Most Frequent Letters

Bruna de Melo Simões
103453 - brunams21@ua.pt

Abstract **– This paper presents an overview of the counting elements in a stream of data, that is, how many different elements of a given type there are in an extensive dataset. In this paper, different approaches to calculate the most frequent letters in different literacy works are performed in order to identify and characterize the different characteristics between them.**

Using an exact count as reference, that is, a way to count the exact number of characters in a data stream by iterating continuously through each character, a counting methodology based on a fixed random probability, that is, an approximate count, and a space-saving algorithm are compared to identify the cases where one of the algorithms might be more advantageous than the other, as well as evaluating its performance in different scenarios.

Resumo **– Este artigo apresenta uma visão geral dos elementos de contagem num fluxo de dados, ou seja, quantos elementos diferentes de um determinado tipo existem num determinado conjunto de dados. Neste artigo são realizadas diferentes abordagens para calcular as letras mais frequentes em diferentes trabalhos de literatura, a fim de identificar e caracterizar as diferentes características entre elas.**

Usando uma contagem exata como referência, ou seja, uma forma de contar o número exato de caracteres num fluxo de dados iterando continuamente cada caractere, uma metodologia de contagem baseada em uma probabilidade aleatória fixa, isto é, uma contagem aproximada, e um algoritmo de economia de espaço são comparados para identificar os casos em que um dos algoritmos pode ser mais vantajoso que o outro, bem como avaliar seu desempenho em diferentes cenários.

Keywords **– Probabilistic Counters, Data Streams, Approximate Counters, Space-Saving Algorithm, Metwally et al. Algorithm**

Palavras chave **– Contadores Probabilísticos, Fluxos de Dados, Contadores Aproximados, Algoritmo de Economia de Espaço, Algoritmo de Metwally et al.**

I. Introduction

With great data streams, the problem of finding the most frequent elements becomes more prominent, since the number of computations necessary to compute and finding these items increases with the number of elements. In this paper, some solutions were implemented which try to find the most frequent letters within literacy works of different languages.

The implemented approaches are as such:

- **Exact counter:** An implementation of a simple exact counter, in which each character from the data stream is counter and added to a data structure which stores the letter and the amount of times that letter was detected. The most frequent letters in this case correspond to the entries that have a greater number of hits detected.
- **Approximate counter:** An implementation of a counter that only counts an element if a certain probability is hit, that is, an algorithm that, in a similar way to the exact counter, tries to count elements, except it only does so when a random number is within a threshold, and therefore, the characters are only counted within a certain probability. Therefore, the overall amount of elements counts should be close to the fixed probability defined multiplied by the total number of characters in the data stream.
- **Space-saving counter:** An implementation of the Metwally et al. (Space-Saving) Algorithm in which only the top k elements are considered. In this algorithm, the data structure which stores the number of hits of each letter never surpasses the k letters.

II. Algorithm Analysis

A. Exact Count

An exact count algorithm was implemented which goes through each items and counts the exact amount of times it appears. This means that every time the algorithm iterates through a new element, adds it to the list, and defines its counter to `one`. Therefore, it always counts the same number of elements and the same number of presences of that element in the data stream, independent of the times it is run.

The Algorithm 1, therefore, consists on the implementation of this strategy.

The algorithm starts by defining an empty set, T, and iterating through each character, i, of the literacy works. If the character is not present in the sets that contains the letters, then it is initialized an entry for the letter i with a value of `zero`, else, the value of `one` is added to the entry created in a previous iteration, c_i, which stores the amount of times a letter has

Algorithm 1: Exact counter

```
T ← ∅;
foreach i do
    if i ∉ T then
        T ← T ∪ {i};
        c_i ← 0;
    end
    c_i ← c_i + 1;
end
```

been counted.

B. Approximate Count

The approximate count algorithm [1] intends to find the most frequent letter by occasionally counting them from the data stream. The algorithm starts by iterating though each character in the data stream, and counting it if a randomized variable, with a uniform distribution between `zero` and `one` which is different for each iteration, and checking if its value is lesser than a fixed variable, which is constant through all iterations. This variable has a value of $\frac{1}{8}$ for all presented solutions. However, this value could potentially be adjusted to count more or less frequently by increasing and decreasing the probability value, respectively, as long as it belongs to the $[0, 1]$ range.

Algorithm 2: Approximate counter

```
Data: 0 < p < 1
T ← ∅;
foreach i do
    x ← uniform(0, 1);
    if p < x then
        if i ∉ T then
            T ← T ∪ {i};
            c_i ← 0;
        end
        c_i ← c_i + 1;
    end
end
```

The algorithm starts by having a premeditated value, p, which defines the static probability in which the algorithm will add elements to the counter with. The T set, which is a set that stores all elements, in this case, letters, is initialized as empty, Then, the algorithm iterates through each character and if the condition is met, that is, when a certain probability is hit, it starts the counting of that character as 0 if it doesn't belong in T yet, and if it already does, then the counter is incremented for that character.

Therefore, after the probability is hit, the algorithm performs in a similar way to the exact counter one.

C. Space-Saving Algorithm

The Space-Saving algorithm is a data stream algorithm which tries to find the most frequent items in a given dataset [2]. For this approach, the *Metwally et al.* algorithm [3] was implemented, as depicted in `Algorithm 3`.

This algorithm is space-efficient because only the top k frequent items are generated, not allowing at any time in the iteration of the characters there be any more `k` elements stored.

Algorithm 3: Space-Saving Counter

```
Data: k ∈ ℕ
T ← ∅;
foreach i do
    if i ∈ T then
        c_i ← c_i + 1;
    end
    else if |T| < k then
        T ← T ∪ {i};
        c_i ← 1;
    end
    else
        j ← arg min_{j∈T} c_j;
        c_i ← c_j + 1;
        T ← T ∪ {i} \ {j};
    end
end
```

This algorithm starts by defining the set of elements, T, as an empty set. Then, it iterates through each character in the data stream and verifies if the character is stored in the T set. If it is, then the counter for that character increases. If not, then it then verifies if the number of elements in T is lesser than k, that is, if the number of stored characters is lesser than the number of most frequent characters intended. If this last condition is true, then the counter for the character is added, c_i, and the element is added to the list of existing characters, T.

If none of the above conditions is met, then the algorithm finds the lowest existing count element in the set, c_j, and the counter of the iterated element, c_i is replaced by c_j and incremented by 1. The lowest previous element j is then replaced from the T set of letters with i.

III. IMPLEMENTED DATASETS

To analyse the correctness and efficiency of each algorithm, three different literacy works were provided as data stream, which consist of the literary work, *"The Odyssey"*, by Homer, in three different languages: english, spanish and french.

For each one of the three documents, which corresponds to one different language, the headlines, corresponding language stop-words and all non alphabetic characters were removed, including punctuation,

spaces and other ASCII characters that were not letters, as well as characters with accents or cedillas, in such way that only the Latin alphabet remained, which afterwards were all converted to uppercase.

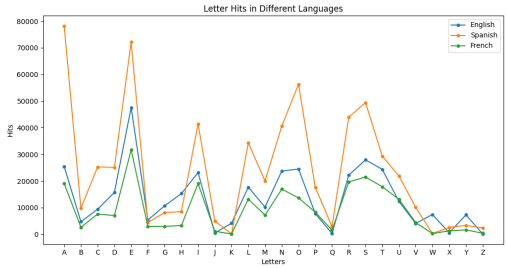

Fig. 1 - Number of letters detected for the same literacy work in spanish, english and french.

In `Fig.1` it becomes possible to analyse that, for the most part, the most frequent letters are similar in all three languages, with some small variations. By analysing the graph it is possible to conclude that the overall most frequent letters for all languages are "`A`", "`E`", "`I`", "`L`", "`N`", "`O`", "`R`", "`S`" and "`T`".

IV. Time efficiency

To evaluate the performance of all implemented algorithms, a series of tests were performed in order to identify the average times each algorithm took to run. Since the approximate counter is dependent on a randomized variable, and therefore, has different values each time is performed, a series of trials were conducted in order to try and get the average time execution time. Similarly, since the space-saving algorithm depends on the input of the number of the top elements to be calculated, the execution time also depends heavily on it. Therefore, the comparison made take into consideration the value of k, that is, the number of most frequent letters to be calculated.

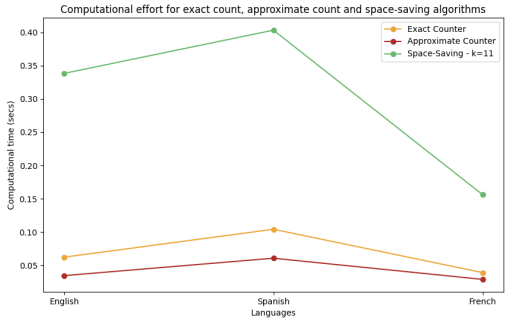

Fig. 2 - Comparison of time efficiency for exact count, approximate count and space-saving algorithms. The tests of computational time for space-saving algorithm was performed with k=11.

In `Fig.2`, it is possible to denote the time efficiency for each algorithm, taking into consideration the different languages. For every language, the most time inefficient algorithm is the space-saving one, while the most time efficient one is the approximate count.

In Table I it is possible to infer about the average time consumption for all languages of each algorithm. While the exact counter and the approximate counter are close in terms of execution time, the space-saving algorithm is much more time inefficient than the previous two. This could be explained due to the fact that more calculations are performed than in the previous ones, since every time a new letter, which does not belong to the existing set of saved letters, is found or the limit size of saved letters is reached, then a set of operations to swap that letter in the less counted one is performed, which increases significantly the execution time for this algorithm. The *Metwally et al. algorithm* is a space-saving algorithm, meaning that it focuses on saving memory rather than having time efficiency.

TABLE I
Average time consumption for all algorithms

Algorithm	Average time consumption
Space-saving ($k = 11$)	0.29003 secs
Exact Counter	0.06853 secs
Approximate counter	0.04141 secs

`Fig.3` allows a better analysis the difference in computational times for the space-saving algorithm and the approximate counter algorithm, by comparing the execution time for different values of k. As mentioned before, since the space-saving algorithm focuses on not occupying too much memory, its time consumption and results depend on the value of k.

It is possible to observe that since for less values of k, the number of stored letters is also lower, then the probability of the letter being iterated not belonging to the set of k most frequent letters at that point of iteration is higher, and therefore, the probability of performing more computations to replace the letter in the set and count it is also higher, which therefore, increases the time consumption.

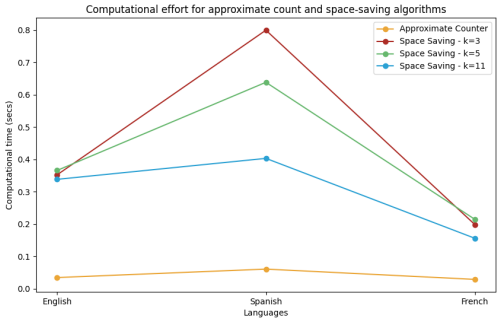

Fig. 3 - Comparison of time efficiency for approximate count and space-saving algorithms. The tests of computational time for space-saving algorithm was performed with k=3, k=5 and k=11.

To sum up, it is expected that with lower values of k, the time consumption is expected to be greater for the space-saving algorithm.

TABLE II allows to visualize the average time consumption for the space-saving algorithm for different k values, in comparison to the approximate counter algorithm.

TABLE II
AVERAGE TIME CONSUMPTION FOR ALL ALGORITHMS

Algorithm	Average time consumption
Space-saving ($k = 11$)	0.29003 secs
Space-saving ($k = 3$)	0.33627 secs
Space-saving ($k = 5$)	0.36477 secs
Approximate counter	0.04141 secs

V. APPROXIMATE COUNT RESULTS

The approximate counter algorithm is an algorithm which depends on a randomized variable. Therefore, every result in each iteration is different, and the counter for the letters detected are different each time the algorithm is executed. Therefore, it is expected that the results for the most frequent letters also change each time.

In order to identify if the approximate counter is mostly reliable, a series of trials were conducted in which the algorithm is executed several times and the results of the most frequent letters were saved. For the conducted experiments, the algorithm was run 100 times, and each result was compared to the exact counter's result.

The following statistics were calculated:

- **Number of events:** the number of total letters of the literacy works detected by the exact counter. This serves as a reference to indicate the total number of letters for each language.
- **Mean:** the average number of counted elements counted by the approximate counter. It is calculated through the division of the sum of the number of elements for each trial, divided by the number of trials:

$$\mu(X) = \frac{1}{n}\sum_{i=1}^{n} x_i$$

  in which n is the number of trials and x_i is the number of elements of the i iteration.
- **Mean absolute deviation:** the average absolute deviation of the elements generated by the absolute counter, given by:

$$mad(X) = \frac{1}{n}\sum_{i=1}^{n} |x_i - \mu|$$

  in which μ is the mean calculated previously.
- **Standard deviation:** the average absolute deviation of the elements generated by the absolute counter, given by:

$$stddev(X) = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2}$$

  in which μ is the mean calculated previously.
- **Maximal deviation:** the maximum value of the element which has the greatest difference between the average. It corresponds to the most further away value from the mean, that is, in this case, the trial iteration in which were counted the lowest number of elements. It is given by:

$$maxdev(X) = max\{|x_i - \mu|, i = 1, 2, ..., n\}$$

  in which μ is the mean calculated previously.
- **Probability:** corresponds to the division of the mean value with the number of elements counted by the exact counter, multiplied by 100.

The absolute and relative errors were also calculated accordingly, detecting the mean, minimum and maximum errors for both.

In TABLE III it is possible to analyse these values for all languages tested.

TABLE III
ANALYSIS OF STATISTICS AND ERROR FOR APPROXIMATE COUNT ALGORITHM

	English	**Spanish**	**French**
Numb. Events	352391	612945	238427
Mean	44049	76617	29818
Mean Abs. deviation	163.06	239.45	144.91
Std. deviation	209.79	299.18	181.45
Max. deviation	699.19	943.85	471.81
Probability	12.5%	12.5%	12.51%
Mean Abs. Error	308342	536328	208609
Mean Rel. Error	22.74762	22.74687	22.74879
Min. Abs. Error	307908	535668	208137
Min. Rel. Error	22.65852	22.65415	22.57944
Max. Abs. Error	309041	537272	209021
Max. Rel. Error	22.87017	22.80972	22.87392

Through its analysis, it is possible to infer that the spanish literacy work is the one with the most deviation, which is to be expected since the number of elements to be counted is also greater, with french being the language with less deviation from the average values.

However, in terms of errors, the three literacy works have similar values when comparing the relative errors, with the absolute errors reflecting the number of letters each literacy work, that is, works with greater number of characters to be counter have a greater absolute error.

In order to test the accuracy of the results, these were compared to the precise results of the exact counter. Each result obtained in the trials was compared to the result obtain in the exact counter in terms of order and number of matching most frequent letters present, that

is, the items in which the order matched completely and the items that were present in both results, regardless of order. Additionally, the percentage of matching items in the same order as the exact counter was also considered, that is, the number of items that have 25%, 50% and 75% items which match the same order as the exact counter.

These results are available in Table IV, which separates results by k values, that is, the k most frequent letters, and language of the literacy work. *"Matching items"* corresponds to the percentage of items in which all elements obtain by the approximate counter algorithm are the same as the exact counter's result, regardless or order, *"Order matching"* are the percentage of items in which the exact order of the most frequent items is the same as the order of the most frequent items for the exact counter. *"x% matching"* corresponds to the percentage of items that have matching x% items in the same order as the exact counter.

TABLE IV
Approximate count most frequent letters matching

		English	**Spanish**	**French**
$k = 3$	Order matching	92.0%	100.0%	83.0%
	Matching items	92.0%	100.0%	83.0%
	25% matching	100.0%	100.0%	100.0%
	50% matching	100.0%	100.0%	100.0%
	75% matching	92.0%	100.0%	83.0%
$k = 5$	Order matching	51.0%	100.0%	45.0%
	Matching items	83.0%	100.0%	100.0%
	25% matching	100.0%	100.0%	100.0%
	50% matching	98.0%	100.0%	90.0%
	75% matching	61.0%	100.0%	45.0%
$k = 11$	Order matching	29.0%	53.0%	20.0%
	Matching items	100.0%	100.0%	97.0%
	25% matching	100.0%	100.0%	100.0%
	50% matching	100.0%	100.0%	97.0%
	75% matching	76.0%	98.0%	61.0%

Through the analysis of the presented results, it possible to infer that the approximate counter algorithm is mostly reliable in finding the most k frequent items. It is also possible to denote that for the literacy work with most letters, in this case, the spanish one, the algorithm is even more reliable. It is also possible to denote that for higher values of k, some of the accuracy of the algorithm is lost.

Therefore, through these results it is possible to conclude that lower k values and higher number of elements to be count contribute significantly to augmenting the reliability of this algorithm, which is logical, since a higher number of elements consist on a greater number of elements counted by the approximate counter and a lower k value consists on more elements in which the lowest ones from the k most frequent elements are more close in terms of elements counted, and therefore, more viable to errors in its order. This is also proved by the results in table IV, that show that even though there is a decrease on the order matching reliability, the present k most frequent letters are the correct ones, even though they appear in the wrong order.

VI. Space-saving algorithm results

The space-saving algorithm is an algorithm that tries to find the k most frequent items, and does not accept at any point of its execution a set with size greater than k. It constantly exchanges the lowest counter for the element being iterated outside of k.

In order to analyze its efficiency and accuracy, the results for different k values were saved and compared to the ones obtained by the exact counter. Percentage of matching items and percentage of items in the same order comparing to the exact counter were registered in table IV as *"Correct items"* and *"Correct order"*.

TABLE V
Space-saving most frequent letters matching

		English	**Spanish**	**French**
$k = 3$	Correct items	33.33%	33.33%	33.33%
	Correct order	0.0%	0.0%	0.0%
$k = 5$	Correct items	60.0%	60.0%	20.0%
	Correct order	20.0%	20.0%	0.0%
$k = 11$	Correct items	72.73%	72.73%	72.73%
	Correct order	18.18%	27.27%	36.36%

By analysing these results, it is possible to retain some conclusions about the *Metwally et al.* algorithm efficiency. For higher k values, the number of correct items increases, that is, the accuracy of the k most frequent letters increases. However, for higher k values, the correct order of items seems to not have much difference from lower k values, which is logical since the space-saving algorithm focuses on obtained the k most frequent elements only, regardless of the order.

The greater accuracy in finding the k most frequent letters regardless of order for higher k values is also explained through the fact that since there exists a greater size to store the most frequent elements, the chance of having to perform exchanges to the lower counter in the existing set is less, and therefore, the algorithm mostly enters the condition of the iterated element is already present on the set, and therefore, the only operation made is in its counter, which is increased.

Therefore, this algorithm has its most efficiency when the purpose of its use it's finding the k most frequent letters in a datastream, without needing much space, and the order of the items not being relevant.

VII. Result Discussion

Through the testings made above, it is possible to infer that the approximate count works better for big datastreams, in which the counter counts more items and therefore has less chance to fail and give worse results.

However, for the space-saving algorithm, the performance of the counter depends heavily on k variable, since for greater k values, the most k frequent letters, regardless of order, are calculated much more reliably. Additionally, the space-saving algorithm is much more reliable in cases where the computational power is great, but the memory is limited, since it has a greater time execution compared to the approximate counter, but doesn't need to store all elements it detects, on the contrary of what happens in the approximate counter, which stores all elements it detects, as well as its respective counters.

Therefore, depending on the problem intending to be solved, one method could perform better than the other. Although the approximate counter performs correctly, its time consumption is dependent on the size of the datastream being analyzed.

VIII. Conclusion

Although some conclusions could be drawn from the performed experiments, further testing with other literacy works should be performed to further prove the reached conclusions. Additionally, more extensive datastreams as well as counting a greater number of elements should be considered. For example, not counting letters but trying to count certain letter combinations or even words.

These should be considered since the use of greater datasets and counting more events, not limited to the number of letters available, is helpful in identifying the cases where one algorithm performs better than the other. For example, in cases where the number of events is too extensive, the space-saving algorithm could be a reliable method of finding the k most frequent elements, even though in the performed tests it had an overall dissatisfying performance when comparing to the approximate counter, and even the exact counter as well.

References

[1] James Schloss, "Approximate counting algorithm", *Approximate Counting Algorithm · Arcane Algorithm Archive*. **URL:** `https://www.algorithm-archive.org/contents/approximate_counting/approximate_counting.html`

[2] Nuno Homem and Joao Paulo Carvalho, "Estimating top-k destinations in data streams", pp. 290–299, 2010.

[3] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi, "Efficient computation of frequent and top-k elements in data streams", pp. 398–412, 2005.