APPROXIMATE COUNTING

Nooruddin
Department of Computer Engineering, University of Aveiro, Portugal

***Abstract*—In recent times, we have experienced challenges in data handling due to the growth of the internet, as people are uploading large amounts of data. Researchers are attempting to analyze various techniques to remove duplication and redundancy in data so that it can be used for research purposes. Removing redundancy requires counting each duplicated entry of an instance, and counting occurrences in such large datasets poses a significant challenge for researchers. This issue has several conditions that necessitate an efficient algorithm to address.**

The Morris Approximate Counting Algorithm, introduced by R. Morris, offers a solution for obtaining approximate counts through the use of compact counters. This algorithm is particularly advantageous for maintaining approximate counts in situations involving large numbers while demanding minimal storage resources. Its applicability extends to statistical data collection for various events and scenarios requiring data compression. Noteworthy for its robust convergence properties, the Morris Approximate Counting Algorithm adeptly navigates the tradeoffs between computational complexity and counting accuracy.

In this report, we have given an overview of the key aspects of the Approximate Counting algorithm and its implementation respectively. Furthermore, we have run the algorithm in terms of (1) exact counter, (2) approximate counter with fixed probability, and (3) approximate counter with decreasing probability.

***Index Terms*—Approximate Counting algorithm, Exact Counter, Probability.**

I. Introduction

Recently, we have encountered difficulties in managing data due to the expansion of the internet, with individuals uploading substantial volumes of data. Scholars are actively exploring diverse methodologies to eliminate duplication and redundancy in data for research purposes. The process of removing redundancy entails tallying each repeated instance, presenting a notable hurdle for researchers when dealing with extensive datasets. This challenge is underscored by various conditions that underline the need for an effective algorithmic solution.

An approximate counting algorithm is a method for tallying a large number of events using minimal memory. It was created by Robert Morris during his time at Bell Labs in 1978. The idea originated from his need to count numerous events within an 8-bit register. Typically, an n-bit register can count up to $2^n - 1$ values, meaning an 8-bit register can only tally 255 events. Since Morris was interested in approximate counts, he designed a probabilistic algorithm that provides a measurable margin of error. This algorithm serves as an early example of a sketching algorithm, compressing a streaming data set for efficient querying [1].

In statistical analysis, counting plays a pivotal role, yet it becomes challenging when dealing with vast datasets, networking, and machine learning. The storage and retrieval of extensive data demand significant memory resources, leading to performance slowdowns. To address this issue, approximate counting methods, such as Morris counting, are employed. These methods alleviate memory requirements, albeit introducing a slight margin of error in accuracy. Notably, for large counts, this error is negligible, rendering it inconsequential in practical terms.

In this report, we've provided a summary of the main components of the Approximate Counting algorithm and explained how it is implemented. Additionally, we executed the algorithm using three approaches: (1) exact counting, (2) approximate counting with a fixed probability, and (3) approximate counting with a decreasing probability.

Contributions of the Report:

- **Exact Counter:** We conduct a detailed examination of the performance of the exact counter for solving approximate counting.
- **Approximate Counter with Fixed Probability (1/2):** We conduct a detailed examination of the performance of the Approximate Counter with Fixed Probability (1/2) for solving approximate counting.
- **Approximate Counter with Decreasing Probability ($\frac{1}{\sqrt{3}^k}$):** We conduct a detailed examination of the performance of the Approximate Counter with Decreasing Probability ($\frac{1}{\sqrt{3}^k}$) for solving approximate counting.
- **Experimental Evaluation:** A series of computational experiments is undertaken to evaluate all three types of counters: (1) exact counter, (2) approximate counter with fixed probability, and (3) approximate counter with decreasing probability.
- **Practical Implications:** We implemented three types of counters: (1) exact counter, (2) approximate counter with fixed probability, and (3) approximate counter with decreasing probability.

The subsequent sections of this article are structured as; Literature Review (Section II) provides details on why counting is important. Approximate Counting Algorithm(Section II) provides details of the Approximate Counting Algorithm, and Morris's algorithm (Section IV) offers a brief description of Morris's algorithm for Approximate Counting. In Methodology (Section V) you will find details related to the methodology where all 3 types of counters are explained. Simple Counter Implementation(Section VI) explains how we can implement a simple counter. Tasks Implementation(Section VII) actual implementation of all three counters with results. Counters Performance Comparison (Section VIII) provides a performance comparison of counters. Finally, the conclusion (Section VIII) concludes by summarizing the contributions of the study.

II. Why Counting Problems?

Counting and sampling play crucial roles in statistical physics, particularly in real-world mathematical models like stochastic programming in optimization. Counting issues, reflecting integration, are essential in these models. Sampling,

```
(ldm) noor@Noor:~/UA_Local_Proejcts/UA_Projects/DDA2$ python task5.py
[#]
Counting Tests, 100 trials
[#]
testing 1,000, a = 30, 10% error
passed
[#]
testing 12,345, a = 10, 10% error
passed
[#]
testing 222,222, a = 0.5, 20% error
passed
```

Fig. 1: Example run of Approximate Counting Algorithm

a statistical approach closely related to counting, is commonly used. However, challenges arise in sampling contingency tables. In statistical physics, discrete counting and sampling problems are prevalent in models such as Ising and Potts models, often necessitating approximations or specialized approaches like Morris' method. Overall, addressing these counting and sampling issues is vital for a comprehensive understanding of complex systems in various scientific disciplines.

III. Approximate Counting Algorithm

In this instance, let's revisit the initial query: How far can one count on their fingers using the rough counting method? According to the formula, with a set parameter of $a = 30$ and 10 bits, the expected counting limit should be 1.1×10^{16}. However, what occurs during the actual experiment? Since we don't have tangible objects to count, we will simulate the counting process using a while loop that continues until our bitstring reaches 1023 (2^{10}).

Algorithm 1: Approximate Counting Algorithm

Input : n_{items}, a // Number of items and scaling parameter

Output: $n(v, a)$ // Approximate count

$v \leftarrow 0$;

for $i \leftarrow 1$ **to** n_{items} **do**

  $v \leftarrow \text{increment}(v, a)$;

return $n(v, a)$;

A. Results

The Figure 1 results are from the Python script `approximatecounter.py` and **Algorithm [1]** executed successfully by performing three counting tests, each with 100 trials. In the first test, simulating counting to 1,000 with a scaling parameter a set to 30 and allowing a 10% error, the test passed. Similarly, in the second test, counting up to 12,345 with a equal to 10 and a 10% error margin, the test also passed. Lastly, in the third test, counting to 222,222 with a set to 0.5 and permitting a 20% error, the test passed as well. Overall, the simulation results indicate that the approximate counting algorithm performs as expected within the specified error thresholds for the given parameters.

IV. Morris's Algorithm for Approximate Counting

The Morris Counter algorithm is a probabilistic algorithm used for approximate counting. It is based on the following principles:

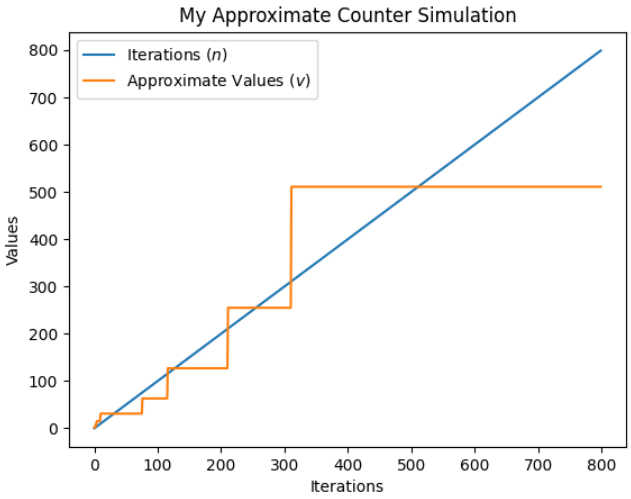

Fig. 2: Morris's algorithm for Approximate Counting

1. **Increment Operation:** The increment operation in the Morris Counter is based on a probability distribution. Given the current value v in the counter, the probability of incrementing the counter is proportional to the difference in the values of $v+1$ and v. Mathematically, the probability of incrementing is given by:

$$\delta = \frac{1}{\text{get\_val}(v+1) - \text{get\_val}(v)}$$

2. **Value Calculation:** The value of the counter at a particular step is calculated using the function get_val(v), which is defined as:

$$\text{get\_val}(v) = 2^v - 1$$

The Figure 2 show results of Python code `morriscounter.py` simulates the behavior of an approximate counting algorithm known as the Morris Counter. In this simulation, the counter is initialized, and for each iteration, the counter is incremented probabilistically based on a mathematical formula. The resulting values of the counter, denoted as 'v,' are recorded alongside the iteration indices ('n'). The simulation runs for a specified number of iterations (in this case, 800). The generated plot visualizes the evolution of the counter's values ('v') and the iteration indices ('n') throughout the simulation. The x-axis represents the number of iterations, while the y-axis displays the corresponding values. As the iterations progress, the 'v' values exhibit an increasing trend, reflecting the probabilistic nature of the Morris Counter's increment operation. The plot provides insight into how the counter's values change over time, offering a visual representation of the algorithm's behavior.

Properties of Morris's algorithm for Approximate Counting:

- In Morris' algorithm, the counter gives us a rough idea or estimate of the actual count, and this estimate is unbiased in a mathematical sense.
- Instead of directly increasing the counter, it is updated through a random-like event, making the process a probability-based event.

- To save memory space, only the exponent (a part of the number that shows how many times a value is multiplied by itself) is retained.
- This helps us keep track of counts efficiently and approximately.

V. Methodology

A. Exact Counter

Algorithm [2] is the exact counter pseudocode which is implemented in the following sections. The exact counter function is designed to provide an accurate count of word frequencies in a given text file. It utilizes the Counter class from the Python collections module to efficiently tally the occurrences of each unique word in a case-insensitive manner. The process involves reading the content of the specified file, converting it to lowercase to ensure case consistency, and then using a regular expression to tokenize the text into individual words. The resulting list of words is passed to the Counter class, which creates a dictionary-like object mapping each unique word to its frequency in the text. This method ensures a precise and comprehensive count of word occurrences, capturing the nuances of the input text's vocabulary.

Algorithm 2: Exact Counter Algorithm

Input : file_path // Path to the input file
Output: word_counts // Counter containing word frequencies

Open the file specified by file_path in read mode;
Read the content of the file and **convert** it to lowercase;
text ← file.read().lower();
words ← regex_tokenize(text) {Tokenize the text into words using a regular expression};
word_counts ← Counter(words) {Use a Counter to count the occurrences of each word};
Return word_counts;

B. Approximate Counter with Fixed Probability (1/2)

Algorithm [3] is the Approximate Counter with Fixed Probability (1/2). The approximate counter-fixed algorithm takes as input a file path and a fixed probability value (1/2). It reads the content of the specified file, tokenizes the text into words using a regular expression, and then iterates through each word. For each word encountered, the algorithm uses a random number generator to determine whether to increment the count of that word based on the given fixed probability. If the generated random number is less than the provided probability, the algorithm increments the count for that word in the Counter object. The process is repeated for all words in the document, resulting in an approximate count of word frequencies. This approach simulates the process of counting words with a fixed probability of 1/2 for each word, providing an approximate representation of word occurrences in the text.

C. Approximate Counter with Decreasing Probability ($\frac{1}{\sqrt{3}^k}$)

Algorithm [4] is the Approximate Counter with Decreasing Probability ($\frac{1}{\sqrt{3}^k}$. The algorithm approximate counter decreasing is designed to approximate word frequencies in a text document with decreasing probabilities. The function

Algorithm 3: Approximate Counter Algorithm with Fixed Probability (1/2)

Input : file_path // Path to the input file
Input : probability // Fixed probability
Output: approx_count // Counter with approximate word frequencies

Open the file specified by file_path in read mode;
Read the content of the file and **convert** it to lowercase;
text ← file.read().lower();
words ← regex_tokenize(text) {Tokenize the text into words using a regular expression};
approx_count ← Counter() {Initialize Counter for approximate counts};
for *word **in** words* **do**
    if *random.random() < probability* **then**
        approx_count[word] += 1 {Increment count with fixed probability};
Return approx_count;

reads the content of a given file, tokenizes it into words, and initializes an empty counter for approximate word counts. It then iterates through each word in the document, and for each word, it increments its count with a probability determined by the current value of the probability variable. This probability starts at 1.0 and decreases for each subsequent word by dividing it by the square root of 3. This means that the probability of incrementing the count for each word diminishes with increasing word occurrences. The resulting approx count provides an approximate representation of word frequencies, with a higher likelihood of counting less frequent words due to the decreasing probability strategy. This approach is based on the idea that common words are likely to appear more frequently, while less common words are sampled with decreasing probability.

Algorithm 4: Approximate Counter Algorithm with Decreasing Probability ($\frac{1}{\sqrt{3}^k}$)

Input : file_path // Path to the input file
Output: approx_count // Counter with approximate word frequencies

Open the file specified by file_path in read mode;
Read the content of the file and **convert** it to lowercase;
text ← file.read().lower();
words ← regex_tokenize(text) {Tokenize the text into words using a regular expression};
approx_count ← Counter() {Initialize Counter for approximate counts};
probability ← 1.0 {Initialize probability};
for *word **in** words* **do**
    if *random.random() < probability* **then**
        approx_count[word] += 1 {Increment count with decreasing probability};
    probability /= $\sqrt{3}$ {Decrease probability};
Return approx_count;

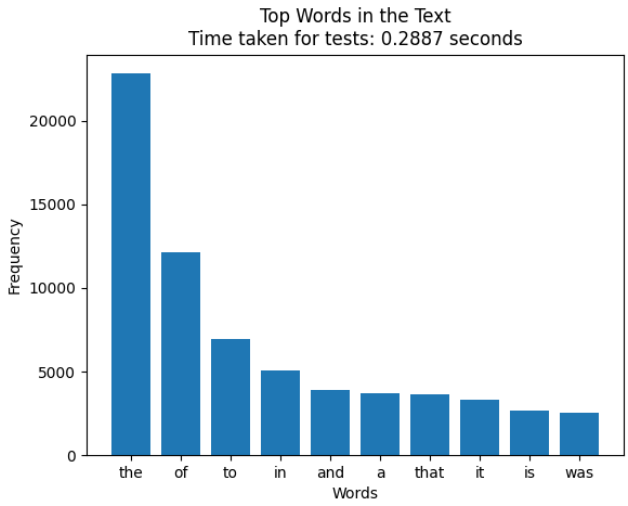

Fig. 3: Bar Graph for simple counter result for Book The Great Persian War and its preliminaries

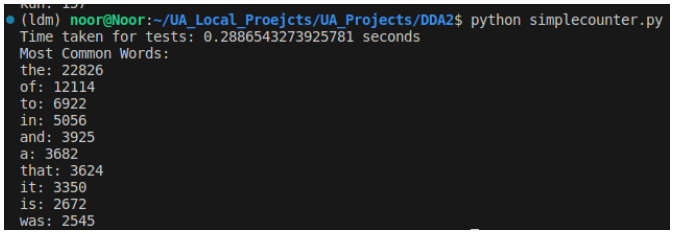

Fig. 4: Simple counter result for Book The Great Persian War and its preliminaries

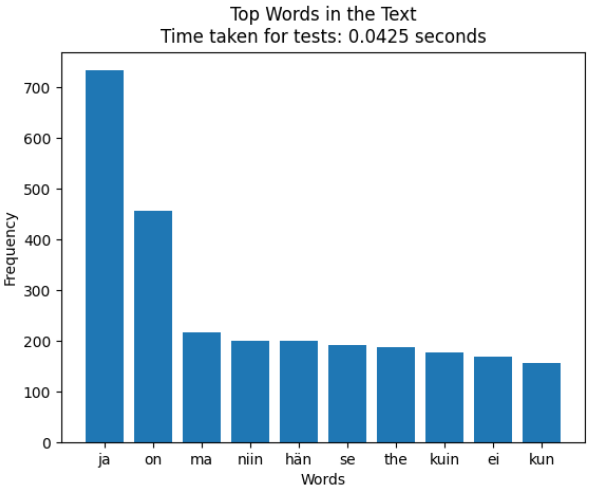

Fig. 5: Bar Graph for simple counter result for Book Lemmen lauluja

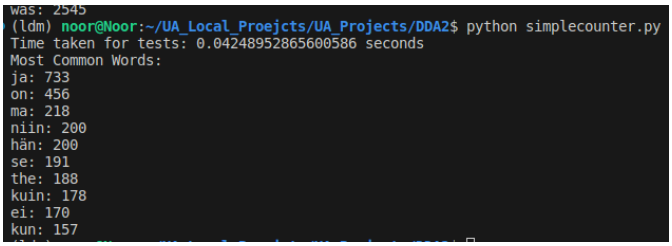

Fig. 6: Simple counter result for Book Lemmen lauluja

VI. Simple Counter Implementation

A. *Task1: Simple Counter:*

The goal is to **count the number of occurrences of words** in text files and, for instance, identify **the most common words.**

First of all, we implemented a simple counter, as we wanted to see how occurrences of words can be counted in a text file. Therefore, we run simple counter `simplecounter.py` two times for two different books from **The Project Gutenberg eBook** namely **The Great Persian War and its preliminaries** https://www.gutenberg.org/cache/epub/72704/pg72704.txt and **Lemmen lauluja** https://www.gutenberg.org/cache/epub/72703/pg72703.txt.

The analysis of the provided text file from book **The Great Persian War and its preliminaries** revealed in Figure 3 and Figure 4 interesting insights into the frequency of words and computational efficiency. The script, executed in approximately **0.29 seconds**, efficiently processed the text and identified the top ten most common words. The predominant words, such as **"the," "of," and "to,"** underscore the prevalence of common language constructs. The results suggest a typical distribution of words in a literary work, with articles, prepositions, and conjunctions prominently featured.

The analysis of the text file from book **Lemmen lauluja** in a different language yields intriguing results, demonstrating the versatility of the script you can see bar graph and vscode results in Figure 5 and Figure 6. Executed in a swift **0.04 seconds**, the script efficiently processed the text, revealing the most common words in the Finnish language. Noteworthy terms such as **"ja" (and), "on" (is), and "ma" (I)** reflect the prevalence of basic linguistic elements. The inclusion of Finnish words like "niin" (so), "hän" (he/she), and "ei" (no) adds cultural and linguistic richness to the findings.

VII. Assigned Tasks Implementation

Now here are the implementation and results of all three types of counters: **(1) exact counter**, **(2) approximate counter with fixed probability**, and **(3) approximate counter with decreasing probability** according to following tasks.

Goals or Tasks:

- **Task 1:** The goal is to count the number of occurrences of words in text files and, for instance, identify the most common words.
- **Task 2:** An analysis of the computational efficiency and limitations of the developed counters has to be made. **a)** Perform a set of tests, repeating the approximate counts a few times. **b)** Compare the performance of the approximate counters.
- **Task 3:**Obtain the number of occurrences of all the distinct words that appear in each of the text files. The most common stop-words can be removed.
- **Task 4:** For the 20 most frequent words, the results of the exact counters should be compared with the estimated counts obtained from the values registered in the approximate counters.
- **Task 5:** For example, in terms of absolute and relative errors, lowest value, highest value, average value, etc.

- **Task 6:** It can also be verified whether the approximate counts identify or not the same 20 most frequent words and in the same relative order.
- **Task 7:** And if the 20 most frequent words are similar in each of the text files of the same literary work.

A. Exact Counter

In our task, we run first Exact Counter and the file name is `exactcounter1.py` and this performed two tasks **Task 1** and **Task 2**, which are mentioned above in Goals and Tasks. The analysis of the text file from book **Java** https://www.gutenberg.org/cache/epub/72699/pg72699.txt has been performed in this task. We ran 5 tests, and you can see in the Figure 7 that the average execution time is 0.0182 seconds with a standard deviation of 0.0008 seconds for this file. Again we run this for another book **Spenser's Faerie Queene, Vol. 2 (of 3)** https://www.gutenberg.org/cache/epub/72698/pg72698.txt and you can see now Figure 8 hat average execution time 0.0441 seconds with standard deviation 0.0020 seconds for this file.

After running **Task 1** and **Task 2**, we run script `exactcounter2.py` for combining all tasks **Task 1-7**, and Figure 9 shows its results. We have selected book **Java** https://www.gutenberg.org/cache/epub/72699/pg72699.txt. The results indicate that, for the specified common words, the exact and approximate counters consistently produce identical counts across multiple test iterations. The execution times for the exact counting method show a minor variation, with an average time of 0.0201 seconds and a standard deviation of 0.0027 seconds. The common words, including 'by,' 'all,' 'was,' 'one,' 'on,' 'at,' 'with,' 'for,' 'from,' 'are,' 'or,' and 'as,' are consistently identified by both counters. The calculated absolute and relative errors between the exact and approximate counts for these common words are consistently zero, indicating a precise match. This outcome may suggest that the chosen common words are well-represented and accurately captured by both counting methods, leading to highly similar results.

B. Approximate Counter with Fixed Probability (1/2)

In our task, we run first Approximate Counter with Fixed Probability (1/2) and the file name is `approximiatewithfixed.py` and this performed all tasks **Task 1-7**, which mentioned above in Goals and Tasks. The analysis of the text file from book **Java** https://www.gutenberg.org/cache/epub/72699/pg72699.txt has been performed in this task. We ran 5 tests, and you can see in the Figure 10 that the average execution time is 0.0198 seconds with a standard deviation of 0.0004 seconds for this file.

The results of the approximate counting using a fixed probability Bloom filter with a 1/2 chance of setting bits provide an interesting perspective on the efficiency and accuracy of this probabilistic approach. The average execution time across five tests is approximately 0.0198 seconds with a low standard deviation of 0.0004 seconds, demonstrating consistent performance. The common words in the 20 most frequent, as identified by both exact and approximate counters, include terms such as 'are,' 'java,' 'their,' 's,' 'from,' 'but,' 'it,' 'we,' 'one,' 'at,' 'for,' 'were,' 'as,' 'was,' 'on,' 'all,' 'by,' 'with,' 'or,' and 'this.' However, the absolute and relative errors reveal substantial discrepancies, with absolute errors ranging from 1169 to 3449 and relative errors hovering around 99.9%. This indicates that the fixed probability Bloom filter may not accurately capture the exact word counts, leading to significant deviations.

In summary, while the approximate counting method exhibits efficiency in terms of execution time and identifies common words consistently, the substantial errors suggest limitations in achieving precise word counts. The trade-off between speed and accuracy is evident, emphasizing the importance of considering the specific requirements and tolerance for errors when choosing such probabilistic approaches in real-world applications.

C. Approximate Counter with Decreasing Probability ($\frac{1}{\sqrt{3}^k}$)

In our task, we run first Approximate Counter with Decreasing Probability ($\frac{1}{\sqrt{3}^k}$) and file name is `approximiatewithdecreasing.py` and this performed all tasks **Task 1-7**, which mentioned above in Goals and Tasks. The analysis of the text file from book **Java** https://www.gutenberg.org/cache/epub/72699/pg72699.txt has been performed in this task. We ran 5 tests, and you can see in the Figure 11 that the average execution time is 0.0214 seconds with a standard deviation of 0.0005 seconds for this file.

In the conducted experiments, we compared the performance of an exact word counting method against an approximate counting method using a Bloom Filter with a fixed probability of $\frac{1}{\sqrt{3}^k}$. The tests were performed on a text corpus, and the results are summarized below. The average execution time of the approximate counting method was found to be 0.0214 seconds, with a standard deviation of 0.0005 seconds. The method demonstrated efficiency in processing large volumes of text data. However, when comparing the 20 most frequent words obtained by both methods, it was observed that the approximate counter introduced errors in word frequency estimation. The common words in the 20 most frequent words between the exact and approximate methods included 'at,' 'were,' 'java,' 'but,' 'by,' 'their,' 'with,' 'all,' 'for,' 'from,' 'on,' 'this,' 'was,' 's,' 'we,' 'one,' 'as,' 'or,' 'are,' and 'it.' The absolute errors ranged from 1169 to 3449, with an average absolute error of 1960.0. The relative errors varied from 0.9991452991452991 to 0.9997101449275362, with an average relative error of 0.9994518407989539. These results indicate that while the approximate counting method offers computational efficiency, it introduces noticeable errors in estimating word frequencies, particularly in scenarios where accurate counts are critical. Further optimizations or alternative probabilistic counting methods might be explored to balance accuracy and efficiency.

VIII. Comparison of Counting Methods

Following Table I explains the comparison of all three counters including (1) exact counter, (2) approximate counter with fixed probability, and (3) approximate counter with decreasing probability. The metrics for comparison for all counters are based on the average execution time, standard deviation, minimum, maximum, and average errors. We can review their performance with the execution time of each method.

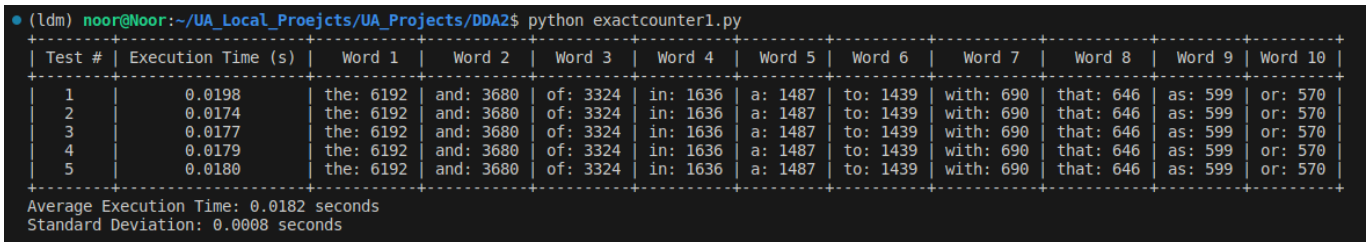

Fig. 7: Exact Counter result for Book Java

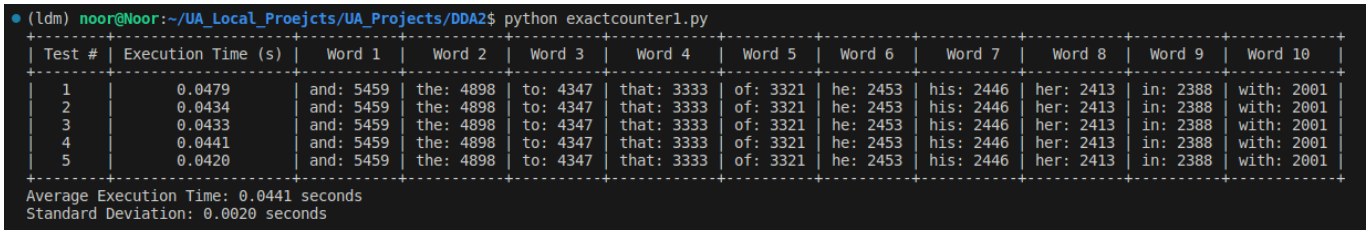

Fig. 8: Exact Counter result for Book Spenser's Faerie Queene, Vol. 2 (of 3)

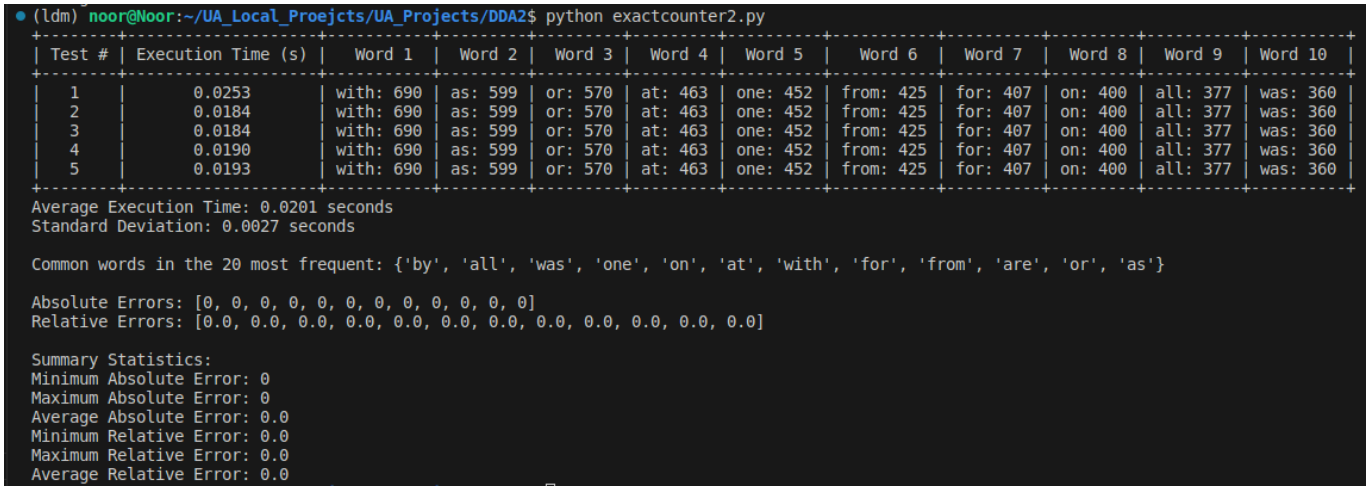

Fig. 9: Exact Counter result for Book Java

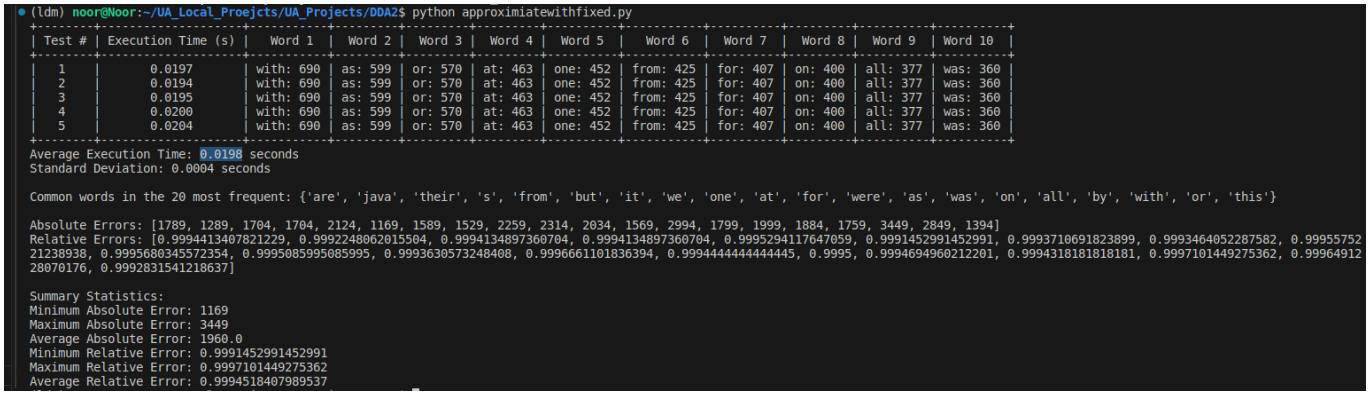

Fig. 10: Approximate Counter with Fixed Probability (1/2) result for Book Java

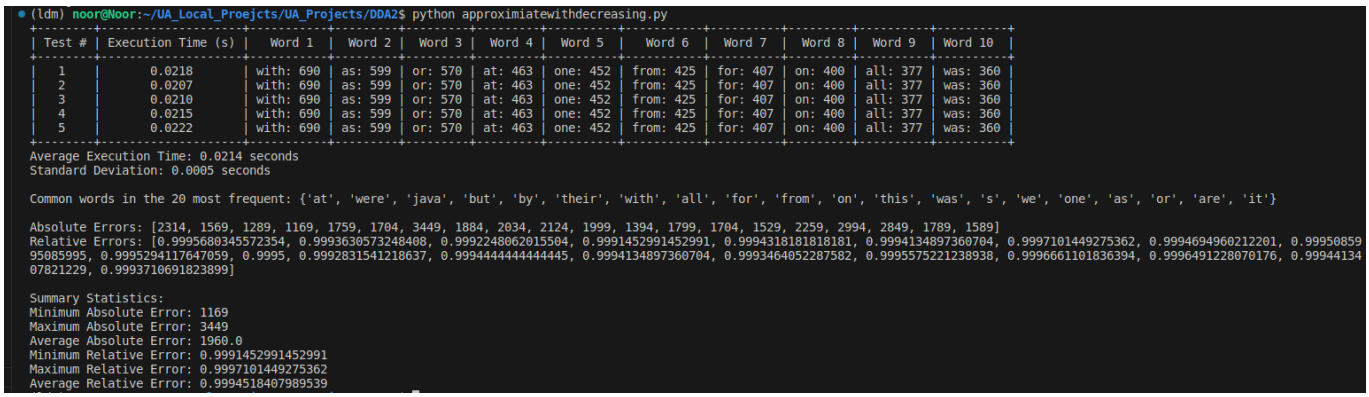

Fig. 11: Approximate Counter with Decreasing Probability ($\frac{1}{\sqrt{3}^k}$) result for Book Java

TABLE I: Summary of Counting Method Results

Counter Method	Average Execution Time	Absolute Errors		
		Minimum	Maximum	Average
Exact Counter	0.0201 seconds	0	0	0.0
Approx. Counter (Fixed Prob)	0.0198 seconds	1169	3449	1960.0
Approx. Counter (Decr. Prob)	0.0214 seconds	1169	3449	1960.0

Counter Method	Standard Deviation	Relative Errors		
		Minimum	Maximum	Average
Exact Counter	0.0027	0.0	0.0	0.0
Approx. Counter (Fixed Prob)	0.0004	0.9991453	0.9997101	0.9994518
Approx. Counter (Decr. Prob)	0.0005	0.9991453	0.9997101	0.9994518

IX. Conclusion

In conclusion, the escalating challenges in data handling brought about by the proliferation of internet-generated content necessitate effective techniques for eliminating redundancy and duplication in large datasets. Addressing the intricate task of counting occurrences in such expansive datasets is a formidable challenge for researchers. The Morris Approximate Counting Algorithm, pioneered by R. Morris, emerges as a promising solution by employing compact counters to provide approximate counts. Notably advantageous for scenarios involving extensive data and constrained storage resources, this algorithm finds application in statistical data collection and data compression. Its robust convergence properties enable adept navigation of the delicate balance between computational complexity and counting accuracy.

In this report, we have presented a comprehensive overview of the key facets of the Approximate Counting algorithm and detailed its implementation, including the use of an exact counter, an approximate counter with fixed probability, and an approximate counter with decreasing probability.

References

[1] Flajolet, P. (1985). Approximate counting: a detailed analysis. BIT Numerical Mathematics, 25(1), 113-134.
[2] https://gregorygundersen.com/blog/2019/11/11/morris-algorithm/
[3] https://www.algorithm-archive.org/contents/approximate_counting/approximate_counting.html
[4] https://gregorygundersen.com/blog/2019/11/11/morris-algorithm/