3rd Project: Most Frequent and Less Frequent Words

Advanced Algorithms

José Miguel Costa Gameiro, 108840

Abstract - **This document presents a brief study conducted within the Advanced Algorithms course to analyse different strategies that identify frequent words in text files, like books, using three different methods: exact counters, approximate counters (with a decreasing probability counter of 1 / sqrt(2)^k) and to evaluate the quality of estimates regarding the exact counts.**

I. INTRODUCTION

The analysis of word frequency in textual data is a cornerstone of many computational linguistics and data analysis tasks, offering insights into language patterns, authorship styles, and information retrieval systems. This report investigates the identification of frequent words in text, specifically books, using a set of different methods. The objective is to evaluate how different approaches estimate exact word counts and to assess their accuracy, computational efficiency, and practical limitations.

To achieve this, three distinct methods were developed and tested:

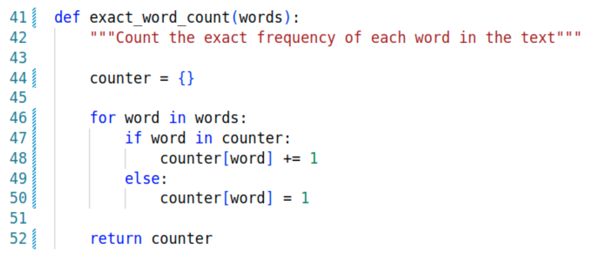
- **Exact counters:** These methods compute precise word frequencies using traditional counting techniques;
- **Approximate Counters:** These algorithms provide estimations of word counts, often trading off exactness for speed and reduced resource storage usage;
- **Algorithms for data streams:** these methods are used to identify frequent items in continuous, large-scale data streams, where storage and computational constraints necessitate innovative solutions.

The evaluation includes an analysis of the computational performance and accuracy of these approaches. Metrics such as absolute and relative errors (including lowest, highest, and average values) are used to compare the methods. Additionally, the report examines whether the most and least frequent words identified by each approach align in relative order and frequency distribution. Special attention is given to comparing word frequency patterns across translations of the same book in different languages, shedding light on linguistic and translational nuances.

II. EXACT WORD COUNTER ALGORITHM

Exact counters are designed to compute precise word frequencies in a given text. These algorithms operate by iterating through the list of words and maintaining a complete count for each unique word encountered. This approach guarantees accuracy but may require substantial memory for large datasets, particularly when the vocabulary is extensive.

The function **exact_word_count**, contains the implementation of a simple exact word counter, where it iterates over a list containing all the words, and checks if a word is in a dictionary created to store the word frequency, if it isn't there, it means the word is new and so it adds a new entry to the dictionary, and sets its frequency to 1, if the word is already in the dictionary, then it adds one to the word thats in the dictionary.



```python
def exact_word_count(words):
    """Count the exact frequency of each word in the text"""

    counter = {}

    for word in words:
        if word in counter:
            counter[word] += 1
        else:
            counter[word] = 1

    return counter
```

Fig. 1 - Exact word count function developed.

A. Advantages

1. **High Accuracy:** The method computes the exact frequency of every word in the input text, ensuring no approximation errors;
2. **Simplicity and Readability:** Using structures like ***Python's Counter***, the algorithm is straightforward to implement and understand;

3. **Deterministic Results:** The output is consistent and reproducible, given the same input, which is essential for verification and debugging;
4. **Versatile Use Cases:** Exact counters can handle various applications, including small-scale text analysis, detailed linguistic studies, and building ground truth datasets for comparisons with other algorithms.

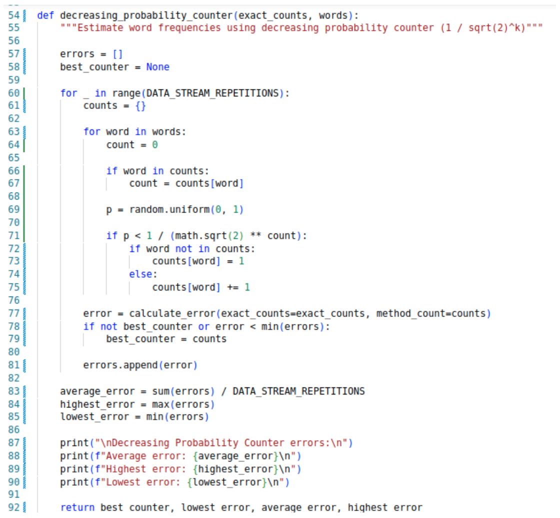
B. Disadvantages

1. **High Memory Usage:** The algorithm requires additional space to store the counts of each unique word. For datasets with a large vocabulary, the memory requirements can grow significantly, making it impractical for systems with limited resources;
2. **Unsuitable for Streaming Data:** Exact word counting requires the entire dataset to be available upfront. It cannot efficiently process data streams where data arrives incrementally or continuously;
3. **Limited Scalability for Rare Words:** While the algorithm provides precise counts, its utility for rare elements is limited to exact values. It cannot provide probabilistic estimates or insights about rare patterns without processing the entire dataset;
4. **Inefficient for Large Datasets:** With a time complexity that scales linearly with the size of the dataset, the algorithm can become computationally expensive for very large datasets, particularly when word frequency analysis is only one part of a larger workflow;
5. **Not Designed for Distributed Systems:** The algorithm assumes all data is processed on a single machine. In distributed environments, coordinating exact counts across multiple nodes introduces significant complexity and inefficiency.

III. Decreasing probability of 1 / sqrt(2)^k Counter

Approximate counting algorithms trade off precision for computational efficiency and reduced memory usage. This method employs a **decreasing probability mechanism**, where likelihood of updating the count diminishes as the count increases. Specifically, the update probability follows the function 1 / sqrt(2) ^ k, where k is the current word count.

The function **decreasing_probability_counter,** contains the algorithm developed for the approximate counter with decreasing probability of 1 / sqrt(2)^k. It maintains a dictionary to track the approximate frequency of each word in the input list. For each word, the current count is retrieved or initialized to zero. A random number p between 0 and 1 is then generated, and the word's count is incremented with a probability inversely proportional to $\sqrt{2}^{k}$, where k is the current count of the word. This mechanism ensures that words with higher existing counts are less likely to increment further.



```python
def decreasing_probability_counter(exact_counts, words):
    """Estimate word frequencies using decreasing probability counter (1 / sqrt(2)^k)"""

    errors = []
    best_counter = None

    for _ in range(DATA_STREAM_REPETITIONS):
        counts = {}

        for word in words:
            count = 0

            if word in counts:
                count = counts[word]

            p = random.uniform(0, 1)

            if p < 1 / (math.sqrt(2) ** count):
                if word not in counts:
                    counts[word] = 1
                else:
                    counts[word] += 1

        error = calculate_error(exact_counts=exact_counts, method_count=counts)
        if not best_counter or error < min(errors):
            best_counter = counts

        errors.append(error)

    average_error = sum(errors) / DATA_STREAM_REPETITIONS
    highest_error = max(errors)
    lowest_error = min(errors)

    print("\nDecreasing Probability Counter errors:\n")
    print(f"Average error: {average_error}\n")
    print(f"Highest error: {highest_error}\n")
    print(f"Lowest error: {lowest_error}\n")

    return best counter, lowest error, average error, highest error
```

Fig. 2 - Approximate word counter algorithm developed.

A. Advantages

1. **Reduced Memory Usage:** By using approximate counts instead of storing exact frequencies, this algorithm requires less memory, making it suitable for large datasets with extensive vocabularies.
2. **Improved Computational Efficiency:** The probabilistic update mechanism reduces the frequency of count updates, lowering the computational overhead compared to exact counting methods;
3. **Scalable for Large Datasets:** This method scales better than exact counters for massive datasets, where memory and processing time are significant constraints;
4. **Bias Toward High-Frequency Words:** The algorithm inherently prioritizes capturing higher-frequency words with more accuracy, which is beneficial in applications focused on identifying dominant patterns or trends;
5. **Suitability for Constrained Environments:** The lower memory and computational demands make this algorithm an excellent choice for environments with limited resources, such as embedded systems or real-time analytics.

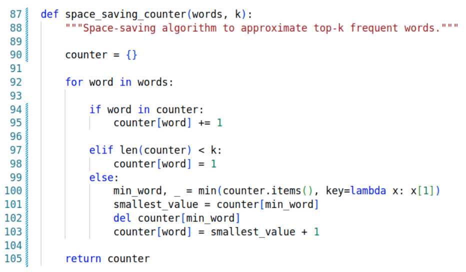
B. *Disadvantages*

1. **Loss of Accuracy:** The algorithm provides only approximate counts, which may deviate significantly from actual values, particularly for low-frequency words;
2. **Error Magnitude Grows for Rare Words:** Rare words are more prone to underestimation because their counts are incremented with a very low probability as the algorithm progresses.
3. **Non-deterministic Results:** Since the algorithm relies on random probability, results can vary across different runs on the same dataset, potentially affecting reproducibility;
4. **Ranking and Distribution Bias:** The approximation can distort the ranking of words with similar frequencies, particularly when counts are close, leading to inaccuracies in the frequency distribution;
5. **Complexity in Interpretation:** The probabilistic nature of the algorithm makes it harder to interpret results directly compared to deterministic exact counters, requiring additional effort to understand and validate the outputs.

IV. Space-Saving Count Algorithm

The space-saving algorithm is a specialized method for tracking the top n most frequent words in a dataset while minimizing memory usage. Instead of maintaining a count for every unique word, this algorithm keeps track of only n words at a time, dynamically replacing less frequent words when necessary. This makes it particularly well-suited for environments where memory is constrained.

The space-saving counter algorithm efficiently tracks the top k most frequent words in a data stream using a fixed-size dictionary. For each word in the input list, its count is incremented if it is already in the dictionary. If the word is not present and the dictionary has fewer than k entries, it is added with a count of one. However, if the dictionary is full, the word with the smallest count is replaced by the new word, and the count of the new word is set to the count of the recently removed word plus 1.



```python
def space_saving_counter(words, k):
    """Space-saving algorithm to approximate top-k frequent words."""

    counter = {}

    for word in words:

        if word in counter:
            counter[word] += 1

        elif len(counter) < k:
            counter[word] = 1
        else:
            min_word, _ = min(counter.items(), key=lambda x: x[1])
            smallest_value = counter[min_word]
            del counter[min_word]
            counter[word] = smallest_value + 1

    return counter
```

Fig. 3 - Space saving counter algorithm developed

A. *Advantages*

1. **Memory Efficiency:** The algorithm tracks only n at a time, making it highly efficient in memory usage regardless of the dataset size
2. **Scalability for Large Datasets:** Unlike exact counting, the algorithm is well-suited for processing large datasets where tracking every unique word is infeasible;
3. **Efficient Identification of Top Words:** By focusing on the n most frequent words, it avoids unnecessary computations for less frequent words;
4. **Dynamic Updates:** The heap structure allows efficient replacement of the least frequent words, maintaining the accuracy of the top n results.

B. *Disadvantages*

1. **Approximation Errors for Low-Frequency Words:** Words not included in the top n tracking may have their counts underrepresented or ignored altogether;
2. **Dependence on n:** The accuracy of the results is highly sensitive to the chosen value of n. A poorly chosen n may exclude important words;
3. **Loss of Global Perspective:** The algorithm does not provide frequency information for words outside the top n words, which may be critical in some analyses;
4. **Additional Computational Overhead:** The use of a heap requires frequent reorganization, adding a minor computational overhead compared to simpler data structures.

V. Study Environment

To analyse the accuracy of each algorithm and compare the different algorithms developed, the book "*Don Quixote*" was used (obtained through the Project Gutenberg). It used 4 different versions of the book, in distinct languages, English, German, Hungarian and Spanish. Also a list containing stop words was used for each language to remove stop words, focusing only on important words. For the space saving count algorithm, it was tested also with multiple values for k, each are 20, 35 and 100. In the approximate counter, it executes the algorithm the number that is stored in the **DATA_STREAMS_REPETITIONS** variable (10).

For each approach, the program computes the absolute and relative error of the approximated counts compared to the exact counts. The absolute error is the difference between the exact and approximated counts for each word, while the relative error is the absolute error divided by the exact count.

VI. Results

A. English Version

Nº	Exact	Approximate	Space Saving
1	said - 1337	sancho - 20	said - 1446
2	quixote - 1122	said - 20	sancho - 1224
3	sancho - 953	quixote - 19	quixote - 1157
4	one - 877	thou - 17	thou - 1100
5	would - 764	go - 16	see - 1087
6	thou - 733	one - 16	one - 1086
7	say - 517	would - 16	eyes - 1085
8	may - 484	see - 15	cart - 1085
9	without - 462	many - 15	devil - 1085
10	see - 443	good - 15	dulcinea - 1084

Table 1 - Top 10 words most frequent words for the English version of Don Quixote using the 3 different approaches, exact, approximate and space Saving with k = 100

The absolute and relative errors for the decreasing probability can be analysed through the following charts:

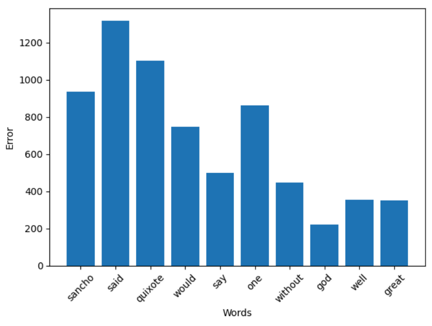


Fig. 4 - Absolute errors for the 10 most frequent words using the approximate counter

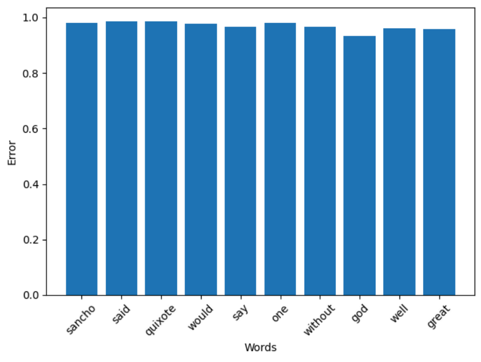


Fig. 5 - Relative errors for the 10 most frequent words using the approximate counter

The performance of the space-saving count for different k values (20, 35, 100) can be seen in the table below, regarding the absolute and relative errors:

Error	k = 20	k = 35	k = 100
Min Absolute	4565.0	1988.0	35.0
Avg. Absolute	5302.5	2923.03	958.53
Max Absolute	5442.0	3109.0	1082.0
Min Relative Error	5.205	1.772	0.03119
Avg Relative	607.63	313.84	149.026
Max Relative Error	5442.0	3109.0	1082.0

Table 2 - Relative and Absolute errors with different values for k with the English version of Dom Quixote

The absolute and relative errors for the space-saving count, with k = 100, are presented in the following charts:

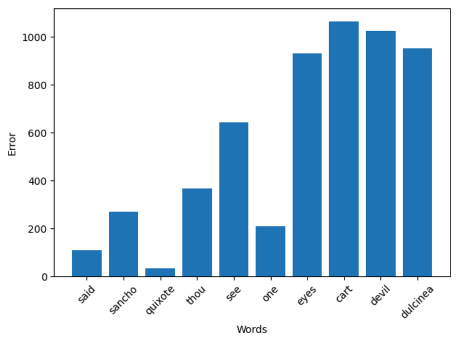


Fig. 6 - Absolute errors for the 10 most frequent words using the space saving counter with k = 100

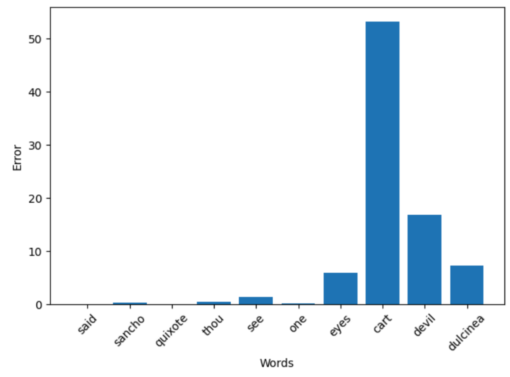


Fig. 7 - Relative errors for the 10 most frequent words using the space saving counter with k = 100

B. Spanish Version

Nº	Exact	Approximate	Space Saving
1	don - 2714	don - 20	don - 2714
2	quijote - 2241	señor - 20	sancho - 2417
3	sancho - 2174	si - 20	quijote - 2268
4	si - 1968	ser - 18	si - 1988
5	dijo - 1808	sancho - 18	dijo - 1895
6	tan - 1245	bien - 18	sobrina - 1824
7	señor - 1065	hacer - 18	muerte - 1823
8	así - 1065	dijo - 18	mal - 1823
9	respondió - 1063	quijote - 17	largo - 1823
10	ser - 1059	tan - 17	hacer - 1823

Table 3 - Top 10 words most frequent words for the Spanish version of Don Quixote using the 3 different approaches

The absolute and relative errors for the decreasing probability can be analysed through the following charts:

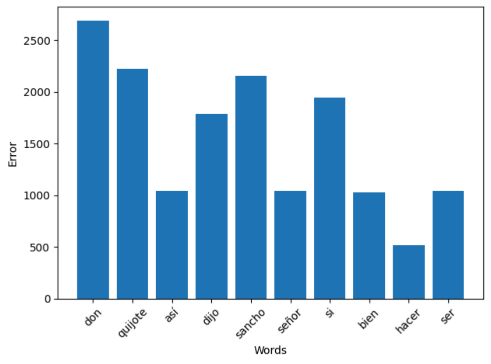


Fig. 8 - Absolute errors for the 10 most frequent words using the approximate counter

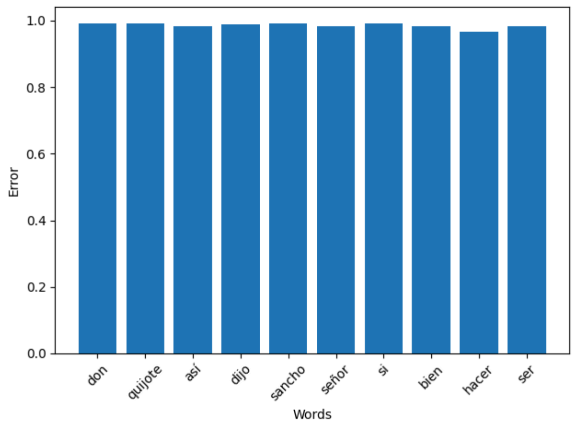


Fig. 9 - Relative errors for the 10 most frequent words using the approximate counter

The performance of the space-saving count for different k values (20, 35, 70) can be seen in the table below, regarding the absolute and relative errors:

Error	**k = 20**	**k = 35**	**k = 70**
Min Absolute	6505.0	2568.0	0.0
Avg. Absolute	8843.0	4985.74	1618.52
Max Absolute	9216.0	5267.0	1821.0
Min Relative Error	2.397	0.946	0.0
Avg Relative	524.113	606.872	213.450
Max Relative Error	3072.0	5267.0	1821.0

Table 4 - Relative and Absolute errors with different values for k with the Spanish version of Dom Quixote

The absolute and relative errors for the space-saving count, with k = 100, are presented in the following charts:

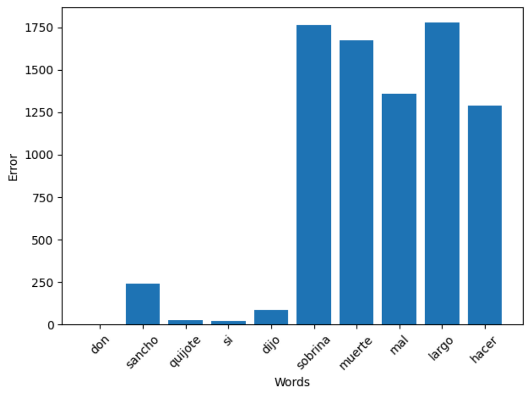


Fig. 10 - Absolute errors for the 10 most frequent words using the space saving counter

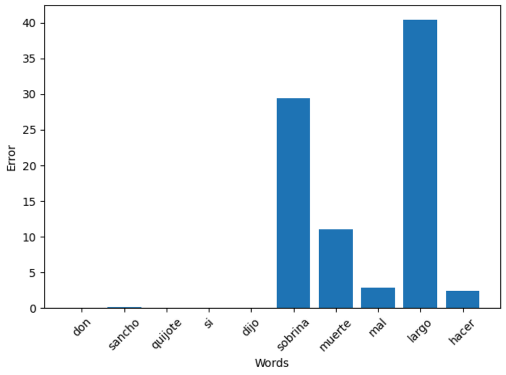


Fig. 11 - Relative errors for the 10 most frequent words using the space saving counter

C. *German Version*

Nº	Exact	Approximate	Space Saving
1	en - 3373	en - 22	en - 3373
2	de - 2853	de - 21	de - 2853
3	van - 1440	dat - 20	van - 1440
4	te - 1305	zijn - 19	te - 1306
5	dat - 1303	het - 19	dat - 1303
6	een - 1253	een - 18	een - 1253
7	hij - 1250	hij - 18	hij - 1250
8	het - 1183	op - 18	het - 1183
9	op - 987	te - 18	ik - 1001
10	ik - 976	sancho - 18	op - 990

TABLE 5 - TOP 10 WORDS MOST FREQUENT WORDS FOR THE GERMAN VERSION OF DON QUIXOTE USING THE 3 DIFFERENT APPROACHES

The absolute and relative errors for the decreasing probability can be analysed through the following charts:

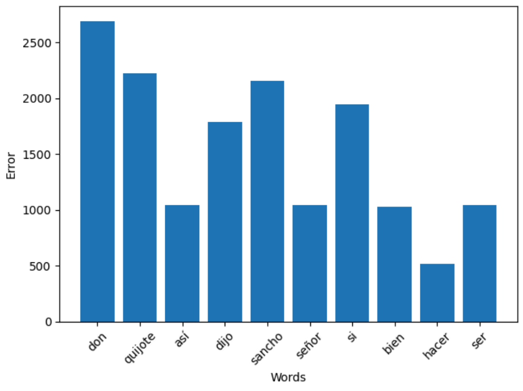


Fig. 12 - Absolute errors for the 10 most frequent words using the approximate counter

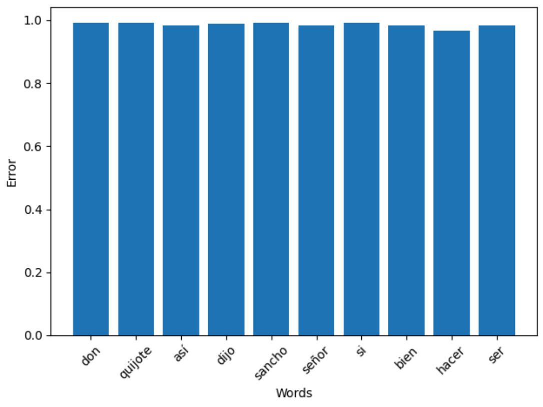


Fig. 13 - Relative errors for the 10 most frequent words using the approximate counter

The performance of the space-saving count for different k values (20, 35, 100) can be seen in the table below, regarding the absolute and relative errors:

Error	k = 20	k = 35	k = 100
Min Absolute	343	0.0	0.0
Avg. Absolute	3060.2	1604.486	456.74
Max Absolute	3697	2052	637
Min Relative Error	0.102	0.0	0.0
Avg Relative	403.798	283.644	123.474
Max Relative Error	3697.0	2052.0	637.0

TABLE 6 - RELATIVE AND ABSOLUTE ERRORS WITH DIFFERENT VALUES FOR K WITH THE GERMAN VERSION OF DOM QUIXOTE

The absolute and relative errors for the space-saving count, with k = 100, are presented in the following charts:

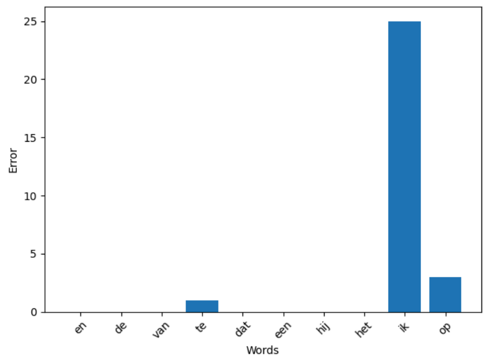


Fig. 14 - Absolute errors for the 10 most frequent words using the space saving counter

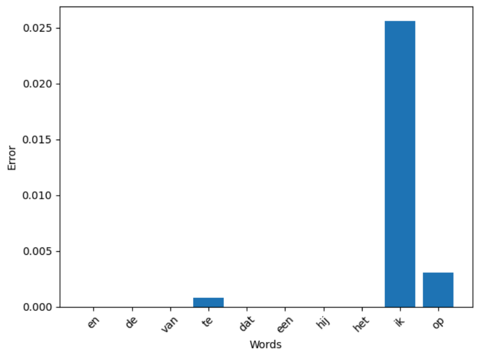


Fig. 15 - Relative errors for the 10 most frequent words using the space saving counter

D. Hungarian Version

Nº	Exact	Approximate	Space Saving
1	is - 2129	is - 20	is - 2129
2	don - 1293	quijote - 19	don - 1299
3	sancho - 1164	mind - 18	sancho - 1231
4	quijote - 1042	ő - 18	quijote - 1049
5	ha - 884	don - 18	ha - 934
6	se - 500	sancho - 17	őket - 867
7	ő - 486	ha - 16	szívből - 867
8	mind - 376	viszonzá - 15	akarom - 866
9	lovag - 353	ember - 14	néki - 866
10	viszonzá - 290	hozzá - 14	végrendeletem - 866

Table 7 - Top 10 words most frequent words for the Hungarian version of Don Quixote using the 3 different approaches

The absolute and relative errors for the decreasing probability can be analysed through the following charts:

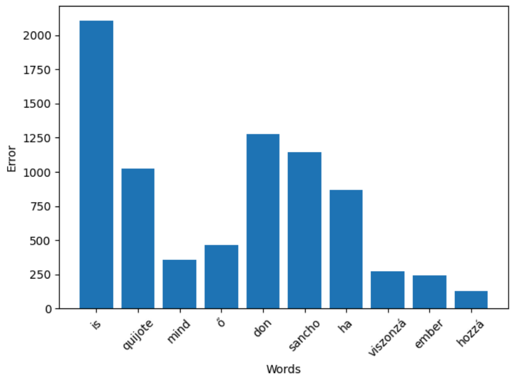


Fig. 16 - Absolute errors for the 10 most frequent words using the approximate counter

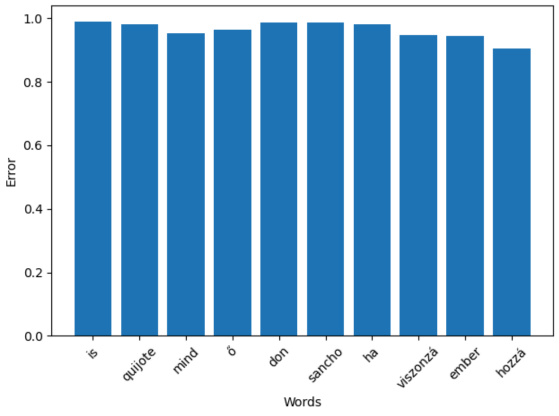


Fig. 17 - Relative errors for the 10 most frequent words using the approximate counter

The performance of the space-saving count for different k values (20, 35, 100) can be seen in the table below, regarding the absolute and relative errors:

Error	k = 20	k = 35	k = 100
Min Absolute	2316.0	413	0.0
Avg. Absolute	4266.75	2431.086	798.46
Max Absolute	4443.0	2539.0	865.0
Min Relative Error	1.0878	0.194	0.0
Avg Relative	1742.011	832.163	317.592
Max Relative Error	4443.0	2539.0	865.0

Table 8 - Relative and Absolute errors with different values for k with the Hungarian version of Dom Quixote

The absolute and relative errors for the space-saving count, with k = 100, are presented in the following charts:

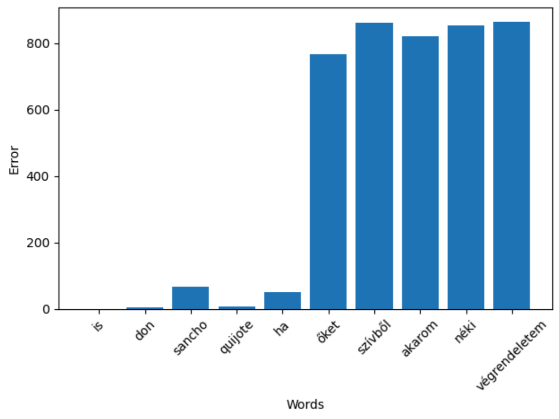


Fig. 18 - Absolute errors for the 10 most frequent words using the space saving counter for the hungarian version of Don Quixote

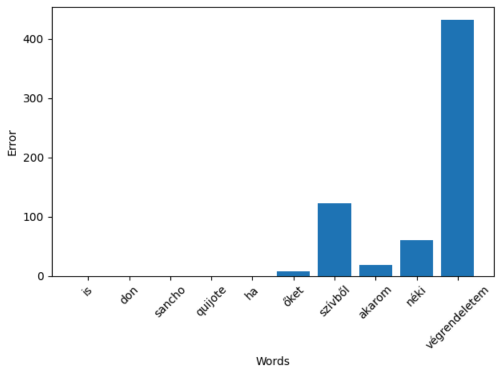


Fig. 19 - Relative errors for the 10 most frequent words using the space saving counter

VI. Conclusion

With the results obtained we can conclude that:

- The **approximate counter with a decreasing probability of $\frac{1}{\sqrt{2}^k}$** is highly inaccurate, yielding word counts significantly lower than their exact values. While it may correctly identify the relative order of the most frequent words (e.g., first, second, third place), the actual frequencies are notably understated;
- **Approximate counting** represents a trade-off between accuracy and memory efficiency. A fixed probability approach could potentially offer more accurate results without excessive memory usage. Nonetheless, this method proves useful for identifying the most frequent words in large text data streams, as it scales efficiently and requires considerably less memory compared to exact counting;
- The space-saving counter demonstrates a better precision when compared to the approximate counter;
- Absolute errors for the space-saving counter are smaller than the approximate one as we can see in the bar charts of the top 10 most frequent words in all of the languages;
- If we analyse the tables **3**, **5** and **7** we can see that the space saving counter got word counts exactly right. This can also be proven by looking at the tables **4**, **6** and **8** where we can see that the **Minimum Absolute and Relative Error** found was 0, meaning that at least one word add the same count has the exact counter obtained;
- The relative errors for the space saving algorithm some times can go over 1, and that's because sometimes, space saving algorithm associates to a word a high count, that belonged to the less frequent word, and in the exact counter the word can actually have a frequency of 1, so the error will give large values;
- For the space saving algorithm, in the tables **2, 4, 6** and **8** the values of the relative and absolute errors decrease when the value of k is increased. This means that the size of the data structure determines the accuracy of the estimates. A larger fixed-size will result in more accurate estimates, but will require more memory. A smaller fixed-size will result in less accurate estimates, but will require less space. By analysing all the top 10 most frequent words we can see that when k = 100, the algorithm can estimate the word frequency correctly for some words.

VII. References

[1] Project Gutenberg, https://www.gutenberg.org/
[2] Optimal Bounds for Approximate Counting https://www.cs.princeton.edu/~hy2/files/approx_counting.pdf
[3] Approximate counting algorithm: https://en.wikipedia.org/wiki/Approximate_counting_algorithm
[4] Natural Language Toolkit (nltk): https://www.nltk.org/
[5] Space Saving Algorithm: https://www.vldb.org/pvldb/vol15/p1215-zhao.pdf