Letter Frequency Counting Algorithms in Literary Texts

Daniel Jorge Bernardo Ferreira

Abstract **– This paper explores different methods for counting letter frequencies in literary texts. In the past, counting methods faced challenges with large datasets due to limited memory. Modern computing allows for more memory, but efficient algorithms are still valuable. We focus on exact and approximate counting algorithms, assessing their accuracy and efficiency. Literary works like *The Republic* are analyzed in English and Portuguese. We preprocess the text and evaluate methods based on metrics like Mean Relative Error and Bits Saved Ratio. We introduce algorithms like the Exact Counter, Fixed Probability Counter, Morris Counter, and Count-Min Sketch. The Lossy Counter is explored for heavy hitters. Results show trade-offs between accuracy and efficiency, aiding informed algorithm choices.**

Resumo **– Este artigo explora diferentes métodos de contagem de frequências de letras em textos literários. No passado, os métodos de contagem enfrentavam desafios com grandes conjuntos de dados devido à memória limitada. A computação moderna permite mais memória, mas os algoritmos eficientes continuam a ser importantes. Centramonos em algoritmos de contagem exactos e aproximados, avaliando a sua precisão e eficiência. São analisadas obras literárias como "A República"em inglês e português. Pré-processamos o texto e avaliamos os métodos com base em métricas como o Erro Relativo Médio e o Rácio de Bits Guardados. Introduzimos algoritmos como o Contador Exato, o Fixed Probability Counter, o Morris Counter e o Count-Min Sketch. O Lossy Counter é explorado para os heavy hitters. Os resultados mostram os compromissos entre exatidão e eficiência, ajudar a efetuar escolhas informadas de algoritmos.**

Keywords **– Text Mining, Data Streams, Approximate Counting, Letter Frequency**

Palavras chave **– Mineração de Texto, Fluxo de Dados, Contagem Aproximada, Frequência de Letras**

I. Introduction

In the early days of computing, when memory was a scarce commodity, traditional counting methods became impractical for large datasets. These challenges led to the development of approximate counting algorithms that provide estimations of item frequencies with significantly reduced memory requirements. While advancements in hardware have alleviated memory constraints, approximate counting algorithms remain relevant in modern computing, offering streamlined solutions for data analysis tasks where precision takes a backseat to efficiency.

The primary objective of this paper is to explore and evaluate different counting algorithms by focusing on their application in identifying the most frequent letters in literary text files. The implemented methods include exact counters, approximate counters, and frequency counters targeting heavy hitters.

The assessment criteria include an analysis of absolute and relative errors, average values, and other relevant metrics to compare the accuracy and efficiency of the developed approaches. Additionally, we will investigate the consistency of results across multiple runs and explore similarities in the most frequent letters between literary works in different languages. The findings of this research aim to contribute insights into the strengths and limitations of each counting algorithm, facilitating informed decisions in their practical application.

II. Methodology

A. Literary Works

The literary works selected for this study include:

- *Orthodoxy* by G. K. Chesterton [1].
- *Mere Christianity* by C.S. Lewis [2].
- *The Republic* by Plato [3].

The texts were obtained in English and Portuguese and sourced from repositories such as Project Gutenberg, Faded Page, and Internet Archive.

The compiled results are available for each literary work, but the focus of this paper will center on presenting and discussing the findings related to *The Republic.*

B. Data Preprocessing

The following preprocessing steps were applied to the raw text data:

(a) **Removal of File Headers:** Any extraneous information, including Project Gutenberg-specific text, was removed to focus solely on the literary content.
(b) **Filtering Non-Alphanumeric Characters:** Punctuation marks and other non-alphanumeric characters were filtered out to isolate the relevant textual content.

(c) **Standardizing Characters:** All characters in the text files were mapped to uppercase to ensure uniformity in letter counts, regardless of case.

C. Pratical Considerations

The non-deterministic algorithms were independently executed ten times to mitigate their inherent randomness and obtain a more dependable estimate of their performance. The reported results reflect the average outcome of these runs.

We did not choose a higher number of iterations, as we are primarily interested in the relative performance of the algorithms, and the additional computational cost would not be justified.

D. Evaluation Metrics

In the assessment of algorithmic performance, four primary metrics were used:

(a) **Mean Relative Error:** Average relative difference between predicted and actual values in a dataset. It is calculated using the formula:

$$\mathrm{MRE}(y, \hat{y}) = \frac{1}{N} \sum_{i=0}^{N-1} \frac{|y_i - \hat{y}_i|}{|y_i|} \tag{1}$$

where N represents the number of observations, y_i denotes the actual values, and $\hat{y}_i$ represents the predicted values for each observation.

(b) **Bits Required:** Minimum number of bits needed to represent the stored information. It is calculated using the formula:

$$\mathrm{BR}(y) = \sum_{i=1}^{N} \lfloor \log_2 y_i \rfloor + 1 + c \tag{2}$$

where N represents the number of observations, y_i denotes the values stored for each observation, and c is a constant term that accounts for any additional overhead or specific requirements of the system. For Hash Maps, this term would include the bits required to store the keys.

(c) **Bits Saved Ratio:** Relative reduction in the number of bits required for storage after compression. It is calculated using the formula:

$$\mathrm{BSR}(y, \hat{y}) = \frac{\mathrm{BR}(y) - \mathrm{BR}(\hat{y})}{\mathrm{BR}(y)} \tag{3}$$

where $\mathrm{BR}(y)$ and $\mathrm{BR}(\hat{y})$ represent the bits required to store the original and compressed data respectively.

(d) **Compression-Error Efficiency:** Trade-off between compression and error. It is calculated using the formula:

$$\begin{aligned} \mathrm{CEE}(y, \hat{y}) = \alpha * \mathrm{BSR}(y, \hat{y}) + \\ (1 - \alpha) * (1 - \mathrm{MRE}(y, \hat{y})) \end{aligned} \tag{4}$$

where $\mathrm{BSR}(y, \hat{y})$ represents the Bits Saved Ratio, $\mathrm{MRE}(y, \hat{y})$ corresponds to the Mean Relative Error, and α is a user-defined parameter that controls the relative importance of each metric. In this study, α is set to 0.5 to give equal weight to both metrics.

(e) **Normalized Discounted Cumulative Gain:** Relevance of recommendations taking in account their order. It is calculated using the formula:

$$\mathrm{nDCG_k} = \frac{\mathrm{DCG_k}}{\mathrm{IDCG_k}} \tag{5}$$

where $\mathrm{DCG_k}$ is the Discounted Cumulative Gain at position k,

$$\mathrm{DCG_k} = \sum_{i=1}^{k} \frac{rel_i}{\log_2(i+1)} \tag{6}$$

$\mathrm{IDCG_k}$ is the Ideal Discounted Cumulative Gain at position k,

$$\mathrm{IDCG_k} = \sum_{i=1}^{|REL_k|} \frac{rel_i}{\log_2(i+1)} \tag{7}$$

and rel_i is the graded relevance of the item at position i, and REL_k represents the ordered list of relevant items up to position k. In this paper, we define the relevance of an item as the inverse of its position in REL_k, reflecting the idea that higher-ranked relevant items contribute more to the cumulative gain.

(f) **Compression-Ranking Efficiency:** Trade-off between compression and ranking quality. It is calculated using the formula:

$$\begin{aligned} \mathrm{CRE_k}(y, \hat{y}) = \alpha * \mathrm{BSR}(y, \hat{y}) + \\ (1 - \alpha) * \mathrm{NDCG_k} \end{aligned} \tag{8}$$

where $\mathrm{BSR}(y, \hat{y})$ represents the Bits Saved Ratio, $\mathrm{NDCG_k}$ corresponds to the Normalized Discounted Cumulative Gain at position k, and α is a user-defined parameter that controls the relative importance of each metric. In this study, α is set to 0.5 to give equal weight to both metrics.

III. COUNTING ALGORITHMS

Exploring the potential for more efficient numerical representations, we consider the notion of a schema that could encode a number using fewer bits. However, the application of the Pigeonhole Principle [4] exposes a fundamental limitation. If distinct integers, generated through different increments, map to the same memory representation, a contradiction ensues as both integers would produce identical values upon query. Until the development of practical Quantum Computers — moving beyond the realm of mere speculation — we find our current numerical representations resistant to further optimization.

In this section, we will explore diverse counting algorithms and assess their performance, aiming to identify the most effective solution for our specific use case.

A. Exact Counter

The Exact Counter, or the Naive Counter, is a simple counting algorithm that maintains the exact number of occurrences for each element in a set. The algorithm uses a data structure, typically a Hash Map, to store the counts of individual elements.

In Algorithm 1, C is the counter represented as a Hash Map. The `Update` function increments the count of the specified element x in the counter, while the `Read` function retrieves its current count.

Algorithm 1 Exact Counter

Require: C – counter

function UPDATE(x)
  $C[x] \leftarrow C[x] + 1$

function READ(x)
  return $C[x]$

The character counts resulting from the execution of the exact algorithm on *The Republic* in both English and Portuguese are shown in Fig. 1. For the English version of the work, the measured BR value is 683, while for the Portuguese version, it is 597.

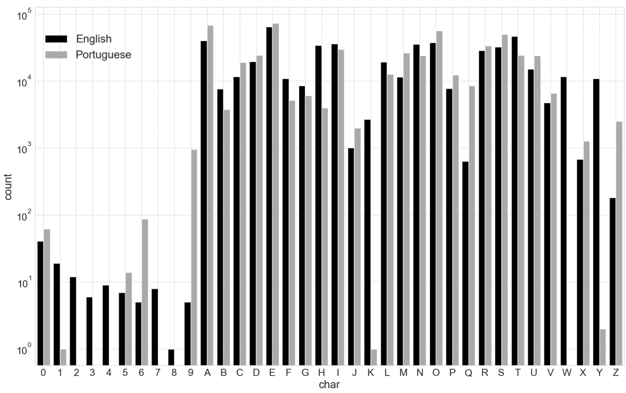

Fig. 1: Log-scaled Character Counts Obtained After Running the Exact Algorithm on *The Republic* in English and Portuguese.

This approach provides an ideal solution for accurate counting. However, as the dataset grows, memory limitations and processing power can make it difficult to use. To address these challenges, we will explore several approximate counting algorithms that offer a trade-off between accuracy and efficiency.

B. Fixed Probability Counter

The Fixed Probability Counter is the simplest implementation of a probabilistic counting algorithm. It introduces a random sampling mechanism to reduce the number of updates performed on the counter, following a binomial distribution.

In Algorithm 2, ρ is a user-defined parameter that controls the probability of updating the counter. The `Update` function increments the count of the specified element x with a probability of ρ. The `Read` function returns an estimate of the true count by dividing the stored count by ρ.

Algorithm 2 Fixed Probability Counter

Require: C – counter; ρ – probability

function UPDATE(x)
  if random() $< \rho$ **then**
    $C[x] \leftarrow C[x] + 1$

function READ(x)
  return $C[x]\rho^{-1}$

In Fig. 2, we observe that the BSR for the Fixed Probability Counter is negatively correlated with the value of ρ. This behavior is expected since a higher value of ρ leads to more updates being performed on the counter, which in turn increases the number of bits required to store the data.

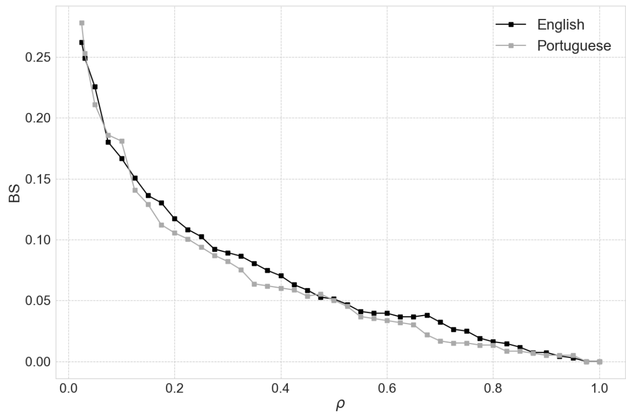

Fig. 2: Bits Saved Ratio Obtained After Running the Fixed Probability Counter Algorithm on *The Republic* in English and Portuguese for Different Values of ρ.

If higher values of ρ lead to more frequent updates, then we would expect the algorithm's accuracy to improve as well. This hypothesis is confirmed in Fig. 3, where we observe that the MRE decreases as ρ increases.

The optimal ρ values, which maximize CEE, for both the Portuguese and English version of the work are depicted in Table I. These values provide a good compromise between accuracy and efficiency in this particular context.

TABLE I: Optimal ρ Values Maximizing Compression-Error Efficiency (CEE) for English (EN) and Portuguese (PT) Using the Fixed Probability Counter Algorithm

	ρ	BS	MRE	CEE
EN	0.175	0.130	0.090	0.520
PT	0.031	0.253	0.164	0.545

The Fixed Probability Counter can be used to reduce

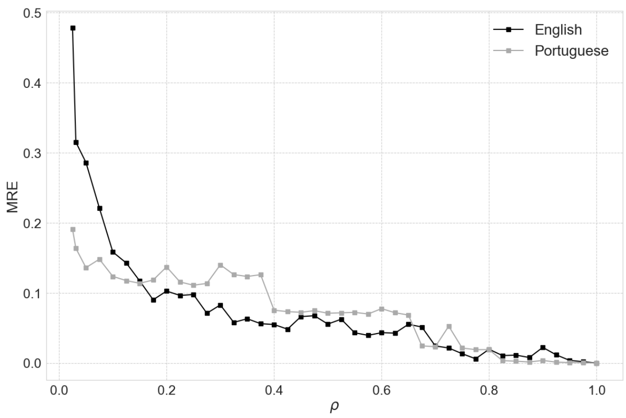

Fig. 3: Mean Relative Error Obtained After Running the Fixed Probability Counter Algorithm on *The Republic* in English and Portuguese for Different Values of ρ.

memory usage, but its simplicity may lead to imprecise estimates, especially in scenarios where elements have significantly different frequencies. Although it serves as a starting point for exploring approximate counting algorithms, more advanced methods such as the Morris Counter or Count-Min Sketch are often favored in real-world applications with large datasets.

C. Morris Counter

In 1985, Robert Morris was working at Bell Labs on the Unix spellchecking program and wanted to keep track of trigram counts, requiring simultaneous counters. To reduce the memory needed, he invented what is now known as the Morris Counter [5] which approximates this count proportional to $O(\log \log n)$ space.

The algorithm is similar to the Fixed Probability Counter, but instead of using a fixed probability, it updates the counter with a probability that decreases exponentially with the current count.

In Algorithm 3, C is the counter, α is some constant that controls the rate of decay, and β is a threshold that ensures the counter is updated at least β times before the decay starts.

In the original implementation of the algorithm, α was set to 1, and there was no threshold β. However, we found that these parameters could be tuned to improve the performance of the algorithm. In our implementation, we fixed β to 8, so the counter behaves deterministically for the first 8 updates.

In Fig. 4, we see similar results to the Fixed Probability Counter, where the BSR is negatively correlated with the value of α. Intuitively, as $\alpha \to \infty$, the algorithm approaches a deterministic counter, so α is directly related to memory usage (bits) and inversely related to variability.

In Fig. 5, we observe that the MRE decreases as α increases, which is consistent with the results obtained for the Fixed Probability Counter and in line with our expectations.

Algorithm 3 Morris Counter

Require: C – counter; α – control constant; β – naivety threshold

```
function UPDATE(x)
    if C[x] < β or random() < (1 + α^{-1})^{-C[x]} then
        C[x] ← C[x] + 1

function READ(x)
    if C[x] > β then
        return (1 + α^{-1})^{C[x]} α − α
    else
        return C[x]
```

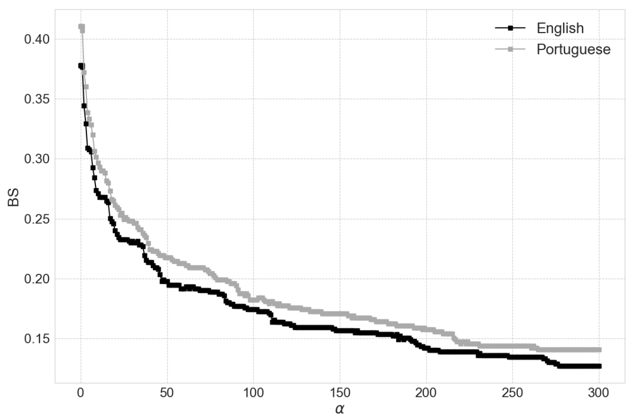

Fig. 4: Bits Saved Ratio Obtained After Running the Morris Counter Algorithm on *The Republic* in English and Portuguese for Different Values of ρ.

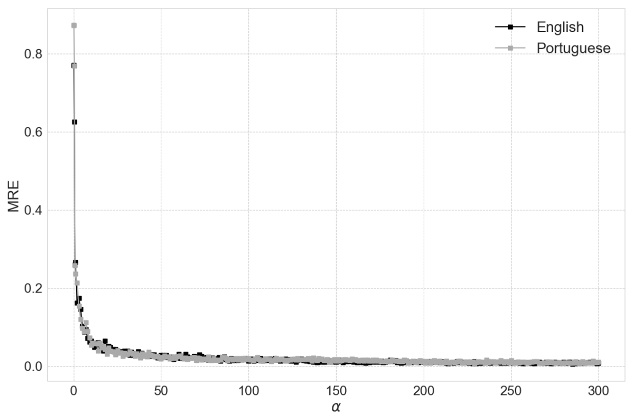

Fig. 5: Mean Relative Error Obtained After Running the Morris Counter Algorithm on *The Republic* in English and Portuguese for Different Values of α.

The optimal α values, which maximize CEE, for both the Portuguese and English version of the work are depicted in Table II. We can observe that the CEE values are higher than those obtained for the Fixed Probability Counter, indicating that the Morris Counter is a more efficient algorithm for this particular use case.

TABLE II: Optimal α Values Maximizing Compression-Error Efficiency (CEE) for English (EN) and Portuguese (PT) Using the Morris Counter Algorithm

	α	BS	MRE	CEE
EN	12	0.268	0.049	0.609
PT	14	0.288	0.040	0.624

D. Count-Min Sketch

The Count-Min Sketch is a probabilistic data structure that serves as a frequency table of events in a stream of data.

It was invented in 2003 by Cormode and Muthukrishnan [6] and has since been applied in a wide range of applications, including network traffic monitoring, frequency estimation, and approximate counting.

It uses hash functions to map events to frequencies, but unlike a hash table, it uses only sub-linear space at the expense of overcounting some events due to collisions.

The estimates produced by the algorithm are biased, meaning they are always equal to or greater than the true value. However, this one-sided error is often acceptable in practice, especially when the algorithm is used to identify heavy hitters.

In Algorithm 4, S is a 2-dimensional array used to store the counts with dimensions $d = \lceil \ln \gamma^{-1} \rceil$ and $w = \lceil \varepsilon^{-1} e \rceil$, where γ is the probability of failure and ε is the error rate. Associated with each row is a pairwise independent hash function h_i, which maps the elements to the array. The `Update` function computes the hash value for the specified element x using each of the hash functions and increments the corresponding count in the array. The `Read` does the same, but instead of incrementing the count, it returns the minimum value across all the hash functions.

Algorithm 4 Count-Min Sketch

Require: S – sketch; $h_1, h_2, \ldots, h_{\lceil \ln \gamma^{-1} \rceil}$ – pairwise independent hash functions; ε – error rate; γ – probability of failure

function Update(x)
  for $i = 1$ to $\lceil \ln \gamma^{-1} \rceil$ **do**
    $j \leftarrow h_i(x) \mod \lceil \varepsilon^{-1} e \rceil$
    $S[i, j] \leftarrow S[i, j] + 1$

function Read(x)
  $\mu \leftarrow \infty$
  for $i = 1$ to $\lceil \ln \gamma^{-1} \rceil$ **do**
    $j \leftarrow h_i(x) \mod \lceil \varepsilon^{-1} e \rceil$
    $\mu \leftarrow \min(\mu, S[i, j])$
  return μ

In Fig. 6, we observe that the BSR is positively correlated with both ε and γ. As ε increases, the number of columns in the array decreases, which in turn reduces the number of bits required to store the data. Similarly, as γ increases, the number of rows in the array decreases, reducing the number of bits required to store the data.

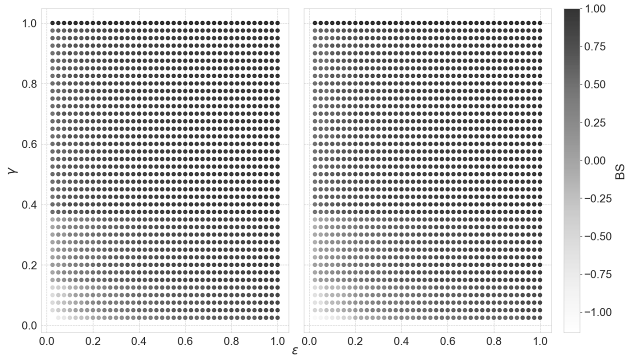

Fig. 6: Bits Saved Ratio Obtained After Running the Count-Min Sketch Algorithm on *The Republic* in English (left) and Portuguese (right) for Different Values of ε and γ.

In Fig. 7, we observe that the MRE is also positively correlated with both ε and γ. If we have a smaller array, we will have more collisions, which will lead to overcounting and a higher MRE.

The MRE reaches extremely high values, which suggests that the Count-Min Sketch algorithm might not be a good fit for our particular use case. However, it might still be useful for identifying heavy hitters, where the MRE is less relevant.

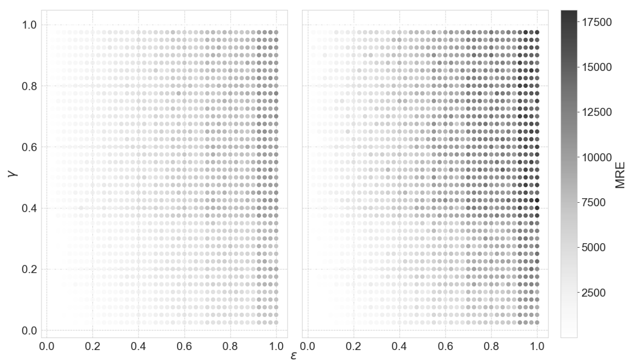

Fig. 7: Mean Relative Error Obtained After Running the Count-Min Sketch Algorithm on *The Republic* in English (left) and Portuguese (right) for Different Values of ε and γ.

The optimal ε and γ values, which maximize CEE, for both the Portuguese and English version of the work are depicted in Table III. As we stated before, the results indicate that the Count-Min Sketch algorithm is not very effective. In fact, the highest CEE score of 0.5 was obtained when the sketch was empty, meaning it saved 100% of the bits but had MRE equal to 1.

TABLE III: Optimal ε and γ Values Maximizing Compression-Error Efficiency (CEE) for English (EN) and Portuguese (PT) Using the Count-Min Sketch Algorithm

	ε	γ	BS	MRE	CEE
EN	0.025	1.000	1.000	1.000	0.500
PT	0.025	1.000	1.000	1.000	0.500

E. Lossy Counter

Lossy counting is a deterministic algorithm well-suited for identifying heavy hitters using a limited amount of memory. It was created by computer scientists Manku and Motwani in 2002 [7]. It finds applications in computations where data takes the form of a continuous data stream instead of a finite data set.

The algorithm works by dividing the data stream into buckets of a fixed size and keeping track of the number of occurrences of each element in the current bucket. When the bucket is full, the algorithm removes all elements whose count is below a certain threshold, hence the name lossy counting.

The basic idea behind the algorithm is to find heavy hitters without having to track every element, periodically removing elements that are unlikely to be heavy hitters based on the data seen so far.

In Algorithm 5, C and D are both Hash Maps used to store the counts and decay/aging factors respectively. The current stream length N is the total number of elements seen so far, and the current window δ is the number or id of the current bucket. The error bound ε is used to determine the bucket size, and the support threshold σ determines the minimum count required for an element to be considered a heavy hitter.

In Fig. 8, we observe that the BSR is positively correlated with both ε and σ. Though, the impact of ε is more pronounced than that of σ. It seems it reaches a plateau very quickly for $\varepsilon > 0.2$, where the BSR is around 0.9 for both the English and Portuguese versions of the work.

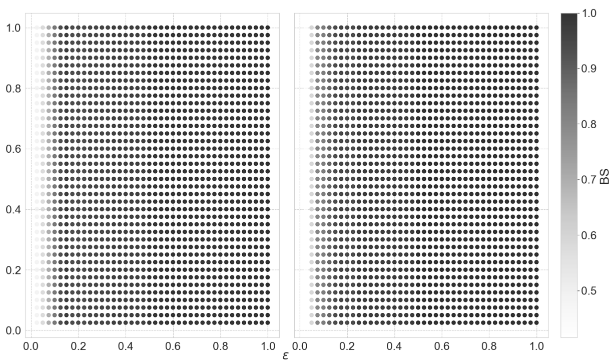

Fig. 8: Bits Saved Ratio Obtained After Running the Lossy Counter Algorithm on *The Republic* in English and Portuguese for Different Values of ε and σ.

Fig. 9 shows the NDGC at position 10 for the Lossy Counter algorithm. The results indicate that as ε and σ increase, the ranking quality decreases. After a certain point, the algorithm starts to remove elements that are likely to be heavy hitters, which leads to a decrease in ranking quality.

Algorithm 5 Lossy Counter

Require: C – counter; N – current stream length; D – decay factors; δ – current window; ε – error bound; σ – support threshold

function UPDATE(x)
  $N \leftarrow N + 1$
  if $x \notin C$ **then**
    $D[x] \leftarrow \delta - 1$
  $C[x] \leftarrow C[x] + 1$
  if $N \mod \lceil \varepsilon^{-1} \rceil$ **then**
    for each $x \in C$ **do**
      if $C[x] + D[x] \leq \delta$ **then**
        delete(C, D, x)
    $\delta \leftarrow \delta + 1$

function READ(x)
  return $C[x]$

function TOP(k)
  $T \leftarrow \emptyset$
  for each $x \in C$ **do**
    if $C[x] \geq N(\sigma - \varepsilon)$ **then**
      $T \leftarrow T \cup \{(x, C[x])\}$
  $T \leftarrow \text{sort}(T, 2, \textit{descending})$
  return $T[0 : k]$

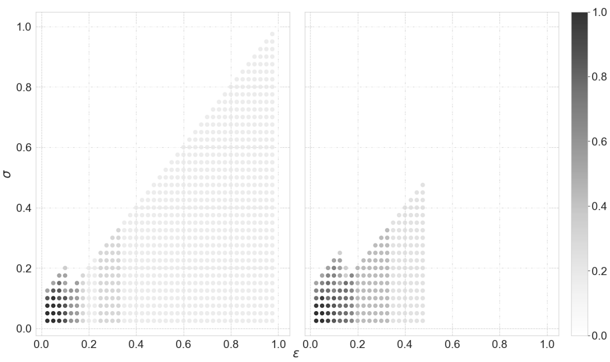

Fig. 9: Normalized Discounted Cumulative Gain at Position 10 Obtained After Running the Lossy Counter Algorithm on *The Republic* in English and Portuguese for Different Values of ε and σ.

The optimal ε and σ values, which maximize CRE, for both the Portuguese and English version of the work are depicted in Table IV. We notice that the σ is always the lowest value recorded, which suggests that the algorithm is more sensitive to changes in ε than in σ. Overall, the Lossy Counter algorithm seems to perform reasonably well in identifying heavy hitters, while reducing the number of bits required to store the data by a considerable margin.

TABLE IV: Optimal ε and σ Values Maximizing Compression-Ranking Efficiency (CRE) for English (EN) and Portuguese (PT) Using the Lossy Counter Algorithm

	k	ε	σ	**BS**	$\mathbf{NDCG_k}$	$\mathbf{CRE_k}$
EN	3	0.075	0.025	0.697	1.000	0.848
	5	0.075	0.025	0.697	1.000	0.848
	10	0.075	0.025	0.697	0.990	0.844
PT	3	0.175	0.025	0.945	0.895	0.920
	5	0.100	0.025	0.822	1.000	0.911
	10	0.100	0.025	0.822	0.870	0.846

IV. Conclusion

In conclusion, our exploration of letter frequency counting methods in literary texts highlights the significance of balancing accuracy and efficiency. With advancements in computing power, we have various algorithms at our disposal, each with its strengths and weaknesses. The Exact Counter provides precise results but may struggle with large datasets, while approximate counters like Morris and Count-Min Sketch offer efficiency at the cost of some accuracy. The choice of method depends on the specific requirements of the task. Our analysis of *The Republic* in English and Portuguese demonstrates the applicability of these methods and the introduced metrics aid in evaluating their performance. As technology evolves, optimizing counting algorithms for letter frequencies becomes crucial for processing vast amounts of textual data effectively.

References

[1] G.K. Chesterton, *Orthodoxy*, Project Gutenberg, 1908.
URL: `https://www.gutenberg.org/ebooks/16769`

[2] C.S. Lewis, *Mere Christianity*, Faded Page, 1952.
URL: `https://www.fadedpage.com/showbook.php?pid=20150620`

[3] Plato, *The Republic*, Project Gutenberg.
URL: `https://www.gutenberg.org/ebooks/1497`

[4] Wikimedia Foundation, “Pigeonhole principle”, Jan. 3 2024, Accessed on 7 Jan. 2024.
URL: `https://en.wikipedia.org/wiki/Pigeonhole_principle`

[5] Robert H. Morris, “Counting large numbers of events in small registers”, *Commun. ACM*, vol. 21, pp. 840–842, 1978.
URL: `https://api.semanticscholar.org/CorpusID:36226357`

[6] Graham Cormode and S. Muthukrishnan, “An improved data stream summary: the count-min sketch and its applications”, *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.
URL: `https://www.sciencedirect.com/science/article/pii/S0196677403001913`

[7] Gurmeet Singh Manku and Rajeev Motwani, “Chapter 31 - approximate frequency counts over data streams”, in *VLDB ’02: Proceedings of the 28th International Conference on Very Large Databases*, Philip A. Bernstein, Yannis E. Ioannidis, Raghu Ramakrishnan, and Dimitris Papadias, Eds., pp. 346–357. Morgan Kaufmann, San Francisco, 2002.
URL: `https://www.sciencedirect.com/science/article/pii/B978155860869650038X`