COMP150 Summer 2018, Final Project

Zachary Goldman: Streaming Algorithms, Sampling and Heavy Hitters

1 Intro: What is a Stream?

Streams are sequences of data which exceed the amount of system memory that is available for a given operation, and may even be of indeterminate size. Streaming algorithms are only allowed a single, or very few, passes on each element of the stream due not only to the space requirements, but to the time requirements, as streams are often based off of real-time data. Streams are used in a number of contexts - video streaming, trending topics, stock volatility analysis, and many other data analysis situations. Streams are faced with a number of problems, and very often the answer is to find a close approximation rather than a certain answer, due to the memory constraints. The distinct elements problem, for example, deals with finding the cardinality of the set of elements in a stream. An exact answer cannot be found due to space constraints, but using a user-defined accuracy parameter, ϵ, an algorithm can, with high probability, find a number that falls between $(1-\epsilon)*f(x)$ and $(1+\epsilon)*f(x)$, where $f(x)$ is the cardinality of the set. Such an algorithm can be done with $O(1/\epsilon^2 + logn)$ space, which is a huge improvement over the $O(n)$ space required to achieve certainty depending on the size of ϵ. However, this paper will primarily deal with two different methods of reducing a stream to fit memory - based on a true-random distribution, and on frequency analysis.

2 Streams: Sampling Algorithms

In this situation, to paint an accurate picture of the stream, a sampling algorithm must be used. Having less memory than the stream contains, as the streams are often massive, to paint an overall representative picture there must be an algorithm to sample the data in one pass (since the stream is ongoing) with a fraction of the stream's space (constant space, rather than linear). Since all the data cannot be stored in memory, and the stream length is often unknown, a probabilistic method of randomly sampling the data must be put in place, such that each element in the stream has an equal chance of making it to memory so no part of the stream has priority. There are many such algorithms, all with the goal of creating an even distribution of data that's representative of the complete stream. This is effectively a form of lossy compression - preserving the best representation of the total stream that can be done, while decreasing the amount of space it takes. This allows for computation over the stream's data to have the highest probability of being correct. If every element has the same chance of making it into memory, then the sampling algorithm is order independent - shaking up the order of the stream wouldn't affect the odds an element is chosen.

2.1 Sampling Example: Reservoir Sampling

One relatively simple sampling algorithm is reservoir sampling, which works on a data stream of any size. Reservoir sampling is best illustrated through the example of memory of size 1 and a stream of unlimited length. Reservoir streaming is set up in such a way that each element in the stream has the same probability of being the element in memory at the end of the stream, regardless of the stream length.

For the nth element in a stream of size N, the probability that it ends up as the element in memory

is $1/N$.
The way this algorithm works is for each element in the stream, generate a random number between 0 and 1. If the number generated is less than $1/n$, where n is the index of the element in the stream, then that element of the stream replaces the existing element in memory.
When working with memory of size k, each index, 0 through $k-1$, is populated with the first k elements of the stream. Then, go through each remaining element in the stream, index k through N. For each element in the stream, indexed n, generate a random number between 0 and 1, and if the number is less than $1/(n-k)$, replace a random element in memory (probability of being chosen is $1/k$ with that element. Each element in the stream has an equal k/N chance of being in memory after the entire stream has been run through.

2.2 Reservoir Sampling: Formal Definition

Let $S = [x_1, x_2, ..., x_n]$ be a stream of N items. Let k be memory where $k < N$. Since $N > k$ we need to obtain a sample, M_i, of k items where $\forall 1 \leq i \leq j \leq N$, $P(x_i \in M_i) = P(x_j \in M_i)$ after i iterations.
Algorithm:
If $i \leq k : M_i = M_{i-1} \cup \{x_i\}$
else: With probability $\frac{k}{i}$ replace a random element in M_{i-1} with x_i.

2.3 Reservoir Sampling: Proof by Induction

To prove that each element has an equal probability of selection, induction can be used.[1]
The base case for our induction proof is this:
$i = k : P(x_i \in M_i) = \frac{k}{i}$
As i is incremented to $i+1$:
Our first i elements have been chosen with the probability $\frac{k}{i}$.
The element at index $i+1$ will be chosen with the probability $\frac{k}{i+1}$.
If this element is chosen, each element in M_i has a $\frac{1}{k}$ probability of being replaced.
Therefore, the probability that an element in memory is replaced with the element in the stream indexed $i+1$ is $\frac{1}{k} * \frac{k}{i+1} = \frac{1}{i+1}$.
By complementary probability, we know that the probability at a given step that an element is not replaced is $\frac{i}{i+1}$.
Every element in M_i is there because it was chosen and not replaced:
$P(x \in M_i + 1) = \frac{k}{i} * \frac{i}{i+1} = \frac{k}{i+1}$
This continues all the way until the final time stamp where $i+1 = N$, so every element in M is there with the probability $\frac{k}{N}$

3 Streaming Algorithms: Heavy Hitters Problem

One common problem in streaming algorithms is how to determine the data that appears with the highest frequency. In the case where the magnitude of the domain of the stream - if measuring search terms, for example, the domain is every possible combination of characters - exceeds the amount of space in memory, a simple counter for each possible piece of data is not possible. An exact solution to this problem using sublinear space does not exist, there are only approximate solutions.

3.1 Heavy Hitters Example: Epsilon-Approximation

One approximation for the Heavy Hitters problem is ϵ-Approximate Heavy Hitters (ϵ-HH).[2] The goal of ϵ-HH is to ensure that, when working with memory of size k, every element in the stream S with n items that occurs at least $\dfrac{n}{k}$ times is in memory, and that every value in memory occurs at least $\dfrac{n}{k} - \epsilon n$ times in the stream, to guarantee a cluster of heavy hitters in memory, where epsilon is an accuracy parameter defined by the user.

3.2 Data Structures for HH: Count-Min Sketch

A data structure known as a Count-Min Sketch, inspired by a data structure known as the Bloom Filter, helps significantly in solving ϵ-HH. Bloom filters are probabilistic data structures based on hashing. Probabilistic, here, means that occasionally there will be a false positive - which is a trade-off that we're willing to make, since it uses a sub-linear space and has $O(1)$ insert and lookup. A Bloom Filter is a bit vector, where every element that is added to the Bloom Filter is hashed by multiple hash functions, and the result of each has flips the bit of the appropriate index to 1. There is no collision resolution if multiple elements hash to the same index, but since there's multiple hash functions, membership can be determined to a reasonable probability of each index that an element hashes to is filled. By contrast, a Count-Min Sketch is a 2d array that can track frequency. A Count-Min Sketch has two parameters - the number of hash functions it uses, h, and the amount of slots in the table, s. Fittingly, it also only supports two functions - $Inc(x)$, which increments the count at the spaces that x is hashed to, and $Count(x)$, which returns the sum of all the increments of x.
The reason a Count-Min Sketch, like a Bloom Filter, uses multiple hash functions is because each element, x, gets hashed by each function whenever $Inc(x)$ is called. The data structure itself is a 2D array of size h by s, where each hash on x corresponds to an index based on which hash function it used in h, and where it hashed to in s. Each hash function will increment the index in the array that it hashes to, to create a frequency count for each variable, x.
Since the hash functions should run in constant time, the time complexity for each call of $Inc(x)$ should also be constant, running in $O(h)$ time. However, there is no collision resolution built in to a Count-Min Sketch, so if an element y collides with x in the table at least once, it still increments the value. So, when $Count(x)$ is called, it returns the minimum amount of increments that it hashes to at each h column, since the error in the Count-Min Sketch can only be overestimated. Since the lookup in $Count(x)$ takes only as long as the hash functions, and the get-min associated with $Count(x)$, it also only takes constant $O(h)$ time.

3.3 Epsilon: Putting it all Together

Maintaining a frequency counter for elements in a stream that operates in constant time for each element allows us to track the frequency of elements in sub-linear time. The solution is trivial if the size of the stream, n, is already known - setting $\epsilon = \frac{1}{2k}$, put each element into the count-min sketch, and whenever the count-min sketch has an estimated frequency of $\geq \frac{n}{k}$, it's added to the array. However, when n is not known it becomes a more complex problem.
When n is not known at the start of the problem, the same value of ϵ is used. However, we will not know when an element has the desired frequency as we would when n is known. To handle this, a counter value, m, is maintained that keeps track of the amount of array entries processed. Although a heap may be used to store the potential heavy hitters, a Fibonacci heap has faster $O(1)$ amortized insert time, and $O(logn)$ delete time. Whenever an element, x, from the stream is processed, if the $Count(x) \geq \frac{m}{k}$, it is inserted into the heap using the count value as its key; if the element already exists in the heap, it's first deleted - altogether this takes up to $O(logn)$ for each element. Additionally, if m gets big enough that elements of the heap are no longer frequent enough to be counted as a potential heavy-hitter, it's removed from the heap. This is done using find-min and extract-min, which are constant and logarithmic-time operations in a Fibonacci heap, respectively. When the stream is done being processed, the heap will consist mostly of the heavy hitters - there cannot be more than k elements in the heap, as each element occurs at least $\frac{n}{k} - \epsilon n$ times. However, a count-min sketch is only an approximation, and prone to over-estimating data values, so it's important that the value of k is set to be roughly one half of the actual available space in memory, to ensure that there is no overload, as our given value of ϵ allows for a max of double the possible elements. Although the count-min sketch minimizes errors by taking the minimum count of the five available hash functions, it's possible that there's sufficient overlap to push extra elements that should not be in the heap into the heap.

3.4 Wrap-Up

One of the positives of a heavy-hitters algorithm is that it also serves as a sampling algorithm. ϵ-HH is an order independent algorithm, just like reservoir sampling. However, rather than an even distribution of elements, ϵ-HH returns the most common elements in the stream. Each has its own distinct advantages - heavy hitters is better for streams that contain a lot of 'junk' data, and the analysis is focused on trends. Reservoir sampling, by contrast, places equal value on each element, and doesn't put as much importance on trends, just a representative picture of the data.

4 Sources Consulted:

[1]https://www.youtube.com/watch?v=dWfo9XGrqKU
[2]http://theory.stanford.edu/ tim/s17/l/l2.pdf
https://www.cs.cmu.edu/ ckaestne/pdf/icse16.pdf
http://minimallysufficient.github.io/2015/08/01/reservoir-sampling.html
https://www.wired.com/story/big-data-streaming/
https://mapr.com/blog/some-important-streaming-algorithms-you-should-know-about/
https://www.geeksforgeeks.org/reservoir-sampling/
https://blog.medium.com/what-are-bloom-filters-1ec2a50c68ff
https://ieeexplore.ieee.org/document/4016143/
http://www.cs.cmu.edu/afs/cs/user/dwoodruf/www/eatcs.pdf