Top K Most Frequent Words in Large Data Streams with Space Saving Counters

Mário Francisco Costa Silva, nMec: 93430

***Abstract* –This main objective of this project is to find approximately the top most frequent words in a document, using the space saving algorithm, which allows doing this process while requiring less memory than the traditional approach.**

***Keywords* –Space Saving Counter, Advanced Algorithms, Memory Optimization, Large Data**

I. Introduction

The goal is to find the **K** most frequent words in text files and to use less memory than a normal approach. The traditional solution for this problem consists of using a counter for each word present on the file, and then ordering the words by their corresponding counters' value. Using this approach is not viable in terms of memory for dealing with very large files, which is the case in this article.

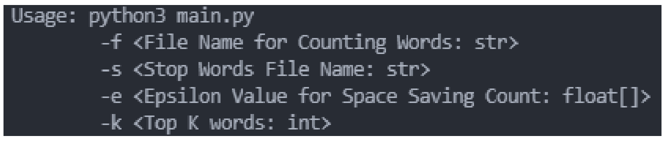
II. Problem Description

The algorithm that solves the **K** most frequent words, while iterating only once through the data stream, when this data is very large, requires to be optimized so that computers with less memory capacity can easily do it with very little error and with shorter amounts of memory.

III. Approach and Implementation

A. Parameters Management

The program allows the user to pass some arguments that have different actions. The user can set a few parameters that allow testing in a more customizable manner. It can be passed the name of the file to be used as data stream, the name of the file that contains stop words, a list of epsilon (ϵ) values and the number of most frequent words **K** to perform the counting.



```
Usage: python3 main.py
        -f <File Name for Counting Words: str>
        -s <Stop Words File Name: str>
        -e <Epsilon Value for Space Saving Count: float[]>
        -k <Top K words: int>
```

Fig. 1
Program arguments

B. Exact Counter

This is a counter with a very simple implementation. All it does is read the given file line by line, tokenizes the line, while removing punctuation marks, converting all letters to lower case, and also removing the stop words present with the given stop words file as an argument.

```
tokens=re.sub('[^a-zA-Z]+',' ', line).lower().split()
```

Reading the file line by line could be crucial since it might be necessary to handle very large files, therefore, it may not have memory to read and store the whole file.

All it does is keep a counter for each word, that corresponds to a simple *python3* dictionary.

```
for word in tokens:
  if word not in self.stop_words:
    self.word_counter[word] += 1
```

C. Space Saving Counter

The space saving counting or approximate counting algorithm allows the counting of numerous events using a small amount of memory.

Given an error rate ϵ, the algorithm uses $k = 1/\epsilon$ counters to monitor all the words of the given text file, which could be translated to a dictionary in *python3* with maximum size k, with the key being a word and the value the number of events.

This error rate ϵ must be passed as an argument, as seen previously, and it is a subject of analysis and fine-tuning.

The implementation is also rather simple, if the word has already a counter associated, increment its value by 1, else, if there is still less counters than the maximum number of counters k, it just creates a counter for the new word, else, replaces the word that has its associated counter with the smallest value (min) out of all counters, and then sets the frequency of the new word to the previous counter value plus one (min+1).

```
for word in tokens:
  if word in self.stop_words:
    continue

  if word in self.word_counter:
    self.word_counter[word] += 1
    continue

  if len(self.word_counter) < self.k:
    self.word_counter[word] = 1
     continue

  min_word =
    min(self.word_counter, key=self.word_counter.get)
  self.word_counter[word] =
    self.word_counter.pop(min_word) + 1
```

According to the algorithm, all elements with $f \geq [\epsilon * S]$, where S is the number of distinct words, are guaranteed to be reported. This means, for example, for $S = 100$ and $\epsilon = 0.05$, we can calculate the minimum frequency, ϕ:

$$\phi = \quad (1)$$
$$= \epsilon * S = \quad (2)$$
$$= 0.05 * 100 = \quad (3)$$
$$= 5 \quad (4)$$

All words with frequency equal or greater than $\phi = 5$, or, in other words, that appear at least 5 times, are guaranteed to have a counter associated (word will be present in the *python3* dictionary).

Other claims can be made using the space saving algorithm [2], such as:

- The sum of the counter values is equal to S items processed.
- Average value for a given word is $\epsilon * S$.
- The smallest counter cannot be larger than $\epsilon * S$.
- The true count of an uncounted item is somewhere between 0 and $\epsilon * S$.
- If ϵ is small enough that $k \geq S$, the countings are exact countings.

D. Results

For the testing part, it was used the text file of the Bible in an English version obtained through the Gutenberg project [1].

E. Results for the Exact Counter

As it can be seen on the table I below, the program shows some measures while performing the counting, such as the total execution time, the number of distinct words, the total number of events (countings) and then the mean, min, and maximum values. These values and the top **K** most frequent words of this exact counter are stored to perform the analysis of the space saving algorithm results.

Measure	Value
Time	0.331
Total Words	11694
Events	344985
Mean	29.501
Minimum	1
Maximum	9454

TABLE I
Exact Counter Results

F. Results for the Space Saving Counter

For this algorithm it was used three different **K** values, $[100, 200, 500]$, and, 5 different ϵ values, $[0.0002, 0.0005, 0.0008, 0.001, 0.002]$.

After performing the counting, the same metrics as the exact counter are calculated, and also, another 3 metrics, accuracy, precision, and average precision.

The accuracy can be calculated by the number of words that were retrieved by the algorithm that are also in the exact words top **K** words, True Positives, and adding that to the True Negatives, the number of words that weren't retrieved and also aren't in the top exact words, and finally dividing that by the total number of distinct words.

The precision is calculated by the number of True Positives dividing by the length of the top words, which is **K**. Recall isn't used because it's equal to the precision in this case, since it is retrieved the same amount of words **K**, therefore the number of False Negatives is equal to the False Positive's.

Finally, the average precision, performs a cumulative precision, but takes into consideration the order of top words relative to the order of the exact words, so it iterates over the exact top words and checks if it is on the same position as the top words retrieved, and then it averages this cumulative precision.

```
right_pos_words, rel_precision = 0, 0
for i, word in enumerate(self.exact_top_k_words):
  if word == top_words[i]:
    right_pos_words += 1
    rel_precision += right_pos_words/(i + 1)
average_relative_precision = rel_precision/self.k * 100
```

F.1 Top 100 Words

For **K** equal to 100, it can be observed from the table II, that the total distinct words, which is the number of counters used, calculated by $k = 1/\epsilon$, increases as ϵ decreases.

The number of events (total words processed), which is calculated by the sum of all counters, are the same on all ϵ values, and also, equal to the exact counter. The same happens with the maximum counter value, because it never gets replaced by another word.

Also, the minimum and mean tend to decrease as ϵ decreases, this because the counters are now able to use more counters, therefore, they will end up storing smaller counters for each word.

Measure	ϵ 0.002	ϵ 0.001	ϵ 0.0008	ϵ 0.0005	ϵ 0.0002
Time	1.32	1.55	1.73	1.82	1.76
Total Words	500	1000	1250	2000	5000
Events	344985	344985	344985	344985	344985
Mean	689.97	344.985	275.988	172.493	68.997
Minimum	487	189	135	63	10
Maximum	9454	9454	9454	9454	9454
Accuracy	99.86	100	100	100	100
Precision	92	100	100	100	100
Avg. Precision	32.31	62.8	78.13	93.2	100

TABLE II
Space Saving Counter Results for **K** = 100

From the 2, it can also be observed the execution time of the program, it can be observed that as ϵ decreases, the execution time also decreases, this is because it processes the same amount of words, but it does a lot less substitutions operations (uses more counters), that require deleting the word from the dictionary and then adding a new one.

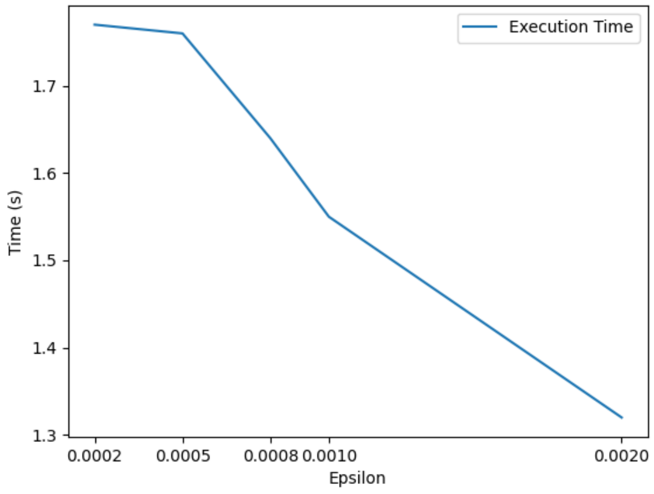


Fig. 2
SPACE SAVING COUNTER EXECUTION TIME FOR **K** = 100

Figure 3 shows a graph with the values of accuracy, precision, and average precision that helps visualize that as ϵ increases, the top **K** gets "worse", especially the average precision metric, even though, the accuracy and precision remain very high.

Accuracy in this case can be very deceiving because it depends on the difference between the chosen **K** and the total number of distinct words, and, for the chosen **K** values, the difference is very big, therefore accuracy will always be very high.

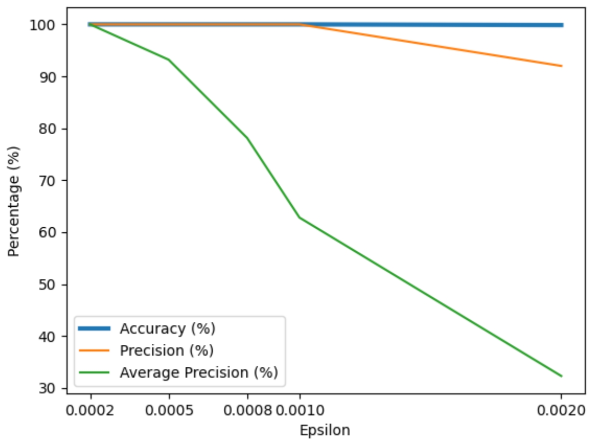


Fig. 3
SPACE SAVING COUNTER ANALYSIS OF TOP **K** = 100

F.2 Top 200 Words

With **K** equal to 200, it can be observed, the overall results get slightly worse, and only the smallest ϵ chosen, 0.0002, maintains the 100% precision.

Measure	ϵ 0.002	ϵ 0.001	ϵ 0.0008	ϵ 0.0005	ϵ 0.0002
Accuracy	99.08	99.85	99.95	99.98	100
Precision	73	95.5	98.5	99.5	100
Avg. Precision	16.16	35.04	49.95	78.77	97.93

TABLE III
SPACE SAVING COUNTER RESULTS FOR **K** = 200

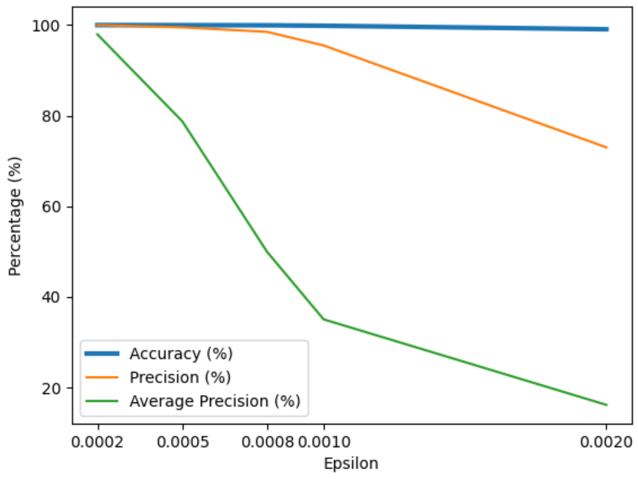


Fig. 4
SPACE SAVING COUNTER ANALYSIS OF TOP **K** = 200

F.3 Top 500 Words

Finally, for **K** equal to 500, the results get even worse than the previous.

The algorithm with the biggest ϵ, which only uses 500 counters, had a precision of 50%. It can also be concluded that maintaining the order of the top **K** is much harder than unordered, even with the smallest ϵ algorithm.

Measure	ϵ 0.002	ϵ 0.001	ϵ 0.0008	ϵ 0.0005	ϵ 0.0002
Accuracy	95.86	98.29	98.99	99.73	100
Precision	51.6	80	88.2	96.8	100
Avg. Precision	6.46	14.09	20.16	32.58	78.06

TABLE IV
SPACE SAVING COUNTER RESULTS FOR **K** = 500

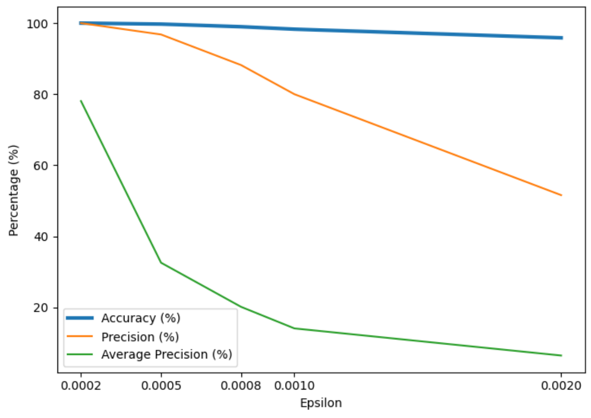


Fig. 5
SPACE SAVING COUNTER ANALYSIS OF TOP **K** = 500 WORDS

IV. Conclusion

Overall, it was tested the space saving algorithm, in order to analyze its results while varying some parameters.

From the results, it can be concluded that it is possible to get, with a decent precision and relative precision, the top **K** most frequent words while having an amount of memory controlled by adjusting the error rate used.

References

[1] Project Gutenberg. (2022, January 21). The King James Version of the Bible.
`https://www.gutenberg.org/ebooks/10`

[2] Mitzenmacher, M., Steinke, T., Thaler, J. (2012). Hierarchical Heavy Hitters with the Space Saving Algorithm. 2012 Proceedings of the Fourteenth Workshop on Algorithm Engineering and Experiments (ALENEX), 160–174.
`https://doi.org/10.1137/1.9781611972924.16`