Advances in Data Mining 2018: Assignment 2

October 12, 2018

1 Introduction

Over the past decades, data has grown in terms of volume, velocity and variety and several other aspects. In the early days, data used to be structured and stored in databases most of the time, where it could be retrieved when needed. However, nowadays data is often so big in volume and comes in at such a high speed that it can not be stored. Data like this is often referred to as data streams. Even though data streams can not be stored and analyzed thoroughly, sometimes it is necessary to be able to interpret properties of the data. In this assignment we study one of those properties, namely counting the amount of distinct elements in a datastream. Since the amount of main memory is not always sufficient to keep track of all of the distinct elements that have passed a data stream, it is necessary to estimate the amount of distinct elements in a stream. There are several ways to do so, of which we discuss two algorithms; the LogLog algorithm and the Flajolet-Martin algorithm. In this report we present the theoretical framework for both of the algorithms, we discuss the way we implemented the algorithms in Python and we compare the results of both algorithms. We set up an experiment to establish the relation between:

- The magnitude of the expected count (N).
- The amount of required memory ($M : number\ of\ bytes$)
- The Relative Approximation Error (RAE):
  $RAE = abs(true\ count - estimated\ count)/true\ count$

Throughout the assignment, we will provide an answer to the following question by means of a practical guide: *'How to count distinct elements in limited memory?'*. For a full description of the assignment, see[1].

2 Methods

2.1 Hashing

As mentioned before, we discuss and two algorithms that estimate the number of distinct items in a stream. Many algorithms, as well as the two algorithms we are discussing here, make use of a hashing procedure in order to do more complex calculations and estimations. Patterns in those bit strings can be used to indicate the amount of distinct values in the collection. Which is what we are doing in this assignment. Hashing is the process of converting data into bit strings according to rules, called hash functions. A problem that can occur when hashing is collision, which happens if two or more distinct values are represented by the same bit string. However, the more different hash functions are used, the smaller the chance of collision will get. To simplify the assignment, we did not have to create hash functions ourselves in order to test our algorithms. Instead, we simulated a stream of 32-bit long integers which looked like 32-bit long hashes of objects.

2.2 Flajolet-Martin Algorithm

The Flajolet-Martin algorithm is named after its creators and is based on the amount of trailing zeroes in the mentioned bitstream at any given moment. With trailing zeroes is meant: The amount of zeroes that have passed without having it intermitted by ones. There are several implementations and version of the Flajolet-Martin algorithm which differ in small details. In this assignment, we implemented the version of the Flajolet-Martin algorithm as it is described in chapter four of the textbook. The algorithm executes the following steps:

1. Pick one or more hash functions h that map each of the n elements to at least $log_2 n$ bits.

2. For each stream element a, let $r(a)$ be the number of trailing zeroes in $h(a)$.
3. Let R be the maximum $r(a)$ seen. In other words, let R be the largest number of trailing zeroes seen so far.
4. Estimate the number of distinct elements with Est $= 2^R$.

2.2.1 Theoretical background

The authors of chapter four in the textbook find reason to believe that this estimation is accurate because of several technical matters, and the following arguments to back their theory:
First, the number of times elements repeat in the stream has no effect on the value of R since every time the same value is hashed, the same trail of zeroes is generated. Therefore, the value of R only depends on the number of distinct items in the stream.[2]
Secondly, the probability that the hash value for any given element n in i zeroes decreases exponentially with i, so when i is increased by one, the amount of different elements should be doubled in order to have a good chance to see $i + 1$ zeroes at the end of some hash value.
This is supported by the following mathematical explanations:

- The probability that a given $h(a)$ ends in at least i zeroes is 2^{-i}.
- If there are m distinct elements, the probability $P(R \geq i) = 1 - (1 - 2^{-i})^m$. Which is the probability that none of $h(a)'s$ end in fewer than i zeroes.
- Since 2^{-1} is small, $1 - (1 - 2^{-i})^m \approx 1 - e^{-m2^{-i}}$.
- Therefore, if $2^i \gg m$, $1 - e^{m2^{-i}} \approx 1 - (1 - m2i) \approx m/2^i \approx 0$
  and if $2^i \ll m$, $1 - e^{m2^{-i}} \approx 1$.

Thus, since 2^R is never much larger or much less than m, it can be said with confidence that 2^R will almost always be around m, which makes it a good estimate.

However, there are two problems with this approach.
First, the number of distinct elements in the stream can be easily overestimated since there is a certain probability of obtaining a hash element that contains only $0's$. As the probability of obtaining a hash element with only $0's$ halves when the amount of maximum trailing zeroes increases with one, the estimated amount of distinct elements doubles. Therefore, the estimation may be way too large, since 2^R becomes a huge number.
A second problem is that the estimated amount of distinct elements is always an even number, since it will always be a power of two.
To overcome these problems, it is possible to use multiple hash functions in order to maximize the precision. This leads to multiple sets of binary strings for the same input values. These sets can be divided into small samples on which an estimation can be calculated. By combining the different estimations from these samples, a more accurate estimation can be done. Flajolet and Martin according to the book combine samples as follows:

- Partition samples into groups with a size of at least a small multiple of $log_2 n$, where n is the size of the universal set. We choose $2 \cdot \log_2 n$
- Within the groups, calculate the median of the estimations.
- Over these medians, take the average.

For the sake of clarity, this is the exact opposite of what is presented in chapter four of the textbook. In the textbook it is proposed to first calculate the median of the group, and take the average over the medians.
According to chapter four in the textbook, the only thing that is kept in main memory is one integer per hash function, which is the maximum amount of trailing zeroes. The relation between the cardinality n and the amount of required memory M in bits for m different groups would for our implementation be: $M = 2 \cdot m \cdot (\log_2 n)^2$.

2.2.2 Implementation

To implement the algorithm in Python, we took the following steps:

1. In order to simulate a stream of hashing values, we first generated an array filled with 32-bit long integers which we used as input for the algorithm. To mimic the effect of using different hash functions, we generated multiple groups of binary integer strings.
2. Subsequently, we created a function that returns the number of trailing zeroes for given integer expressed in binary by deleting the most right end of the bit string one by one, while increasing the variable containing the number of trailing zeroes if a zero passes, without being interrupted by ones.
3. To estimate the cardinality, we created a function that returns 2^R where R is is the maximum amount of trailing zeroes that was seen in the bit stream.
4. In order to combine the estimates of multiple samples to get a more accurate estimate, we created a function to calculate the average of the group medians, which uses the number of elements and the number of groups as input. Each group consists of $2 \cdot \log_2 n$ estimates of different hash functions. The number of groups times the group size can be seen as the number of different hash functions used. However, since we did not use hash functions, we generated new sets of random 32 bit integers. This is equivalent to using multiple hash functions.

2.3 LogLog Algorithm

2.3.1 Theory

The main idea behind the Durand-Flajolet LogLog Algorithm is the same as the Flajolet-Martin Algorithm: map the data-values of the stream using a hash algorithm to a binary representation of an integer and then count the number of trailing zeroes to estimate the cardinality of the dataset. The main difference between the algorithms however is that the Durand-Flajolet algorithm groups the hashed data-values of a given stream into buckets. The number of buckets m can be be expressed as a power of 2: $m = 2^k$. The grouping is done by using the first (least significant) k bits of the hashed data-value to represent the index of the bucket in binary, note that k bits has $2^k = m$ different ways of arrangement thus making it possible to have m different groups. After the hashed data-value has been placed in its bucket the binary representation of the hashed data-value can be stripped off the first (least significant) k bits and the remaining bits are used to find the maximum number of trailing zeroes for each bucket. Define R(j) as the maximum number of trailing zeroes for values with bucket index j. The way R(j) is found is identical to how R is found in the Flajolet-Martin algorithm. In the limit of many different buckets m the average of the values R(j) over all buckets: $\frac{1}{m}\sum_{j=1}^{m} R(j)$ can be expected to approach the number of trailing zeros that the cardinality would have in binary were it to be stripped of its last k bits. This number would be of course be equal to $\log_2 n/m$. However the estimate of the total cardinality is slightly off what one might expect:

$$E := \alpha_m m 2^{\frac{1}{m}\sum_{j=1}^{m} R(j)}, \tag{1}$$

as there is an extra multiplication with the constant α_m. This constant corrects the systematic bias of the averaging of $R(j)$ and depends in the following way on the number of buckets m:

$$\alpha_m := (\Gamma(-1/m)\frac{1-2^{1/m}}{\ln 2})^{-m}, \tag{2}$$

where Γ is the gamma function.[3] Since the method is still probabilistic, the estimate is not perfect and repeating the method many times will give a range of estimates that will be distributed in a similar way to that of a Poisson distribution. The standard error of this distribution for N samples is defined as:

$$\text{Std. error} = \frac{1}{n}\sqrt{\mathbb{V}_n(E)} = \frac{1}{n}\sqrt{\frac{\sum_{i=1}^{N}(E_i - n)^2}{N}}, \tag{3}$$

where n is the cardinality of the stream and N the number of times the expectation has been computed. Note that this definition is exactly equal to the definition of the Relative Absolute Error defined in the assignment if N=1.[1] It turns out that the standard error depends on the number of buckets m approximately in the following way:

$$\text{Std. error} \approx \frac{1.30}{m}.[3] \tag{4}$$

The advantage of this algorithm primarily lies in the (substantially) lower memory usage as the amount of memory M (in bits) needed for the algorithm is equal to:

$$M = m \log_2 \log_2 (\frac{n}{m}).[3] \tag{5}$$

2.3.2 Implementation

To implement the algorithm in Python similar steps as in the Flajolet Martin algorithm were taken:

1. Instead of using hash functions we use datasets of randomly generated 32 bit integers to mimic the same effect.
2. We use a function that returns the number of trailing zeroes for a given binary integer representation in the same way as before.
3. To estimate the cardinality, the number of different buckets m has to be specified beforehand. Based on the number of bits $k = \log_2 m$ each integer of the stream will have its bucket identified by its (least significant) first k bits. These first k bits will be the index of the bucket and then the number of trailing zeroes will be counted for the binary representation thats left over after the integer is stripped of the first (least significant) k bits. If this number of trailing zeroes is larger than the maximum value already assigned for integers in that bucket it will replace the previous value, otherwise it will be discarded. After having gone through the procedure for all integers, the values $R(0), R(1), ..., R(m)$ will be mathematically combined as in equation 1 and returned as the estimate for the cardinality.

2.4 Experimental setup

To analyze and verify any theoretical relationships between the error, the amount of required memory and the number of distinct elements in the stream, we run multiple experiments. We run slightly different experiments for the two algorithms, however the idea behind the experiments are similar. The error could be dependent on both the cardinality of the stream as well as the memory used in the algorithm. So the algorithms were run for a range of different cardinalities n and memories m and the errors were tracked and saved to see if the theoretical relationships hold up.

2.4.1 Flajolet-Martin

In order to study the relation between the error and the number of distinct elements in the stream we set up an iterator that runs the algorithm with input cardinality starting from $n = 10^4$ up to $n = 10^5$ in incremental step sizes of $\Delta n = 10^4$ all for the same number of groups $m = 10$. For each cardinality n the algorithm is run 10 different times and the standard error is calculated over these 10 runs for each n.

For analyzing if there is any relationship between the standard error (which is the relative absolute error for 1 run) and the amount of memory used, we do a similar experiment. The amount of memory m for the Flajolet-Martin will be the number of groups the algorithm averages over to come to the final estimation. We run the algorithm with input memory starting from $m = 10$ groups up to $m = 100$ groups in incremental step sizes of $\Delta m = 10$ all for the same cardinality of the dataset $n = 10^4$. For each number of groups m the algorithm is run 10 different times and the standard error is calculated over these 10 runs for each m.

2.5 Durand-Flajolet

To determine the theoretical relationship between the standard error and the number of buckets m we will run the algorithm 1000 times for the following values of m: $m = [64, 128, 256, 512, 1024, 2048]$. This is done for a range of cardinalities: $n = [10^4, 2 \cdot 10^4, 3 \cdot 10^4]$. Then we will calculate from the range of estimates that were calculated for a single value of m the standard error in the distribution according to the definition of the standard error as in equation 3. To verify that the relationship between the number of buckets and the error is in the following form: $\alpha/\sqrt{m}$, we will use the scipy function curve-fit to fit the function $\alpha/\sqrt{m}$ through the obtained datapoints. The curve-fit function will give back the parameter α that minimizes the squared error between the curve and the datapoints. If the relationship holds up the function should return a value for the parameter α close to 1.30 and the curve should go smoothly through the datapoints.

3 Results and interpretation

3.1 Flajolet-Martin

Figure 1 shows a plot of the standard error computed over the 10 runs of the Flajolet-Martin algorithm for each cardinality n for a fixed number of groups $m = 10$. There does not seem to be any clear relationship between the standard error and the cardinality of the dataset as the error seems to fluctuate about a constant value of roughly 0.11. This result would agree with the literature (chapter four of the textbook) as no clear relationship was reported between the standard error and the cardinality of the dataset.

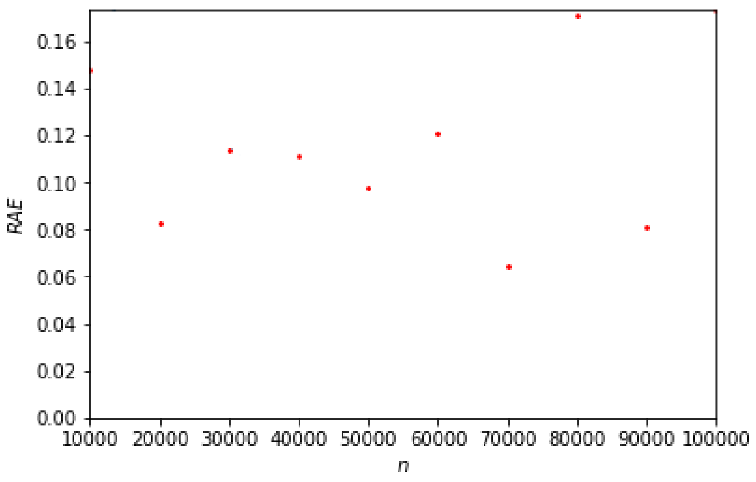

Figure 1: Plot of the standard error obtained from the 10 runs of the Flajolet-Martin algorithm for each cardinality n all for the same number of groups $m = 10$

Figure 2 shows a plot of the standard error computed over the 10 runs of the Flajolet-Martin algorithm for each number of groups m for the same cardinality of $n = 10^4$. There does not seem to be any relationship between the standard error and the number of groups m. In the original paper of the PCSA algorithm they reported the following relationship between the error and the memory: standard error $= 0.78/\sqrt{m}$ where m is the memory in units of words of (24-32 bits).[2] Even though the original PCSA algorithm is set up fundamentally different as the memory unit is a bitmap instead of a group of estimates as in our case, we still expected that the more groups the algorithm was run for the better the estimate would be.

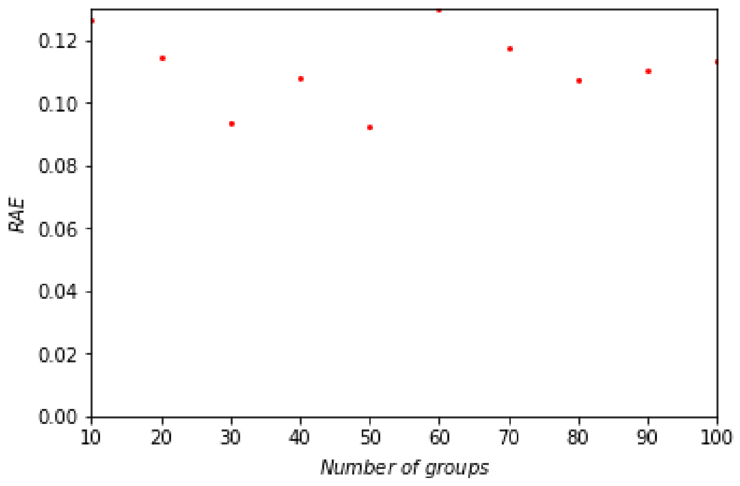

Figure 2: Plot of the standard error obtained from the 10 runs of the Flajolet-Martin algorithm for each number of groups m all for the same cardinality $n = 10^4$

3.2 Durand-Flajolet

Figure 3 shows how the distribution for the estimates of the Durand-Flajolet looks like after 1000 runs for a dataset with cardinality $n = 10^4$ and $m = 128$ buckets. As predicted in section 2.3 the distribution looks like a Poisson distribution. The estimate with the most counts in this histogram is close to the cardinality of 10^4, which is not surprising for $m = 128$ buckets.

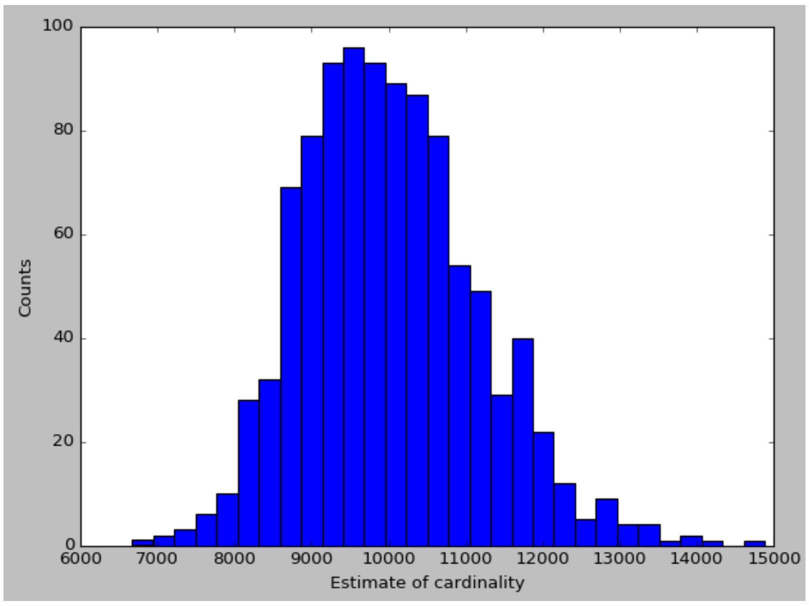

Figure 3: Histogram of the 1000 estimations obtained by running the Durand-Flajolet algorithm with a cardinality of $n = 10^4$ and $m = 128$ buckets.

Similarly the algorithm was run 1000 times for the other number of bucket values m. The results of the standard error for the different number of buckets m can be seen in figure 4 as the individual dots. What can also be seen is the fitted curve $\alpha/\sqrt{m}$ with α having the value that minimizes the squared error between the curve and the datapoints. For this data set the parameter $\alpha = 1.30$ was the best fit, which agrees exactly with the theoretical prediction in equation 4.

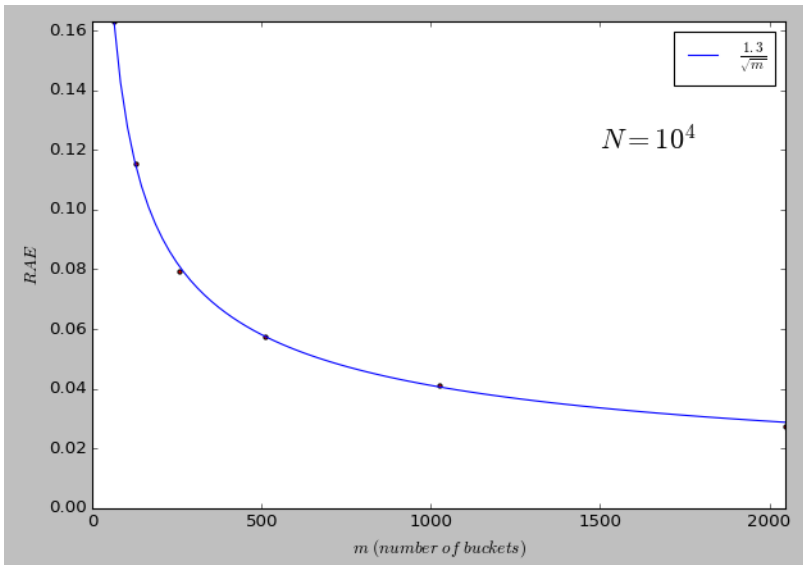

Figure 4: Plot of the standard error against the number of bucket m obtained from running the Durand-Flajolet algorithm for $N = 10^4$. The curve plotted is the optimal curve of that form going through the data points according to the least squared error.

The same experiment was done, but for cardinalities of $N = 2 \cdot 10^4$ and $N = 3 \cdot 10^4$, the results of which can be seen in figure 5 and 6. The resulting best fitting curves have parameters $\alpha = 1.31$ and $\alpha = 1.28$ respectively, both agree closely with the theoretical prediction of 4.

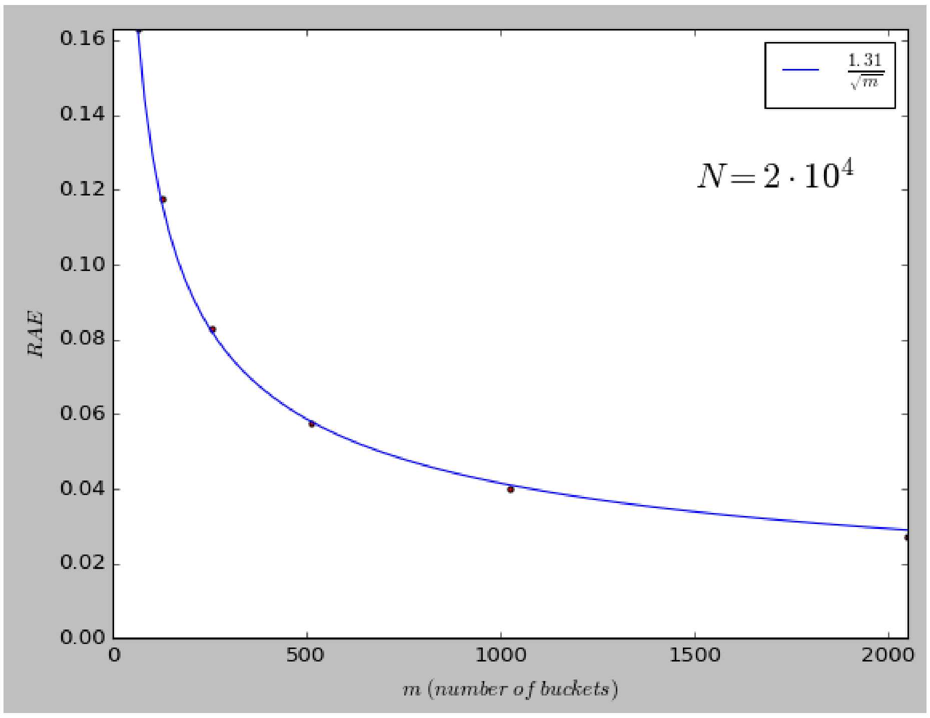

Figure 5: Same as Figure 4 but now ran with N=$2 \cdot 10^4$

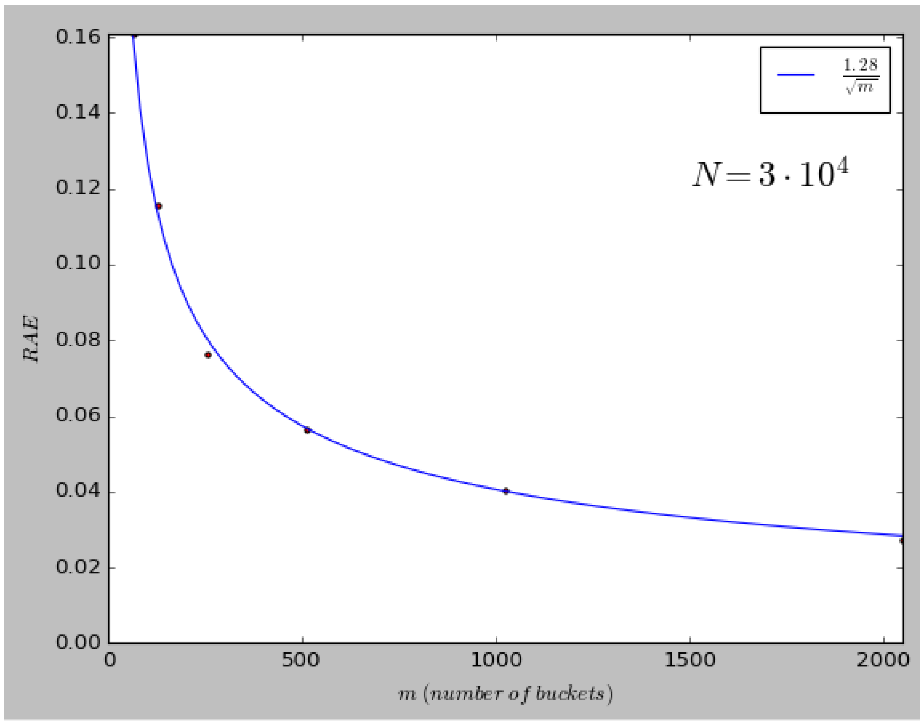

Figure 6: Same as Figure 4 but now ran with N=$3 \cdot 10^4$

4 Conclusion

4.1 Practical guide

We will present here a step by step practical guide of how to use the algorithms based on how accurate you want your estimate to be, while also giving you the maximum amount of memory you will need in the process.

4.2 Flajolet-Martin

Since the accuracy does not seem to become better for larger number of groups our advice would be to pick a small number of different groups of around 10. The reason being you will need less memory for less groups, however you will still need a small number of groups to have a meaningfull average. The amount of memory in bits needed for m groups and a dataset with cardinality n for the algorithm we implemented is equal to:

$$M = 2 \cdot m \cdot (\log_2 n)^2, \tag{6}$$

since there are m groups and each group has $2 \cdot log_2 n$ estimates and each estimate on average will have $log_2 n$ number of trailing zeroes. This means that for a dataset with cardinality of $n = 10^4$ and a run with $m = 10$ different groups the amount of memory needed will be equal to $M = 3531$ bits$\approx$ 441 bytes.

4.3 Durand-FLajolet

1. Pick a certain accuracy you want your estimate to achieve.
2. Use the standard error equation 4 to determine how many buckets m (must be a power of 2) you need.
3. Determine the amount of memory you will need. This depends in principle on the cardinality of your dataset, however an upper limit can be calculated from equation 5 by setting n equal to the size of you data set. The reason being the cardinality can not exceed the size of the data set. Make sure that each bucket has a whole number of bits by rounding the value up to the closest integer.
4. Run the algorithm for that number of buckets.

An example calculation: say you want roughly 6% accuracy, then according to equation 4 you will need $m = (1.3/0.06)^2 \approx 469$ buckets. However since the number of buckets must be a power of two the closest value that would give us atleast 6% accuracy is 512 since $2^9 = 512$. The maximum amount of memory we will need depends on the size of the data set n, but let's say it is $n = 2^{32}$. Then the amount of memory will be equal to $2^9 \log_2 \log_2 \frac{2^{32}}{2^9} = 2^9 \cdot 4.52 -> 2^9 \cdot 5 = 2560$ bits = 320 bytes. Notice that we made sure that each bucket had sufficient and a whole number of bits to store the maximum number of zeros, by rounding the value of 4.52 up to the closest integer 5.

4.4 Application

There are numerous applications of the algorithms that can help to solve problems. Algorithms that count distinct elements in a stream are often used to approach the amount of different visitors on sites such as Google or Facebook. There are, however, also less common problems that can be solved using an algorithm that counts distinct elements. For example, one could find spam websites or artificial websites using these algorithms when looking for the amount of different words that are present at a given website. If one would set up a page crawler, which is a program or algorithm that scans through websites, it is possible to implement an algorithm to estimate the distinct words on a site. Unusually low or unusually high numbers of distinct words could indicate artificial web pages (spam). By applying algorithms for this purpose, there is a chance to find scammers or hackers that make such websites.

References

[1] Kowalczyk, W. J. (2018). *Assignment 2: Counting distinct elements.* Retrieved October 2, 2018, from https://blackboard.leidenuniv.nl

[2] Flajolet, P., Martin, G. N. (1985). Probabilistic counting algorithms for data base applications. *Journal of Computer and System Sciences, 31*(2), 182–209.

[3] Durand, M, Flajolet, P. (2003). LogLog counting of Large Cardinalities. *Algorithms - ESA 2003. Lecture Notes in Computer Science. 2832.*