Identify the Most Frequent Letters using Exact Counters, Approximate Counters and Lossy Count Algorithm

Pedro Monteiro 97484
Advanced Algorithms
Course Instructor: Joaquim Madeira
DETI, Aveiro University
Aveiro, Portugal
pmapm@ua.pt

***Abstract*—The main objective of this article is to identify the number of frequent letters in text files, from the Project Gutenberg [1], following three different approaches. Using exact counters, approximate counters and finally create a lossy count algorithm. The limitations of each approximation will also be evaluated, as well as the error.**

***Keywords*—Frequent Letters, Exact Counters, Approximate Counters, Lossy Count, Absolute Error, Relative Error**

I. Introduction

This article comes within the scope of the third advanced algorithms project, where the main objective is to identify the most frequent letters.

In order to make a better and more complete analysis, three different approaches were followed.

The first approach is more direct, uses exact counters, and basically, as a letter is read, its counter changes. In the second approach, approximate counters were used, where there was a fixed probability of $\frac{1}{4}$. Finally, an approach that follows the lossy count algorithm [6] was implemented.

All these algorithms will be discussed below, as well as an analysis of the limitations and errors associated with each one.

II. Description of the Problem

Identifying the most frequent letters in a sequence of characters is a problem that often arises in various contexts, such as text analysis, data compression, and cryptography. The goal of this problem is to identify which letters (or characters) appear the most frequently.

There are several algorithms that try to solve this problem efficiently, that is, with a small memory footprint and good performance. However, they also have limitations and can produce errors under certain circumstances.

For example, some algorithms may be better suited for identifying the most frequent letters in a sequence with a high degree of noise or variability, while others may be better suited for more predictable or structured sequences and also, the fact that the most frequent letters may not be significantly more frequent than other letters, i.e., if the frequency distribution of the letters in the sequence is relatively even, it may be difficult to identify a clear majority element.

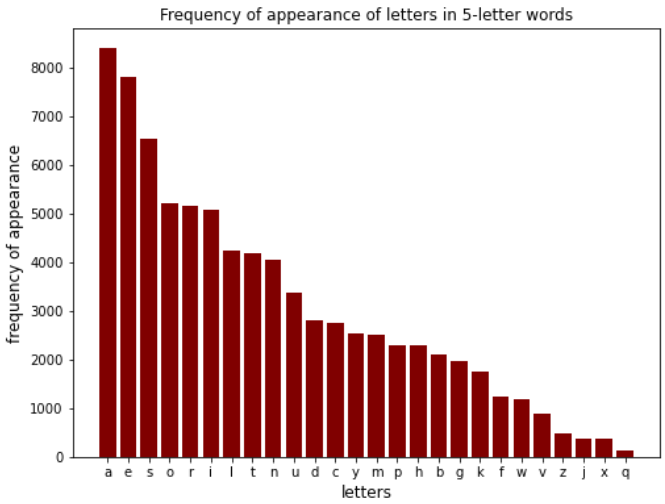

Fig. 1: Frequency of Letters in 5-letters words.

Just to exemplify, in figure 1 it's possible to observe the most frequent letters in words with five letters in the English language, according to Towards Data Science article [2]. Looking more closely, is easily observed that the vowels a, e and o stand out in the top 5 most frequent, with the consonants s and r also being present.

III. Description of the Solution

As already mentioned, in carrying out this project, 3 different approaches were used to identify the number of frequent letters in text files.

To carry out the project, a Gutenberg Project book was used. First, the book's headers were removed as well as all punctuation marks, leaving only the letters. To facilitate the identification process, all letters were converted to uppercase, with no distinction between uppercase and lowercase letters, which means, the implemented algorithms are not case sensitive.

For this, the **string** [3] library and the **isalpha()** [4] method were used. The pseudocode below represents the function that allows to get all the letters in the book.

Algorithm 1 Get Letters Uppercase

```
function GET_UPPER_LETTERS(file)
    letters ← {}
    open the text file
    text ← contents of file
    words ← text.split()

    for word in words do
        for char in word do
            if char ∉ string.punctuation ∧ char.isalpha() then
                letters.append(char.upper())
return letters
```

As can be seen from the pseudocode, only if the character is a letter and not a punctuation mark, string.punctuation, or a number, isalpha(), is added to the list that is returned. This function can be found in the utils.py file.

After obtaining the letters, the three different approaches mentioned above were then implemented, which will be discussed below.

A. Exact Counter

The first approach implemented was the method using exact counters. This is, perhaps, the simplest approach to understand, since as the algorithm reads a letter, its counter value changes.

The first thing that is done is to get all the letters, which, as already mentioned, is done using the `get_upper_letters` function.

This algorithm is very straight forward, since it is only necessary to initialize an empty dictionary and iterate over all the letters and, if the letter is already in the dictionary, the letter counter is incremented by 1, and if not, the letter is created in the dictionary with the counter value of 1.

The pseudocode below describes the behavior of the algorithm, as already explained.

Algorithm 2 Exact Counters

```
letterCounts ← {}
for letter in letters do
    if letter in letterCounts then
        letterCounts[letter] += 1
    else
        letterCounts[letter] ← 1
```

To finish this algorithm, the dictionary containing the letter count was ordered according to the frequency number of each letter, with the most frequent letters being placed first.

For testing reasons, which will be discussed later, the k most frequent letters were saved, with k ∈[3, 5, 10].

B. Approximate Counter

In the case of this approach, which uses approximate counters, a fixed probability counter of $\frac{1}{4}$ was used.

This algorithm is somewhat similar to the previous one, in that it checks whether the letter is already in the dictionary, and if it is, its counter value increases by 1, otherwise its counter becomes 1. Differs in the fact that this change on letter counters does not happen every time, as there is a condition.

The condition concerns the probability referred to at the beginning, $\frac{1}{4}$. For this, a pseudo random number is generated, using random library [5], and if the generated value is less than $\frac{1}{4}$, the value of the letter's counter is changed. Otherwise, the algorithm moves on to the next letter and the counter remains unchanged.

Algorithm 3 Approximate Counters

```
letterCounts ← {}
for letter in letters do
    if random.random() ≤ 1/4 then
        if letter in letterCounts then
            letterCounts[letter] += 1
        else
            letterCounts[letter] ← 1
```

As can be seen from the pseudocode presented above, the only difference in relation to the first approximation is the condition that verifies whether the generated number is less than the stipulated probability.

C. Lossy Count Algorithm

The last approach implemented corresponds to the lossy count streaming algorithm [6].

This algorithm is often used to identify elements in a data stream whose frequency exceeds a user-given threshold. The input to the algorithm is a list of letters, and two parameters: ϵ and σ. ϵ (epsilon) is the error bound and determines the width of the buckets. σ (sigma) is the minimum number of times that a letter must appear in the data stream for it to be considered frequent.

Smaller values of ϵ mean that the counters for each letter will be updated less frequently, resulting in the algorithm being less accurate.

The algorithm initializes a dictionary, *letter_counters*, to store the counters for each letter in the data stream. It then processes the data stream by dividing it into buckets of width w, where w is determined by ϵ. For each bucket, the algorithm increments each letter counter in the bucket and decrements all counters by 1.

After all the buckets have been processed, the algorithm checks which letters have a counter value that is greater than or equal to σ. These letters are considered to be frequent and are stored in a dictionary, frequent_letters. Finally, the algorithm returns the dictionary of frequent letters.

As already described, the pseudocode below represents the implemented lossy count algorithm:

Algorithm 4 Lossy Count

```
function LOSSY_COUNT(letters, epsilon, sigma)
    letterCounters ← {}
    frequentLetters ← {}
    for letter in letters do
        if letter ∉ letterCounters then
            letterCounters[letter] ← 0
    w ← ⌊1/epsilon ⌋
    buckets ← [letters[i:i+w] for i in range(1, len(letters)+1,
w)]
        for bucket in buckets do
            for letter in bucket do letterCounters[letter] += 1
                for letter in letterCounters do letterCoun-
ters[letter] -= 1
            for letter, value in letterCounters.items() do
                if value ≥ sigma then
                    frequentLetters[letter] ← value
return frequentLetters
```

IV. Solutions Analysis

These algorithms sometimes have some limitations and problems that can affect their performance. Many problems are associated with the algorithm not being able to handle non-letter characters and the time needed to identify all the letters. In the case of the algorithms implemented in this article, this is not a problem, since only the letters were identified.

Next, each of the implemented algorithms will be analyzed.

A. Exact Counters

This approach uses an if statement to check if a letter is already in the dictionary, which has a time complexity of O(n) in the worst case.

The complexity of this algorithm is O(n), because it uses a dictionary to store the letters and letter frequencies, and the size of the dictionary is proportional to the number of unique letters in the text file.

Since the time complexity of this approach is O(n), this means that the time taken to count the frequencies of letters is proportional to the size of the text file.

Therefore, this could be a limitation if you need to process very large text files efficiently. If the input text file is very large, this implementation may take a long time to run and may use a large amount of memory.

The fact that the letters are all converted to uppercase eliminates the possibility that the algorithm is not case sensitive and therefore counts the same letter several times, in lowercase and uppercase, and therefore there is no such limitation.

B. Approximate Counters

Similar to the previous approach, this one also presents a time complexity of O(n) which indicates that the time taken to count the approximate frequencies of letters is proportional to the size of the text file.

The main limitation of this approximation, which differs from the others, is that it is an approximate algorithm and the results may not be accurate. The accuracy of the results will depend on the size of the text file and the probability used.

In terms of computational efficiency, this implementation is less efficient than the previous one, because it uses a probability to determine if a letter should be counted, which introduces an additional step in the process. This may result in longer processing times.

C. Lossy Count

The computational efficiency and limitations of this function would depend on the size of the text file and the values of the epsilon and sigma parameters.

A potential limitation of this function is that it reads the entire text file once to initialize the dictionary of letter counters, then creates the buckets and updates the counters. This means that the time complexity of this function is at least O(n), where n is the number of characters in the text file. If the text file is very large, the function may run slowly.

The value of the epsilon parameter also affects the computational efficiency of the function. A smaller value of epsilon results in a larger value of w and therefore more buckets, which means that the counters will be updated less frequently. This can potentially improve the performance of the function, but at the cost of accuracy.

The value of the epsilon parameter also affects the computational efficiency of the function. The smaller the value of epsilon, the larger the value of w, which results in more buckets and less frequent counter updates. This may improve function performance, but at the cost of accuracy.

The sigma parameter also affects the efficiency of the function. This is because the higher the sigma value, the fewer characters are considered common/frequent and added to the common character dictionary. This may improve function performance, but at the cost of the ability of the function to accurately count the number of occurrences of all letters in the text file.

In general, in all three approaches its performance will depend on the size of the input file, with larger and more extensive files requiring more time and more resources to be processed efficiently.

In this article was used the book Nineteen Hundred? A Forecast and a Story, from Gutenberg's project [1], which contains about 227 pages.

V. Experiments

Regarding experiments, the results of each approximation for k frequent letters were saved, with k ∈[3, 5, 10], as already mentioned.

For a better and simpler visualization of the results, bar graphs were constructed for each approximation, as shown in figures 2, 3 and 6, with the help of matplotlib [7] and numpy [8] libraries. These functions are defined in the `utils.py` file.

Also, a table was created, table 1, which contains all the results obtained for each approximation.

In this section, an analysis of the absolute errors will also be made, since using the results obtained, the difference in performance of each algorithm is more easily perceived.

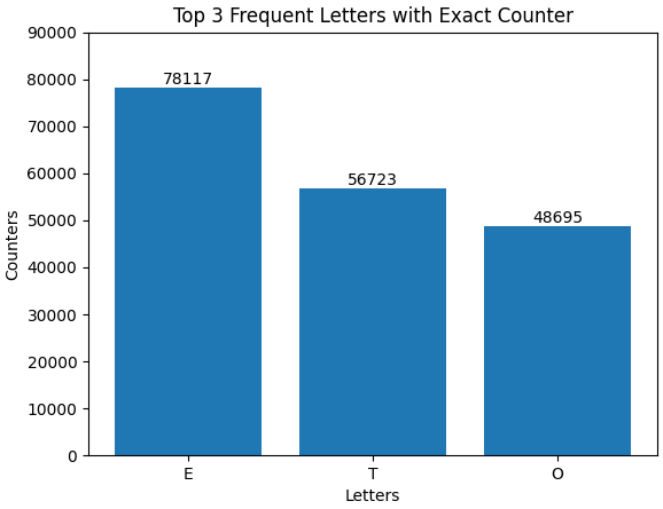

Fig. 2: Top 3 Frequent Letters Using Exact Counter.

Looking at figure 2, where the three most frequent letters in the text are represented, it is observed that the letter E is the one that appears most often, 78117, still being significantly separated from the second most frequent letter, the letter T, which appears 56723 times. Finally, the third most frequent is the letter O, which appears 48695 times.

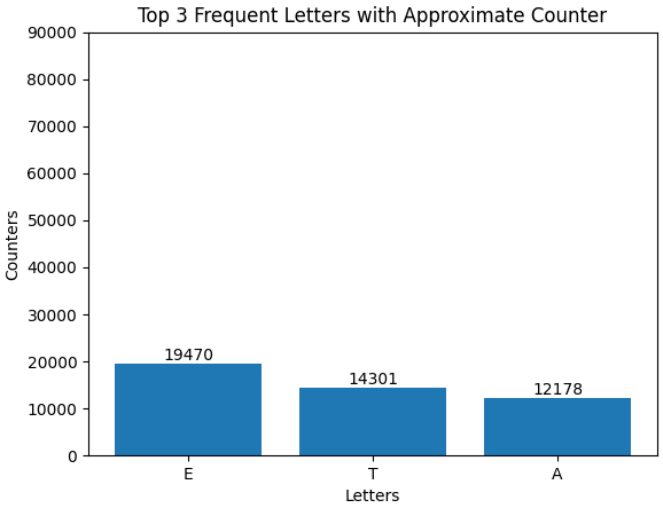

Fig. 3: Top 3 Frequent Letters Using Approximate Counter.

In figure 3 there is a great visual difference in relation to figure 2, since the same scale was purposely left. Frequency values are much lower than those obtained with exact counter. This is because a fixed probability of $\frac{1}{4}$ was used for each letter's counter to be updated.

That said, we verify that, using exact counter, the value for the most frequent letter, E, is 78117, and in the case of the approximate counter is 19470, using the probability of $\frac{1}{4}$, 78117 * $\frac{1}{4}$ = 19529, we observe that the values are similar, which confirms what was said.

Looking at figure 3 it is also easily noticeable that the 3 most frequent letters in the case of exact counter and approximate counter are not the same.

In the case of the exact counter, the three most frequent letters are E, T, O and in the case of the approximate counter are E, T, A, which differs in the last letter. This is due to the fact that a random number is generated to be compared with a probability of $\frac{1}{4}$, which means that the algorithm can sometimes count fewer letters and more others.

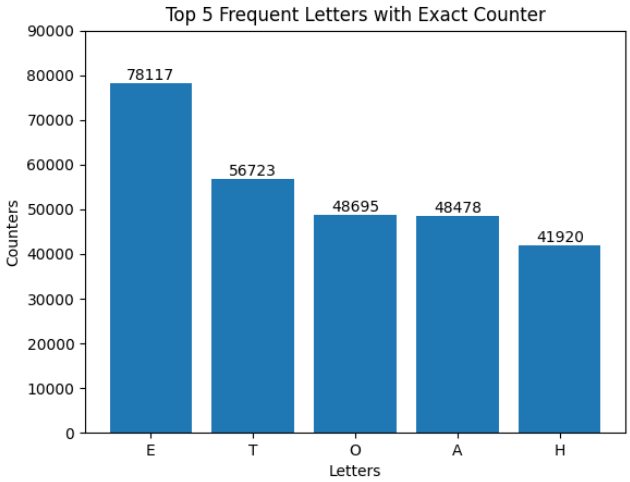

Fig. 4: Top 5 Frequent Letters Using Exact Counter.

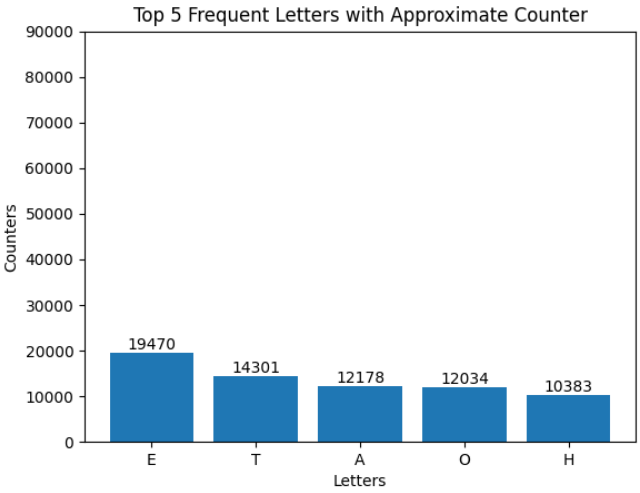

Fig. 5: Top 5 Frequent Letters Using Approximate Counter.

To reinforce the previous idea a little, figures 4 and 5 represent the top 5 most frequent letters in the case of exact counter and approximate counter, respectively.

As can be easily observed, the most frequent letters in one case and the other are different.

As already mentioned, this is due to the addition of random in decision making, since only if the value is less than the probability, in this case $\frac{1}{4}$, the letter's counter change.

Added to this, is the fact that the values for the letters A and O, which are found in different order, are quite similar, which leads the algorithm to sometimes count one letter more than the other.

Through the results obtained for the approximate counter and compared with the true values, obtained in the exact counter, it is possible to calculate the absolute error and the relative error.

The absolute error [9] corresponds to the difference between the calculated value and the actual value, and therefore, in the case of the letter E, for example,

$$78117 * \frac{1}{4} - 19470 = 59$$

Note that the actual total value of the frequency of the letter E was multiplied by the probability associated with the algorithm, $\frac{1}{4}$.

Looking now at the relative error [10] associated with the frequency of the letter E, which is obtained by dividing the absolute error by the actual value, expressed as a percentage

$$\frac{59}{78117 * \frac{1}{4}} = 0.3\%$$

The error obtained is relatively low, which indicates a good performance of the algorithm, but only 1/4 of the total number of letters was considered.

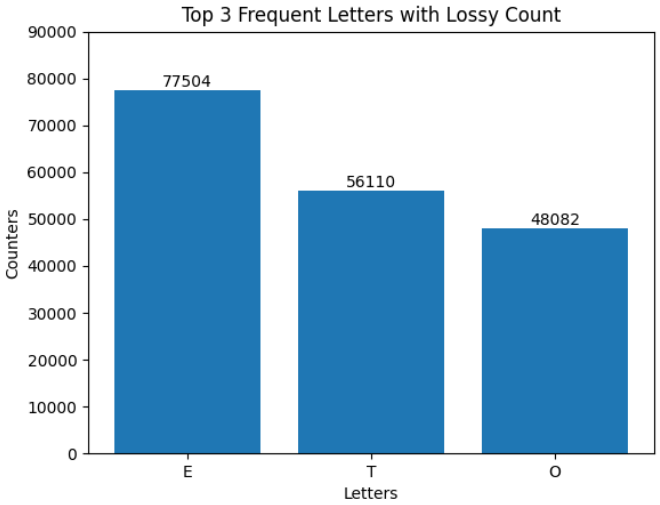

Fig. 6: Top 3 Frequent Letters Using Lossy Count.

In the case of the lossy count algorithm, shown in Figure 6, where $\epsilon = 0.001$ and $\sigma = 20000$ were used, values are quite similar to those obtained with exact counter. This indicates that the lossy count algorithm has a good performance, relatively correctly counting the frequency of the letters.

Also the top 3,5 and 10 most frequent letters are the same.

Regarding the errors associated with the lossy count algorithm, a grouped bar chart was constructed to demonstrate the differences between the values obtained.

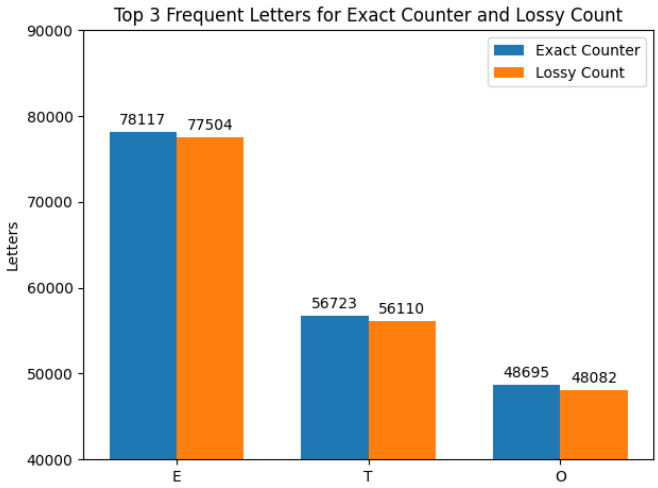

Fig. 7: Values Obtained Comparison Between Exact Counter and Lossy Count.

As shown in figure 7, the frequency value for the letter E for the exact counter is 78117 and for the lossy count it is 77504.

The difference between the two corresponds to the absolute error [9], i.e.

$$78117 - 77504 = 613$$

Regarding the relative error [10] associated with the letter E, expressed as a percentage, the absolute error, 613, was divided by the real value, that is, the one obtained by the exact counter. That said,

$$\frac{613}{78117} = 0.79\%$$

As mentioned at the beginning of this section, a table with all the results obtained for each value of k is presented below.

Results with each approach			
Letter	Exact Counter	App. Counter	Lossy Count
E	78117	19470	77504
T	56723	14301	56110
O	48695	12034	48082
A	48478	12178	47865
H	41920	10383	41307
N	41451	10372	40838
I	38731	9686	38118
S	37375	9199	36762
R	35616	8966	35003
D	26803	6796	26190

TABLE I: Results Obtained by each Approach

As already mentioned and just to remind, the 10 most frequent letters for the exact counter and the lossy count are the same, differing for the case of the approximate counter where the letters A and O change positions.

VI. Conclusion

To conclude this article, there were implemented three different approaches that can be used to identify frequent

letters in text files. The first approach is to use a simple loop to count the number of occurrences of each letter in the file. The second approach is to use a probability to count the number of occurrences. The third approach corresponds to the well-known lossy count streaming algorithm [6].

Each of these approaches has its own advantages and trade-offs, and the most suitable approach will depend on the specific requirements and constraints of the task at hand.

ACKNOWLEDGMENT

I would like to thank Professor Joaquim Madeira, regent of AA course, for being available to answer questions to discuss the project.

REFERENCES

[1] Project Gutenberg https://www.gutenberg.org/
[2] Most Frequent Letters in 5-letter words https://towardsdatascience.com/a-frequency-analysis-on-wordle-9c5778283363
[3] Python String Library https://docs.python.org/3/library/string.html
[4] isalpha() Method https://docs.python.org/3/library/stdtypes.html
[5] Random Library https://docs.python.org/3/library/random.html
[6] Lossy Count Algorithm https://en.wikipedia.org/wiki/Lossy_Count_Algorithm
[7] Matplotlib Docs https://matplotlib.org/
[8] NumPy Docs https://numpy.org/
[9] Absolute Error https://mathworld.wolfram.com/AbsoluteError.html
[10] Relative Error https://mathworld.wolfram.com/RelativeError.html