Project Name: Count-Min Sketch

Your Name(s): Ethan Pyke, David Lee

University of Washington: CSE 312 Summer 2020

1 Application

A *Count-Min Sketch* is most useful when trying to determine the approximate frequency of a piece of data without having to store any of the actual data. Thus, a CMS becomes useful when dealing with very large amounts of data. For example, online retailers such as Amazon can utilize the CMS to find frequent items sold with a particular product, in order to provide useful suggestions for a shopping customer. Rather than storing and keeping track of every single item and related frequencies (in a Hash-Table-like structue), the retailer can simply record the approximate number of times two items are sold together in a designated, small space.

A *Count-Min Sketch* is very useful for solving problems like the Heavy Hitter Problem, which may be the most common application of the CMS. Problems like such entail determining the majority element in a given input array. Applying this problem to real life gives us examples like the one previously mentioned, where Amazon wants to determine and display the most popular items on their website. Another example would be determining the most frequent searches on Google by estimating the most frequent search queries for a given day using the CMS. These big companies utilize the CMS because it saves them a great amount of space while still being able to work with huge amounts of data in their efforts to portray useful information to their customers.

2 Core Functionality

The *Count-Min Sketch* algorithm first initializes a table (or two-dimensional arrays/vectors) with dimensions $k \times b$ that represents k unique hash functions that map values into b different buckets. Every value in each bucket starts at 0. For every ith hash function up to the kth, we produce the hash value for each piece of data, x, to indicate which bucket in the ith row of the table to increment by 1, regardless of its previous value. In order to do this, there are three major functions:

- initialize(k, b): Given numbers k, and b, initialize a $k \times b$ array of 0's.
- increment(x): Given an element x, for each hash function $h_0, \ldots, h_k$, increment the resulting bucket at index $h_i(x)$ by 1.

- count(x): Given an element x, for each row i and hash function h, determine the value of $h_i(x)$ and the value being stored at that index. Compare all the values found from iterating through every hash function and return the lowest value.

Algorithm 1 Count-Min Sketch

```
1: function INITIALIZE(k,b)
2:     for $i = 1, \ldots, k$: do
3:         $t_i$ = new vector of $b$ 0's
4: function INCREMENT(x)
5:     for $i = 1, \ldots, k$: do
6:         $[t_i][h_i(x)] += 1$
7: function COUNT(x)
8:     return $\min([t_1][h_1(x)],\ [t_2][h_2(x)],\ \ldots,\ [t_k][h_k(x)])$
```

Since a *Count-Min Sketch* is a probabilistic data structure, there are trade-offs over using a deterministic data structure such as a Hash Table. For example, the CMS only holds the count of each item, which means we are unable to have access to a list of what elements have been added to it. Additionally, since `count(x)` is probabilistic, the value returned will simply be an approximation of the exact count.

3 Explanation and Intuition

A *Count-Min Sketch* is a probabilistic data structure used to determine and store the frequency of a given element in a certain set of elements. Instead of storing every element mapped to its frequency (as one would in a Hash Map), the CMS is able to store a significantly smaller amount of data but still approximate the frequency with accuracy. To do this, we have multiple unique hash functions, and every time an element is inserted into the CMS, each hash function is applied to the element to output a particular bucket index at its row. The value at the unique index for each row is then incremented by 1 to update the number of times the inputted element was seen.

Since each hash function can potentially map different values to the same bucket, we can expect collisions to cause an over-estimation when determining the count of an element. However, it becomes far less likely for all of the hash functions to map two different values to the same bucket at each row. In other words, it's likely that there is at least one hash function that does not map a particular value to the same bucket as another value. We can take advantage of this when searching for a count of an element by determining that the smallest count of the element across all the hash functions is the best estimated count. This also means that our approximated counts will never be less than the actual count.

To determine the number of hash functions and buckets to use, we will use the method described in the original CMS paper. This method describes the use of two variables, δ and ϵ, where the error in determining the frequency of an item is within a factor of ϵ with probability δ. Then, once we know these parameters, we can set k (number of hash functions) $= \lceil ln\frac{1}{\delta} \rceil$ and b (number of buckets) $= \lceil \frac{e}{\epsilon} \rceil$.

4 Advantage(s) over Deterministic Counterpart(s)

Even though the *Count-Min Sketch* can occasionally result in an overestimate of an elements frequency, a CMS is a more efficient way of determining frequencies when taking into account how much space is saved in comparison to other deterministic data structures. We will compare our CMS to a traditional Hash Table. The following is the pseudocode for a traditional hash table:

Algorithm 2 Hash Table

```
function INSERT(key, value)
    index = hash(key)
    array[index] = (key, value)
function FIND(key)
    index = hash(key)
    return array[index]
```

This implementation of a Hash Table contains 2 main functions, `insert(key, value)` and `find(key)`, and holds all of its elements in an array of size n where n is the number of elements in a set. The function `insert`, inserts a key value pair into the array at an index determined by a hash function and `find` returns a value associated with the given key. To compare the Hash Table to a CMS, in order to increase the frequency of item within the Hash Table would involve calling `insert` and `find` and then `find` could be used in the same way that `count` is used in the CMS. If we compare these two data structures, we can see that the CMS will have the following advantages.

4.1 Space Efficiency

A *Count-Min Sketch* will take up considerably less space than an equivalent hash table. Take, for example, a set of hundreds of thousands of elements. The hash table will need to store both the elements themselves and each elements count which would be hundreds of thousands of bytes needed. With n elements, the space complexity of the Hash Table would be $O(n)$. On the other hand, since the the CMS only stores $k \times b$ elements where k is the number of has functions and b is the number of possible values the hash functions can map to, the resulting amount of space the CMS will use is significantly less. So, while the hash table stores its data with a linear space complexity, $O(n)$, a CMS is able to store its data utilizing a sub-linear space complexity.

Another benefit of the CMS is that it is space-constant. Its space complexity will always be $O(kb)$ where k is the number of hash functions and b is the number of buckets. This means that no matter how many elements we are counting, its space complexity will always be the same. On the other hand, a Hash Table will *always* be entirely dependent on the number of unique items within a data set making it hard to keep track of its overall space usage.

5 Implementation in Python

```
import numpy as np
import hashlib

class CountMinSketch(object):
    def __init__(self, num_hash, buckets):
        """
        Initialize a [num_hash x buckets] table with all zeroes.

        :param num_hash: The number of hash functions / rows in the table
        :param buckets: The number of buckets that each hash function will
                        hash to
        """
        self.num_hash = num_hash
        self.buckets = buckets
        self.table = np.zeros([num_hash, buckets])

    def insert(self, key):
        """
        Inserts an element into the table by incrementing its given hash
        values at each row in the table (according to the hash function
        at that row).

        :param key: The element to be inserted.
        """
        for i in range(self.num_hash):
            hash_val = self.hash(key, i)
            self.table[i, hash_val] = self.table[i, hash_val] + 1

    def hash(self, key, i):
        """
        Computes the hash value of 'key' according to the i-th hash function
        in the table.

        :param key: The element to be hashed
        :param i: Indicates which hash function is being used
        :return: Hash value of 'key'
        """
        return int((hash(key) + self.hash2(key) * i)) % self.buckets

    def count(self, key):
        """
        Estimates the number of occurrences of 'key' that we have seen.
        :param key: The element whose count we are estimating
        :return: The estimated total count of 'key'
        """
        min_count = self.table[0, self.hash(key, 0)]
        for i in range(1, self.num_hash):
            min_count = min(min_count, self.table[i, self.hash(key, i)])
        return int(min_count)

    def hash2(self, key):
        """
        Helper function for self.hash()
        """
        return int(hashlib.md5(key.encode('utf-8')).hexdigest()[:8], 16)
```

6 Visual Results and Analysis

6.1 Count-Min Sketch: Counting

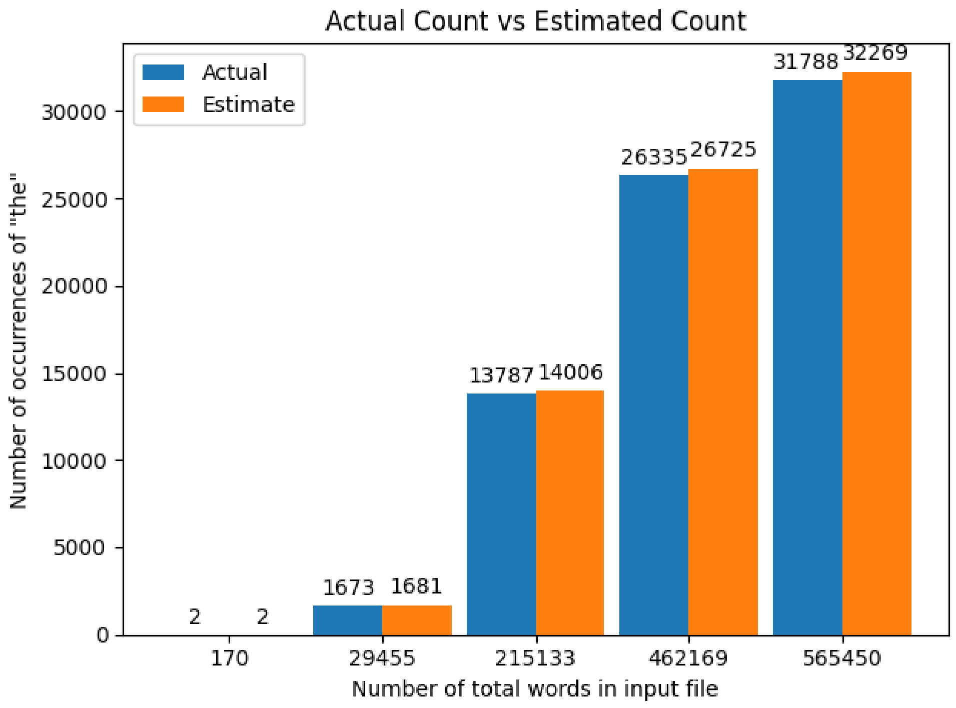


Figure 1: Bar graph showcasing the estimated counts from our CMS algorithm, side by side with the actual counts, of the word "the" in various text files of differing lengths. The dimensions of the CMS used was 5 × 272 (5 hash functions with 272 buckets each).

6.2 CMS vs Hash Table: Time

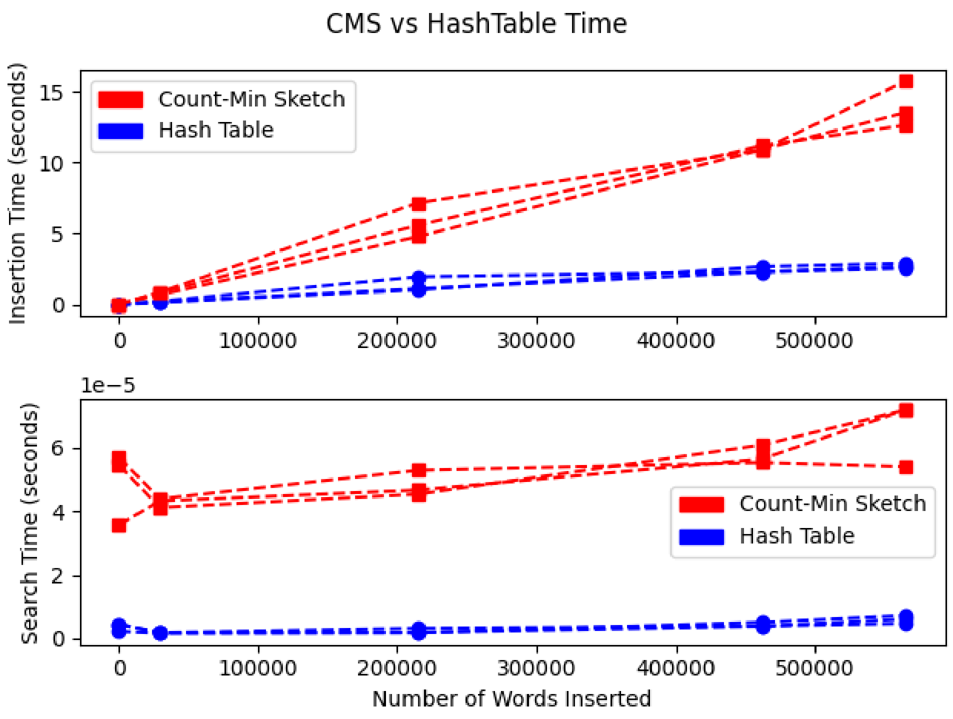
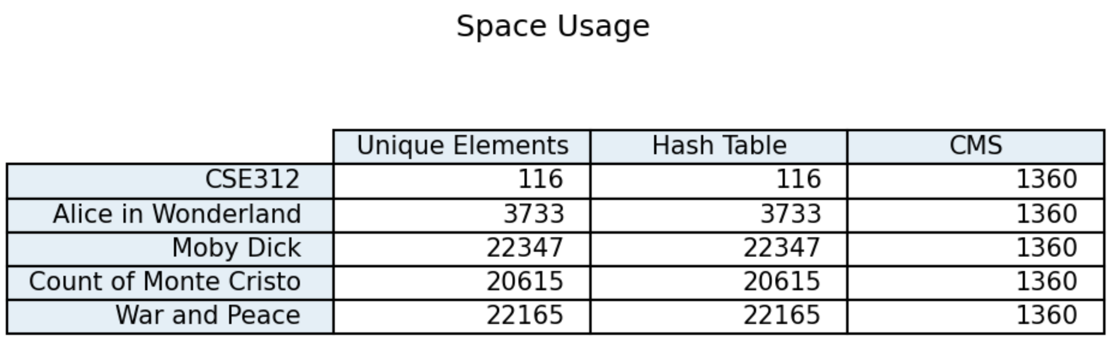


Figure 2: Plot of time comparisons (across three trials) between a Count-Min Sketch and a Hash Table, for inserting/searching in data of various sizes. Note that the ”Search Time” graph is scaled to $x * e^{-5}$, which means the observed performance differences between the CMS and the hash table are negligibly small.

6.3 CMS vs Hash Table: Space Usage



Space Usage

	Unique Elements	Hash Table	CMS
CSE312	116	116	1360
Alice in Wonderland	3733	3733	1360
Moby Dick	22347	22347	1360
Count of Monte Cristo	20615	20615	1360
War and Peace	22165	22165	1360

Figure 3: Comparison of amount of space needed to be used by each data structure where each number represents the number of distinct elements needed to be stored. Unique elements represents the number of unique words within each text file.

7 Probabilistic Analysis

To analyze the effectiveness of the *Count-Min Sketch*, we are going to analyze the CMS `count` function. To do this, we are going to determine the probability that the function's estimated frequency of a given element is within a factor of ϵ. In other words, we will determine $P(A \leq E \leq A + (A * \epsilon))$, where A represents the actual frequency and E represents the returned estimated frequency.

First, we know that given a ("good") hash function, the probability that an input x is hashed to the same bucket as another input y is $\frac{1}{b}$, where b is the total number of buckets belonging to the hash function. To take this one step further, this means that given multiple hash functions $h_1, ..., h_k$, the probability that an input x is hashed to the same buckets as another input y across all of the hash functions is $\frac{1}{b^k}$. However, this probability only applies between two inputs, and as we increase the number of distinct elements to input, this probability grows larger. Thus, we must find a way to choose appropriate numbers of hash functions k as well as their corresponding buckets b.

By the definition of a CMS, we know that $k = \lceil ln\frac{1}{\delta} \rceil$ and $b = \lceil \frac{e}{\epsilon} \rceil$, where ϵ is a factor of error and $1 - \delta$ is the probability that our estimate is within that factor. In other words, a CMS says that our estimate will always be greater than or equal to the actual count and less than or equal to $Actual + (Actual * \epsilon)$ with a probability of $1 - \delta$. Therefore, we are going to be proving the following

$$P(Actual \leq Estimate \leq Actual + Actual * \epsilon) = 1 - \delta$$
$$P(Estimate > Actual + Actual * \epsilon) = \delta$$

To start, we introduce an indicator variable $I_{x,y,z}$ which is 1 if $(x \neq z)$ and $(h_y(x) = h_y(z))$ (when two values are mapped to the same bucket), and 0 otherwise. Then, we can define a random variable $X_{x,y}$ to be $X_{x,y} = \sum_{z=1}^{n} I_{x,y,z} a_z$ where x is the hash function, y is the bucket, z represents each unique element, and a_z is the actual frequency for element z. By construction, when we call $count(y, h_y(x))$, it will be equal to $a_x + X_{x,y}$, thus showing that min $count(y, h_y(x)) \geq a_x$. Additionally, we can determine the following by pairwise independence of h_y:

$$E[X_{x,y}] = P(h_y(x) = h_y(x)) = \frac{\epsilon}{e}$$

Then, by using the Markov inequality, we can determine the following:

$$\begin{aligned} P(Estimate > Actual + Actual * \epsilon) &= P(count(y, h_y(x)) > Actual + Actual * \epsilon) \\ &= P(a_x + X_{x,y} > a_x + a_x * \epsilon) \\ &= P(X_{x,y} > eE[X_{x,y}]) \leq \delta \end{aligned}$$

Since we have successfully proven that our estimate will be outside the range of $[Actual, Actual + Actual * \epsilon]$ with a probability of δ, we know that this means it will be within that range within a probability of $1 - \delta$. Because of this, we have show how probability makes the CMS work.

References

[1] https://coderbook.com/@marcus/how-to-create-a-hash-table-from-scratch-in-python/

[2] https://matplotlib.org/api/_as_gen/matplotlib.pyplot.html

[3] https://web.stanford.edu/class/cs168/l/l2.pdf

[4] http://www.eecs.harvard.edu/ michaelm/CS222/countmin.pdf

[5] https://florian.github.io/count-min-sketch/