Most Frequent and Less Frequent Words

Pedro Ponte
Licenciatura em Engenharia Informática

Abstract **– This project presents an analysis of three different approaches for word frequency counting in text processing: exact counting, probabilistic counting with decreasing probability ($\frac{1}{\sqrt{2^k}}$), and streaming based lossy counting. The methods were evaluated using various texts, including different language versions of "Os Lusíadas" and "Don Quixote," as well as large random text files up to 1GB in size. Performance metrics included accuracy, memory usage, and execution time. Results show that while the exact counter provides perfect accuracy, it faces significant memory constraints with large files. The probabilistic counter achieves 93-96% accuracy with 40+ runs, offering a balance between precision and resource usage. The streaming algorithm demonstrates superior memory efficiency, using only 1108kB compared to 4952kB for exact counting on "Don Quixote," while maintaining 93% accuracy with $\epsilon = 0.001$. It was the only method capable of processing the 1GB file, completing it in 8.43 seconds with a 2.5MB memory footprint.**

Keywords **–Word Count, Frequent Words, Exact Count, Approximate Count, Streaming Counter, Accuracy, Memory**

I. Introduction

The main goal of this project is to count the number of words and determine the most/less frequent words in text files, in this case books obtained from *Project Gutenberg* [1].
To do this we'll use three different approaches (explained in II):

1. Exact Counters
2. Approximate Counters - decreasing probability of $\frac{1}{\sqrt{2^k}}$
3. Streaming Counters - Lossy Counter

The analyzed books for this project will be:

- Lusíadas - Portuguese [2]/English [3]/Spanish [4]
- Don Quixote - English [5]
- Metamorphosis - English [6]
- Moby Dick - English [7]

The main focus will be on the Lusíadas book in Portuguese, because it's one of the greatest portuguese's work of art, along with Don Quixote, due to it's large word count (187737 words). All the books were processed to only include the relevant text, excluding any *Gutenberg* file headers and tails, remove punctuation, make all letters lowercase and remove stopwords in all languages.
Additionally, very large files were created to test the algorithms in large scale problems.

II. Methods

A. Exact Counter

The **Exact Counter** method is the simplest of the three approaches. It works by maintaining a dictionary where each unique word serves as a key, with its value being the number of ocurrences of said word. The algorithm processes the text word by word:

1. If the word is already in the dictionary, increment its count by 1
2. If the word is not in the dictionary, add it with an initial count of 1

While this method provides exact counts and is easy to implement, its main problem is memory usage. The time complexity is $O(n)$ where n is the number of words in the text, and the space complexity is $O(k)$ where k is the number of unique words, requiring storage for both the input text and the frequency dictionary.

B. Probabilistic Counter

The **Probabilistic Counter** [8] uses a decreasing probability approach to estimate word frequencies. For each word occurrence, the counter decides whether to increment the count based on a probability that decreases as the current count increases, following the formula $\frac{1}{\sqrt{2^k}}$ where k is the current count of the word.

1. If the word is not in the dictionary, add it with count 1 (probability of $\frac{1}{\sqrt{2^0}}$)
2. If the word is already counted k times:
   - Generate a random number $r \in [0, 1]$
   - If $r \leq \frac{1}{\sqrt{2^k}}$, increment the count

As a word appears more frequently, the chance of it being counted is lower, reducing the number of accesses to our dictionary. The time complexity remains $O(n)$, but now we need multiple runs, while the space complexity stays $O(k)$ where k is the number of unique words.

C. Streaming Counter

The **Streaming Counter** [9] follows the Lossy Counting Algorithm [10], which processes text in a

streaming process while maintaining approximate frequency counts within an error bound ϵ. The algorithm divides the input stream into buckets of width $w = \lceil \frac{1}{\epsilon} \rceil$ and processes them sequentially:

1. For each word in the stream:
   - If the word exists in the dictionary, increment its count
   - If not, add it with count 1
2. At the end of each bucket:
   - Decrement all counters by 1
   - Remove entries with count ≤ 0

This approach filters out infrequent words by periodically pruning the dictionary, as words must appear frequently enough to survive the decrements at bucket boundaries. For better accuracy we can lower the value of ϵ which would increase the bucket value and reduce the number of decrements, but this comes at the cost of memory efficiency. The time complexity is $O(n)$ where n is the number of words, and the space complexity is $O(\frac{1}{\epsilon} \log(\epsilon N))$ where N is the stream length, making it memory efficient for tracking frequent items in large streams.

III. Experimental analysis

With our methods all defined, we can go to the experimental part of the project. We'll start be running the **Exact Counter** and seeing the top words for **Don Quixote** and then compare it with the other methods, to calculate accuracy.
We'll then take **Lusíadas**, analysing the "same" book in three different languages and checking if the most frequent words match.
Finally, we'll try to use our algorithms in very large files and see the results.

A. Don Quixote

Starting with **Don Quixote** we can run our three algorithms and check the results (*exhaustive_wc.py*, *probablistic_wc.py*, *streaming_wc.py*).
These results will give us the total amount of words processed along with the number of ocurrences for every method.

A.1 Word count

As we can see (Table I, Table II, Table III), the probablistic and streaming methods achieve similar results to the exact counter, but with some variations. To better understand how precise both algorithms are we can compare the results by different metrics.

A.2 Accuracy

To check the accuracy of our methods we'll be checking for every word of the **top *N*** words of the exact count if the word also appears in the **top *N*** of the other methods.
Through these values (Table IV) we can see that although the probablistic algorithm takes more time to execute (since it needs several runs to have decent results), it is more precise than the streaming algorithm.

TABLE I: Exact Word Counter Results (Total Words: 187,177)

	Word	Count
1	said	2,624
2	quixote	2,117
3	sancho	2,089
4	one	1,571
5	would	1,246
6	thou	1,228
7	say	903
8	good	848
9	may	834
10	see	767

TABLE II: Lossy Word Counter Results ($\epsilon = 0.005$, Total Words: 187,177)

	Word	Count
1	said	1,724
2	sancho	1,201
3	quixote	1,182
4	one	637
5	thou	334
6	would	317
7	say	28
8	thee	27
9	good	24
10	niece	10

TABLE III: Probabilistic Word Counter Results (Decreasing probability: $\frac{1}{\sqrt{2}^k}$, Number of runs: 10, Total Words: 187,177)

	Word	Avg. Count
1	said	20.4
2	quixote	19.0
3	one	18.8
4	sancho	18.5
5	thou	17.5
6	would	17.4
7	say	17.1
8	good	16.8
9	come	16.6
10	time	16.4

It's also important to see how the accuracy changes depending on both the number of runs for the probablistic approach and epsilon value in the streaming one. The code for these comparisons can be found in *varying_accuracy.py*.
The probabilistic counter shows (table V) good performance improvements with increased runs. With just a single run, it achieves 73% accuracy, indicating good base performance. A significant jump occurs between 1

TABLE IV: Accuracy Comparison for Different Top-N Values

N	Probabilistic (%)	Streaming (%)
3	66.67	100.00
5	80.00	80.00
10	80.00	80.00
15	80.00	66.67
20	80.00	55.00
25	64.00	44.00
30	53.33	36.67
40	40.00	27.50
50	32.00	22.00

Runs	**Accuracy (%)**	**Exec. Time (ms)**
1	73	57.15
5	88	165.37
10	88	316.75
20	92	554.01
40	93	1089.33
60	93	1585.21
80	95	2144.64
100	96	2660.68

TABLE V: Accuracy and Execution Time of Probabilistic Counter vs Number of Runs

and 5 runs, reaching 88% accuracy. The accuracy continues to improve but with smaller values, stabilizing at around 93% with 40-100 runs. This suggests that while more runs generally improve accuracy, there may be no point in exceeding 40 runs, as the accuracy gain becomes insignificant considering the execution time.

Epsilon	**Accuracy (%)**
0.1000	0
0.0500	0
0.0100	9
0.0050	26
0.0010	93
0.0005	96

TABLE VI: Accuracy of Streaming Counter vs Epsilon Value

The streaming counter's accuracy shows how important the *epsilon* value is, as there's a clear trade-off between memory efficiency and accuracy. With larger *epsilon* values (0.1 and 0.05), the algorithm fails to identify any of the top words correctly, resulting in 0% accuracy. However, as *epsilon* decreases, accuracy improves dramatically. The most notable improvements occur between $\epsilon = 0.01$ (9%) and $\epsilon = 0.001$ (93%). This suggests that while the streaming counter requires very small epsilon values for high accuracy, it can achieve decent results when com+ared to the probabilistic approach if properly tuned. Another interesting insight into this variance of *epsilon* would be to see how the memory usage increases for lower values of *epsilon*, but due to python's cache and garbage collection features it was not possible to verify, as consecutive runs are optimized.

A.3 Memory

Our next comparison is checking a memory footprint of each method, this will be done using the package *psutil* [11], code can be checked in *memory.py*

Algorithm	**Memory Usage (kB)**
Exhaustive	4952.00
Probabilistic	3912.00
Streaming	1108.00

TABLE VII: Memory Usage Comparison for Don Quixote

As we can see, the streaming algorithm uses way less memory than the other two, making it a better choice when memory is an issue and we can afford some uncertainty.

B. Lusíadas

For the **Lusíadas** book we used our **Exact Counter** on the books of the three languages (Portuguese, English and Spanish)

Portuguese		
Rank	**Word**	**Count**
1	gente	230
2	terra	222
3	tão	210
4	rei	203
5	mar	188
6	onde	177
7	grande	138
8	ali	103
9	assim	102
10	mundo	101
11	bem	98
12	reino	86
13	toda	85
14	tudo	83
15	céu	83

TABLE VIII: Most frequent words in Os Lusíadas (Portuguese)

Looking at our tables (VIII, IX, X), we can make some interesting observations.

1. Common thematic words
   - words related to the king, with "rei" being 4th in portuguese, "king" 2nd in english and "rey" 1st in spanish, which makes sense as Camões dedicated this book to the then king D. Sebastião
   - maritime and geographic words like "mar", "terra", "shore" are also very common, since the work talks about the "Descobrimentos", an

English		
Rank	**Word**	**Count**
1	oer	445
2	king	367
3	thy	324
4	gama	320
5	shall	318
6	great	308
7	one	299
8	yet	270
9	though	239
10	portuguese	232
11	whose	227
12	every	225
13	shore	223
14	camoëns	220
15	thus	196

TABLE IX: Most frequent words in Os Lusíadas (English)

Spanish		
Rank	**Word**	**Count**
1	rey	256
2	gente	247
3	mar	238
4	tierra	231
5	tan	220
6	si	190
7	fué	142
8	grande	116
9	aquí	115
10	gran	109
11	tal	101
12	pecho	100
13	reino	97
14	mundo	95
15	allí	92

TABLE X: Most frequent words in Os Lusíadas (Spanish)

era where the portuguese kingdom sailed the seas discovering the unknown
- the portuguese people are also heavily mentioned, "gente" 1st in portuguese, "portugese" 10th in english and "gente" 2nd in spanish, showing Camões' intention of exalting his people ("grande", "great" and "grande" are also very common words)

2. English words show archaic forms with "thy" and "oer" being common, as well as more *function words* like "shall" and "whose"
3. The portuguese and spanish counts show very similar words and distributions, this really shows how the two languages are close linguistically

This comparison shows how translation choices and linguistic structures affect the frequency of words while maintaining the book's core themes across languages.

C. Large files

Purely for experimentation, we'll also go through two very large files created using [12]

```
$ tr -dc "A-Za-z 0-9" < /dev/urandom | \
fold -w100|head -n 5000000 > bigfile.txt
$ tr -dc "A-Za-z 0-9" < /dev/urandom | \
fold -w100|head -n 10000000 > bigfile.txt
```

which creates files with 5 000 000 and 10 000 000 lines respectively, with the first one being 505MB in size and the second one 1,0GB. These files only have random numbers and letter, so the results won't really be relevant, but it will still be interesting to measure the execution time and memory usage of the three approaches. The code to run these tests will be on *memory.py*.

Algorithm	**Time (s)**
Exhaustive	11.25
Probabilistic	99.76
Streaming	4.34

TABLE XI: Performance for 505MB file

For the **1GB** file (and also for an extra **750MB**), the program was killed before being able to complete the exhaustive count. However, when running the lossy algorithm with an ***epsilon* of 0.001** (which, as seen before, has an accuracy of around 93%), it was able to finish the program in only **8.43 seconds** and a memory footprint of about **2.5MB**, this shows the pratical use that streaming algorithms have when dealing with Big Data.

IV. Conclusion

This project compared three different approaches to word counting in text analysis: exact counting, probabilistic counting, and streaming algorithms.

The exact counter provided perfect accuracy but demonstrated scalability problems, failing to process files larger than 750MB due to memory constraints.

The probabilistic counter with decreasing probability showed good accuracy (93-96% with 40+ runs) and moderate memory usage, making it a viable option for medium-sized texts. However, its requirement for multiple runs to achieve high accuracy results in longer execution times.

The streaming (lossy counting) algorithm proved to be the most memory efficient solution, using way less memory than the other approaches (**1108kB** for **Don Quixote** compared to **4952kB** for the exact counter). While it sacrifices some accuracy with larger epsilon values, it can achieve 93% accuracy with $\epsilon = 0.001$. Most importantly, it was the only method capable of processing the 1GB file, completing the task in 8.43 seconds while maintaining a small memory footprint of 2.5MB.

The linguistic analysis of **Os Lusíadas** in three languages demonstrated the practical application of these counting methods and revealed interesting patterns in word frequencies across translations.

For practical applications, the choice of method should depend on the specific requirements:

- For small to medium texts requiring perfect accuracy, the exact counter is suitable
- For applications tolerating some uncertainty but requiring high accuracy, the probabilistic counter offers a good balance, particularly useful in social media trend analysis or network traffic monitoring
- For large-scale text analysis or streaming applications with memory constraints, the lossy counting algorithm provides the best scalability while maintaining acceptable accuracy, while also providing an adjustable parameter

References

[1] "Project Gutenberg — gutenberg.org", https://www.gutenberg.org/, [Accessed 10-12-2024].

[2] "Os Lusíadas by Luís de Camões — gutenberg.org", https://www.gutenberg.org/ebooks/27236, [Accessed 10-12-2024].

[3] "The Lusiad; Or, The Discovery of India, an Epic Poem by Luís de Camões — gutenberg.org", https://www.gutenberg.org/ebooks/32528, [Accessed 10-12-2024].

[4] "Los Lusíadas: Poema épico en diez cantos by Luís de Camões — gutenberg.org", https://www.gutenberg.org/ebooks/64775, [Accessed 10-12-2024].

[5] "Don Quixote by Miguel de Cervantes Saavedra — gutenberg.org", https://www.gutenberg.org/ebooks/996, [Accessed 10-12-2024].

[6] "Metamorphosis by Franz Kafka — gutenberg.org", https://www.gutenberg.org/ebooks/5200, [Accessed 10-12-2024].

[7] "Moby Dick; Or, The Whale by Herman Melville — gutenberg.org", https://www.gutenberg.org/ebooks/2701, [Accessed 10-12-2024].

[8] Joaquim Madeira, "Probabilistic counters", Slides, University of Aveiro, Aveiro, November 2024.

[9] Joaquim Madeira, "Data stream algorithms i", Slides, University of Aveiro, Aveiro, November 2024.

[10] "Lossy Count Algorithm - Wikipedia — en.wikipedia.org", https://en.wikipedia.org/wiki/Lossy_Count_Algorithm, [Accessed 28-12-2024].

[11] "psutil documentation &x2014; psutil 7.0.0 documentation — psutil.readthedocs.io", https://psutil.readthedocs.io/en/latest/, [Accessed 28-12-2024].

[12] "Looking for large text files for testing compression in all sizes — stackoverflow.com", https://stackoverflow.com/questions/44492576/looking-for-large-text-files-for-testing-compression-in-all-sizes, [Accessed 30-12-2024].