given expected number of unique elements. The number of buckets should never exceed the number of unique elements, as it will clearly cause unreliable estimates. But, the number of buckets should neither be to small since the accuracy continually improves until the point where the number of buckets equals the number of unique elements.

To demonstrate the much higher stability of this algorithm compared to the Flajolet-Martin algorithm, we again display the histogram of repeating a cardinality estimate 100 times with a fixed parameter configuration. In Fig. 5 a near normal distribution can be seen, indicating that drawing a sample from this distribution by taking the mean over a 20 iterations is a likely to give a fair estimate of what the RAE would on average be, if we were to average over many runs.

5 Complexity

In this section we will discuss the time and memory complexity of the two algorithms for the case where they are simply used with one given parameter configuration, instead of probing the parameters space as done in this report. The complexity will be indicated using the big-O notation, which indicates the scaling of the complexity with the change of certain variables. The time complexity is taken as the number of steps the computer has to take to execute the algorithm as a function of the algorithm's parameters. The memory complexity is the amount of storage needed for the algorithm to run as a function of the algorithm's parameters.

5.1 Flajolet-Martin

This algorithm has the following parameters: the number of groups (G), the number of different hashes (H) and the number of elements (E). The simplest possible version of the algorithm uses only one hash for every element and keeps in memory only the maximum number of trailing zeroes encountered so far. Therefore the memory requirement is $O(\log \log E)$ since we need to keep at most the number of trailing zeroes in the bit string of length E in memory. The time complexity scales linearly with the number of elements: $O(E)$. Our implementation uses multiple hashing functions that are combined to give a certain estimate. For every hash function we need to keep in memory the maximum number of trailing zeroes so far, thus memory scales with $O(H * \log \log E)$. The number of operations increases from 1 per element to H per element, thus the time complexity is $O(H * E)$. Since it is recommended to use $H =$ a small multiple of $\log_2 E$, this algorithm scales fairly poorly in time with increasing number of elements.

5.2 LogLog

This algorithm has as parameters: number of masked bits(k), number of elements(E). The maximum length of the bit string required is again $log_2 E$. Thus, again, the memory required scales with $O(\log \log E)$ if we use only 1 bucket. But the algorithm partitions the hash function into 2^k buckets, thus the total memory required scales with $O(2^k * \log \log E)$. This algorithm uses only one hash function to achieve this, which is the main advantage over the FM algorithm. Thus the time complexity is linear with the number of elements: $O(E)$.

6 Conclusion

We have tested two distinct cardinality estimation algorithms which both rely on the maximum number of trailing zeroes present in the bit-wise representation of a list of elements. The two algorithms have widely varying performance and tractability. Where the LogLog algorithm is both fast, easy to interpret and has a high accuracy, the Flajolet-Martin algorithm is slow, extremely hard to interpret given the large fluctuations in the accuracy and generally has a lower accuracy. The Flajolet-Martin algorithm seem to have a relatively high RAE, with the lowest error recorded being 2.5%, whereas the LogLog algorithm can reach an error of less than 1% for the largest number of distinct elements. Having said that, for a small number of distinct elements, the LogLog algorithm struggles to obtain even a 10% accuracy, where the Flajolet-Martin algorithm can at times still obtain 2.5%, however it is much less stable.

The near-normal distribution of the RAE results for the LogLog algorithm indicated that we can make a useful prediction based on our results for different parameter settings. This is not the case for the Flajolet-Martin algorithm, since the distribution is far from Gaussian, any prediction we make might be based on our sampling of one particular feature of the distribution which might only be reproduced very sporadically. However, the distribution did show that the mode prediction was quite accurate (RAE < 0.05), indicating that taking the mode of several 100 runs of the algorithm might yield a good estimation for the cardinality. This was only tested for a single parameter configuration so clearly, more work should be put into exploring the statistics of the Flajolet-Martin algorithm to pursue this lead further.

From these conclusions we have provided a 'Users Guide' to counting distinct elements in limited memory. This guide can be found in Appendix I.

References

Durand, Marianne and Philippe Flajolet (2003). *Loglog Counting of Large Cardinalities*.

Flajolet, P. and G. N. Martin (1983). "Probabilistic counting". In: *24th Annual Symposium on Foundations of Computer Science (sfcs 1983)*, pp. 76–82. DOI: 10.1109/SFCS.1983.46.

Rajaraman, Anand and Jeffrey David Ullman (2011). *Mining of Massive Datasets*. New York, NY, USA: Cambridge University Press. ISBN: 1107015359, 9781107015357.

Appendix I: How to Count Distinct Elements in Memory

We provide here a users guide to counting N distinct elements in a data stream which does not fit entirely into memory. Two algorithms are available for this task: the Flajolet-Martin (FM) algorithm and the LogLog algorithm, which have been investigated in this report. The first thing the user should do is estimate the order of magnitude (N) of the number distinct elements that can be reasonably expected from the data stream.

Then, we have found that the FM algorithm should be used if:

- The user favours accuracy over computation time.
- The expected N is relatively small ($< 10^5$)

Additionally we recommend the following about the parameters of the FM algorithm

- The number of hash functions used is ideally more than at least $4 * \log_2 N$, although more seem to increase the accuracy.
- The number of groups does not seem to have a big influence on the accuracy.

These conclusions come from the fact that the LogLog algorithm only outperformed the FM algorithm in the case that $N \geq 10^5$. But this increased accuracy comes at the cost of more computation time as the FM algorithm requires more and more hash functions with increasing N.

The LogLog algorithm should be used if:

- The user favours quick computation times over accuracy.
- The expected N is larger than 10^5.

We have also found that the k value which indicates the number of masked bits as a bucket-ID should ideally be picked to be as large as possible, but within the limit $2^k < N$ as buckets will be largely empty then. Therefore we recommend the user to choose $k = \log_2 N - 2$.

In closing, given that 10^6 distinct elements can be easily counted in memory on almost any given modern computer, the LogLog algorithm seems to be the only realistic choice for a real-world application of probabilistic counting. This is due to the time complexity of the algorithm scaling better as well as the accuracy of the algorithm scaling better with increasing numbers of distinct elements.

If $\mathrm{N} < 10^5$
FM with $> 4 \log_2 \mathrm{N}$ **hashes**

If $\mathrm{N} > 10^5$
LogLog with $\mathrm{k} = \log_2 \mathrm{N} - 2$