CS110 Final - Building a Plagiarism Detector Using CountingBloom Filters (CBFs)

Kennedy Emeruem

April 19, 2023

Contents

Counting Bloom Filters as a Data Structure

Counting bloom filters are an extension of the bloom filter data structure. Bloom filters are data structures that are used to determine the presence of an element in a set of elements. For example, given a set of 10 words [“apple”, “banana”, “orange”, “grape”, “pear”, “pineapple”, “watermelon”, “mango”, “kiwi”, “strawberry”], if we were to check if the word “kiwi” is in this set, we would have to go though every element in the list to determine if it is present. This is a very inefficient way of determining if an element is in the list because in the worst case, we would have to go through every element in the list just to see that the element is not present. As the number of words in the set increases, the time it takes to determine if an element is in the set increases linearly.

This is not good when we have large sets of data. Bloom filters are the solution to this problem. They are very efficient as they can determine if an element is in a set in constant time. The bloom filter is a bucket of bits that are all initialized to 0. each of those bits can either be on or off (have a value of 1 or 0).

To add an element to the bloom filter, we follow the following steps:

1. Run the element through a hash function to get a hash value. This value will be between 0 and the size of the bloom filter.
2. Set the bit at the index of the hash value to 1.

Note: This hash function has to be deterministic. This means that if we run the same element through the hash function multiple times, we will get the same hash value each time.

To determine if an element is in the set, we follow the following steps:

1. Run the element through the same hash function to get a hash value.
2. Check if the bit at the index of the hash value is 1. If it is, then the element is in the set. If it is not, then the element is not in the set.

However, it is important to note that running two different elements through the same hash function could result in the same hash value (This is called a collision). This means that here is a chance that the element is not in the set, but the bloom filter will return that it is in the set. This is called a false positive. The probability of a false positive is called the false positive rate.

There are two ways to reduce the false positive rate. Some are as follows

1. Increase the size of the bloom filter: This will reduce the false positive rate by reducing the chance of a collision.
2. Using a better hash function.
3. Use multiple hash functions to add and search for elements. This is done by running the element through the hash function a seed and checking if all the bit at the index is already set to 1. if it is then we hash the element with a new seed again and check the but at the index.

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, n is the number of elements in the set, and m is the size of the bloom filter.(corte.si, 2023)

The counting bloom filter extends this data structure by adding a counter to each bit in the bloom filter. This counter is used to determine the number of times a bit has been set to 1. This is done by incrementing the counter by 1 every time a bit is set to 1. Doing this allows us to remove elements from the bloom filter. A function that could not be done with the regular bloom filter as regardless of how many times an element is added to the bloom filter, the bit at the index of the hash value will always be set to 1. This means that turning a bit off could remove more than one element from the set.

Counting Bloom Filter Operations

As mentioned above, the counting bloom filter extends the bloom filter by adding the ability to remove elements from the set. As such the counting bloom filter has the same operations as the bloom filter. The operations are as follows:

insert. Insertion is done by running the element through the hash function to get a hash value. We then increment the counter at the index of the hash value by 1. The pseudo code for this process is as follows:

```
Insert(element):
    Index = Hash(element)
    Counter[Index] += 1
```

As we can see from the pseudo code, the insertion process is very simple. Running the element through the hash function is a constant time operation $O(1)$. Incrementing the counter at the index of the hash value is also a constant time operation $O(1)$. As such, the insertion process is a constant time operation $O(1)$. This means that regardless of the size of the bloom filter, the runtime of the insertion process will always be the same.

Remeber that to prevent collisions, we need to use multiple hash functions. Adjusting the pseudo code to use multiple hash functions is as follows:

```
Insert(element):
    for i in range(0, k):
        Index = Hash(element, i)
        Counter[Index] += 1
```

In this case, the insertion process is still a constant time operation $O(1)$ because the number of hash functions is constant (k).

Search. Searching is done by running the element through the hash function to get a hash value. We then check if the counter at the index of the hash value is greater than 0. If it is, then the element is in the set. If it is not, then the element is not in the set. The pseudo code for this process is as follows:

```
Search(element):
    Index = Hash(element)
    if Counter[Index] > 0:
        return True
    else:
        return False
```

As we can see from the pseudo code, the search process is very simple. Running the element through the hash function is a constant time operation $O(1)$. Checking if the counter at the index of the hash value is greater than 0 is also a constant time operation $O(1)$. As such, the search process is a constant time operation $O(1)$. This means that regardless of the size of the bloom filter, the runtime of the search process will always be the same.

Remeber that to prevent collisions, we need to use multiple hash functions. Adjusting the pseudo code to use multiple hash functions is as follows:

```
Search(element):
    for i in range(0, k):
        Index = Hash(element, i)
        if Counter[Index] == 0:
            return False
    return True
```

In this case, the search process is still a constant time operation $O(1)$ because the number of hash functions is constant (k).

Delete. Deleting is done by running the element through the hash function to get a hash value. We then decrement the counter at the index of the hash value by 1. The pseudo code for this process is as follows:

```
Delete(element):
    Index = Hash(element)
    Counter[Index] -= 1
```

As we can see from the pseudo code, the deletion process is very simple. Running the element through the hash function is a constant time operation $O(1)$. Decrementing the counter at the index of the hash value is also a constant time operation $O(1)$. As such, the deletion process is a constant time operation $O(1)$.

As we previously stated for insertion and deletion, we want to avoid collisions by running the element though multiple hash functions. Adjusting the pseudo code to use multiple hash functions is as follows:

```
Delete(element):
    for i in range(0, k):
        Index = Hash(element, i)
        Counter[Index] -= 1
```

In this case, the deletion process is still a constant time operation $O(1)$ because the number of hash functions is constant (k) and k is small compared to n as n approaches infinity.

Hashing. As explained above, we need a hash function to map elements to an index in the counting bloom filter. We discussed that in order to reduce the risk of collision we will need a good hashing function. For this implementation of Counting bloom filters, I have decided to use a regular hashing function with double hashing for open addressing. Open addressing is the process of reducing collisions in the filter by systematically searching for a new index to increment. With double hashiing, we use the first hash function to determine the sirat index to check and then we step the index using the second hash function. The formula is given by

$$\text{double hash}(key, i) = (hash_1(k) + i \times hash_2) \mod x$$

For my implementation, below are the two hash functions I have opted to use

```
hash_1(key):

    return key % num_slots



hash_2():

    prime = 31

    return prime - (key mod prime)
```

The overall hash function is

“‘Java hash(key, i): (hash_fn1(key) + idx * hash_fn2(key)) mod num_slots

Applications of Counting Bloom Filters

Since lookups and insertions in a counting bloom filters are very efficient (constant time operations), they are often used in applications where we need to check if an element is in a set very quickly. Some of these applications are as follows:

1. **IP Address Filtering:** IP addresses are used to identify devices on a network. IP addresses are often used to filter traffic on a network. For example, a company may want to block all traffic from a specific IP address. In this case, the company would add the IP address to the counting bloom filter. When a packet is received, the IP address of the packet is checked against the counting bloom filter. If the IP address is in the counting bloom filter, then the packet is dropped. If the IP address is not in the counting bloom filter, then the packet is allowed to pass through the network.

   A counting bloom filter is a good choice for this application because the number of

IP addresses that need to be checked is very large and we do not want to spend a lot of time checking each IP address in a database or a list (linear time operations $O(n)$ or $O(n \log n)$ with optimized search algorithms). As such, a counting bloom filter is a good choice because lookups are constant time operations $O(1)$.

Additionally, flagging IP addresses by adding them to the counting bloom filter is efficient because insertion is a constant time operation $O(1)$. The deletion process is also important in this application because the company may want to unblock an IP address. As such, the company would remove the IP address from the counting bloom filter. This is also a constant time operation $O(1)$.

2. **Inventory management:** Stores and libraries usually have a large inventory of items. When a user wants to buy an item or borrow a book, the store or library will have to check if they have the item that the user requested. Since the databases of items are very large, a counting bloom filter is a good choice for this application because lookups are constant time operations $O(1)$. As such, the store or library can quickly determine if the item is available before going to search for the item in the database. This constant lookup is better than searching through a database or a list (linear time operations $O(n)$ or $O(n \log n)$ with optimized search algorithms). Additionally, when an item is sold or borrowed, the item is removed from the inventory. As such, the item is removed from the counting bloom filter. This is also a constant time operation $O(1)$. If the item is returned or restocked, then the item is added back to the inventory. As such, the item is added back to the counting bloom filter. This is also a constant time operation $O(1)$.

Other applications of counting bloom filters include: 1. password checking 2. spam filtering 3. malware detection

Python Implementation of Counting Bloom Filters

[34]:

```
class CountingBloomFilter():

    def __init__(self, num_items, num_hashfn):
        """
        Initialize a Counting Bloom Filter

        Parameters
        ----------
        num_items: int, number of items to be inserted
        num_hashfn: int, number of hash functions to be used

        Returns
        -------
        None
        """
        self.num_items = num_items
        self.num_hashfn = num_hashfn
        # multiply the number of items by 10 to get the bucket size to␣
 ↪reduce the number of collisions
```

```
        self.bucket_size = num_items * 10
        self.bucket = [0] * self.bucket_size
        self.stored_item_count = 0

    def string_to_int(self, string, base=10):
            '''
            Converts a string to an integer by summing the product of the␣
 ↪ASCII value of each character and
            128 to the power of the position of the character in the␣
 ↪string.

            Inputs:
                string: string
                base: integer
            Output:
                integer
            '''
            #keep track of the length of the string
            string_length = len(string)
            final_int = 0
            # add the product of the ASCII value of each character and␣
 ↪base to the power of the position of the character in the string
```

```
            for position,char in enumerate(string):
                final_int += ord(char) * (base ** (string_length -␣
↪position - 1))
            return final_int

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

        Parameters
        ----------
        item: string, item to be hashed
        i: integer, index of the hash function

        Returns
        -------
        hash_values: list, list of hash values
        """
        # convert the string to an integer
        key = self.string_to_int(item)

        def hash_fn1(key):
```

```
        '''
        Hashes a string using the first hash function.

        Inputs:
            key: string
        Output:
            integer
        '''
        return key % self.bucket_size

    def hash_fn2(key):
        '''
        Hashes a string using the second hash function.

        Inputs:
            key: string
        Output:
            integer
        '''
        prime = 31
        return prime - (key % prime)
    return (hash_fn1(key) + idx * hash_fn2(key)) % self.bucket_size


def search(self, item):
    """
    Determine if an item is in the counting bloom filter or not

    Parameters
    ----------
        item: string, item to be queried

    Returns
    -------
        boolean: True if item is in the filter, False otherwise
    """
    for i in range(self.num_hashfn):
        index = self.hash_cbf(item, i)
        if self.bucket[index] == 0:
            return None
    return True

def insert(self, item):
    """
    Insert an item to the filter

    Parameters
    ----------
        item: str, item to be inserted

    Returns
    -------
        None
    """
    # ensure that the filter is not full
    if self.stored_item_count == self.num_items:
        raise Exception("The filter is full")

    # loop through the number of hash functions
    for i in range(self.num_hashfn):
        # get the index of the hash function
        index = self.hash_cbf(item, i)
        # if the index is 0, set it to 1 and if it is not 0,␣
↪increment it by 1
        if self.bucket[index] == 0:
            self.bucket[index] = 1
        else:
            self.bucket[index] += 1
    # increment the number of stored items by 1
    self.stored_item_count += 1

def delete(self, item):
    """
    Delete an item from the filter

    Parameters
    ----------
    item: str, item to be deleted

    Returns
    -------
    None
    """
    # ensure that the item exists in the CBF
    item_exists = self.search(item)
    # if the item exists, loop through the number of hash functions and decrement the index of the hash function by 1
    if item_exists:
        for i in range(self.num_hashfn):
            index = self.hash_cbf(item, i)
            self.bucket[index] -= 1
            # if the index is 0, break out of the loop because we cannot decrement it any further
            if self.bucket[index] == 0:
                # decrement the number of stored items by 1 and break out of the loop
                self.stored_item_count -= 1
                break
        # decrement the number of stored items by 1 if we did not break out of the loop
        self.stored_item_count -= 1

def __str__(self):
    """
    Returns a string representation of the filter
    """
    return str(self.bucket)
```

Note: I multiplied the number of slots by 10 to reduce the risk of false positives by spreading out the elements in the counting bloom filter. This is not necessary, but it is a good practice to reduce the risk of false positives.

Testing the Counting Bloom Filter

[33]:
```
# a list of all the letters in the alphabet
letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l',␣
 ↪'m', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z']
# create a cbf to store 26 elements with 7 hash functions
letters_CBF = CountingBloomFilter(26, 7)


# Testing the insert functionality
for letter in letters:
    letters_CBF.insert(letter)


try:
    for letter in letters:
        assert letters_CBF.search(letter) == True, f"Expected True but␣
 ↪got {letters_CBF.search(letter)} for {letter}"
    print("Insertion Tests passed ")
except Exception as e:
    print(e)
```

```
# test cases for the search function
test_cases = {
    'a': True,
    'b': True,
    'c': True,
    'd': True,
    'qwerty': None,
    'z': True,
    'ab': None,
    'abfc': None,
}

try:
    #going through the test cases
    for item, expected in test_cases.items():
        actual = letters_CBF.search(item)
        assert actual == expected, f"Expected {expected} but got {actual}␣
 ↪for {item}"
    print("Search Tests passed ")
except Exception as e:
    print(e)
```

```
# Testing the delete functionality
to_delete = ['a', 'b', 'c', 'd', 'z']

try:
    # deleting the items
    for item in to_delete:
        letters_CBF.delete(item)

    # searching for the deleted items
    for item in to_delete:
        assert letters_CBF.search(item) == None, f"Expected None but got␣
 ↪{letters_CBF.search(item)} for {item}"
    print("Deletion Tests passed ")
except Exception as e:
    print(e)
```

```
Insertion Tests passed
Search Tests passed
Deletion Tests passed
```

Verifying the effectiveness of the Counting Bloom Filter

```
[43]: from requests import get

url = 'https://gist.githubusercontent.com/raquelhr/
 ↪78f66877813825dc344efefdc684a5d6/raw/
 ↪361a40e4cd22cb6025e1fb2baca3bf7e166b2ec6/'

def get_txt_into_list_of_words(url):
    '''Cleans the text data
    Input
    ----------
    url : string
    The URL for the txt file.
    Returns
    -------
    data_just_words_lower_case: list
    List of "cleaned-up" words sorted by the order they appear in the␣
 ↪original file.
    '''
    bad_chars = [';', ',', '.', '?', '!', '_', '[', ']', '(', ')', '*']
    data = get(url).text
    data = ''.join(c for c in data if c not in bad_chars)
    data_without_newlines = ''.join(c if (c not in ['\n', '\r', '\t'])␣
↪else " " for c in data)
    data_just_words = [word for word in data_without_newlines.split(" ")␣
↪if word != ""]
    data_just_words_lower_case = [word.lower() for word in␣
↪data_just_words]
    return data_just_words_lower_case
```

[45]:
```
all_words = get_txt_into_list_of_words(url)


# get the unique words from the list of all words
unique_words = set(all_words)


print(f"There are {len(unique_words)} unique words in the text")
```

```
There are 33153 unique words in the text
```

[49]:
```
def calculate_fpr(input_words, test_words, num_hashfn):
    '''
    Calculates the false positive rate of the counting bloom filter
    Parameters
    ----------
        input_words: list, list of words to be inserted into the counting␣
↪bloom filter
```

```
        test_words: list, list of words to be tested for existence in the counting bloom filter

    Returns
    -------
    fpr: float, the false positive rate of the counting bloom filter
    '''
    # create a counting bloom filter to store the input words
    num_items = len(input_words)
    cbf = CountingBloomFilter(num_items, num_hashfn)
    # insert the input words into the counting bloom filter
    for word in input_words:
        cbf.insert(word)

    # check if all the test words for existence in the counting bloom filter
    false_positives = 0
    for word in test_words:
        if cbf.search(word) == True:
            false_positives += 1
    # calculate the false positive rate
    fpr = false_positives / len(test_words)
    return fpr
```

How False Positives Rate scales with the number of hash functions

To see how the false positive rate scales with the number of hash functions, we will run the following experiment:

1. Split up the unique words from shakespear's works into n equal parts.
2. For each part, create a counting bloom filter and add all the words from the part to the counting bloom filter.
    1. Randomly select 5 other parts and check if the words from the other parts are in the counting bloom filter.
    2. For each part, calculate the false positive rate.
    3. Average the false positive rates from the 5 parts.

This experiment will be run for 1 to n hash functions.

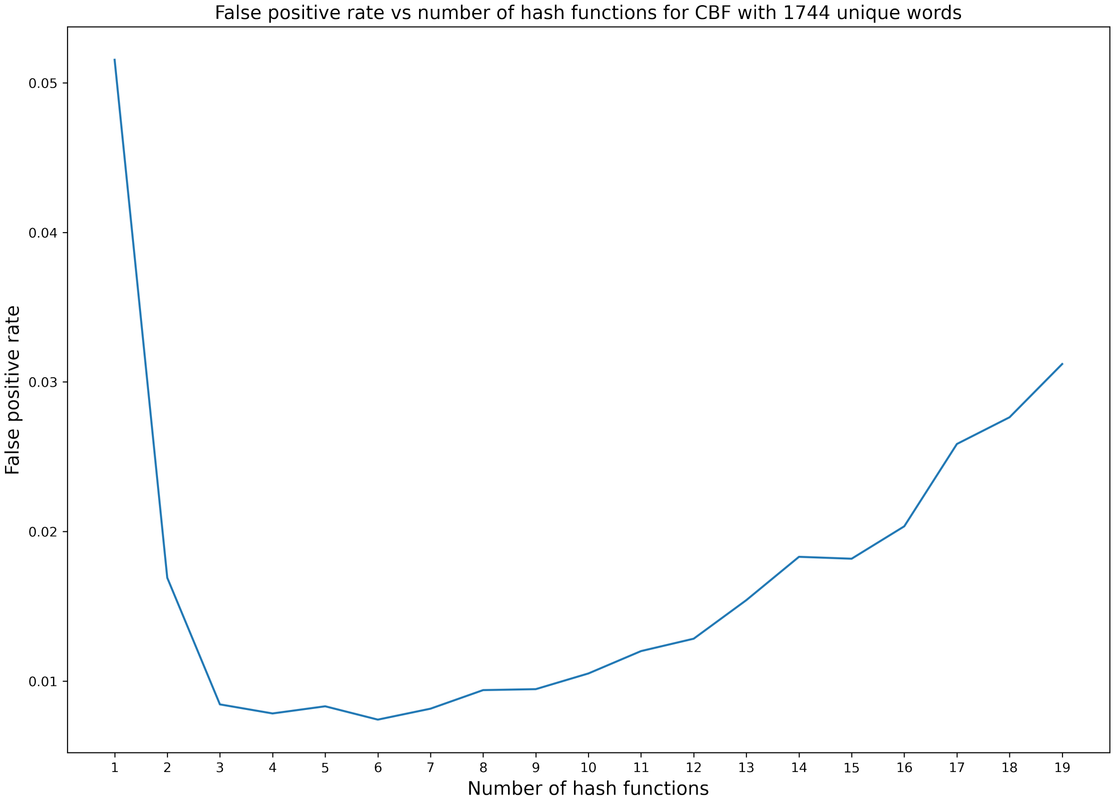
[96]:

```
import random
import matplotlib.pyplot as plt
num_hash_functions = [x for x in range(1,20)]
fpr_list = []
unique_words_list = list(unique_words)
unique_words_list_len = len(unique_words_list)
hash_fns_count = len(num_hash_functions)

# split the list of unique words into a certian number of parts based on␣
 ↪the number of hash functions
```

```
parts_dict = {}

for idx in range(hash_fns_count):
    # get the index of the words to be inderted into the counting bloom␣
 ↪filter
    start_idx = idx * unique_words_list_len // hash_fns_count
    end_idx = (idx + 1) * unique_words_list_len // hash_fns_count

    # get the words to be inserted into the counting bloom filter
    words = unique_words_list[start_idx:end_idx]

    # add the words to the dictionary
    parts_dict[idx] = words

for idx in range(hash_fns_count):
    # get the words to be inserted into the counting bloom filter
    input_words = parts_dict[idx]
    fa_pr = 0
    non_idx = [x for x in range(hash_fns_count) if x != idx]
    tests = random.sample(non_idx, 5)
    for test in tests:
        # get the words to be tested for existence in the counting bloom␣
 ↪filter
        test_words = parts_dict[test]
        # calculate the false positive rate for the counting bloom filter
        fa_pr += calculate_fpr(input_words, test_words,␣
 ↪num_hash_functions[idx])
    # calculate the average false positive rate
    fa_pr = fa_pr / (hash_fns_count - 1)
    fpr_list.append(fa_pr)
```

[102]:

```
plt.figure(figsize=(14, 10), dpi=300)
plt.plot(num_hash_functions, fpr_list)
plt.xlabel("Number of hash functions", fontsize=14)
plt.ylabel("False positive rate", fontsize=14)
plt.title(f"False positive rate vs number of hash functions for CBF with␣
 ↪{len(parts_dict[0])} unique words", fontsize=14)
plt.xticks(num_hash_functions)
plt.show()
```



Theorerically, we know that there is an optimal number of hash functions that will minimize the false positive rate. The formula for the optimal number of hash functions is as follows:

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

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, 2023)

This is derived from 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 elements in the counting bloom filter, and e is the base of the natural logarithm. (Corte.si, 2023)

The graph above shows that the false positive rate decreases as the number of hash functions increases but then increases again as the number of hash functions crosses 7. As such, the optimal number of hash functions is somewhere between 5 and 7. This is consistent with the theoretical assumptions of how the false positive rate scales with the number of hash functions.

How access time to hashed values scale with the number of items stored at a fixed False Positive Rate

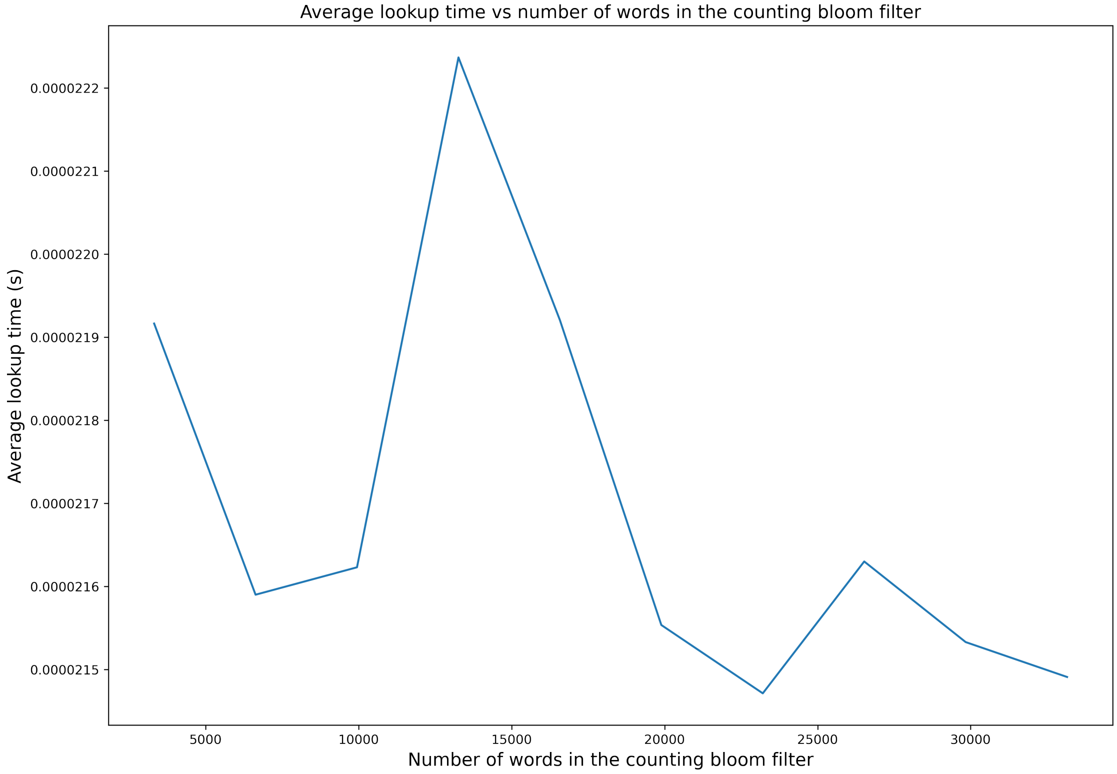
To determine how the access time to hashed values scale with the number of items stored at a fixed False Positive Rate, we will computer the average lookup time for a counting bloom filter with a certain number of words hashed to it ans see how the average will change as we increase the number of words hashed to it.

[135]:

```
import math

import time





fpr = 0.01
```

```
# calculate the relevant parameters for the counting bloom filter given␣
 ↪the false positive rate
n = len(unique_words)
fpr = 0.01
m = round(- (n * math.log(fpr)) / (math.log(2) ** 2))
k = round((m / n) * math.log(2))

sections = 10
section_dict = {}
times = []

for idx in range(sections):
    # get the index of the words to be inderted into the counting bloom␣
 ↪filter
    start_idx = 0
    end_idx = (idx + 1) * unique_words_list_len // sections

    # get the words to be inserted into the counting bloom filter
    words = unique_words_list[start_idx:end_idx]

    # add the words to the dictionary
    section_dict[idx] = words

for idx, words in section_dict.items():
    # create a counting bloom filter to store the input words
    num_items = len(words)
    cbf = CountingBloomFilter(num_items, k)
    # insert the input words into the counting bloom filter
    for word in words:
        cbf.insert(word)

    # loop though the words in the section and check if they exist in the counting bloom filter
    total_time = 0
    for word in words:
        start = time.time()
        cbf.search(word)
        end = time.time()
        total_time += end - start
    # calculate the average time it takes to check if a word exists in the counting bloom filter
    avg_time = total_time / len(words)
    times.append(avg_time)
```

[134]:
```
plt.figure(figsize=(14, 10), dpi=300)

plt.ticklabel_format(style='plain', axis='y')

plt.plot([len(section_dict[x]) for x in range(sections)], times)

plt.xlabel("Number of words in the counting bloom filter", fontsize=14)

plt.ylabel("Average lookup time (s)", fontsize=14)

plt.title(f"Average lookup time vs number of words in the counting bloom␣
 ↪filter", fontsize=14)

plt.show()
```



As we can see from the plots above, the average time to access a hashed value is fairly constant () as the number of items stored in the counting bloom filter increases for a fixed False Positive Rate. This is the expected behavior according to the theoritical ananlysis of lookup time in a counting bloom filter.

How does memeory size scale with False Positive Rate

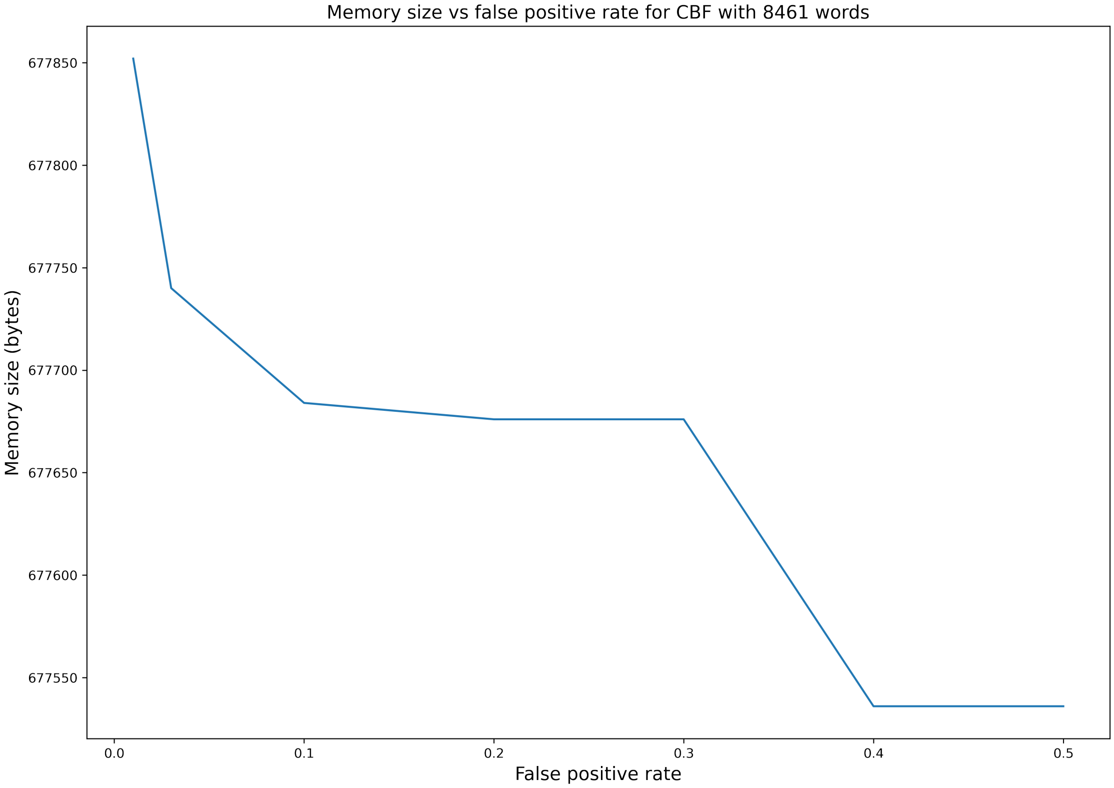
Theoretically, the larger the memory size (the number of counters in the counting bloom filter), the lower the false positive rate for a fixed number of hash functions and the number of items hashed to the counting bloom filter. This is the case because with more counters, we will reduce the number of collisions when hashing an element. As such, the false positive rate will decrease. Testing this experimentally:

[154]:

```python
import tracemalloc
fprs = [0.01, 0.02, 0.03, 0.1, 0.2, 0.3, 0.4, 0.5]
memory_sizes = []
url_version_1 = "https://bit.ly/39MurYb"
version_1 = get_txt_into_list_of_words(url_version_1)

for fpr in fprs:
    # number of elements to be stored in the counting bloom filter
    n = len(version_1)
    # ideal size of the counting bloom filter
    m = round(- (n * math.log(fpr)) / (math.log(2) ** 2))
    # ideal number of hash functions
    k = round((m / n) * math.log(2))

    tracemalloc.start()

    cbf = CountingBloomFilter(n, k)

    for word in version_1:

        cbf.insert(word)

    current, peak = tracemalloc.get_traced_memory()

    memory_sizes.append(peak)

    tracemalloc.stop()
```

[156]:

```
plt.figure(figsize=(14, 10), dpi=300)

plt.ticklabel_format(style='plain', axis='y')

plt.plot(fprs, memory_sizes)

plt.xlabel("False positive rate", fontsize=14)

plt.ylabel("Memory size (bytes)", fontsize=14)

plt.title(f"Memory size vs false positive rate for CBF with␣

 ↪{len(version_1)} words", fontsize=14)

plt.show()
```



Examining the graph above, we can see that fpr increases as the memory size decreases. This is consistent with the theoretical analysis of how the false positive rate scales with the memory size.

How does the memory size scale with the number of stored for a fixed False Positive Rate

Plagerism Detection using Counting Bloom Filters

In order to use a counting bloom filter to detect plagerism, we will first define what plagerism between two texts is. In this case, we will define plagerism as the following:

> How much of the text in one document is also present in another document.

As such, we will use the following formula to calculate the extent of plagerism of one document with respect to another document:

$$Plagerism = \frac{Number\ of\ words\ in\ document\ 1\ that\ are\ also\ in\ document\ 2}{Number\ of\ words\ in\ document\ 1}$$

Usina a counting bloom filter, we can quickly determine if a given word is in a document by checking if the word is in the counting bloom filter for that document. As such, we can quickly determine the number of words in document 1 that are also in document 2. As such, we can quickly determine the extent of plagerism between two documents. comparing the words one by one is feasible but it is not efficient. If we have a document with n words, for each word, we will search through the other document to see if the word is present. This is a linear time operation $O(n)$ or $O(\log n)$ with optimized search algorithms. Hence as the number of words in a document increases, the time complexity for determining the extent of plagerism between two will scale quadratically $O(n^2)$ or $O(n \log n)$ with optimized search algorithms. However, using a counting bloom filter, we can determine the extent of plagerism between two documents in linear time $O(n)$ because for every n words in document 1, lookups are constant time operations $O(1)$.

Note: By words, we mean unique words.

Below is a python implementation this method of plagerism detection using a counting bloom filter.

[192]:

```
def compute_plagiarism_extent(text_words,reference_text_words):
    '''
    Determines how similar a text is to a given reference text.

    Parameters
    ----------
    text_words : list of str
        The words of the text to be checked for plagiarism.
    reference_text_words : list of str
        The words of the reference text.

    Returns
    -------
    float
        The extent of plagiarism in the text, as a percentage.
    '''
    # create a counting bloom filter to store the words in the reference␣
 ↪text
    num_items = len(reference_text_words)
    cbf = CountingBloomFilter(num_items, 7)
    # insert the words in the reference text into the counting bloom␣
 ↪filter
```

```
    for word in set(reference_text_words):
        cbf.insert(word)

    similar_words = 0
    # loop through the words in the text to be checked for plagiarism
    for word in set(text_words):
        # check if the word exists in the counting bloom filter
        if cbf.search(word):
            similar_words += 1
    # calculate the percentage of words in the text to be checked for␣
 ↪plagiarism that are similar to the reference text
    plagiarism_extent = similar_words / len(text_words) * 100
    return plagiarism_extent

url_version_1 = 'https://bit.ly/39MurYb'
url_version_2 = 'https://bit.ly/3we1QCp'
url_version_3 = 'https://bit.ly/3vUecRn'

version_1 = get_txt_into_list_of_words(url_version_1)
version_2 = get_txt_into_list_of_words(url_version_2)
version_3 = get_txt_into_list_of_words(url_version_3)
```

[194]:
```
similarity_1_2 = compute_plagiarism_extent(version_1, version_2)

similarity_1_3 = compute_plagiarism_extent(version_1, version_3)

similarity_2_3 = compute_plagiarism_extent(version_2, version_3)


print(f"Similarity between version 1 and version 2: {similarity_1_2:.
 ↪2f}%")

print(f"Similarity between version 1 and version 3: {similarity_1_3:.
 ↪2f}%")

print(f"Similarity between version 2 and version 3: {similarity_2_3:.
 ↪2f}%")
```

```
Similarity between version 1 and version 2: 26.33%

Similarity between version 1 and version 3: 26.33%

Similarity between version 2 and version 3: 26.33%
```

This method of plagerism detection is flawed because of how we are defining plagerism. We are defining plagerism as the number of similar words between two documents. However, it is possible that the two documents use simialr words but the words are used in different contexts. As such, the two documents are not plagerized. Hence, extending our definition of plagerism,

> Plagerisom is the extent to which groups of text are similar between two documents.

it is better to use groups of words instead of individual words to determine the extent of plagerism because a groupd of words will convery more information and context about the text than a single word. We still benefit from using a counting bloom filter because we can quickly determine if a group of words is in a document. As such, we can quickly determine the extent of plagerism between two documents.

The formula for determining the extent of plagerism between two documents is as follows:

$$Plagerism = \frac{Number\ of\ groups\ of\ words\ in\ document\ 1\ that\ are\ also\ in\ document\ 2}{Number\ of\ groups\ of\ words\ in\ document\ 1}$$

where a group of words is a set of words that are adjacent to each other in the document.

Note that we will have to determine how many words to include in a group of words as the grouping would determine the extent of plagerism.

Below is a python implementation of this method of plagerism detection using a counting bloom filter.

[199]:

```python
def group_words(words, n):
    '''
    Groups words into groups of n words.

    Parameters
    ----------
    words : list of str
        The words to be grouped.
    n : int
        The number of words in each group.

    Returns
    -------
    new_words of list of str
        The list of groups of words.
    '''
    new_words = []
    for idx in range(0, len(words), n):
        group = ' '.join(words[idx:idx + n])
        new_words.append(group)
    return new_words

def compute_plagerism_extent_2(text_words, reference_text_words, n):
    '''
    Determines how similar a text is to a given reference text.

    Parameters
    ----------
    text_words : list of str
        The words of the text to be checked for plagiarism.
    reference_text_words : list of str
        The words of the reference text.

    Returns
    -------
    float
        The extent of plagiarism in the text, as a percentage.
    '''
    # group the words in groups of n words
    text_words = group_words(text_words, n)
    reference_text_words = group_words(reference_text_words, n)

    # create a counting bloom filter to store the words in the reference text
    num_items = len(reference_text_words)
    cbf = CountingBloomFilter(num_items, 4)
    # insert the words in the reference text into the counting bloom filter
    for word in reference_text_words:
        cbf.insert(word)

    similar_words = 0
    # loop through the words in the text to be checked for plagiarism
    for word in text_words:
        # check if the word exists in the counting bloom filter
        if cbf.search(word):
            similar_words += 1
    # calculate the percentage of words in the text to be checked for␣
 ↪plagiarism that are similar to the reference text
    plagiarism_extent = similar_words / len(text_words) * 100
    return plagiarism_extent
```

[201]:

```
#group words into groups of 5 words


similarity_1_2 = compute_plagerism_extent_2(version_1, version_2, 5)
similarity_1_3 = compute_plagerism_extent_2(version_1, version_3, 5)
similarity_2_3 = compute_plagerism_extent_2(version_2, version_3, 5)


print(f"Similarity between version 1 and version 2: {similarity_1_2:.
 ↪2f}%")
print(f"Similarity between version 1 and version 3: {similarity_1_3:.
 ↪2f}%")
print(f"Similarity between version 2 and version 3: {similarity_2_3:.
 ↪2f}%")
```

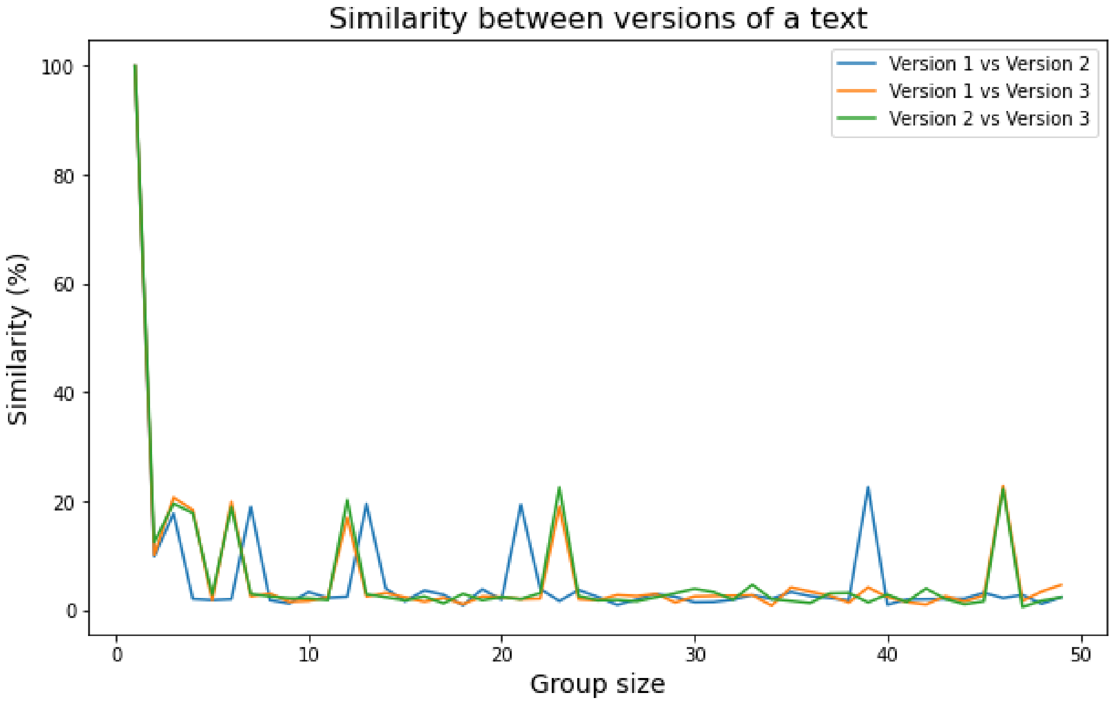
```
Similarity between version 1 and version 2: 1.83%

Similarity between version 1 and version 3: 1.77%

Similarity between version 2 and version 3: 2.84%
```

As we can see, the extent of plagerism between two documents is much lower than when we used individual words. The higher the number of words in a group of words, the lower the extent of plagerism we will detect between two documents. See the graph below to see how the extent of plagerism changes as the number of words in a group of words changes. the similarity score reduced exccluding random spikes in the graph.

[203]:

```python
group_sizes = [n for n in range(1, 50)]
similarities = []
n_trials = 5


# loop through the different group sizes
for group_size in group_sizes:
    # compute the similarity between the three versions of the text
    similarity_1_2 = 0
    similarity_1_3 = 0
    similarity_2_3 = 0
    for _ in range(n_trials):
        # for each trial, compute the similarity between the three␣
 ↪versions of the text
```

```
        similarity_1_2 += compute_plagerism_extent_2(version_1,␣
 ↪version_2, group_size)
        similarity_1_3 += compute_plagerism_extent_2(version_1,␣
 ↪version_3, group_size)
        similarity_2_3 += compute_plagerism_extent_2(version_2,␣
 ↪version_3, group_size)
    similarities.append([similarity_1_2 / n_trials, similarity_1_3 /␣
 ↪n_trials, similarity_2_3 / n_trials])


plt.figure(figsize=(10, 6))
plt.plot(group_sizes, similarities)
plt.legend(["Version 1 vs Version 2", "Version 1 vs Version 3", "Version␣
 ↪2 vs Version 3"])
plt.xlabel("Group size", fontsize=14)
plt.ylabel("Similarity (%)", fontsize=14)
plt.title("Similarity between versions of a text", fontsize=16)
plt.show()
```



Comparison with other methods of plagerism detection

Word for word comparison: With the strategy, for every word in document 1, we will have to search through document 2 to see if the word is present. we will then keep track of the number of words that are present in both documents and divide it by the number of words in document 1. Below is a python implementation of this method of plagerism detection.

[185]:

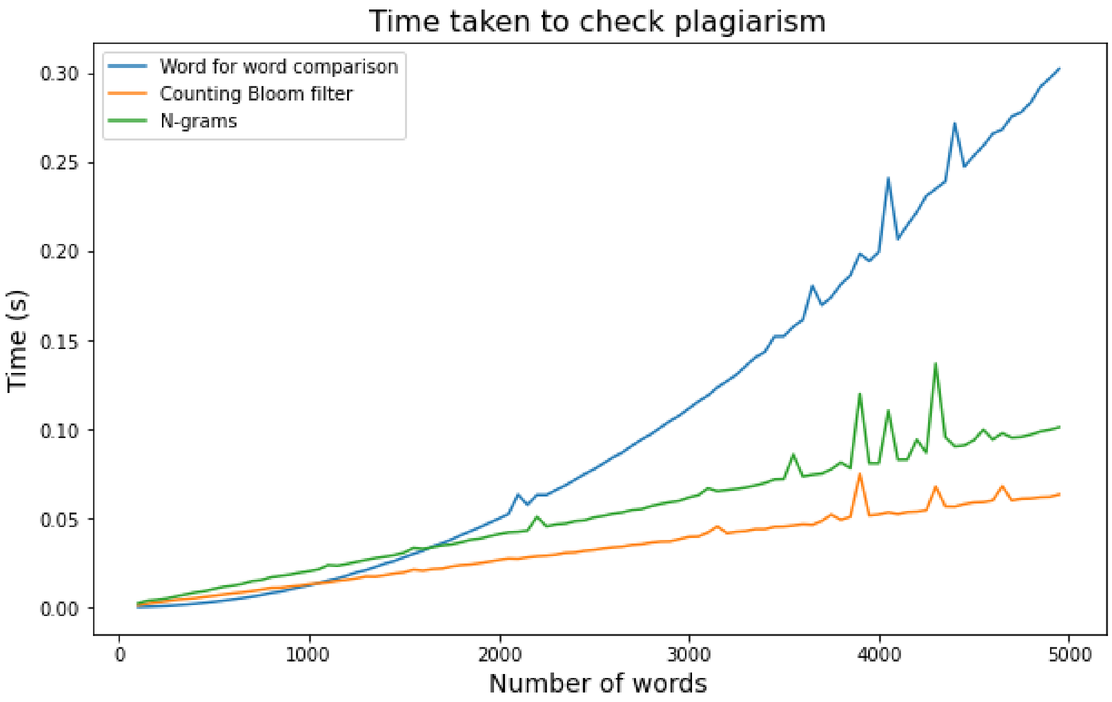
```
def word_for_word_comparison(text_1, reference_text_1):
    '''
    Determines how similar a text is to a given reference text.
    Parameters
    ----------
    text_1 : list of str
        The words of the text to be checked for plagiarism.
    reference_text_1 : list of str
        The words of the reference text.

    Returns
    -------
    float
        The extent of plagiarism in the text, as a percentage.
    '''
    similar_words = 0
    # loop through the words in the text to be checked for plagiarism
    for word in text_1:
        # check if the word exists in the reference text
        if word in reference_text_1:
            similar_words += 1
    # calculate the percentage of words in the text to be checked for␣
 ↪plagiarism that are similar to the reference text
    plagiarism_extent = similar_words / len(text_1) * 100
    return plagiarism_extent


similarity_1_2 = word_for_word_comparison(version_1, version_2)
similarity_1_3 = word_for_word_comparison(version_1, version_3)
similarity_2_3 = word_for_word_comparison(version_2, version_3)


print(f"Similarity between version 1 and version 2: {similarity_1_2:.
 ↪2f}%")
print(f"Similarity between version 1 and version 3: {similarity_1_3:.
 ↪2f}%")
print(f"Similarity between version 2 and version 3: {similarity_2_3:.
 ↪2f}%")
```

```
Similarity between version 1 and version 2: 100.00%
Similarity between version 1 and version 3: 100.00%
Similarity between version 2 and version 3: 100.00%
```

As we can see above, this method tells is that the entire document is plagerized. However, this is not the case because the works are different works of shakespeare. As such, this method of plagerism detection is flawed.

Comparing runtimes of different methods of plagerism detection. Below are experiments to compare the runtimes of different methods of plagerism detection as the number of words in a document increases. From the experiments, we can see that the counting bloom filter methods both outperform the word for word comparison method.

[223]:
```python
def generate_test_words(n):
    '''
    Generates a list of n random words.

    Parameters
    ----------
    n : int
        The number of words to generate.

    Returns
    -------
    list of str
        The list of random words.
    '''
    words = []
    letters = [chr(i) for i in range(ord('a'), ord('z') + 1)]
    for _ in range(n):
        word = ''
        for _ in range(5):
            word += random.choice(letters)
        words.append(word)
    return words


tests = [x for x in range(100, 5000, 50)]
wfw_times = []
cbf_times = []
n_groups_times = []
trials = 5

# run the tests for each algorithm and time the results
for n in tests:
    version_1 = generate_test_words(n)
    version_2 = generate_test_words(n)

    total_time = 0
    for _ in range(trials):
        start_time = time.time()
        compute_plagerism_extent_2(version_1, version_2, 5)
        end_time = time.time()
        total_time += end_time - start_time
    cbf_times.append(total_time / 5)
    total_time = 0
    for _ in range(trials):
        start_time = time.time()
        compute_plagiarism_extent(version_1, version_2)
        end_time = time.time()
        total_time += end_time - start_time
    n_groups_times.append(total_time / 5)

    total_time = 0
    for _ in range(trials):
        start_time = time.time()
        word_for_word_comparison(version_1, version_2)
        end_time = time.time()
        total_time += end_time - start_time
    wfw_times.append(total_time / 5)

plt.figure(figsize=(10, 6))
plt.plot(tests, wfw_times, label="Word for word comparison")
plt.plot(tests, cbf_times, label="Counting Bloom filter")
plt.plot(tests, n_groups_times, label="N-grams")
plt.legend()
plt.xlabel("Number of words", fontsize=14)
plt.ylabel("Time (s)", fontsize=14)

plt.title("Time taken to check plagiarism", fontsize=16)

plt.show()
```



HC and LO Appendix

#cs110-Professionalism: In the group LBA, my teammmate and I did not present our document in a professional manner. This time I ensured that my document contained a title page, table of contents, a proper appendix, and was formatted in APA style.

#cs110-ComplexityAnalysis: In the first assignment a comment I found helpful was the following: `You've made a great start to deriving the time complexities for` both algorithms. There are some important alterations to make here. Firstly, since O, Omega, and Theta are descriptions of runtime as the input size approaches infinity, we can drop any terms involving k from the equation (since this is an arbitrary value unrelated to the input size). You should also consider the runtime for the 'best case' input for insertion sort. Here, the runtime is O(n) for put insertion sort. You should also draw conclusions about the interpretation of each runtime (eg. what happens to the runtime when we double the input size). Using the feedback from this comment, I was able to improve my complexity analysis by correctly identifying the the time complexity for all processes in the counting bloom filter. I also Improved my analysis when talking about the time complexity when we are using more than one hash function. I coreectly identified how the size of the number of hash functions is arbitrary compared to the size of the input as it approaches infinity.

#cs110-ComputationalCritique: In previous assignments I faild to draw relevant conclusions from my experiments. This time I ensured that I drew relevant conclusions from my experiments. I connected the results of the experiment to show how the fpr of the counting bloom filter changes as the number of hash functions increases to the theoretical conclusions. I accurately identified that the fpr of the counting bloom filter reduces as the number of hash functions increases till it reacheds an optimal number of hash functions after which the fpr increases.

#cs110-CodeReadability: I ensured that my code used consistent naming conventions (snake case in this document). I also ensured that my variable names were easy to follow from my algorithmic explaination. For example, I used the attributw name `stored_item_count` rather than `count` to make it clear that the attribute is storing the number of items stored in the counting bloom filter.

#**cs110-AlgoStratDataStruct:** In the last assignment, the professor encouraged me to provide justifications for specific design choices of an algorithm or data structure. This time I ensured that I provided justifications for the design when creating the counting bloom filter. For example, I made sure to briefly explain why I set the number of counters to be twice the number of items to be stored in the counting bloom filter.

#**Audience:** While completing this assignment, I followed the assignment prompt by ensuring that the report was tailored to abeginner audience. I avoided using techical jargon when explaining key concepts. I ensured to explain new conepts using short easy examples that are familiar to a beginner audience and then segue from those examples into key concepts necessary to understand counting bloom filters as a data structure.

#**Dataviz:** I created easy to follow graphs when explaining the results of my experiments. I ensured that the graph axes were labeled and the graph has a title. I also ensured that I used an appropriate scale to display the data. I also ensured that the graphs were easy to read and understand.

AI Policy

No significant use of AI tools in the generation of this project except Grammerly for grammar and spell check.

References

corte.si. (2023). Corte.si. https://corte.si/posts/code/bloom-filter-rules-of-thumb

Appendix

Appendix A: Counting Bloom Filter

```
[ ]: class CountingBloomFilter():

         def __init__(self, num_items, num_hashfn):
             """
             Initialize a Counting Bloom Filter

             Parameters
             ----------
             num_items: int, number of items to be inserted
             num_hashfn: int, number of hash functions to be used

             Returns
             -------
             None
             """
             self.num_items = num_items
             self.num_hashfn = num_hashfn
```

```
        # multiply the number of items by 10 to get the bucket size to␣
↪reduce the number of collisions
        self.bucket_size = num_items * 10
        self.bucket = [0] * self.bucket_size
        self.stored_item_count = 0

    def string_to_int(self, string, base=10):
            '''
            Converts a string to an integer by summing the product of the␣
↪ASCII value of each character and
            128 to the power of the position of the character in the␣
↪string.

            Inputs:
                string: string
                base: integer
            Output:
                integer
            '''
            #keep track of the length of the string
            string_length = len(string)
            final_int = 0
            # add the product of the ASCII value of each character and␣
↪base to the power of the position of the character in the string
            for position,char in enumerate(string):
                final_int += ord(char) * (base ** (string_length -␣
↪position - 1))
            return final_int

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

        Parameters
        ----------
        item: string, item to be hashed
        i: integer, index of the hash function

        Returns
        -------
        hash_values: list, list of hash values
        """
        # convert the string to an integer
        key = self.string_to_int(item)
        def hash_fn1(key):
            '''
            Hashes a string using the first hash function.

            Inputs:
                key: string
            Output:
                integer
            '''
            return key % self.bucket_size

        def hash_fn2(key):
            '''
            Hashes a string using the second hash function.

            Inputs:
                key: string
            Output:
                integer
            '''
            prime = 31
```

```
            return prime - (key % prime)

        return (hash_fn1(key) + idx * hash_fn2(key)) % self.bucket_size


    def search(self, item):
        """
        Determine if an item is in the counting bloom filter or not

        Parameters
        ----------
            item: string, item to be queried

        Returns
        -------
            boolean: True if item is in the filter, False otherwise
        """
        for i in range(self.num_hashfn):
            index = self.hash_cbf(item, i)
            if self.bucket[index] == 0:
                return None
        return True
    def insert(self, item):
        """
        Insert an item to the filter

        Parameters
        ----------
            item: str, item to be inserted

        Returns
        -------
            None
        """
        # ensure that the filter is not full
        if self.stored_item_count == self.num_items:
            raise Exception("The filter is full")

        # loop through the number of hash functions
        for i in range(self.num_hashfn):
            # get the index of the hash function
            index = self.hash_cbf(item, i)
            # if the index is 0, set it to 1 and if it is not 0, increment it by 1
            if self.bucket[index] == 0:
                self.bucket[index] = 1
            else:
                self.bucket[index] += 1
        # increment the number of stored items by 1
        self.stored_item_count += 1

    def delete(self, item):
        """
        Delete an item from the filter

        Parameters
        ----------
        item: str, item to be deleted

        Returns
        -------
        None
        """
        # ensure that the item exists in the CBF
        item_exists = self.search(item)
        # if the item exists, loop through the number of hash functions and decrement the index of the hash function by 1
        if item_exists:
            for i in range(self.num_hashfn):
                index = self.hash_cbf(item, i)
                self.bucket[index] -= 1
                # if the index is 0, break out of the loop because we cannot decrement it any further
                if self.bucket[index] == 0:
                    # decrement the number of stored items by 1 and break out of the loop
                    self.stored_item_count -= 1
                    break
            # decrement the number of stored items by 1 if we did not break out of the loop
            self.stored_item_count -= 1

    def __str__(self):
        """
        Returns a string representation of the filter
        """
        return str(self.bucket)
```

Appendix B: Testing the Counting Bloom Filter

```
[ ]: from requests import get

     url = 'https://gist.githubusercontent.com/raquelhr/
      ↪78f66877813825dc344efefdc684a5d6/raw/
      ↪361a40e4cd22cb6025e1fb2baca3bf7e166b2ec6/'

     def get_txt_into_list_of_words(url):
         '''Cleans the text data
         Input
         ----------
         url : string
         The URL for the txt file.
         Returns
         -------
         data_just_words_lower_case: list
         List of "cleaned-up" words sorted by the order they appear in the␣
      ↪original file.
         '''
         bad_chars = [';', ',', '.', '?', '!', '_', '[', ']', '(', ')', '*']
```

```
    data = get(url).text
    data = ''.join(c for c in data if c not in bad_chars)
    data_without_newlines = ''.join(c if (c not in ['\n', '\r', '\t'])␣
↪else " " for c in data)
    data_just_words = [word for word in data_without_newlines.split(" ")␣
↪if word != ""]
    data_just_words_lower_case = [word.lower() for word in␣
↪data_just_words]
    return data_just_words_lower_case
```

[ ]:

```
all_words = get_txt_into_list_of_words(url)

# get the unique words from the list of all words
unique_words = set(all_words)

print(f"There are {len(unique_words)} unique words in the text")
```

```
There are 33153 unique words in the text
```

[ ]:

```
def calculate_fpr(input_words, test_words, num_hashfn):
    '''
    Calculates the false positive rate of the counting bloom filter
    Parameters
    ----------
```

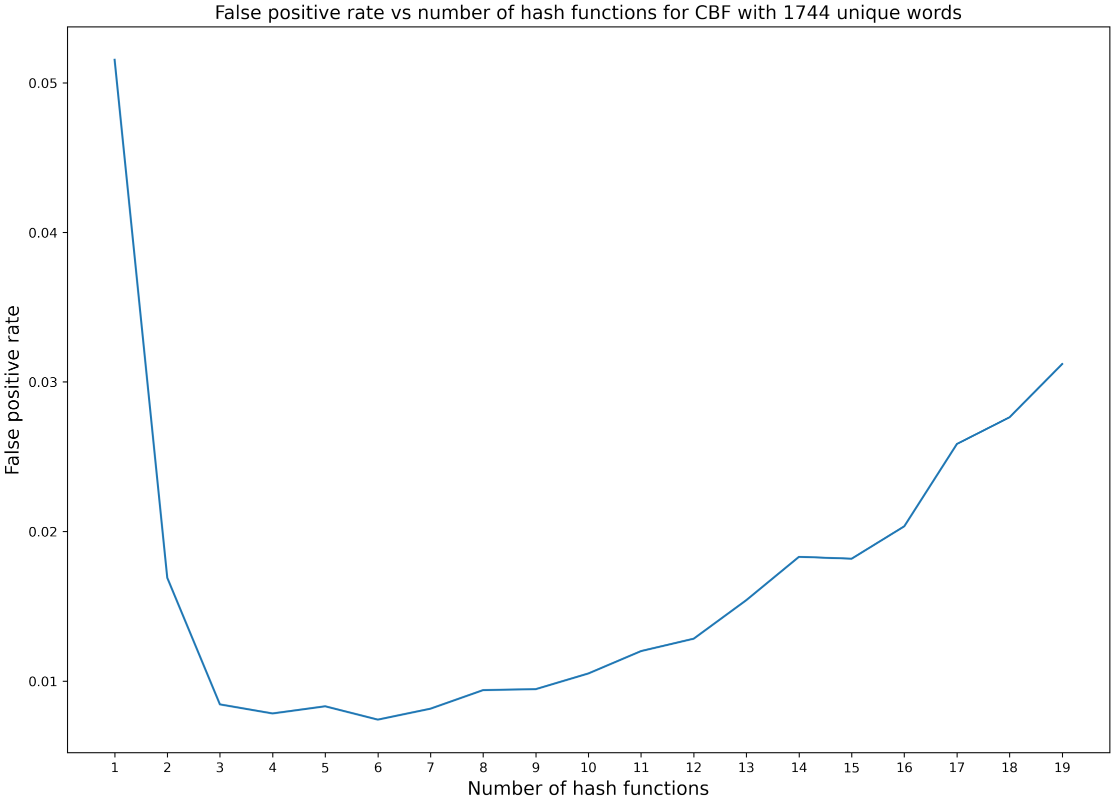
```
        input_words: list, list of words to be inserted into the counting␣
↪bloom filter
        test_words: list, list of words to be tested for existence in the␣
↪counting bloom filter
    Returns
    -------
    fpr: float, the false positive rate of the counting bloom filter
    '''
    # create a counting bloom filter to store the input words
    num_items = len(input_words)
    cbf = CountingBloomFilter(num_items, num_hashfn)
    # insert the input words into the counting bloom filter
    for word in input_words:
        cbf.insert(word)

    # check if all the test words for existence in the counting bloom␣
↪filter
    false_positives = 0
    for word in test_words:
        if cbf.search(word) == True:
            false_positives += 1
    # calculate the false positive rate
    fpr = false_positives / len(test_words)
    return fpr
```

[ ]:

```
import random
import matplotlib.pyplot as plt
num_hash_functions = [x for x in range(1,20)]
fpr_list = []
unique_words_list = list(unique_words)
unique_words_list_len = len(unique_words_list)
hash_fns_count = len(num_hash_functions)

# split the list of unique words into a certian number of parts based on␣
 ↪the number of hash functions
parts_dict = {}

for idx in range(hash_fns_count):
    # get the index of the words to be inderted into the counting bloom␣
 ↪filter
    start_idx = idx * unique_words_list_len // hash_fns_count
    end_idx = (idx + 1) * unique_words_list_len // hash_fns_count

    # get the words to be inserted into the counting bloom filter
    words = unique_words_list[start_idx:end_idx]

    # add the words to the dictionary
    parts_dict[idx] = words

for idx in range(hash_fns_count):
    # get the words to be inserted into the counting bloom filter
    input_words = parts_dict[idx]
    fa_pr = 0
    non_idx = [x for x in range(hash_fns_count) if x != idx]
    tests = random.sample(non_idx, 5)
    for test in tests:
        # get the words to be tested for existence in the counting bloom␣
 ↪filter
        test_words = parts_dict[test]
        # calculate the false positive rate for the counting bloom filter
        fa_pr += calculate_fpr(input_words, test_words,␣
 ↪num_hash_functions[idx])
    # calculate the average false positive rate
    fa_pr = fa_pr / (hash_fns_count - 1)
    fpr_list.append(fa_pr)
```

[ ]:

```
plt.figure(figsize=(14, 10), dpi=300)
plt.plot(num_hash_functions, fpr_list)
plt.xlabel("Number of hash functions", fontsize=14)
plt.ylabel("False positive rate", fontsize=14)
plt.title(f"False positive rate vs number of hash functions for CBF with {len(parts_dict[0])} unique words", fontsize=14)
plt.xticks(num_hash_functions)
plt.show()
```

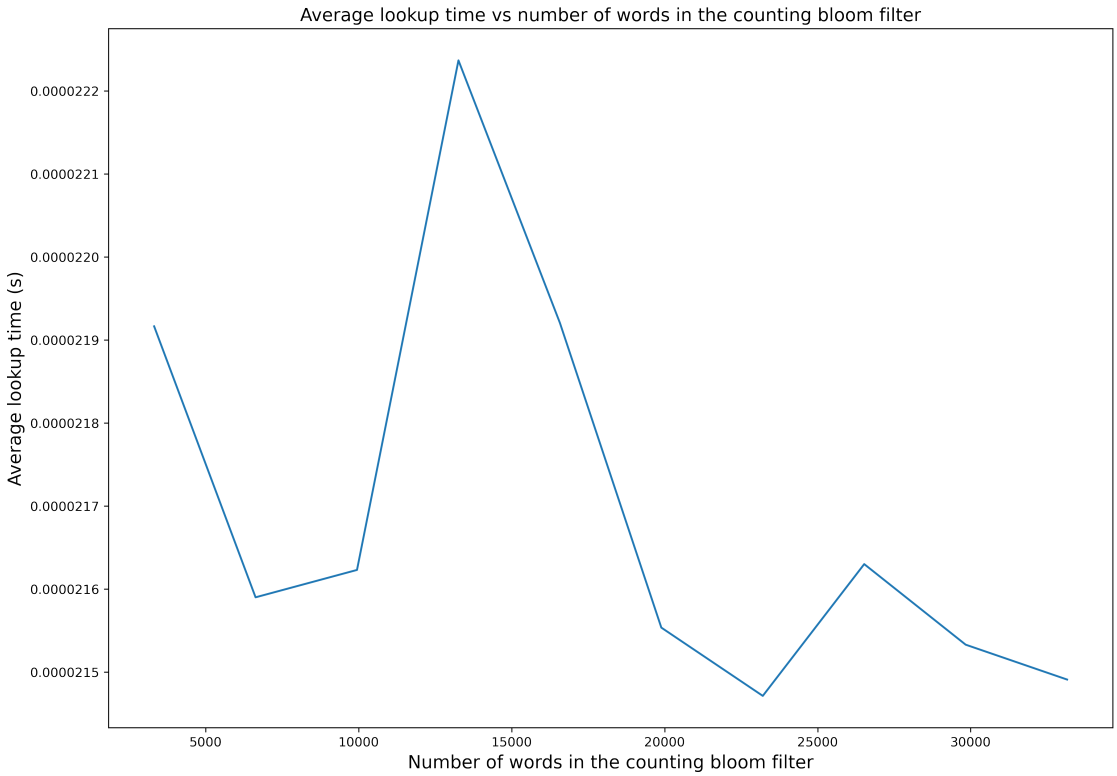


[ ]:

```
import math
import time


fpr = 0.01

# calculate the relevant parameters for the counting bloom filter given␣
 ↪the false positive rate
n = len(unique_words)
fpr = 0.01
m = round(- (n * math.log(fpr)) / (math.log(2) ** 2))
k = round((m / n) * math.log(2))

sections = 10
section_dict = {}
times = []

for idx in range(sections):
    # get the index of the words to be inderted into the counting bloom␣
 ↪filter
    start_idx = 0
    end_idx = (idx + 1) * unique_words_list_len // sections
    # get the words to be inserted into the counting bloom filter
    words = unique_words_list[start_idx:end_idx]

    # add the words to the dictionary
    section_dict[idx] = words

for idx, words in section_dict.items():
    # create a counting bloom filter to store the input words
    num_items = len(words)
    cbf = CountingBloomFilter(num_items, k)
    # insert the input words into the counting bloom filter
    for word in words:
        cbf.insert(word)

    # loop though the words in the section and check if they exist in the␣
 ↪counting bloom filter
    total_time = 0
    for word in words:
        start = time.time()
        cbf.search(word)
        end = time.time()
        total_time += end - start
    # calculate the average time it takes to check if a word exists in␣
 ↪the counting bloom filter
    avg_time = total_time / len(words)
    times.append(avg_time)
```

[ ]:

```
plt.figure(figsize=(14, 10), dpi=300)
plt.ticklabel_format(style='plain', axis='y')
plt.plot([len(section_dict[x]) for x in range(sections)], times)
plt.xlabel("Number of words in the counting bloom filter", fontsize=14)
plt.ylabel("Average lookup time (s)", fontsize=14)
plt.title(f"Average lookup time vs number of words in the counting bloom␣
 ↪filter", fontsize=14)
plt.show()
```

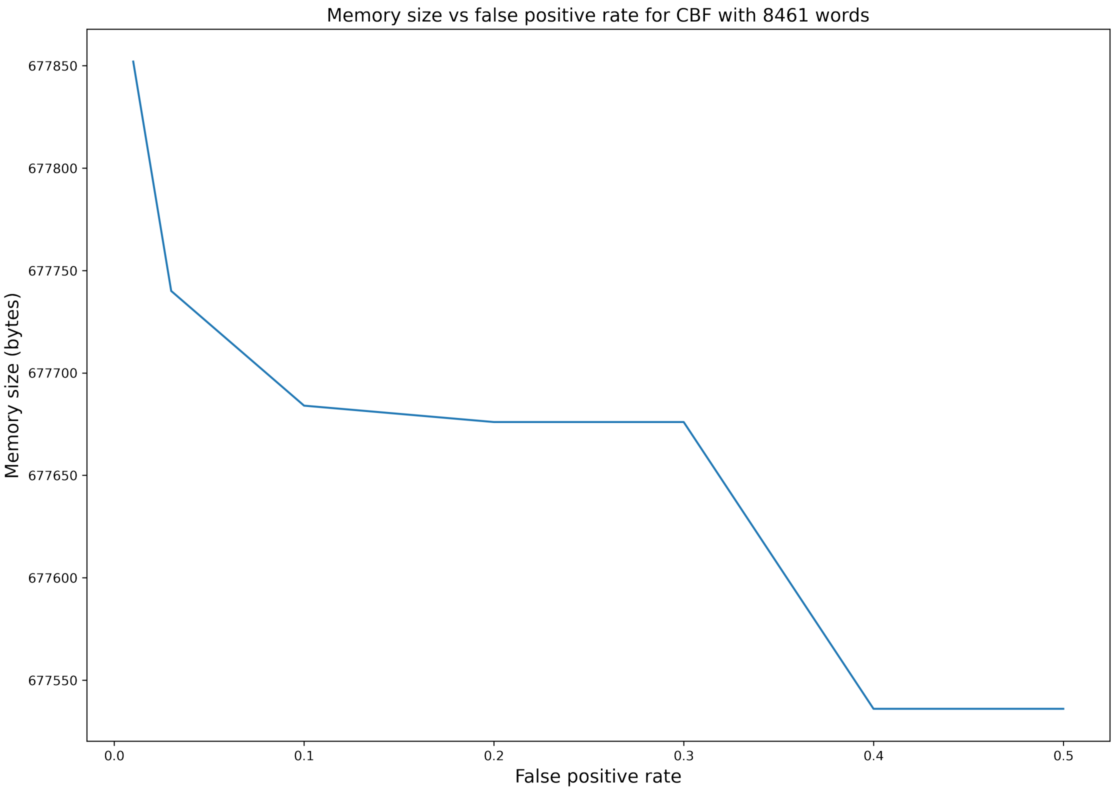


[ ]:

```
import tracemalloc
fprs = [0.01, 0.02, 0.03, 0.1, 0.2, 0.3, 0.4, 0.5]
memory_sizes = []
url_version_1 = "https://bit.ly/39MurYb"
version_1 = get_txt_into_list_of_words(url_version_1)

for fpr in fprs:
    # number of elements to be stored in the counting bloom filter
    n = len(version_1)
    # ideal size of the counting bloom filter
    m = round(- (n * math.log(fpr)) / (math.log(2) ** 2))
    # ideal number of hash functions
    k = round((m / n) * math.log(2))
    tracemalloc.start()
    cbf = CountingBloomFilter(n, k)
    for word in version_1:
        cbf.insert(word)
    current, peak = tracemalloc.get_traced_memory()
    memory_sizes.append(peak)
    tracemalloc.stop()
```

[ ]:

```
plt.figure(figsize=(14, 10), dpi=300)
plt.ticklabel_format(style='plain', axis='y')
plt.plot(fprs, memory_sizes)
plt.xlabel("False positive rate", fontsize=14)
plt.ylabel("Memory size (bytes)", fontsize=14)
plt.title(f"Memory size vs false positive rate for CBF with␣
 ↪{len(version_1)} words", fontsize=14)
plt.show()
```



Appendix C: Plagerism Detection

```
[ ]: def compute_plagiarism_extent(text_words,reference_text_words):

         '''

         Determines how similar a text is to a given reference text.



         Parameters

         ----------

         text_words : list of str

             The words of the text to be checked for plagiarism.
```

```
    reference_text_words : list of str
        The words of the reference text.

    Returns
    -------
    float
        The extent of plagiarism in the text, as a percentage.
    '''
    # create a counting bloom filter to store the words in the reference text
    num_items = len(reference_text_words)
    cbf = CountingBloomFilter(num_items, 7)
    # insert the words in the reference text into the counting bloom filter
    for word in set(reference_text_words):
        cbf.insert(word)

    similar_words = 0
    # loop through the words in the text to be checked for plagiarism
    for word in set(text_words):
        # check if the word exists in the counting bloom filter
        if cbf.search(word):
            similar_words += 1

    # calculate the percentage of words in the text to be checked for␣
 ↪plagiarism that are similar to the reference text
    plagiarism_extent = similar_words / len(text_words) * 100
    return plagiarism_extent


url_version_1 = 'https://bit.ly/39MurYb'
url_version_2 = 'https://bit.ly/3we1QCp'
url_version_3 = 'https://bit.ly/3vUecRn'


version_1 = get_txt_into_list_of_words(url_version_1)
version_2 = get_txt_into_list_of_words(url_version_2)
version_3 = get_txt_into_list_of_words(url_version_3)
```

[ ]:

```
similarity_1_2 = compute_plagiarism_extent(version_1, version_2)
similarity_1_3 = compute_plagiarism_extent(version_1, version_3)
similarity_2_3 = compute_plagiarism_extent(version_2, version_3)


print(f"Similarity between version 1 and version 2: {similarity_1_2:.
 ↪2f}%")
print(f"Similarity between version 1 and version 3: {similarity_1_3:.
 ↪2f}%")
```

```
print(f"Similarity between version 2 and version 3: {similarity_2_3:.
 ↪2f}%")
```

```
Similarity between version 1 and version 2: 26.33%
Similarity between version 1 and version 3: 26.33%
Similarity between version 2 and version 3: 26.33%
```

```
[ ]: def group_words(words, n):
         '''
         Groups words into groups of n words.

         Parameters
         ----------
         words : list of str
             The words to be grouped.
         n : int
             The number of words in each group.

         Returns
         -------
         new_words of list of str
             The list of groups of words.
         '''
```

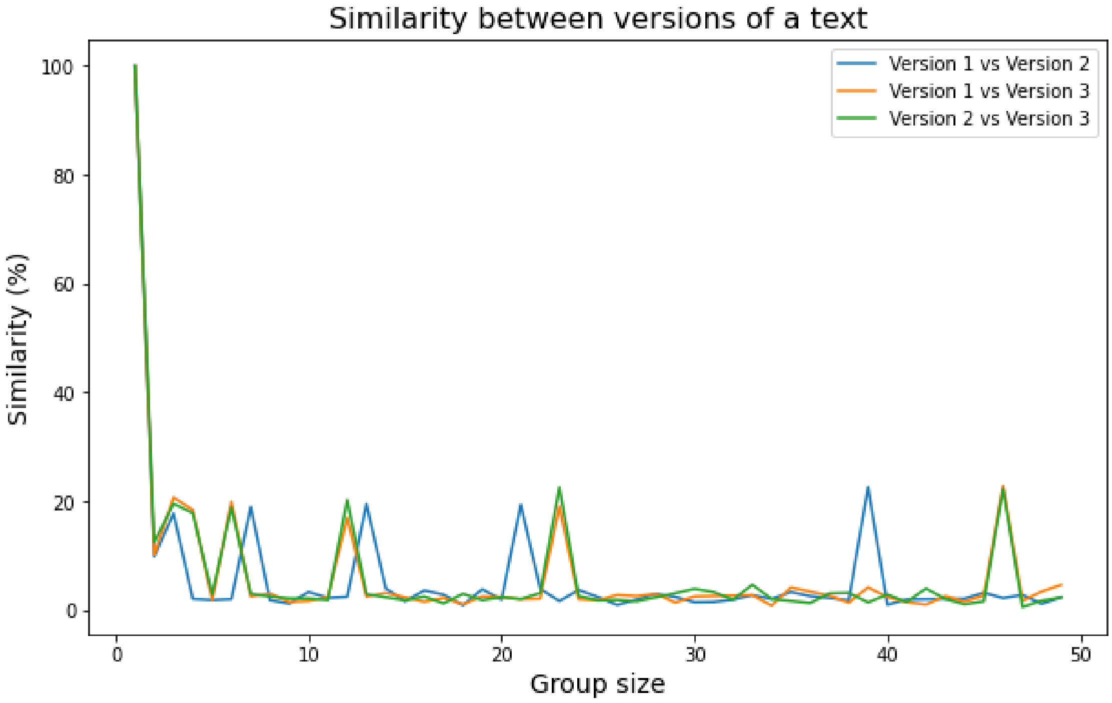
```
    new_words = []
    for idx in range(0, len(words), n):
        group = ' '.join(words[idx:idx + n])
        new_words.append(group)
    return new_words

def compute_plagerism_extent_2(text_words, reference_text_words, n):
    '''
    Determines how similar a text is to a given reference text.

    Parameters
    ----------
    text_words : list of str
        The words of the text to be checked for plagiarism.
    reference_text_words : list of str
        The words of the reference text.

    Returns
    -------
    float
        The extent of plagiarism in the text, as a percentage.
    '''
    # group the words in groups of n words
    text_words = group_words(text_words, n)
    reference_text_words = group_words(reference_text_words, n)

    # create a counting bloom filter to store the words in the reference␣
↪text
    num_items = len(reference_text_words)
    cbf = CountingBloomFilter(num_items, 4)
    # insert the words in the reference text into the counting bloom␣
↪filter
    for word in reference_text_words:
        cbf.insert(word)

    similar_words = 0
    # loop through the words in the text to be checked for plagiarism
    for word in text_words:
        # check if the word exists in the counting bloom filter
        if cbf.search(word):
            similar_words += 1
    # calculate the percentage of words in the text to be checked for␣
↪plagiarism that are similar to the reference text
    plagiarism_extent = similar_words / len(text_words) * 100
    return plagiarism_extent
```

```
[ ]: #group words into groups of 5 words

     similarity_1_2 = compute_plagerism_extent_2(version_1, version_2, 5)
     similarity_1_3 = compute_plagerism_extent_2(version_1, version_3, 5)
     similarity_2_3 = compute_plagerism_extent_2(version_2, version_3, 5)

     print(f"Similarity between version 1 and version 2: {similarity_1_2:.
      ↪2f}%")
     print(f"Similarity between version 1 and version 3: {similarity_1_3:.
      ↪2f}%")
     print(f"Similarity between version 2 and version 3: {similarity_2_3:.
      ↪2f}%")
```

```
Similarity between version 1 and version 2: 1.83%
Similarity between version 1 and version 3: 1.77%
Similarity between version 2 and version 3: 2.84%
```

```
[ ]: group_sizes = [n for n in range(1, 50)]
     similarities = []
     n_trials = 5

     # loop through the different group sizes
```

```
for group_size in group_sizes:
    # compute the similarity between the three versions of the text
    similarity_1_2 = 0
    similarity_1_3 = 0
    similarity_2_3 = 0
    for _ in range(n_trials):
        # for each trial, compute the similarity between the three␣
 ↪versions of the text
        similarity_1_2 += compute_plagerism_extent_2(version_1,␣
 ↪version_2, group_size)
        similarity_1_3 += compute_plagerism_extent_2(version_1,␣
 ↪version_3, group_size)
        similarity_2_3 += compute_plagerism_extent_2(version_2,␣
 ↪version_3, group_size)
    similarities.append([similarity_1_2 / n_trials, similarity_1_3 /␣
 ↪n_trials, similarity_2_3 / n_trials])


plt.figure(figsize=(10, 6))
plt.plot(group_sizes, similarities)
plt.legend(["Version 1 vs Version 2", "Version 1 vs Version 3", "Version␣
 ↪2 vs Version 3"])
plt.xlabel("Group size", fontsize=14)
plt.ylabel("Similarity (%)", fontsize=14)
plt.title("Similarity between versions of a text", fontsize=16)
plt.show()
```



[ ]:
```
def word_for_word_comparison(text_1, reference_text_1):
    '''
    Determines how similar a text is to a given reference text.

    Parameters
    ----------
    text_1 : list of str
        The words of the text to be checked for plagiarism.
    reference_text_1 : list of str
        The words of the reference text.

    Returns
    -------
    float
        The extent of plagiarism in the text, as a percentage.
    '''
    similar_words = 0
    # loop through the words in the text to be checked for plagiarism
    for word in text_1:
        # check if the word exists in the reference text
        if word in reference_text_1:
            similar_words += 1
    # calculate the percentage of words in the text to be checked for plagiarism that are similar to the reference text
    plagiarism_extent = similar_words / len(text_1) * 100
    return plagiarism_extent

similarity_1_2 = word_for_word_comparison(version_1, version_2)
```

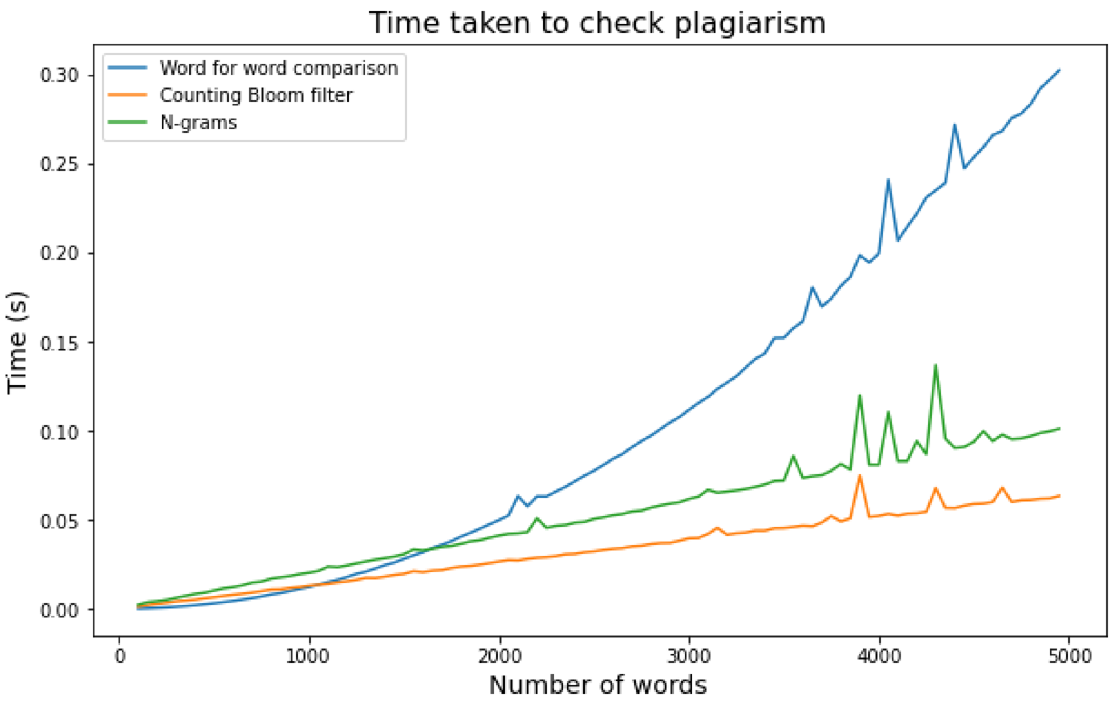
```
similarity_1_3 = word_for_word_comparison(version_1, version_3)
similarity_2_3 = word_for_word_comparison(version_2, version_3)

print(f"Similarity between version 1 and version 2: {similarity_1_2:.
 ↪2f}%")
print(f"Similarity between version 1 and version 3: {similarity_1_3:.
 ↪2f}%")
print(f"Similarity between version 2 and version 3: {similarity_2_3:.
 ↪2f}%")
```

```
Similarity between version 1 and version 2: 100.00%
Similarity between version 1 and version 3: 100.00%
Similarity between version 2 and version 3: 100.00%
```

[ ]:

```
def generate_test_words(n):
    '''
    Generates a list of n random words.

    Parameters
    ----------
    n : int
        The number of words to generate.

    Returns
    -------
    list of str
        The list of random words.
    '''
    words = []
    letters = [chr(i) for i in range(ord('a'), ord('z') + 1)]
    for _ in range(n):
        word = ''
        for _ in range(5):
            word += random.choice(letters)
        words.append(word)
    return words


tests = [x for x in range(100, 5000, 50)]
wfw_times = []
cbf_times = []
n_groups_times = []
trials = 5

# run the tests for each algorithm and time the results
```

```
for n in tests:
    version_1 = generate_test_words(n)
    version_2 = generate_test_words(n)

    total_time = 0
    for _ in range(trials):
        start_time = time.time()
        compute_plagerism_extent_2(version_1, version_2, 5)
        end_time = time.time()
        total_time += end_time - start_time
    cbf_times.append(total_time / 5)

    total_time = 0
    for _ in range(trials):
        start_time = time.time()
        compute_plagiarism_extent(version_1, version_2)
        end_time = time.time()
        total_time += end_time - start_time
    n_groups_times.append(total_time / 5)

    total_time = 0
    for _ in range(trials):
        start_time = time.time()
        word_for_word_comparison(version_1, version_2)
        end_time = time.time()
        total_time += end_time - start_time
    wfw_times.append(total_time / 5)

plt.figure(figsize=(10, 6))
plt.plot(tests, wfw_times, label="Word for word comparison")
plt.plot(tests, cbf_times, label="Counting Bloom filter")
plt.plot(tests, n_groups_times, label="N-grams")
plt.legend()
plt.xlabel("Number of words", fontsize=14)
plt.ylabel("Time (s)", fontsize=14)
plt.title("Time taken to check plagiarism", fontsize=16)
plt.show()
```


Time taken to check plagiarism
Word for word comparison
Counting Bloom filter
N-grams
Time (s)
0.30
0.25
0.20
0.15
0.10
0.05
0.00
0
1000
2000
3000
4000
5000
Number of words