Project 3: Frequent Items Counting
A Study on Memory-Efficient Algorithms

Filipe Pires [85122] & João Alegria [85048]

Advanced Algorithms
Department of Electronics, Telecommunications and Informatics
University of Aveiro

Abstract **– Determining the most frequent items on a data stream has many applications and is a hot topic on the research community of the field. The challenges inherent of data stream processing in a memory efficient way are very much worth exploring and some of the existing solutions already provide with great optimization strategies.**

In this report, we focus on one of the most famous approximate counters to determine an estimation of the most frequent words of literary works from several authors in several languages and compare it to an exact counter. We also present a few conclusions drawn from the study applied to the dataset.

Keywords **– Probabilistic Counter, Count-Min Sketch, Memory-Efficient Algorithms**

I. Problem Contextualization

When dealing with large datasets there are many operations that require the calculation of properties of each element. The presence of frequent events throughout the data might be a potential aspect worth exploring. However, counting problems present obstacles when dealt with in very large contexts, and the difficulties are undoubtedly accentuated when the contexts morph into data streams. As continuous sources of real time events, data streams present big challenges when processing each event in a precise and reliable way. Many solutions have appeared throughout the years as means of dealing with these challenges by creating reliable approximations that, without using too many resources, still permit inferring the same conclusions that one is in search of. These solutions are typically based on probabilistic approaches that do not compromise the results but add an error factor.

The previous works of both authors of this paper address the application of approximate counters with fixed or descending probabilities of updating counts on the context of word counting in literary works [*1*] [*2*]. This time, the same literary works will be used to estimate the most frequent words, with the text corpus considered as data streams with the help of a different and more complex strategy to reduce resource usage. This report was written for the course of 'Advanced Algorithms', taught by professor Joaquim Madeira for the master's in Informatics Engineering at DETI UA and it describes the work done for the course's 3rd assignment [*3*]. We chose the hypothesis "Hipótese A-2 – Contagem dos Itens Mais Frequentes" as we were interested in knowing how low could memory usage go without compromising the outcome, i.e. how much memory could be saved while ensuring that the code still accomplished its purpose and returned approximations not too distant from the exact counters' values.

We create data streams to simulate a real world scenario and test the effectiveness and performance of the algorithm Count-Min Sketch. Other available alternatives were: Frequent-Count, proposed by Misra & Gries; Lossy-Count, by Manku & Motwani; Space-Saving-Count, by Metwally. We also discuss the differences and similarities between book translations.

II. Dataset

The dataset used, as already mentioned in chapter I, is based on literary works and in some of their translations. All of the books were obtained from *Project Gutenberg* [*4*]. The chosen books and respective languages are presented below:

- *Alice in the Wonderland*, by Lewis Carol - written in English, German, French and Italian
- *A Christmas Carol*, by Charles Dickens - written in English, Finnish, German, Dutch and French
- *King Solomon's Mines*, by H. Rider Haggard - written in English, Finnish and Portuguese
- *Oliver Twist*, by Charles Dickens - written in English, French and German
- *The Adventures of Tom Sawyer*, by Mark Twain - written in English, Finnish, German and Catalan

Since provident from the *Project Gutenberg*, the books come with headers and annotations that identify authors, sources and additional metadata that was not relevant for our study; with this in mind, we preprocessed the various book files to remove the metadata that could cause unwanted variation on results. This process was quite simple and was only necessary the usage of a text editor and the manipulation of some regular expressions.

III. Frequent Word Identification

Identifying the most frequent words in a text stream was achieved using two very different strategies. On the first one, the program keeps record of all words that appear in the information stream along with an exact count of each word - this allows it to determine at any time which words are the most frequent. The second, on the other hand, is a far more memory-friendly solution that uses hash functions to avoid storing the pair $(word, counter)$. In this chapter we discuss the implementation of both solutions for the use case of identifying the most frequent words on literary works.

A. Exact Counter

The exact counter is the gold standard when creating alternative solutions that save time, memory or other resource at the cost of precision. It is by the values computed by it that the other counters are evaluated.

Its implementation consists of a regular dictionary of words as keys and counts as values, constantly updated as new words appear on the text stream. There are conditions, however, to update this dictionary. The incoming word must contain at least two characters and cannot belong to a list of stop-words defined *a priori*. This is done to prevent considering words that are irrelevant, i.e. do not offer any practical value to the study such as articles or prepositions.

Stop-words are managed by an external library [*5*], capable of dealing with all languages of our dataset.

B. Count-Min Sketch

The Count–Min Sketch (CM Sketch) is a probabilistic data structure that serves as a frequency table of events in a data stream. This means it works essentially like Bloom-Filters (BFs). However, they are used differently and therefore sized differently: a CM Sketch typically has less entries than the total number of different events (or words in our case) of the stream, related to the desired approximation quality, while a counting BF is sized to match the total number of different events.

The solution uses hash functions to map word occurrences to frequencies, but saves memory space at the expense of over-counting some words due to collisions. The effect of these phenomena is addressed ahead.

The implementation of our CM Sketch was taken from the resources made available by the course's professor. As it served perfectly for our purposes, we found it unreasonable to reach our own implementation at the risk of adding imperfections that could compromise our study. Nevertheless, we offer a description of how it works and how we introduced it to the use case.

If the same conditions defined in the Exact Counter are verified for the current word on the text stream, the word is passed to the *update()* method of the `CountMinSketch` class. This method then applies all its hash functions to the word and increments the values on the hash tables on the respective entries.

As the word counts either have the exact count or suffer from over-counting, the value considered as more trustworthy is the one whose result from hashing and updating returned the smallest value - the one with less occurrences of collisions - (hence the name count-MIN sketch). It is this value that is returned when the class instance is queried.

`CountMinSketch` receives two regulatory parameters that allow us to control the overestimation factor. These are `m`, the size of the hash tables, and `d`, the number of hash tables. The first regulates how significant is the overestimation since only the smallest value is returned by *query()* so the more tables there are, the smallest will be the difference between the real count and the one returned. The second reduces the probability of collisions since more hash tables means more complex hashes less likely to be equal. Two alternative parameters could have been used, `delta` and `epsilon`, that regulate the probability of query error and the query error factor and internally define `m` and `d` but in our case we sticked with the first two.

In relations to the hash functions used, they have the same basis and over that a differencing operation is perform to achieve as many hash functions asked from the user. The Python library `hashlib` [*?*] is used to generate secure hashes and digests, being the most used features its MD5 function and a hexadecimal digest. The dataflow of the hashing process goes as follows: firstly a base is created by generating a MD5 digest over the hash of the provided string; then for each one of the hash functions needed, the MD5 digest is updated with the successive indicators of the hash functions(from 0 to N) until the wanted function is reached, then the hexadecimal digest is created, converted to an integer and applied the modulus based on the size of the hash tables.

In this process the successive addition of the hash function indicators is the portion that enables the creation of several hash function, since by starting from a base hash and adding always the same N numbers, it is equivalent of having N different hash function. The portion of code that calculates the hexadecimal digest and executes the modulus based on the size of the hash tables is the code that certifies that a valid index is always generated, since assuming that the hash tables have a size of K, the modulus of any number by K will return only values in the range [0, K-1].

IV. Literary Study

As a form of automation and to enable us to quickly study the behavior of the chosen algorithm, we created a script in Python that allows us to change some of the necessary parameters and run the solution for a given input stream in a straightforward, command-based manner. The signature of our script is as follows:

```
python3 frequentWordFinder.py [-h]
[-o outputFolder] [-d numHash]
[-m numColumns] <inputFolder>
```

The *outputFolder* is a parameter that indicates the folder where the resulting files should be stored (the default folder is called "out" and is generated

if it does not exist), *numHash* and *numColumns* correspond to the number of hashes and the number of columns the user wants to be used when creating the `CountMinSketch`, explained in chapterIII; these 3 parameters are optional and they all have default values (outputFolder:"out", numHash:1000, numColumns:5). The *inputFolder* is a mandatory parameter since it's the folder where the source data should be present; this data is assumed to be preprocessed, since that will generate the best results, without any noise form the headers and annotations mentioned in chapter II.

What we did to study the performance of CM Sketch was to record its results for various inputs and parameter combinations and resort to data visualization tools to understand its behavior. The tested inputs were all of the literary works from our dataset, and the tested parameters were combinations of `numColumns` within the range [200,4000] and `numHash` within the range [1,10].

$$E_r = \frac{E}{X} \times 100 \quad (1)$$

The exact count is considered to compute both the error of the approximate count and its relative error. The error (E) of an estimation for a word is defined as the difference between its exact count (X) in the text stream and the count given by the Sketch. The relative error (E_r) corresponds to the error percentage given by equation 1.

V. Results & Discussion

The first analysis we did was on the words considered most frequent by CM Sketch and how accurate were its count estimations. This is possible to do by reading the bar charts present in figures 3 and 4. These bar charts present the counts of both solutions in descending order of the top 20 words's exact counts on the data stream; they allow to see which `CountMinSketch` configurations offer reasonably good results. .....

Figures 1 and 2, on the other hand, present two line charts that follow the evolution of the average error associated to the Sketch counts according to the parameters *numHash* and *numColumns*, respectively. .....

In table I, we present an output example for the book *Alice in the Wonderland*, by Lewis Carol, written in English. Here, CM Sketch was instantiated with a hash table of 2000 columns and 5 hash functions. .....

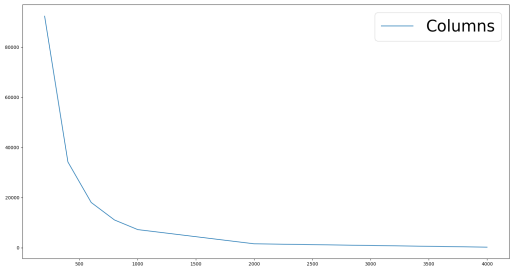


Fig. 1: Column Study with English Alice.

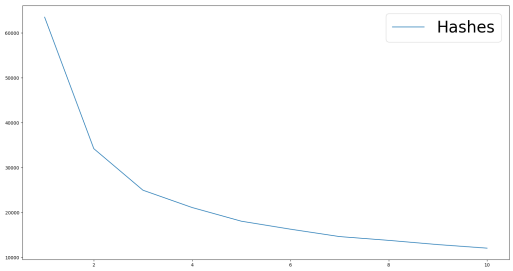


Fig. 2: Hashes Study with English Alice.

Words	All	Sketch	SA Abs Err	SA Rel Err (%)
said	462	462	0	0
alice	397	397	0	0
little	128	128	0	0
one	104	104	0	0
know	87	90	3	3.45
like	85	85	0	0
went	83	83	0	0
queen	75	75	0	0
thought	74	76	2	2.7
time	71	71	0	0
see	67	67	0	0
well	63	64	1	1.59
can	63	63	0	0
king	63	63	0	0
don	61	61	0	0
now	60	60	0	0
turtle	59	59	0	0
began	58	59	1	1.72
ll	57	57	0	0
way	56	56	0	0

TABLE I: English Alice Results(2000 cols,5 hash)

Key	All
said	462
alice	397
little	128
one	104
know	87
like	85
went	83
queen	75
thought	74
time	71
see	67
can	63
king	63
well	63
don	61
now	60
turtle	59
began	58
ll	57
mock	56

(a) Exact Counts

Key	Sketch
said	473
alice	414
little	154
like	127
one	125
can	124
mock	117
know	115
running	109
went	107
go	105
see	101
don	97
began	95
queen	93
thought	93
savage	93
much	92
king	91
first	91

(b) Sketch Counts

TABLE II: Sketch Comparison

VI. Conclusion

Reducing memory usage while maintaining accuracy on determining the most frequent words on a text stream is not a trivial task. Hash functions offer a great benefit by allowing solutions to not have to store the words being counted, but they bring the problem of different words returning the same hash, resulting in counting collisions. Managing this brings complexity to any solution that takes advantage of such memory saving strategy.

Count-Min Sketch proved to be able to determine the most frequent words with a very high precision with much less memory usage than an exact counter. And, by studying the evolution of the algorithm's behavior for different configurations, we were able to achieve a balanced solution for the dataset available.

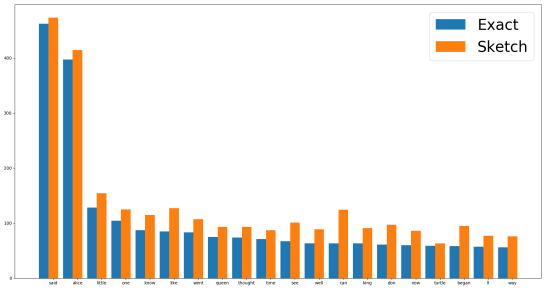
For future work, we believe it would be a good idea to implement other algorithms such as the ones presented in chapter I so that a deeper analysis and comparison between these solutions could be achieved.

Our overall perspective on the project is very positive. We found that hash tables were very practical and adequate to problems such as this, and, even though we did not formally compare the performances of the algorithm here presented and the ones from our previous works, we were still able to conclude that Count-Min Sketch is far more promising than the regular approximate counters based on a given probability of updating counts. As we were both already introduced to the problem when the project began, we were able to properly segment the work to be done very rapidly and kept a good balance and work distribution between group elements throughout the development.

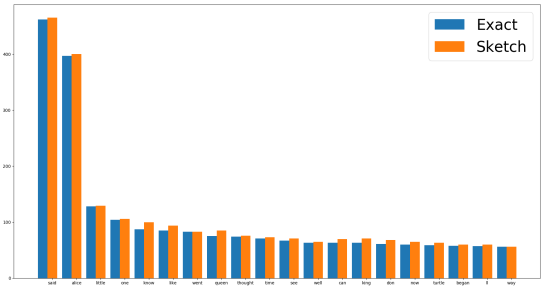
References

1. João Alegria, "Literary work word approximate occurrence counters".
2. Filipe Pires, "Project 2: Word counting in large texts".
3. Joaquim Madeira, "3o trabalho - algoritmos avançados".
4. Michael Hart, "Free ebooks - project gutenberg", `https://www.gutenberg.org/`.
5. Python SW Foundation, "Pypi - stop-words", `https://pypi.org/project/stop-words/`.

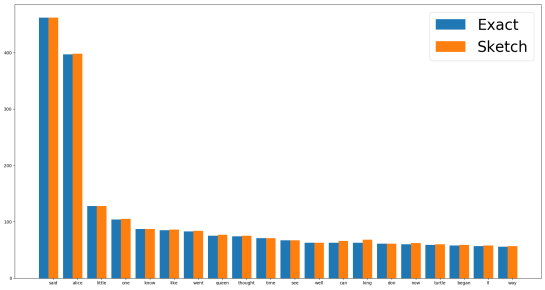
Appendix



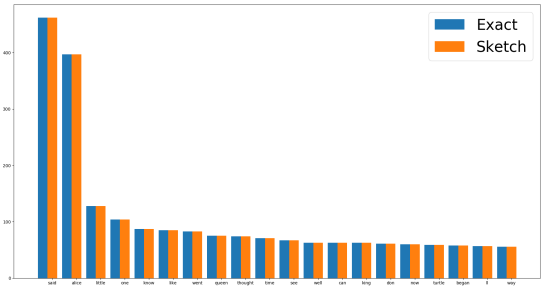
(a) 200 columns, 5 hashes



(b) 600 columns, 5 hashes

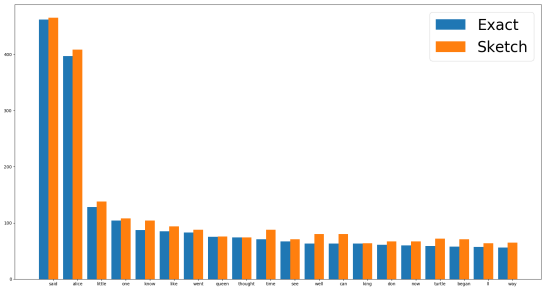


(c) 1000 columns, 5 hashes

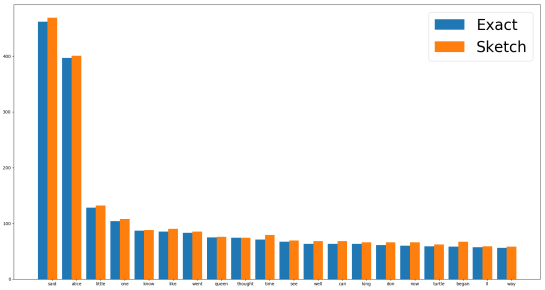


(d) 4000 columns, 5 hashes

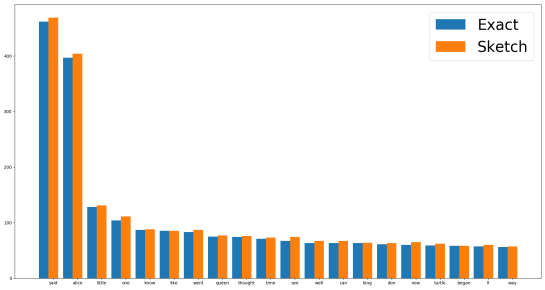
Fig. 3: Column Study Comparison



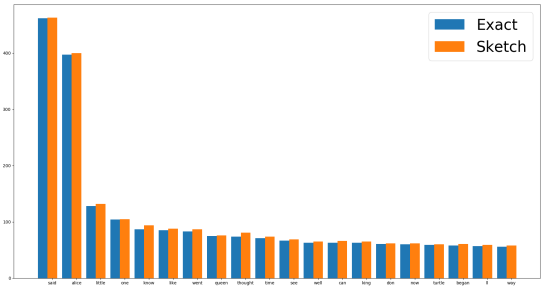
(a) 600 columns, 2 hashes



(b) 600 columns, 4 hashes



(c) 600 columns, 6 hashes



(d) 600 columns, 10 hashes

Fig. 4: Hash Study Comparison