Finding the K-most Frequent Words in a Large Data Stream using Space Saving Counters

Tomás O. Costa <89016>, tomascosta@ua.pt

***Abstract* - This article's main objective is to give an approximate to the most frequent words in a book or text acting as a data stream, using the algorithm for space saving counters, which as the name indicates, allows for memory space usage being reduced.**

I. INTRODUCTION

The problem of finding the K most frequent words is a very frequent problem and there's multiple approaches to it. On this work we focus on large data streams as input and that changes the paradigm of the problem since processing must be done on the run and not after the data has been processed.

II. DESCRIPTION OF THE PROBLEM

Since the input is a very large data stream of text, previously tokenized in this work, for better word processing (including parsing and stopwords), we must process these tokens and update the counters while reading from the data stream.

In this approach we cover both the exact counters and the space saving counters for a measure of comparison between the two.

III. SPACE SAVING ALGORITHM

The Space Saving algorithm is very simple and straightforward, and its pseudocode is as follows:

```
n ← 0;
T ← ∅;
foreach i do
    n ← n + 1;
    if i ∈ T then c_i ← c_i + 1;
    else if |T| < k then
        T ← T ∪ [i];
        c_i ← 1;
    else
        j ← arg min_{j∈T} c_j;
        c_i ← c_j + 1;
        T ← T ∪ [i] \ [j];
```

Fig. 1 – Space saving with k counters

The code represented above is for a given k, this derives from the following formula:

$$k = 1 / \varepsilon$$

This ε must be manually inserted in the loop and it's the only parameter that is subject to change in this analysis. Fine tuning of this parameter is done further down the document.

After inputing our ε, the program will have created k counters or 1 / ε counters, we then create a dictionary with size k that will contain the most frequent word and their respective counters.

As long as there is space to fill in the dictionary, the token will be added and the counter will be incremented to 1. If the word has already been added to the dictionary, we shall increment its counter by 1 and move to the next counter.

As for the last case of the word not being in the dictionary and there is no more space left on our k-sized dictionary, we will evaluate the words stored and their counters and remove the word with the least increments, but, instead of discarding its counter, we take that value and assign it to the new word that is going in the dictionary. To finalize we increment that previous word value by 1.

With the way we were processing this algorithm, there's some claims we can make:

- The counter values sum is equal to n items processed.
- Average value for a given word is n / k, which is equal to ε x n.
- The smallest counter cannot be larger than ε x n.
- The true count of an uncounted item is somewhere between 0 and ε x n.
- All items whose true count is bigger than ε x n are stored correctly in the array.
- If ε is small enough that k equals n, the countings are exact countings.

These claims are further discussed by Metwally et al. [2] and M. Mitzenmacher [1].

That way, we can ensure that by keeping only k items and counters in hand, we can process very large datasets on the go without much usage of resources or processing. And for a fine tuned ε, the results achieved are very good.

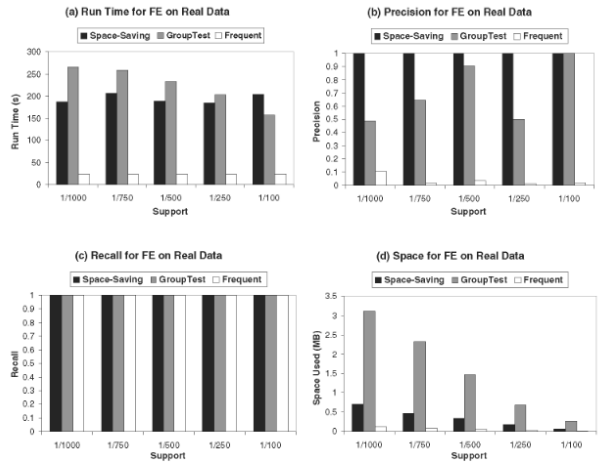

Fig. 2 - Performance comparison for the frequent elements problem using a real click stream. Metwally et al. [2]

IV. RESULTS AND ANALYSIS

A. *The testing dataset*

For the testing dataset, we need something to emulate a very large textual data stream, and therefore several books were parsed and tested. Since books were relatively small on the scale of testing done by Metwally [2], we decided to compile several texts into one big file and use it as a main testing tool. These books can be found under the texts folder in the project's GitHub [4].

B. *Method of evaluation*

To evaluate the algorithm a simple accuracy measure was created. This measure takes into the account the k most frequent words in a given text, using the exact counter and then compares it with the k most frequent words from the space saving counting.

accuracy % = exact_words ∩ retrieved_words / exact_words * 100

The results obtained using this measure were very good as long as we were using a low enough ε. They would vary from around 20% to 100% depending on the epsilon.

```
# Words
Exact Count:              7001
Space Saving Count:       7001

# Distinct Words
Exact Distinct Count:             2874
Space saving Distinct Count:      500

Accuracy of the 500 most frequent words is: 41.20
```

Fig. 3 - Evaluation of german2.txt with ε=0.002

Further analysis on how the decrement of epsilon improves results is down below.

C. *Multiple evaluation tool*

Since manually changing ε was a costly process and didn't make sense in this testing stage, I developed a simple jupyter notebook to evaluate how the change of epsilon was greatly influencing the accuracy of the algorithm.

As we can see in the image below, tests were conducted varying ε from 0.1 tending to 0 (tending but never reaching), and it was possible to evaluate that the most frequent words accuracy would be achieved on a epsilon of about 0.0003 for almost every text included in the testing folder.

In this evaluation the minimum value for epsilon used was 0.0002 which is around 5000 counters. Since the smallest text contains 7001 words, in these tests, k never reached the exact counters number, but still gave exact results.

We can also see below that the patterns in every book/language are very similar, so ε has a very similar impact regardless of the number of tokens in the data stream.

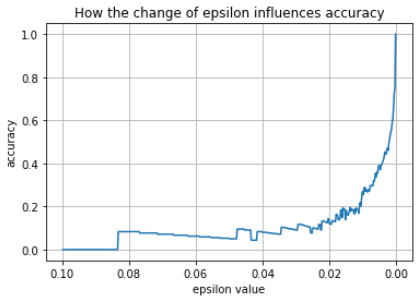

Fig. 4 - Evaluation of english1.txt

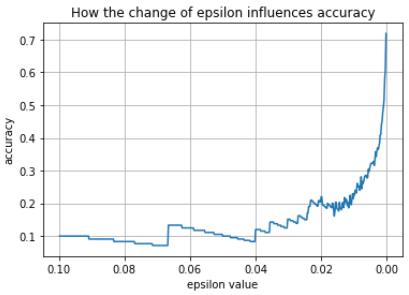

Fig. 5 - Evaluation of czech1.txt (largest data stream of the repository)

REFERENCES

[1] Hierarchical HeavyHitters with the SpaceSaving Algorithm, https://epubs.siam.org/doi/pdf/10.1137/1.9781611972924.16

[2] An Integrated Efficient Solution for Computing Frequent and Top-k Elements in Data Streams, https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.486.6006&rep=rep1&type=pdf

[3] Data Stream Algorithms I Slides, Joaquim Madeira, e-Learning 2020

[4] Space Saving Counter GitHub, Tomás Costa, https://github.com/TomasCostaK/Space_Saving_Count/