Most Frequent Letters

Vicente Barros

Abstract **–This report investigates three algorithms for identifying heavy hitters in data streams: Exact Counter, Approximate Counter, and Lossy Count. It assesses their performance in terms of accuracy, memory efficiency, and processing speed, using diverse language versions of 'Don Quixote' from Project Gutenberg as test data. The Exact Counter offers precise results but suffers from high memory usage. The Approximate Counter balances accuracy and memory use, while the Lossy Count algorithm significantly reduces memory needs at the cost of lower accuracy. This study offers practical guidance for selecting the most suitable algorithm for different data scenarios, balancing precision with resource efficiency.**

Resumo **–Este relatório investiga três algoritmos para identificar os elementos mais frequentes em fluxos de dados: Exact Counter, Approximate Counter e Lossy Counter. Avalia o desempenho deles em termos de precisão, eficiência de memória e velocidade de processamento, usando versões em diferentes línguas de "Dom Quixote"do Projeto Gutenberg como dados de teste. O Exact Counter oferece resultados precisos, mas com alto uso de memória. O Approximate Counter equilibra precisão e uso de memória, enquanto o algoritmo de Lossy Counter reduz significativamente a necessidade de memória, mas com menor precisão. Este estudo fornece insights valiosos para a escolha do algoritmo adequado em diferentes cenários de fluxo de dados, destacando os compromissos entre precisão e otimização de recursos.**

Keywords **–Most Frequent Letters, Exact Count, Fixed Probability Counter, Lossy Count**

I. Introduction

In the realm of data stream processing, identifying the most frequent elements—often termed 'heavy-hitters'—is a critical task that necessitates efficient algorithms. This report explores three distinct approaches to tackle this challenge: the Exact Counter, the Approximate Counter based on Morris' counter with a fixed probability, and the Lossy Count algorithm. Each of these methods presents unique trade-offs between accuracy and resource utilisation. Through a comprehensive analysis of their performance using various metrics, this study aims to provide insights into their suitability for different data-intensive applications. The algorithms are evaluated on diverse language versions of 'Don Quixote' sourced from Project Gutenberg, offering a real-world context to assess their efficacy in handling voluminous data streams. This report aims to guide practitioners in selecting the appropriate algorithm based on their specific requirements, balancing accuracy, memory efficiency, and processing speed.

II. Algorithms

A. Exact Counter

The exact counter, also known as the exact frequency counter is a simple approach where a hash table - a dictionary in Python - records every item from the text with their counter which is increased every time the same symbol appears.

Despite giving a precise count of the symbols present in the text, it requires more memory than the other algorithms present in this paper, occupying proportionally the amount of memory of the number of unique symbols.

To obtain the most frequent letters, it is necessary to sort the hash table. To accomplish this, the built-in sort function from Python was used to sort the items in a descendant way - using it the time complexity is on average $O(n \log n)$ [1] which means that the sorting step depends on the number of unique letters from the text.

Is important to note that for the implementation of the algorithms used **defaultdict** factory from the Python *collections* built-in library to remove the need to check if the key already exists in the map. Another optimization that would probably increase the efficiency of the algorithm would be the use of the **OrderDict** also from the *collections* library to keep the map sorted while iterating over the input text.

B. Approximate Counter

The Approximate Counter, often referred to as the probabilistic frequency counter, adopts a different strategy compared to the exact counter. At its core, this method employs a stochastic approach where each letter in the text is counted with a fixed probability - in this implementation $probabilty = \frac{1}{32}$.

This strategy reduces drastically the amount of space needed for the counters by increasing their sizes based on a random number generator. This process effectively samples a subset of occurrences, thereby reducing the memory footprint significantly compared to the exact counter, which is particularly advantageous for large datasets.

After the probabilistic counting, each count in the hash table is multiplied by the inverse of the proba-

bility. This scaling step approximates the actual frequency of each letter, striking a balance between accuracy and memory efficiency.

Sorting the hash table to identify the most frequent letters remains necessary, similar to the exact counter. The sorting operation, using Python's built-in function, has an average time complexity of $O(n \log n)$, where n is the number of unique letters.

C. Lossy Count

The Lossy Count algorithm departs from the previous algorithms addressing the unique items issue. Unlike the exact counter and the approximate counter which keep track of every unique letter encountered, the Lossy count sets a threshold to remove the less frequent items making it useful to identify heavy-hitters.

The implementation of Lossy Counting operates on a simple yet effective principle. It maintains a running total of items processed and a dynamic threshold that divides this total by the specified k value. This threshold, denoted as 'delta' in the code, is key to the algorithm's memory efficiency. When the total number of processed items crosses a multiple of k, the algorithm increases the delta and re-evaluates the items in the map.

Each letter's count is compared against this delta. If a letter's count is less than the current delta, it is considered a light-hitter and is removed from the map. This approach ensures that only those letters that occur with a frequency above the calculated threshold are retained, effectively filtering out less frequent items. As a result, the Lossy Count algorithm is particularly well-suited for large data streams where maintaining counts of all unique items is not feasible due to memory constraints.

An important characteristic of this algorithm is its adaptability to different values of k. A smaller k results in a larger threshold, leading to more aggressive filtering of items. Conversely, a larger k value lowers the threshold, allowing more items to be retained. This tunability makes the Lossy Count algorithm versatile, as it can be adjusted according to the specific requirements of a task, balancing between memory usage and the granularity of frequency data retained.

Algorithm 1 Lossy Counting Algorithm

```
function LossyCounting(text, k)
    map ← {}
    total ← 0
    δ ← 0
    for each letter in text do
        if letter not in map then
            map[letter] ← δ + 1
        else
            map[letter] ← map[letter] + 1
        end if
        total ← total + 1
        threshold ← ⌊total/k⌋
        if threshold ≠ δ then
            δ ← threshold
            for each item, count in map do
                if count < δ then
                    map ← map \ {item}
                end if
            end for
        end if
    end for
    return map
end function
```

III. Methodology

The algorithms were tested on different books provided by Project Gutenberg. For the sake of simplicity, all the benchmarks presented in this paper will be made under Don Quixote by Miguel de Cervantes in different languages that simulate the data stream. Each book was processed to be prepared for the analysis. Firstly, the header and the footer, which contained meta-information and licensing about it, followed by the removal of the stop-words[1] and punctuation. For the tokenization and removal of the stopwords Natural Language Toolkit (NLTK) [2] was used to speed up the process.

The final step was to uppercase all the alphabetic values and store the output. It is noteworthy that all the special characters with accentuation were kept. For benchmarking purposes, the lossy counter was initialised with $k \in \{3, 5, 10\}$, which are the top k most frequent letters. The approximate counter ran 500 times for each book.

Books	Total	Unique
Don Quixote (Hungarian)	749,921	36
Don Quixote (Spanish)	1,565,545	35
Don Quixote (Dutch)	365,670	34
Don Quixote (English)	1,103,436	36
Don Quixote (Finnish)	301,427	28
Don Quixote (French)	1,394,837	45

TABLE I
TOTAL AND UNIQUE NUMBER OF LETTERS

[1]Common words that do not contribute to the meaning of the text

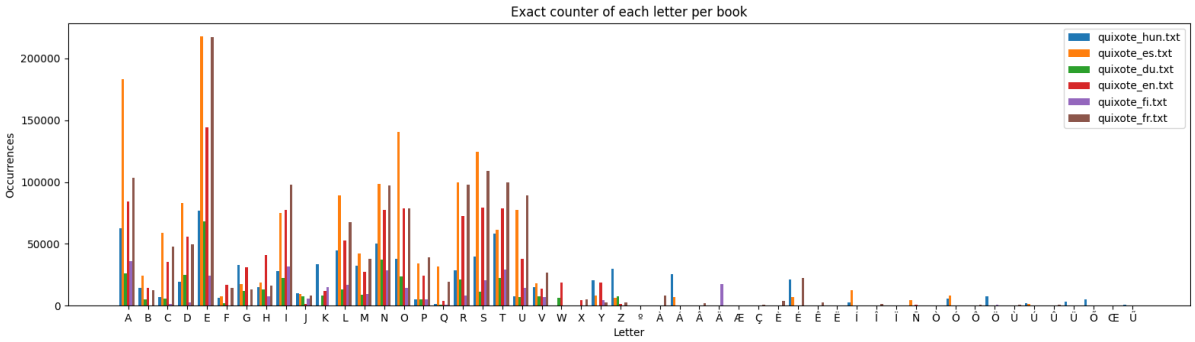

Fig. 1 - Exact Counter of each letter in the analysed per book

After processing the texts, we employed a set of statistical metrics to analyze the performance of each algorithm:

1. **Mean Absolute Error (MAE)**: Reflects the average size of errors in frequency estimations, with lower values indicating greater accuracy.
2. **Mean Relative Error**: Assesses error magnitude relative to the actual counts, providing insights into proportional accuracy.
3. **Mean Accuracy Ratio**: Evaluates how closely the algorithm's frequency estimates align with actual frequencies on average.
4. **Smallest and Largest Values**: Identifies the range of frequency counts, giving an idea of the spread in the algorithm's predictions.
5. **Mean, Mean Absolute Deviation, and Standard Deviation**: These measure the central tendency and variability in the frequency estimates, indicating the average count and consistency of predictions.
6. **Variance and Maximum Deviation**: Examine the spread and the furthest count from the mean, respectively, providing insights into the consistency and reliability of the algorithms.

IV. Results

The analysis of the most frequent letters is depicted in Figures 2, 3 and 4 where is possible to observe the difference between the performance of each algorithm. It is noteworthy, that the missing values from the lossy are caused by the incapability of it to identify them as the most frequent letters.

Tables II through VII present a side-by-side comparison of the most frequent letters as determined by the exact counter, the fixed probability counter, and the lossy counting algorithm at various k values. The exact and fixed probability counters demonstrate a strong alignment in all cases, which underscores the reliability of the fixed probability counter in approximating the exact frequency counts across different languages.

The lossy counting algorithm shows variability in its performance, particularly at lower k values where it sometimes fails to capture the most frequent letters identified by the exact method. However, as k increases, the lossy counting results begin to more closely approximate the exact counts, suggesting its potential effectiveness in applications where resource constraints necessitate a balance between accuracy and computational efficiency.

Tables VIII through XI provide a detailed breakdown of the statistical metrics for the English translation. The mean absolute error and mean relative error metrics indicate that the lossy counting algorithm's accuracy improves with higher k values, as the counts converge closer to the true frequencies. The mean accuracy ratio further substantiates this observation, with lossy counting demonstrating an acceptable level of precision at $k = 10$.

The smallest and largest values, along with the mean, give insight into the range and central tendency of the counts. Notably, the variance and standard deviation values are significantly lower for the fixed probability counter compared to the lossy counting, suggesting less spread and more consistency in the former's frequency estimations.

It's worth noting that the statistical analysis presented in the paper from the tables XI, VIII, IX and X are related to the English version of the book. However, all the remaining statistics from the other available languages are in the results folder from the project.

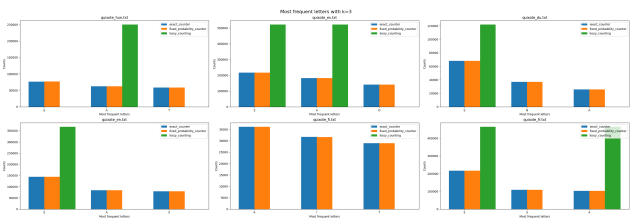

Fig. 2 - Most frequent letters with $k = 3$ comparison between each algorithm

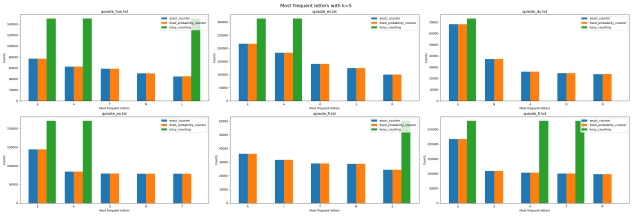

Fig. 3 - Most frequent letters with $k = 5$ comparison between each algorithm

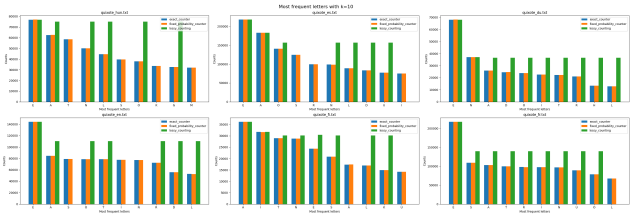

Fig. 4 - Most frequent letters with $k = 10$ comparison between each algorithm

TABLE II
MOST FREQUENT LETTERS FOR QUIXOTE_HUN.TXT

Algorithm	**Most Frequent Letters**
Exact	E A T N L S O K G M
Approximate	E A T N L S O K G M
Lossy($k = 3$)	Y V R A G
Lossy($k = 5$)	V L E U R A G Y
Lossy($k = 10$)	E L A O V N C U R G Y

TABLE III
MOST FREQUENT LETTERS FOR QUIXOTE_ES.TXT

Algorithm	**Most Frequent Letters**
Exact	E A O S R N L D U I
Approximate	E A O S R N L D U I
Lossy($k = 3$)	E A V L
Lossy($k = 5$)	A L V E
Lossy($k = 10$)	E A O D L N U V G

TABLE IV
MOST FREQUENT LETTERS FOR QUIXOTE_DU.TXT

Algorithm	**Most Frequent Letters**
Exact	E N A D O I T R H L
Approximate	E N A D O I T R H L
Lossy($k = 3$)	E I F
Lossy($k = 5$)	E L I F
Lossy($k = 10$)	E N A I O D H R T L F

TABLE V
MOST FREQUENT LETTERS FOR QUIXOTE_EN.TXT

Algorithm	**Most Frequent Letters**
Exact	E A S O T I N R D L
Approximate	E A S T O I N R D L
Lossy($k = 3$)	L E W
Lossy($k = 5$)	E L R F A W
Lossy($k = 10$)	E L O R T D F A V W

TABLE VI
MOST FREQUENT LETTERS FOR QUIXOTE_FI.TXT

Algorithm	**Most Frequent Letters**
Exact	A I T N E S Ä L K U
Approximate	A I T N E S Ä L K U
Lossy($k = 3$)	O K S E B
Lossy($k = 5$)	E O K S W B
Lossy($k = 10$)	A I E O S T R W B K L N H

TABLE VII
MOST FREQUENT LETTERS FOR QUIXOTE_FR.TXT

Algorithm	**Most Frequent Letters**
Exact	E S A T R I N U O L
Approximate	E S A T R I N U O L
Lossy($k = 3$)	A U M B E
Lossy($k = 5$)	A U E T O M B
Lossy($k = 10$)	E A R T N O M B U I C H S

TABLE VIII
STATISTICS FOR QUIXOTE_EN.TXT WITH LOSSY COUNTING ($k = 3$)

Metric	**Value**
Mean Absolute Error	49310.42
Mean Relative Error	1.64663
Mean Accuracy Ratio	0.81330
Smallest Value	367687
Largest Value	367689
Mean	367688.0
Mean Absolute Deviation	0.66667
Standard Deviation	0.81650
Maximum Deviation	1.0
Variance	0.66667

TABLE IX
Statistics for quixote_en.txt with Lossy Counting ($k = 5$)

Metric	Value
Mean Absolute Error	45804.08
Mean Relative Error	1.67851
Mean Accuracy Ratio	1.01184
Smallest Value	220612
Largest Value	220614
Mean	220612.67
Mean Absolute Deviation	0.88889
Standard Deviation	0.94281
Maximum Deviation	1.33333
Variance	0.88889

TABLE X
Statistics for quixote_en.txt with Lossy Counting ($k = 10$)

Metric	Value
Mean Absolute Error	28002.03
Mean Relative Error	1.31207
Mean Accuracy Ratio	0.86763
Smallest Value	110306
Largest Value	144152
Mean	113691.0
Mean Absolute Deviation	6092.2
Standard Deviation	10153.67
Maximum Deviation	30461.0
Variance	103096947.2

TABLE XI
Statistics for quixote_en.txt with Fixed Probability Counter

Metric	Value
Mean Absolute Error	31.44
Mean Relative Error	0.11603
Mean Accuracy Ratio	0.88718
Smallest Value	5
Largest Value	144213
Mean	34471.41
Mean Absolute Deviation	29150.67
Standard Deviation	35275.41
Maximum Deviation	109741.59
Variance	1244354310.0

While the fixed probability counter offers a consistent and reliable approximation across various languages, the lossy counting algorithm's performance is more nuanced, improving with higher k values and demonstrating sensitivity to the linguistic characteristics of the books used.

V. Time Analysis

A. Exact vs. Approximate Counting

The exact counter and the fixed probability counter demonstrate close proximity in processing times across all translations. This observation, as seen in Tables XII, XIII, XIV, XV, XVI, and XVII, highlights the efficiency of the approximate method in delivering similar results to the exact method with a marginal increase in time. This slight increase in time for the fixed probability counter can be attributed to the additional computation involved in the random sampling and scaling process.

B. Impact of k on Lossy Counting

A critical observation from the tables is the notable increase in processing time for the lossy counting algorithm as the k value escalates. For instance, in the English translation (Table XV), the processing time almost doubles from $k = 3$ to $k = 10$. This trend is consistent across other translations as well, indicating that while lossy counting effectively reduces memory usage, its computational time is adversely affected by higher k values.

C. Language-Specific Observations

Interestingly, the processing times vary across different languages, reflecting the unique characteristics and complexities of each language. For example, the Hungarian and Finnish translations (Tables XII and XVI) show relatively lower processing times, potentially due to linguistic factors such as letter frequency distribution or text length. In contrast, the Spanish and French translations (Tables XIII and XVII) exhibit higher processing times, indicating a possible correlation with the complexity or size of the text.

D. Practical Implications

For tasks requiring rapid text processing with reasonable accuracy, the fixed probability counter emerges as a viable option. However, for applications where memory usage is constrained, and exact counts are not critical, the lossy counting at lower k values offers a beneficial trade-off.

TABLE XII
Computational times for quixote_hun.txt

Algorithm	Time (seconds)
Exact Counter	0.09312
Fixed Probability Counter	0.10511
Lossy Counting ($k = 3$)	0.49728
Lossy Counting ($k = 5$)	0.47296
Lossy Counting ($k = 10$)	0.39604

TABLE XIII
COMPUTATIONAL TIMES FOR QUIXOTE_ES.TXT

Algorithm	**Time (seconds)**
Exact Counter	0.15081
Fixed Probability Counter	0.21579
Lossy Counting ($k = 3$)	1.15358
Lossy Counting ($k = 5$)	1.08607
Lossy Counting ($k = 10$)	0.78080

TABLE XIV
COMPUTATIONAL TIMES FOR QUIXOTE_DU.TXT

Algorithm	**Time (seconds)**
Exact Counter	0.03391
Fixed Probability Counter	0.04650
Lossy Counting ($k = 3$)	0.24020
Lossy Counting ($k = 5$)	0.21990
Lossy Counting ($k = 10$)	0.18646

TABLE XV
COMPUTATIONAL TIMES FOR QUIXOTE_EN.TXT

Algorithm	**Time (seconds)**
Exact Counter	0.11213
Fixed Probability Counter	0.13865
Lossy Counting ($k = 3$)	0.71213
Lossy Counting ($k = 5$)	0.66843
Lossy Counting ($k = 10$)	0.56022

TABLE XVI
COMPUTATIONAL TIMES FOR QUIXOTE_FI.TXT

Algorithm	**Time (seconds)**
Exact Counter	0.03047
Fixed Probability Counter	0.03685
Lossy Counting ($k = 3$)	0.19691
Lossy Counting ($k = 5$)	0.17412
Lossy Counting ($k = 10$)	0.14390

TABLE XVII
COMPUTATIONAL TIMES FOR QUIXOTE_FR.TXT

Algorithm	**Time (seconds)**
Exact Counter	0.13152
Fixed Probability Counter	0.17155
Lossy Counting ($k = 3$)	0.95266
Lossy Counting ($k = 5$)	0.83047
Lossy Counting ($k = 10$)	0.68735

VI. CONCLUSION

In conclusion, this report has thoroughly investigated the practicality of the Lossy Count algorithm and two counter-based methods (Exact Counter and Approximate Counter) in identifying the most frequent letters in data streams. While the counter-based approaches deliver precise results, their memory consumption increases significantly with the diversity of items in the stream. Though less accurate in capturing the frequency and order of items, the Lossy Count significantly reduces memory usage. This advantage is pivotal in real-world applications with unknown varieties of items. Although the Approximate Counter also lessens memory demands, its impact is less pronounced in scenarios with various items. Hence, the choice of algorithm depends on the balance between accuracy and resource constraints in specific applications.

REFERENCES

[1] PrepBytes, "Sort function in python", 2023, Accessed: 2024-01-02.
URL: `https://www.prepbytes.com/blog/python/sort-function-in-python/`

[2] NLTK's Team, "Nltk", 2023, Accessed: 2024-01-02.
URL: `https://www.nltk.org/`