Final Project: Counting Bloom Filter (CBF)

Minerva University

CS110 - Problem-Solving with Data Structures and Algorithms

December 13, 2023

Table of Contents

What is it? **3**

Data Structure **5**

Hashing Technique **7**

Applications **11**

Python Implementation **13**

Time complexity **15**

How does the memory size scale with FPR? 15

How does the memory size scale with the number of items stored for a fixed FPR? 17

How does the actual FPR scale with the number of hash functions? 20

How does the access time to hashed values scale with the number of items stored in a CBF kept at constant FPR? 24

Plagiarism Detection using Counting Bloom Filter **25**

Similar Words Method 25

Similar Consecutive-Word Method 27

Strengths and Limitations 31

Other Algorithmic Strategies to Detect Plagiarism **32**

Word-by-word comparison method 32

Set of words similarity comparison method 33

Four approaches time complexity comparison 35

LOs and HCs **37**

References **39**

Appendix **40**

Counting Bloom Filters (CBFs)

What is it?

Imagine that you're the owner of a winter coat store. Customers come in and ask for a specific coat e in set A, which is your storage where all the store items are kept. How do you confirm that you have this coat in stock as quickly as possible?

The simplest, but inefficient, solution:

The simplest solution to achieve this would be linear search: iterating and checking each of the positions in the check whether the element is in the set or not. This would involve iterating through A and checking each of the positions. This process would take time complexity of $O(n)$. Another technique is binary search, which takes $O(log\ n)$ time complexity, but the elements must be sorted. This would mean going through each of the coats and looking for *the one*, which could take hours if you owned a big shop. One may certainly go out of business if we have to go through that process for every request from customers. Thus, we need a more efficient approach.

A quicker solution:

We can use a hash table that allows us to store and access the elements in $O(1)$ time. However, we are only interested in checking whether the elements are in set A or not, but we are wasting space and time to store all elements.

Instead, we can just record their "existence" in the set. We use 1 to record if e is in A, or 0 if e is not in set A. The starting hash table would be filled with 0's, and after the elements of A are inserted, we would switch the bits in the appropriate cells to 1's.

However, in this approach, we cannot differentiate whether the 1 in a given cell indicates the actual presence of the queried element e or that 1 indicates the presence of a different element. For example, when you search for the Gucci Spring 2019 coat, the hash table will only tell you that there is a 1 in the corresponding cell given the index, but it won't specify if it's the Gucci Spring 2019 coat or another item that happens to have the same hash value.

An actually better solution:

A more advanced solution involves employing multiple hash functions ($h1,\ h2,\ h3,\ldots$) to replace a 1 in each of the different slots that the set of hash functions being used return for a given element e. This approach minimizes the likelihood of multiple elements pointing to the exact same set of slots, reducing the chance of false positives.

Despite this improvement, the possibility of **false positives**, where an element is mistakenly identified as in the set when it is not, still exists. The frequency of false positives depends on factors such as the number of items and the number of hash functions employed.

Overall, although we do not eliminate the possibility of false positives, Bloom Filters can make such a possibility more unlikely (compared to the solutions described above).

However, we encounter another problem when removing items. Bloom Filters using binary values (0 or 1) may lead to a problem when an item is deleted, it affects the slots of other elements. A solution to this problem is the **Counting Bloom Filters**. Instead of a single bit of information (0 or 1), each cell of the table has a (n-bit) counter. This counter increments every time a value is inserted, and decrements every time an item is removed.

For example, when you sell a coat e, all the bloom filter counters of the corresponding slots of this element go down by one. Later, when we search for e, we would expect that at least one of the corresponding slots would have gone down to 0 and return "not found". In a less likely case, if enough other elements were "inserted" in the same slots corresponding to e, the search would return a false positive.

Data Structure

Hash Functions:

Hashing is to map our item to the set of indexes in the bit array with the set of hash functions. The number of indexes produced equals the number of hash functions, as each function produces a separate index in the bit array for every item hashed. The hash function has to be deterministic, meaning that the same hash values are returned for a specific element every time we run it.

The goal of hash functions is to transform input data into hash values, which are then used to determine the index within the bit array. These hash codes are "fingerprints" for the items being hashed, allowing for efficient storage in the CBF.

Each hash function has an O(1) time complexity, independent of the size of the input.

Back to the Gucci Spring 2019 coat, we first add this coat to our store inventory (using the insert method), and then it undergoes the hashing process. Each hash function maps the unique features of the Gucci coat to specific indexes within the storage spaces. Because they are deterministic, the same hash code is generated every time the hash functions are called. This ensures consistency in the store and the identification of each coat. Now, when a customer requests to try on this coat, you employ the same set of hash functions to quickly compute the hash codes, get the index, and confirm whether the coat is in stock (search method). The customer tries it on and loves it. Congrats, another jacket sold in a breeze! Lastly, you can use the delete method to remove this item from your inventory.

Insert method:

To add an element to the counting bloom filter, we first the element through the hash functions to obtain the hash value. This hash value is then divided by the total number of slots for all elements (size of the CBF) to get the exact index where this element will be inserted. The counter at the index of that hash value will be incremented by 1 to signal that at this slot, at least one value exists. Running the element through the hash function takes a constant time of O(1). Incrementing the counter at the index of the hash value also takes a constant time of O(1). Thus, the time complexity of this method is also O(1).

Search method:

To look for an element using the hash function, we first use hash functions to find the hash value of that element. We check the counter at the index of that hash value. If the counter value is greater than 0, the element probably is in the set, as in a less likely scenario, it could be a false positive. If the counter value equals 0, the item is definitely not in the set.

Running the element through the hash function takes a constant time of O(1). Then checking the counter value at the index of the hash value also takes a constant time of O(1). Thus, the time complexity of the search operation is O(1). As such, regardless of the size of the bloom filter, the runtime of the lookup operations stays constant.

Delete method:

To remove an element from the set, we also start by running the element through the hash function to get the hash value. After getting the index value of the element to de deleted, we decrementer the counter at the index of the hash value by 1. Similar to the previous two operations, running the element through the hash function takes a constant time of O(1). Decrementing the counter at the index of the hash value also takes a constant time of O(1). Thus, the time complexity of the delete operation is O(1).

The false-positive rate is calculated by the following formula:

$$P = (1 - e^{-kn/m})^k$$

Where P is the false positive rate, k is the number of hash functions, and n is the number of items in the set, and m is the size the bloom filter (Corte.si, 2010)

Hashing Technique

The formula for the optimal number of hash functions is:

$$k = \frac{m}{n} \ln 2$$

where m is the number of bits and n is the number of items in the counting bloom filter.

There are 2 types of functions: 1-to-1 function (h(x) = x) and many-to-one function. A hash function is the later type of function.

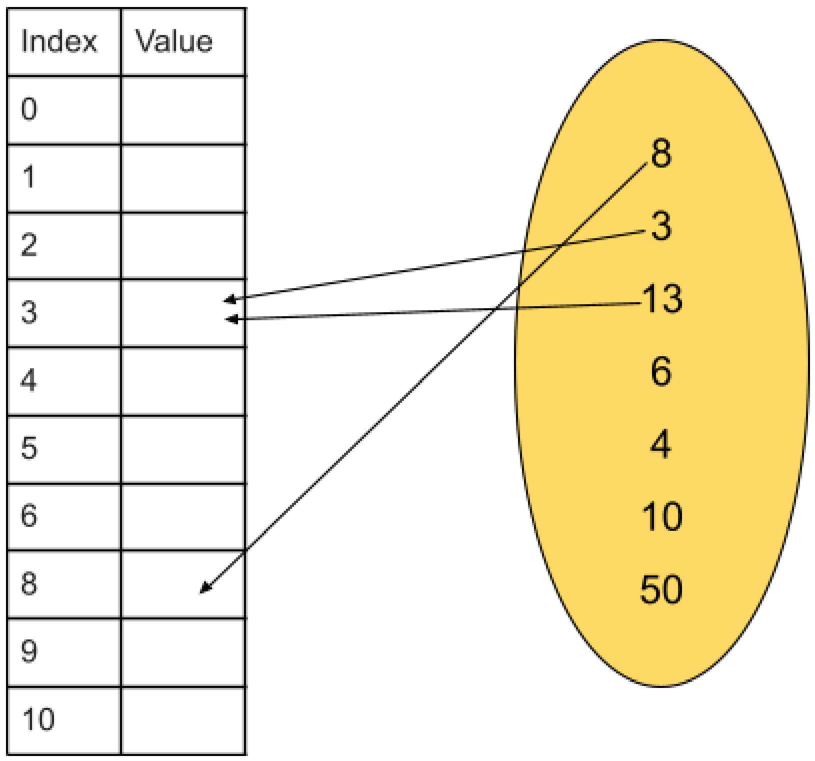
Let's study this in the form of an example.

Here's an empty hash table:

Index	0	1	2	3	4	5	6	7	8	9	10
Value											

We have a list of keys to be inserted:

8	3	13	6	4	10	50



We have h(x) = x. Number “8” will be placed at index 8. Number “3” will be placed at index 3.

Size of a hash table: h(x) = x % size. => For number “13”, h(13) = 13 % 10 = index 3.

Because 2 or more elements are mapped at the same index (3), we have a **collision**, which is a situation where we have multiple key-value pairs with the same indices after hashing.

Although collisions are inevitable, we can use open addressing to resolve this problem. In open addressing, each slot can only have at most 1 element and it reduces collisions in the filter by systematically searching for a new index to increment. In this assignment, double hashing is being employed as it is one of the best methods for open addressing.

$$\text{double\_hashing}(k,j) = (\text{hash\_function1}(k) + j\ \text{hash\_function2}(k)) \bmod m$$

Double hashing is the process of using two hash functions to compute two different hash values for a given key. The first hash function is used to compute the initial hash value, and the second hash function is to compute the step size for the probing sequence. Because this method uses two hash functions to compute the hash value and the step size, the probability of a collision occurring is lower than other collision resolution methods such as linear or quadratic probing.

Let’s take an example.

Note: mod or % returns the remainder of a division. We have the formula for hash function 1 and hash function 2.

hash1(k) = k mod 13

hash2(k) = 7 - k mod 7

Keys = (18, 41, 22, 44)

k	**Hash 1**	**Hash 2**
18	5	3
41	2	1
22	9	6
44	5	5

We will fill in our bucket array using the double hashing formula starting with j = 0:

$$\text{hash1(k)} + \text{j hash2 (k)}$$

k = 18 => $5 + 0 \cdot 3 = \mathbf{5}$

k = 22 => $9 + 0 \cdot 6 = 9$

k = 44 => $5 + 0 \cdot 5 = \mathbf{5}$

k = 41 => $2 + 0 \cdot 1 = 2$

At this point, index 5 is not empty because it already has value k=18 in it. Thus, we increase "j" by 1. We have, k = 44 => $5 + 1 \cdot 5 = \mathbf{10}$. This index is empty, and we can add 44 to the 10th index.

Value			41			18				22	44		
Index	0	1	2	3	4	5	6	7	8	9	10	11	12

Thus, j goes from 0 to the needed number. This happens until we find an empty index and we can fill that index.

Uniform distribution:

A good choice of hash function also maps the expected inputs as evenly as possible over its output range. This means that every hash value in the output range is generated with roughly the same probability. A uniform distribution of hash values is vital as if certain hash values are more likely to occur than others, it leads to a higher chance of multiple inputs colliding and creating collisions in the lookup process. In such cases, more lookup operations would need to run through a larger set of overlapping table entries, making the hashing process less efficient.

Applications

Real-life computational applications of CBF include:

Username Checker:

When signing up for an account, we need to create a username and password. The purpose of this process is for the program to distinguish among users. Thus, when you choose a username that has been used by other people, CBF can be used to search whether that username is available or not. CBF makes this process more efficient as it only takes O(1) to look up an item. Similarly, when a user deletes one’s account, CBF can easily remove that item and make it available for other users. This approach makes it process secure and efficient for thousands or millions of users compared to a list, which would take O(n) time complexity and a lot of space.

IP Address Filtering:

IP Addresses are used to identify devices on a network. An IP Address Filter is usually used to control what IP traffic will be allowed into and out of the network. For example, a company may want to block all traffic from a specific IP address for safety reasons. The company would add this IP address to a counting bloom filter. When a packet arrives, it is inspected by this network router. If it exists in the counting bloom filter, the packet is dropped, and if it does not, the packet is allowed to enter the network.

A counting bloom filter can be particularly useful in this case where the number of items (IP addresses) is very large. Checking each IP address in a database or a list would result in a linear time complexity $O(n)$ or $O(log\ n)$ with more optimized search algorithms. Instead, CBF’s search operation takes a constant time of $O(1)$. Moreover, if the company wants to remove or add an IP address, insertion and deletion operations also take a constant time $O(1)$.

Library inventory system:

Another application of CBF is library inventory where a detailed record of the items, materials, and resources is held. Since the databases of items are very large, a counting bloom filter is useful to look up books when requested or for periodic check-ups. The constant lookup time is significantly more efficient than other algorithmic strategies such as linear search or more optimized search algorithms. When a book is borrowed/ lost, the item can be removed from the counting bloom filter in a constant time of $O(1)$. When the book is returned, the item is added back to the inventory. The CBF also does this operation in $O(1)$.

Python Implementation

```python
    def hash_cbf(self, item, idx):
        """
        Returns hash values of an item

        Parameters:
            item(str): The item to hash
            i(int): The index of the hash function

        Returns:
            hash_values(lst): A list of hash values
        """
        key = self.str_to_int(item)

        #going through each hash function to find hash value
        def hash_fn1(key):
            """
            Hashes a string using the first hash function
            """
            return key % self.bucket_size

        def hash_fn2(key):
            """
            """
            prime = 18
            return prime - (key % prime)
        return (hash_fn1(key) + idx * hash_fn2(key)) % self.bucket_size


    def search(self, item):
        """
        Queries the membership of an element in the filter

        Parameters:
        item(str): The item to search for

        Returns:
        bool: tells if the item is there or not
        """
        for i in range(self.num_hashfn):
            # Getting the index of the hash function
            index = self.hash_cbf(item, i)
            if self.bucket[index] == 0:
                return False
        return True
    def hash_cbf(self, item, idx):
        """
        Returns hash values of an item

        Parameters:
            item(str): The item to hash
            i(int): The index of the hash function

        Returns:
            hash_values(lst): A list of hash values
        """
        key = self.str_to_int(item)

        #going through each hash function to find hash value
        def hash_fn1(key):
            """
            Hashes a string using the first hash function
            """
            return key % self.bucket_size

        def hash_fn2(key):
            """
            """
            prime = 18
            return prime - (key % prime)
        return (hash_fn1(key) + idx * hash_fn2(key)) % self.bucket_size


    def search(self, item):
        """
        Queries the membership of an element in the filter

        Parameters:
        item(str): The item to search for

        Returns:
        bool: tells if the item is there or not
        """
        for i in range(self.num_hashfn):
            # Getting the index of the hash function
            index = self.hash_cbf(item, i)
            if self.bucket[index] == 0:
                return False
        return True
    def hash_cbf(self, item, idx):
        """
        Returns hash values of an item

        Parameters:
            item(str): The item to hash
            i(int): The index of the hash function

        Returns:
            hash_values(lst): A list of hash values
        """
        key = self.str_to_int(item)

        #going through each hash function to find hash value
        def hash_fn1(key):
            """
            Hashes a string using the first hash function
            """
            return key % self.bucket_size

        def hash_fn2(key):
            """
            """
            prime = 18
            return prime - (key % prime)
        return (hash_fn1(key) + idx * hash_fn2(key)) % self.bucket_size


    def search(self, item):
        """
        Queries the membership of an element in the filter

        Parameters:
        item(str): The item to search for

        Returns:
        bool: tells if the item is there or not
        """
        for i in range(self.num_hashfn):
            # Getting the index of the hash function
            index = self.hash_cbf(item, i)
            if self.bucket[index] == 0:
                return False
        return True
```

Time complexity

How does the memory size scale with FPR?

According to Corte.si (2010), the length of a Bloom filter m can be described as:

$$m = -n \cdot \frac{ln(p)}{(ln(2))^2}$$

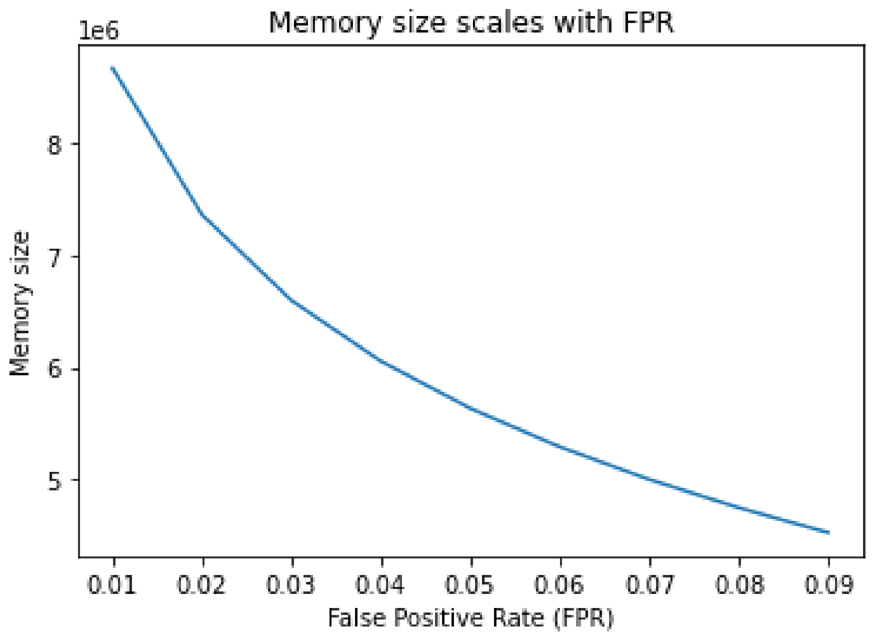
where m is the size of the CBF (in bits), n is the number of items stored in the filter, and FPR is the false-positive rate

Assuming a fixed n, as the false-positive rate (FPR) of a Bloom filter increases in the range from 0 to 1, $ln(p)$ increases from negative infinity to 0. The memory size m of the Bloom filter grows proportionally to $ln(p)$.

However, $0 \leq p \leq 1$. Thus, m is inversely proportional to p. This means that a lower false-positive rate (p) means a higher memory size (m). Conversely, increasing the false-positive rate would decrease the memory size needed.

Intuitively, this correlation makes sense because, with more counters, we will reduce the number of collisions when hashing an element, which is the cause of the false positive rate. Testing this experimentally,

```
# Memory size scales with FPR
fprs = [0.01 * i for i in range(1, 10)]
memory_sizes = []

for fpr in fprs:
    # Number of items stored in the counting bloom filter
    n = len(whole_text)

    #Optimal size of the counting bloom filter
    m = round(- (n * math.log(fpr)) / (math.log(2) ** 2))

    #Optimal number of hash functions
    k = round((m / n) * math.log(2))

    cbf = CountingBloomFilter(n, k)
    for word in whole_text:
        cbf.insert(word)
    memory_sizes.append(m)

# Plotting graphs
plt.plot(fprs, memory_sizes)
plt.title("Memory size scales with FPR")
plt.xlabel("False Positive Rate (FPR)")
plt.ylabel("Memory size")
plt.show()
```



The downward slope showcases that as the False-positive rate increases, the memory size decreases, which matches our theoretical analysis. Thus, achieving a low, ideal false-positive rate requires a larger array size which helps reduce collisions. It is the tradeoff between efficiency and accuracy.

How does the memory size scale with the number of items stored for a fixed FPR?

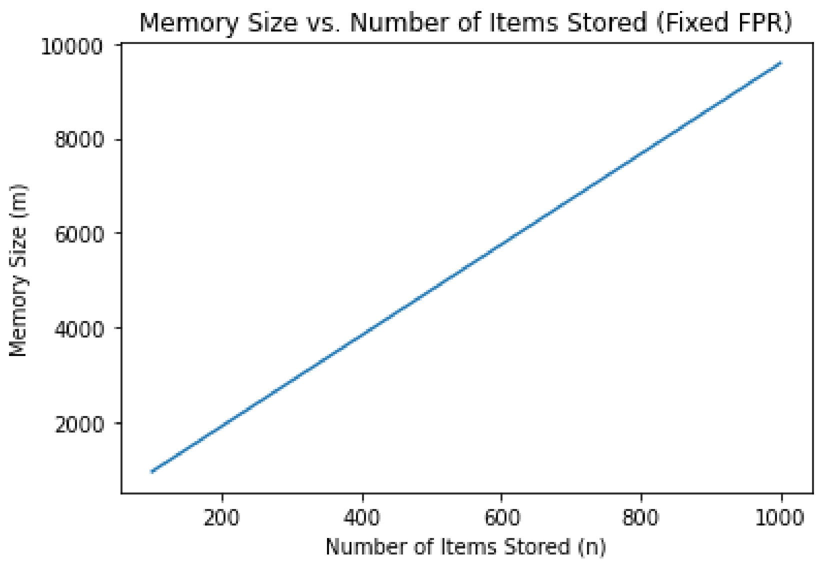
Using the same formula,

$$m = -n \cdot \frac{ln(p)}{(ln(2))^2}$$

we observe that for a *fixed* false positive rate (or probability) p, the length of a Bloom filter m, is proportionate to the number of items stored n. p is within the range from 0 to 1 and $ln(p)$ is always ≤ 0 in this range. The negative sign is to make sure that m is positive as the size of a data structure cannot be negative. Thus, as n increases, m also increases.

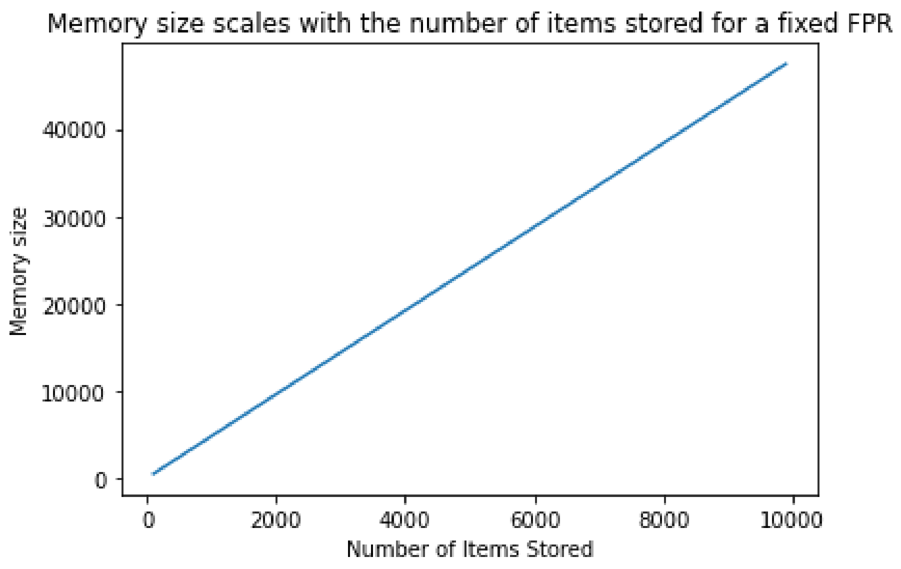
Intuitively, a larger n implies more items need to be stored in the CBF, and to maintain the fixed false positive rate, the CBF needs to be larger. Thus, the memory size needs to also grow to accommodate the increased number of distinct items stored in the CBF.

```python
# THEORETICAL ANALYSIS
# Memory size scales with the number of items stored for a fixed FPR

import math
import matplotlib.pyplot as plt

# Fixed False Positive Rate
FPR = 0.01

# Vary the number of items stored (n)
n_values = list(range(100, 1001, 100))

# Calculate memory sizes (m) for each n
memory_sizes = [- (n * math.log(FPR)) / (math.log(2) ** 2) for n in n_values]

# Plotting graph
plt.plot(n_values, memory_sizes)
plt.title("Memory Size vs. Number of Items Stored (Fixed FPR)")
plt.xlabel("Number of Items Stored (n)")
plt.ylabel("Memory Size (m)")
plt.show()
```



This plot showcases the relationship between m and n theoretically based on the formula. Comparing this to an empirical experiment:

```python
# EMPIRICAL ANALYSIS
# Memory size scales with the number of items stored for a fixed FPR
num_items = [100 * i for i in range(1, 100)]
fixed_fpr = 0.1  # Fixed False Positive Rate

memory_sizes = []

for n in num_items:
    # Optimal size of the counting bloom filter for the fixed FPR
    m = round(- (n * math.log(fixed_fpr)) / (math.log(2) ** 2))

    # Optimal number of hash functions
    k = round((m / n) * math.log(2))

    cbf = CountingBloomFilter(n, k)
    for word in whole_text[:n]:  # Use the first n items from the dataset
        cbf.insert(word)

    # Append the size of the counting bloom filter's memory
    memory_sizes.append(m)

# Plotting graphs
plt.plot(num_items, memory_sizes)
plt.title("Memory size scales with the number of items stored for a fixed FPR")
plt.xlabel("Number of Items Stored")
plt.ylabel("Memory size")
plt.show()
```



A positive correlation between the memory size and the number of items is shown, as expected. As we increase the number of items, the memory size grows linearly. The linear scalability is exhibited in both theoretical and empirical analysis.

How does the actual FPR scale with the number of hash functions?

We have the formula for the optimal number of hash functions:

$$k = \frac{m}{n} ln\, 2$$

where m is the number of bits in the counting bloom filter and n is the number of elements in the counting bloom filter. (Corte.si, 2010)

This was deprived of the false positive rate formula:

$$P_{false\, positive} = (1 - e^{-kn/m})^k$$

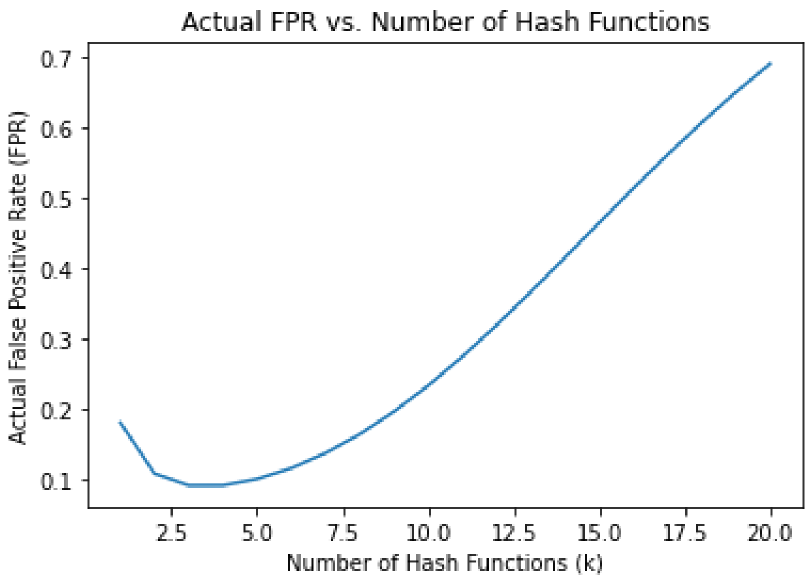
where k is the number of hash functions, m is the number of bits in the counting bloom filter, n is the number of items stored in the CBF, and e is the base of the natural logarithm. (Corte.si, 2010)

An optimal number of hash functions will minimize the false positive rate as the probability of collisions when hashing an element is reduced. Assuming that the number of bits in the filter (m) and the number of items stored (n) are held constant, as the number of hash functions (k) increases, the false positive rate decreases.

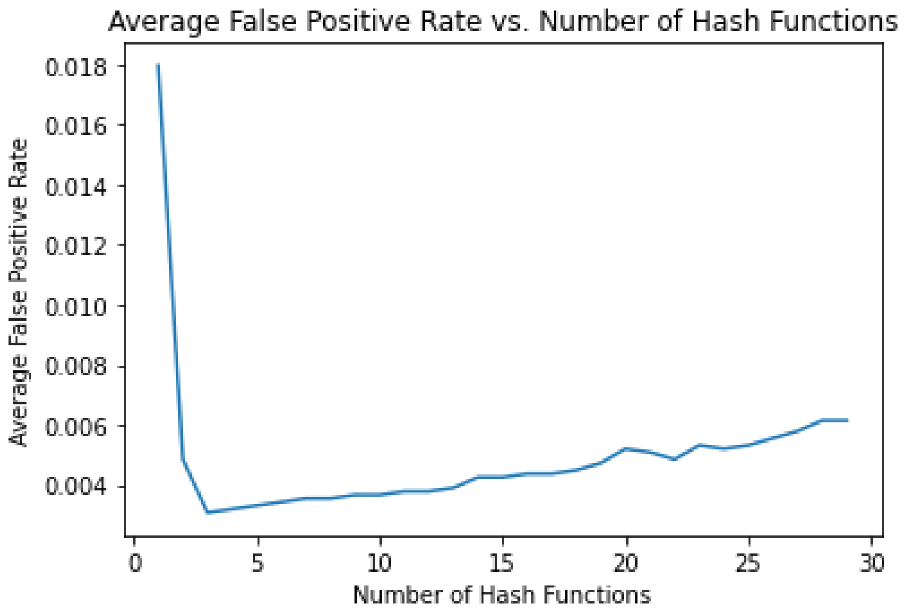
Intuitively, each hash function contributes a unique location in the filter where an item’s presence can be marked. Thus, more hash functions mean a more uniform distribution of set bits in the filter, reducing the likelihood of collisions where multiple items are hashed to the same set of bits, resulting in a lower false positive rate.

In this graph, we demonstrate the relationship between $P_{false\ positive}$ and k using the false positive rate formula mentioned earlier. m and n are fixed at 200 and 10000 - these values were taken from the theoretical data in the graph of how memory size scales with the number of items stored for a fixed FPR.

```python
# THEORETICAL ANALYSIS
# Actual FPR scales with the number of hash functions

import math
import matplotlib.pyplot as plt

# Fixed bits
m = 2000
n = 200

# Vary the number of hash functions (k)
num_hashfn_values = list(range(1, 21, 1))

# Calculate actual FPR for each number of hash functions
actual_fpr_values = [(1 - math.exp(-k * n / m)) ** k for k in num_hashfn_values]

# Plotting graph
plt.plot(num_hashfn_values, actual_fpr_values)
plt.title("Actual FPR vs. Number of Hash Functions")
plt.xlabel("Number of Hash Functions (k)")
plt.ylabel("Actual False Positive Rate (FPR)")
plt.show()
```



```python
# EMPIRICAL ANALYSIS
# Actual FPR scales with the number of hash functions
def fpr_scales_hashfns(version_1, num_hashfn_values, num_trials):
    fpr_values = []

    for num_hashfn in num_hashfn_values:
        total_false_positives = 0

        for _ in range(num_trials):
            # Create a counting bloom filter with the varying number of hash functions
            cbf = CountingBloomFilter(len(version_1), num_hashfn)

            # Insert all words from version_1 into the counting bloom filter
            for word in version_1:
                cbf.insert(word)

            # Check if other random words are falsely identified as present
            test_words = [f"random_word_{i}" for i in range(len(version_1))]
            false_positives = sum(cbf.search(word) for word in test_words)

            total_false_positives += false_positives

        # Calculate the average false positive rate for the current configuration
        average_fpr = total_false_positives / (num_trials * len(test_words))
        fpr_values.append(average_fpr)

    return fpr_values


num_hashfn_values = list(range(1, 30))
num_trials = 10

fpr_values = fpr_scales_hashfns(version_1, num_hashfn_values, num_trials)

print("Num Hash Functions:", num_hashfn_values)
print("Average FPR Values:", fpr_values)

# Plotting graphs
plt.plot(num_hashfn_values, fpr_values)
plt.xlabel("Number of Hash Functions")
plt.ylabel("Average False Positive Rate")
plt.title("Average False Positive Rate vs. Number of Hash Functions")
plt.show()
```



Similar to the theoretical graph, the false positive rate initially decreases as the number of hash functions increases, suggesting that a higher number of hash functions contributes to a more accurate Counting Bloom Filter (CBF). However, after reaching a minimum, the false positive rate begins to increase again as the number of hash functions surpasses 3. This observed trend implies that the optimal number of hash functions likely falls within the range of 2 to 3.

Beyond the optimal number of hashes, the average false positive rate shows a slight increase. This could be due to a specific implementation of the code. However, even with this slight increase, the false positive rates at 3 to 30 hash functions remain considerably lower than those at 0 to 3 hash functions. The initial range exhibited false positive rates ranging from 0.002 to 0.018, while the rates at 3 to 30 hash functions range from 0.002 to 0.006. This indicates that, despite the minor increase, employing more than the optimal number of hash functions still results in a CBF with relatively low false positive rates

How does the access time to hashed values scale with the number of items stored in a CBF kept at constant FPR?

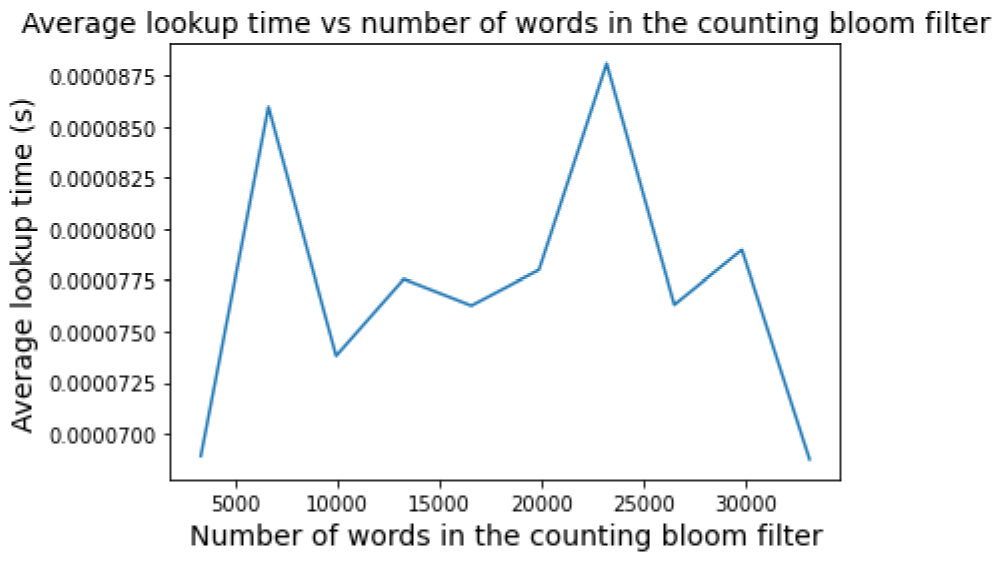
From the two formulas, (1) $m = -n \cdot \frac{ln(p)}{(ln(2))^2}$ and (2) $k = \frac{m}{n} ln\,(2)$, we have:

$$\frac{m}{n} = -\frac{ln(p)}{(ln(2))^2}$$

$$=> k = -\frac{ln(p)}{ln(2)} = -log(2p)$$

From this result, the number of items stored in a CBF (n) does not directly affect the number of hash functions (k). The lookup operations in a CBF are constant operations O(1) as regardless of the number of items stored in the filter, the number of hash functions used for lookups (k) remains constant. The lookup time is proportional to the number of hash functions but it does not depend on the size of the dataset stored in the filter.

To determine how the access time to hashed values scales with the number of items at a fixed False Positive Rate, we will compute the average search time for a counting bloom filter with a certain number of words hashed to it. We will analyze how the average time changes as the number of words hashed to it increases.



The graph shows that the average lookup time to access a hashed value is fairly constant as the number of words in the counting bloom filter increases at a constant false positive rate.

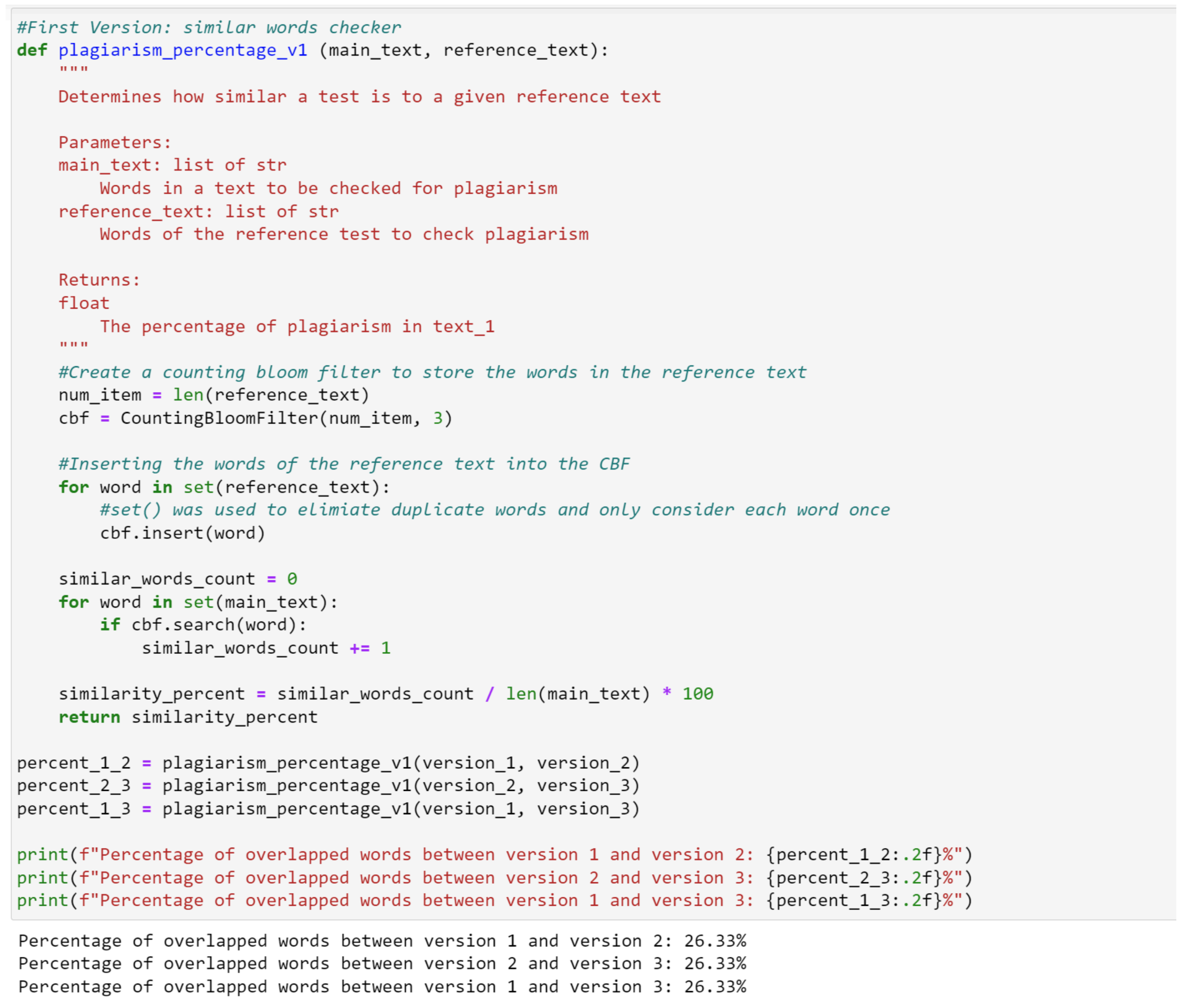
Plagiarism Detection using Counting Bloom Filter

Similar Words Method

Starting with the most straightforward and simple method, we define plagiarism as the number of words in both documents. There's a main text to be checked for the extent of plagiarism it has and a reference text, serving as the base to check plagiarism against.

In this first version, we use CBF to count how many words in the main text are also in the reference text and divide that number by the total number of words in the main text to get the percentage of plagiarism.

Since the lookup operation in the Counting Bloom Filter is O(1), we can quickly check whether a word of the main text is also in the reference text. For a document of n words, we can check the extent of plagiarism in O(1) * n = O(n) time complexity. Whereas, if we are going through each word in a list, the search operation could be $O(n)$ or $O(log\ n)$ for more optimized sorting algorithms. As such, the time complexity for checking the extent of plagiarism would be $O(n^2)$ or $O(n\ log\ n)$.



```
#First Version: similar words checker
def plagiarism_percentage_v1 (main_text, reference_text):
    """
    Determines how similar a test is to a given reference text

    Parameters:
    main_text: list of str
        Words in a text to be checked for plagiarism
    reference_text: list of str
        Words of the reference test to check plagiarism

    Returns:
    float
        The percentage of plagiarism in text_1
    """
    #Create a counting bloom filter to store the words in the reference text
    num_item = len(reference_text)
    cbf = CountingBloomFilter(num_item, 3)

    #Inserting the words of the reference text into the CBF
    for word in set(reference_text):
        #set() was used to elimiate duplicate words and only consider each word once
        cbf.insert(word)

    similar_words_count = 0
    for word in set(main_text):
        if cbf.search(word):
            similar_words_count += 1

    similarity_percent = similar_words_count / len(main_text) * 100
    return similarity_percent

percent_1_2 = plagiarism_percentage_v1(version_1, version_2)
percent_2_3 = plagiarism_percentage_v1(version_2, version_3)
percent_1_3 = plagiarism_percentage_v1(version_1, version_3)

print(f"Percentage of overlapped words between version 1 and version 2: {percent_1_2:.2f}%")
print(f"Percentage of overlapped words between version 2 and version 3: {percent_2_3:.2f}%")
print(f"Percentage of overlapped words between version 1 and version 3: {percent_1_3:.2f}%")
```

```
Percentage of overlapped words between version 1 and version 2: 26.33%
Percentage of overlapped words between version 2 and version 3: 26.33%
Percentage of overlapped words between version 1 and version 3: 26.33%
```

The percentages of overlapped words are all the same and quite high, which requires us to reevaluate our settings. If we examine the definition of plagiarism for this method, it is not accurate and comprehensive. Having certain numbers of similar words in both documents is not sufficient to determine plagiarism as those words may be from the prompt or used in different contexts. For example, in a literature class, when everyone gets assigned to write about the same topic, similar words are naturally expected. Thus, we need to improve this definition in version 2 of the plagiarism checker using counting bloom filters.

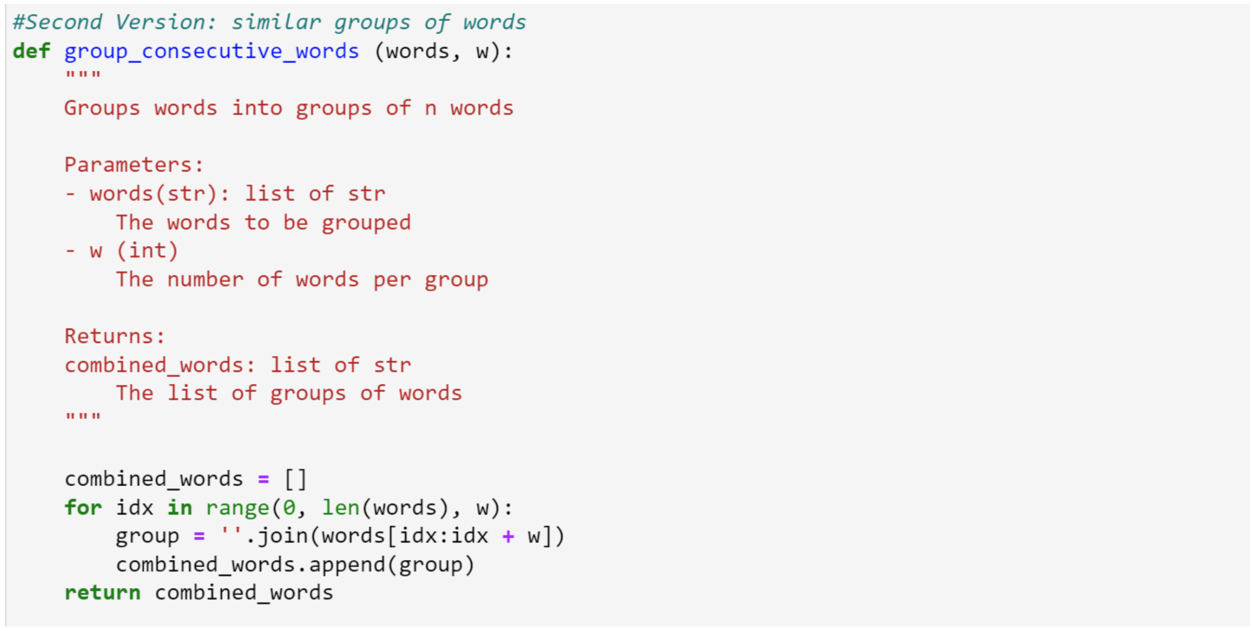
Similar Consecutive-Word Method

In this version, we aim to give more context and information about how a word is used by grouping consecutive words together. A phrase, both long and short, can describe the intention and originality of the words better than a singular word can.

We divide the list of words in the main text into groups of *w* words and count the number of groups in the main text that is also in the reference text. We then divide that number by the total number of groups of words in the main text to obtain the percentage of similarity.

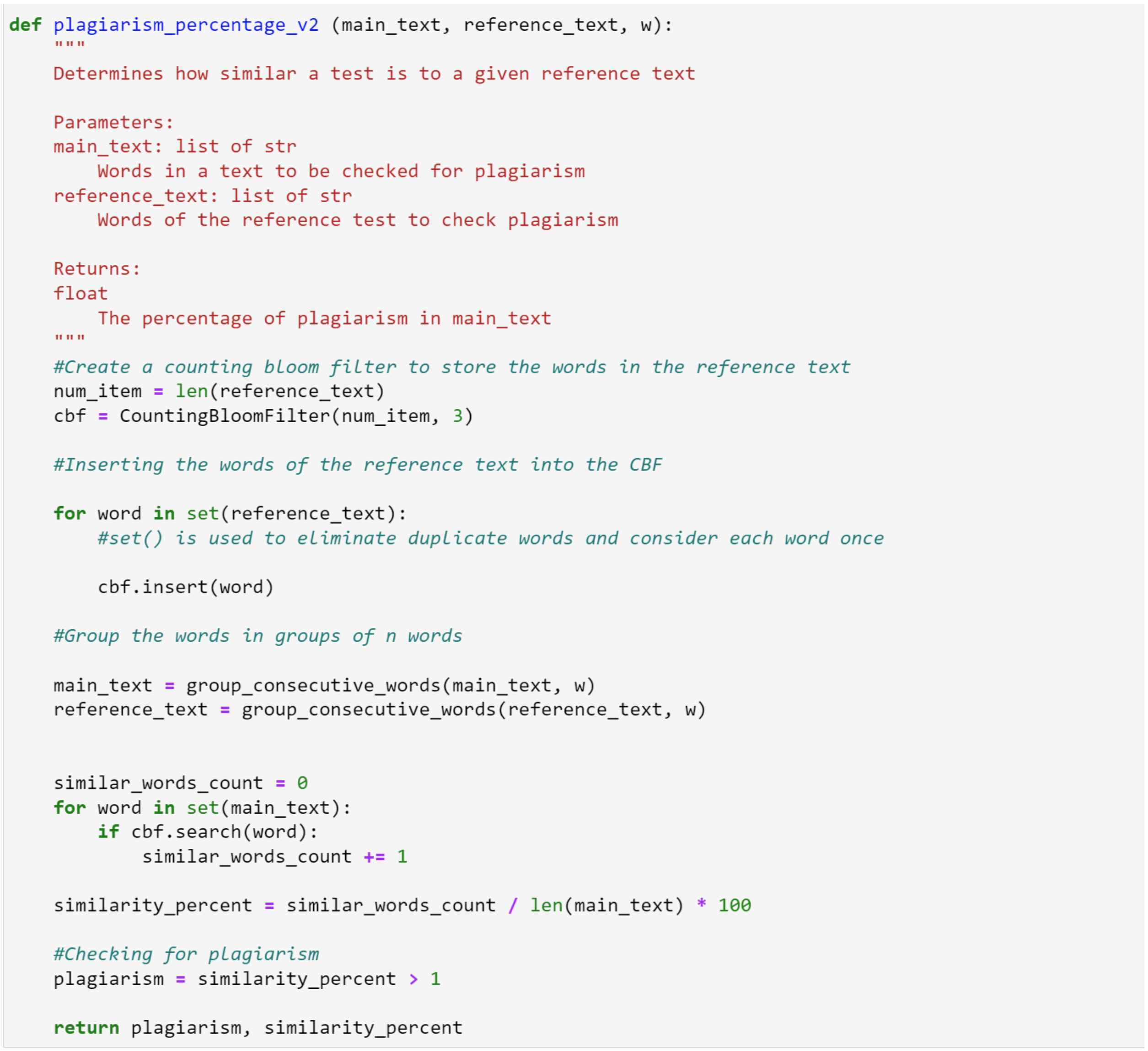
Counting Bloom Filter is also efficient in this version as the search operation to look up whether a group of words in one document is present in the other is also O(1).

Firstly, this function iterates through the list of words with a step size of ‘w’ to create consecutive groups of words, with w being the number of words in each group. The groups formed are then stored in the ‘combined_words’ list.



```python
#Second Version: similar groups of words
def group_consecutive_words (words, w):
    """
    Groups words into groups of n words

    Parameters:
    - words(str): list of str
        The words to be grouped
    - w (int)
        The number of words per group

    Returns:
    combined_words: list of str
        The list of groups of words
    """

    combined_words = []
    for idx in range(0, len(words), w):
        group = ''.join(words[idx:idx + w])
        combined_words.append(group)
    return combined_words
```

Secondly, this function calculates the plagiarism percentage between ‘main_text’ and ‘reference_text’. It creates a counting bloom filter with the number of items being the number of words in reference text and 3 hash functions (as identified as optimal in the complexity analysis section). After inserting the words into the CBF, we group the words in both documents using the ‘group_consecutive_words’ function. The plagiarism percentage is calculated based on the count of similar words out of the total of words in the main text. If this percentage is greater than 1, it is considered as plagiarism.



```python
def plagiarism_percentage_v2 (main_text, reference_text, w):
    """
    Determines how similar a test is to a given reference text

    Parameters:
    main_text: list of str
        Words in a text to be checked for plagiarism
    reference_text: list of str
        Words of the reference test to check plagiarism

    Returns:
    float
        The percentage of plagiarism in main_text
    """
    #Create a counting bloom filter to store the words in the reference text
    num_item = len(reference_text)
    cbf = CountingBloomFilter(num_item, 3)

    #Inserting the words of the reference text into the CBF

    for word in set(reference_text):
        #set() is used to eliminate duplicate words and consider each word once

        cbf.insert(word)

    #Group the words in groups of n words

    main_text = group_consecutive_words(main_text, w)
    reference_text = group_consecutive_words(reference_text, w)


    similar_words_count = 0
    for word in set(main_text):
        if cbf.search(word):
            similar_words_count += 1

    similarity_percent = similar_words_count / len(main_text) * 100

    #Checking for plagiarism
    plagiarism = similarity_percent > 1

    return plagiarism, similarity_percent
```

For different numbers of words in each group (w), we obtain different results of the percentage of similarity between 2 documents, which determines whether plagiarism is detected.

Groups of 2 words:

```
result_1_2 = plagiarism_percentage_v2(version_1, version_2, 2)
result_2_3 = plagiarism_percentage_v2(version_2, version_3, 2)
result_1_3 = plagiarism_percentage_v2(version_1, version_3, 2)

print(f"Plagiarism between version 1 and version 2: {result_1_2[0]}, Percentage: {result_1_2[1]:.2f}%")
print(f"Plagiarism between version 2 and version 3: {result_2_3[0]}, Percentage: {result_2_3[1]:.2f}%")
print(f"Plagiarism between version 1 and version 3: {result_1_3[0]}, Percentage: {result_1_3[1]:.2f}%")
```

```
Plagiarism between version 1 and version 2: False, Percentage: 0.52%
Plagiarism between version 2 and version 3: False, Percentage: 0.85%
Plagiarism between version 1 and version 3: True, Percentage: 1.09%
```

Groups of 4 words, as we assume this is the average number of words in a phrase:

```
result_1_2 = plagiarism_percentage_v2(version_1, version_2, 4)
result_2_3 = plagiarism_percentage_v2(version_2, version_3, 4)
result_1_3 = plagiarism_percentage_v2(version_1, version_3, 4)

print(f"Plagiarism between version 1 and version 2: {result_1_2[0]}, Percentage: {result_1_2[1]:.2f}%")
print(f"Plagiarism between version 2 and version 3: {result_2_3[0]}, Percentage: {result_2_3[1]:.2f}%")
print(f"Plagiarism between version 1 and version 3: {result_1_3[0]}, Percentage: {result_1_3[1]:.2f}%")
```

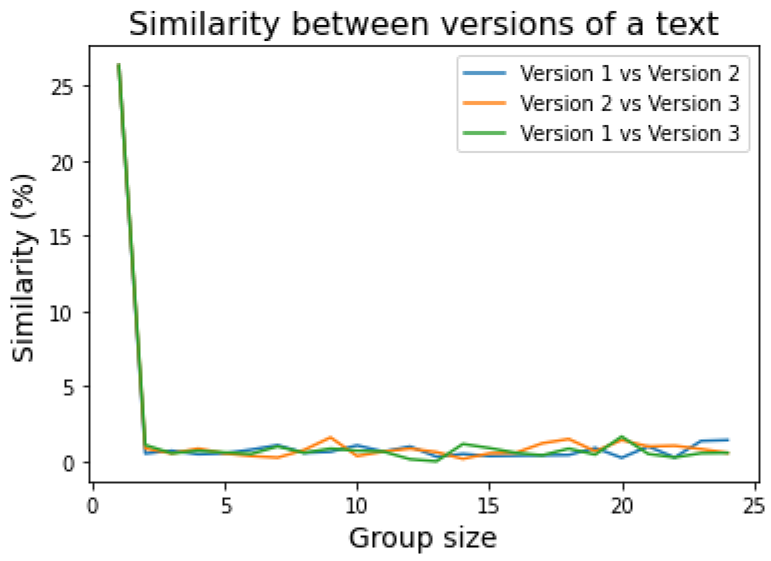
```
Plagiarism between version 1 and version 2: False, Percentage: 0.47%
Plagiarism between version 2 and version 3: False, Percentage: 0.85%
Plagiarism between version 1 and version 3: False, Percentage: 0.71%
```

Group of 15, as we assume this is the average number of words in a sentence:

```
result_1_2 = plagiarism_percentage_v2(version_1, version_2, 15)
result_2_3 = plagiarism_percentage_v2(version_2, version_3, 15)
result_1_3 = plagiarism_percentage_v2(version_1, version_3, 15)

print(f"Plagiarism between version 1 and version 2: {result_1_2[0]}, Percentage: {result_1_2[1]:.2f}%")
print(f"Plagiarism between version 2 and version 3: {result_2_3[0]}, Percentage: {result_2_3[1]:.2f}%")
print(f"Plagiarism between version 1 and version 3: {result_1_3[0]}, Percentage: {result_1_3[1]:.2f}%")
```

```
Plagiarism between version 1 and version 2: False, Percentage: 0.35%
Plagiarism between version 2 and version 3: False, Percentage: 0.53%
Plagiarism between version 1 and version 3: False, Percentage: 0.88%
```

Thus, finding and choosing an optimal number of words per group depends on the materials of the documents (type, length, purpose). We can perform an experiment to see how the average percentage of plagiarism changes as the number of words in a group increases and find the optimal number.



Theoretically, the higher the number of words in a group of words, the lower the similarity percentage between two documents should be, as it is harder to find 25 consecutive words that are exactly the same in both documents than a similar phrase of 3 words. However, this graph shows that except for comparing single words (group of 1), the similarity percentage stays fairly unchanged with slight increases and decreases, as the number of words in a group increases. This matches the results we produced.

Strengths and Limitations

Strengths:

Besides its intriguing name, Counting Bloom Filters offers comparably much more efficiency and time complexity. The search, insert, and delete operations all have the scalability of O(1), compared to other algorithms with lookup's time complexity of O(n) and O(log n), which are more ideal in terms of efficiency. CBFs also provide a space-efficient approach for storing a large number of unique items. This is particularly beneficial when dealing with extensive datasets, where the compact data structure of CBFs helps minimize storage requirements.

Additionally, because CBF is designed to use hash functions and map words to positions in the filter, instead of storing the actual information (words), it can be more secure and private.

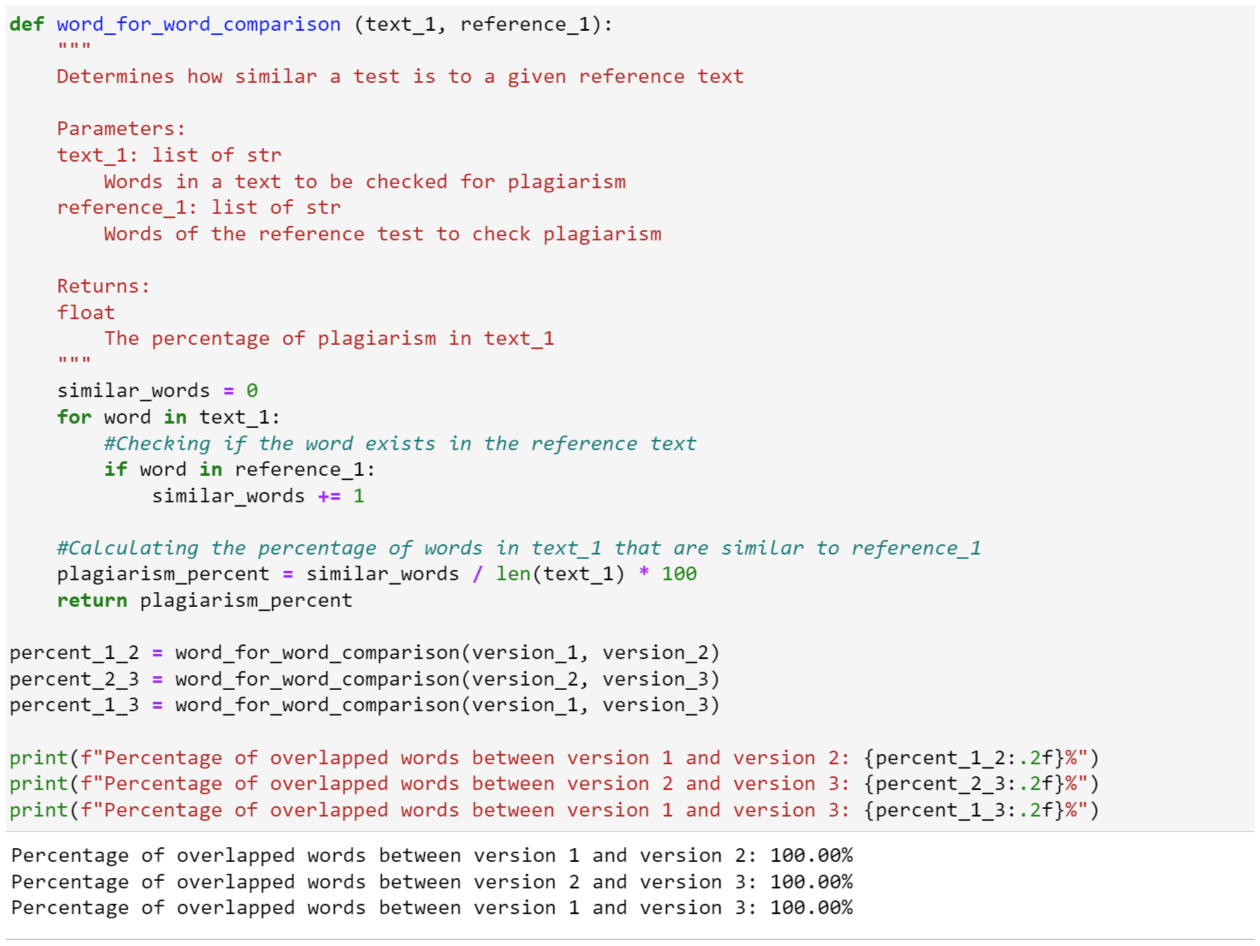
Limitations:

As discussed earlier, while the false positive rate can be controlled to some extent, it is a non-zero probability, especially when handling similar word frequencies or documents with short lengths. This is due to hash collisions where multiple words are mapped to the same positions in the filter. Although there are ways to reduce the likelihood of collisions such as chaining and open addressing, the choice and implementation of hash functions must be optimal to minimize false positives.

Other Algorithmic Strategies to Detect Plagiarism

Word-by-word comparison method

Using this approach, for every word in the main text, the algorithm goes through all the words in the reference text to check if such a word is present. The algorithm then counts the number of words present in both documents and divides it by the number of total words in the main text.



```
def word_for_word_comparison (text_1, reference_1):
    """
    Determines how similar a test is to a given reference text

    Parameters:
    text_1: list of str
        Words in a text to be checked for plagiarism
    reference_1: list of str
        Words of the reference test to check plagiarism

    Returns:
    float
        The percentage of plagiarism in text_1
    """
    similar_words = 0
    for word in text_1:
        #Checking if the word exists in the reference text
        if word in reference_1:
            similar_words += 1

    #Calculating the percentage of words in text_1 that are similar to reference_1
    plagiarism_percent = similar_words / len(text_1) * 100
    return plagiarism_percent

percent_1_2 = word_for_word_comparison(version_1, version_2)
percent_2_3 = word_for_word_comparison(version_2, version_3)
percent_1_3 = word_for_word_comparison(version_1, version_3)

print(f"Percentage of overlapped words between version 1 and version 2: {percent_1_2:.2f}%")
print(f"Percentage of overlapped words between version 2 and version 3: {percent_2_3:.2f}%")
print(f"Percentage of overlapped words between version 1 and version 3: {percent_1_3:.2f}%")
```

```
Percentage of overlapped words between version 1 and version 2: 100.00%
Percentage of overlapped words between version 2 and version 3: 100.00%
Percentage of overlapped words between version 1 and version 3: 100.00%
```

A whole document of Shakepeare's work being plagiarised 100% does not sound plausible. All three versions are different works of Shakespeare, and just because singular words appear in both documents does not mean the whole document is plagiarized. Thus, this method does not provide plausible results.

Set of words similarity comparison method

Upgrading from the last version, instead of comparing every single word, we look for the number of unique common words between both documents. This approach iterates through the unique words in text_2, checks if each word is present in reference_2, and calculates the percentage of common words relative to the total number of words in text_2.

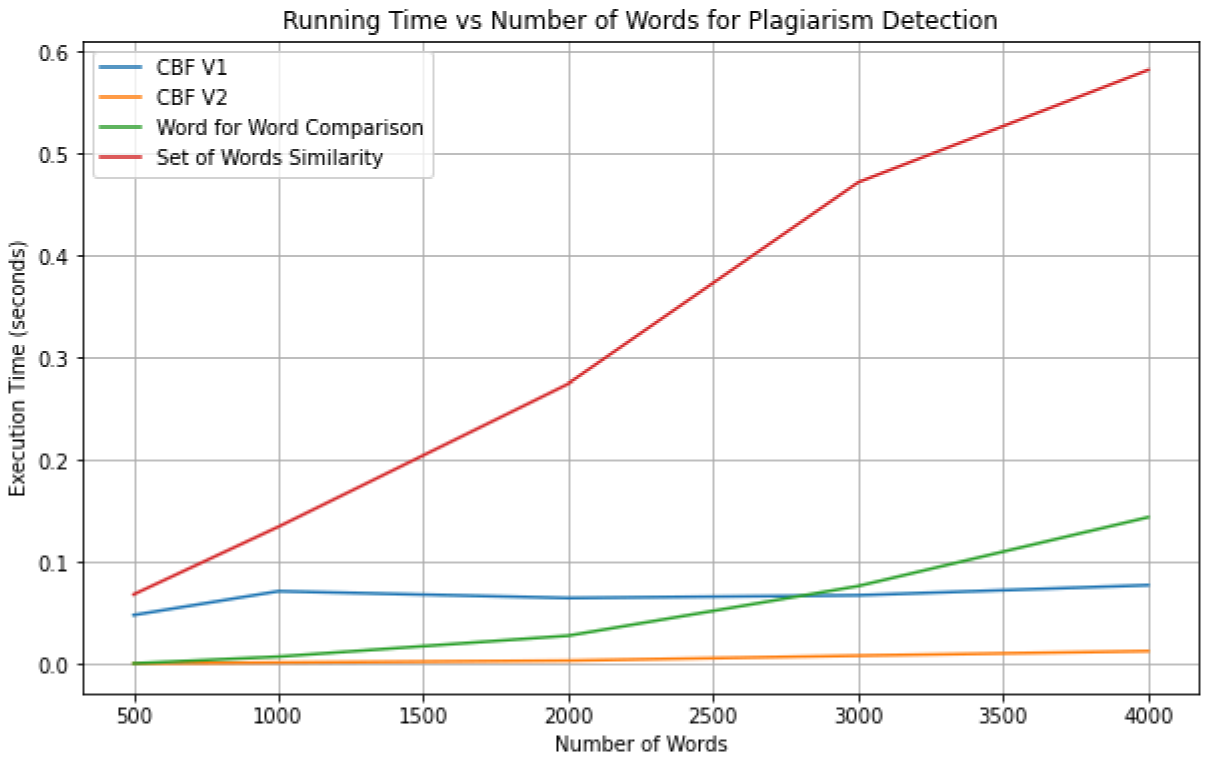
```python
def set_of_words_similarity(text_2, reference_2):
    """
    Determine how similar a test is based on the number of common words

    Parameters:
    text_2: list of str
        The words in the first document
    reference_2: list of str
        The words in the second document

    Returns:
    float
        The similarity percentage between the two documents
    """

    common_words = set()
    for word in set(text_2):
        if word in set(reference_2):
            common_words.add(word)

    # Calculate common words relative to the size of text_2
    similarity = len(common_words) / len(text_2) * 100

    return similarity

percent_1_2 = set_of_words_similarity(version_1, version_2)
percent_2_3 = set_of_words_similarity(version_2, version_3)
percent_1_3 = set_of_words_similarity(version_1, version_3)

print(f"Percentage of overlapped words between version 1 and version 2: {percent_1_2:.2f}%")
print(f"Percentage of overlapped words between version 2 and version 3: {percent_2_3:.2f}%")
print(f"Percentage of overlapped words between version 1 and version 3: {percent_1_3:.2f}%")
```

```
Percentage of overlapped words between version 1 and version 2: 26.33%
Percentage of overlapped words between version 2 and version 3: 26.33%
Percentage of overlapped words between version 1 and version 3: 26.33%
```

The results are the same as those obtained from the Counting Bloom Filter Version 1 approach. This is because both approaches assess the overlap of unique words between two documents but with different mechanisms. The “set of words similarity” approach directly compares unique words using set() and calculates a similarity score. Whereas, CBF Version 1 uses a probabilistic data structure to check the presence of words after inserting unique words from reference text into the filter.

Four approaches time complexity comparison

Comparing these two approaches' complexity with the two versions of the Counting Bloom Filter, we have:



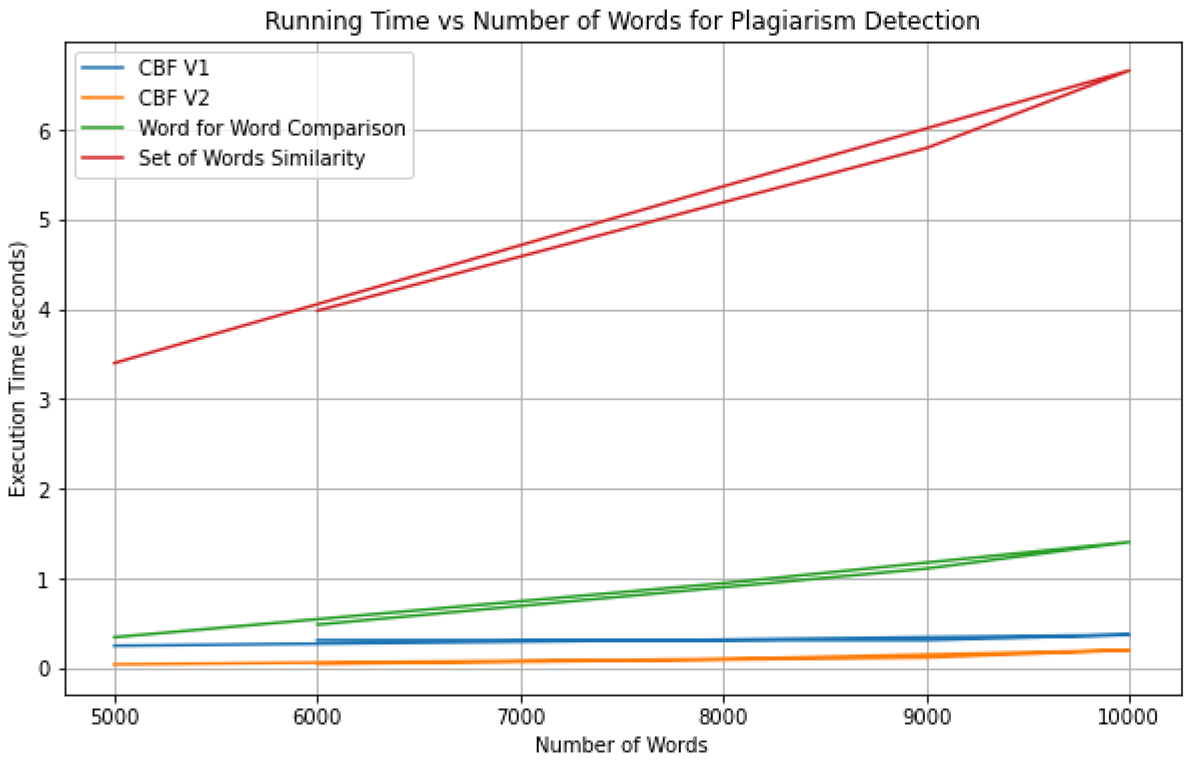
For the **"word for word comparison"** approach, 1000 words = 0.01s, 2000 words = 0.04s, 4000 words = 0.15s. Although the running time is low, it is suggesting major increases in running time as the number of words increases. Specifically, the results show that as the number of words doubles, the execution time approximately quadruples, $O(n^2)$. This is due to the nested loop structure, where the outer loop iterates over each word in text_1 and the inner loop searches for the word in reference_1. If they each have a length of n, the time complexity would be O(n*n) = $O(n^2)$. This is very inefficient for a large number of elements and not practical for real life applications.

For the **“set of words similarity”** approach, 1000 words = 0.13s, 2000 words = 0.27s, 4000 words = 0.58s. These results exhibit a linear scalability $O(n)$. This makes sense as for each word in the main text, the algorithm checks for its presence which takes O(1) time complexity. The iteration through n number of unique words would take O(1) * n = O(n). Similar to

For the **CBF Version 1** where we use the Counting Bloom Filter to check for unique words in both documents, it is fairly constant, as expected. Although for the small number of words, its running time is quite high compared to the “word for word” method, it stays constant as the number of words increases, which the other method does not.

CBF Version 2 where we check for groups of words that are in both documents, where the number of words per group is adjustable. For this experiment, each group of words contains 3 words. This approach has the best results in terms of plagiarism accuracy and time complexity. The execution time stays constant as the number of words increases and the actual running time is also lower compared to other approaches.

We further increase the number of words and observe the scaling behaviors:



For very large numbers of words, we can see the clear difference between CBF and non-CBF approaches. This result showcases how CBF is particularly useful for handling large amounts of data efficiently, O(1).

LOs and HCs

#ComputationalCritique

Strengths and limitations of CBF were discussed thoroughly. From Prof's suggestion, I also performed an empirical analysis to examine how the similarity percentage changes as the number of words/ group increases. I also explained the purpose and mechanism of each approach and why CBF version 2 was the most efficient and accurate one. (53)

#ComplexityAnalysis

This LO has been challenging for me. Based on previous feedback, I tried to reduce the noise in my graphs but providing two graphs with different ranges in the x-axis for the comparison of four approaches. For theoretical analysis, I explained specific formulas and data structure in examining whether the empirical analysis matches. (53)

#CodeReadability

I have received fine grades for this LO before. In this assignment, besides clear variable names, adequate comments, and concise organization, I added more space between different parts and lines of my code. This small detail made the overall codes look less intimidating and more precise, especially for an audience with limited knowledge about this topic. (56)

#PythonProgramming

I was advised to add the tests in the Jupiter notebook file to show that they work as described and I have made sure to include all the tests and graphs in this assignment. The tests (for operations in CBF and its application as a plagiarism checker) were also meaningful and comprehensive to demonstrate this data structure's uses. (58)

#AlgoStratDataStruct

I provided contextualized discussions about why the Counting Bloom Filter is beneficial and explained the three operations of constant scalability. I also provided a particular example to demonstrate the hashing technique and discussed CBF's strengths and limitations. Prof. encouraged me to elaborate on specific details, and I improved by explaining formulas and giving the purpose of every term/ variable. (58)

#Professionalism

This is my lowest LO score. Based on the feedback, I will make sure to submit the PDF file as the primary source besides the zip file. I also double-checked to make sure the graph images are presented as expected in PDF format.

#dataviz

Line graphs were utilized in this assignment for complexity analysis to explain the results of the empirical analysis. All graphs include titles, legends, labels, and explanations. I made sure that the scale was appropriate for an effective display of information, and to draw inferences from. (45)

#analogies

I used the example of a store manager searching for a coat in their inventory to explain how Counting Bloom Filter and hash functions work. This example is employed throughout the first part of the report to demonstrate different methods, making the information more concrete and the report more consistent. (50)

#audience

Tailored for readers who are new to this topic, the report strikes a balance between technical details and simplicity. I've defined key terms and explained their importance to simplify complex ideas. The inclusion of specific examples with tables ensures accessibility for the audience without compromising the depth of the report. (50)

References

(2023, March 2)- YouTube. Retrieved December 13, 2023, from https://www.youtube.com/watch?v=mFY0J5W8Udk&ab_channel=AbdulBari

Bloom filter. (n.d.). Wikipedia. Retrieved December 13, 2023, from https://en.wikipedia.org/wiki/Bloom_filter

3 rules of thumb for bloom filters. (2010, August 25). corte.si. Retrieved December 13, 2023, from https://corte.si/posts/code/bloom-filter-rules-of-thumb/

Appendix

AI statement: I used Grammarly to proofread. I used ChatGPT to brainstorm the difference between “set of words similarity” and “plagiarism_percentage_v1” approaches. Otherwise, no significant use of AI tools for this assignment.