1 Min Sketches

1.1 Background

So far our goal has been making estimates on streams that look like \( L = [x_1, x_2, \ldots x_t] \)
Where \( t \) consists of a value and an increment to that value \( x_t = (i, \Delta t) \)

What we’ve shown can be helpful in understanding the makeup of the stream is to use a ‘sketch’
which is a smaller vector that we can access to figure out what the value of a component in \( L \) is.

1.2 Counting Unique Elements in a Stream

Q: What if we want to measure how many unique items are in \( L \)?

We can immediately think of 3 ideas from previous topics

- Use a dictionary → Would take up far too much space
- Use a bloom filter → Would work but we think we can do better
- Use reservoir sampling → Could work but you’d need a reasonably large buffer size

Instead, a more elegant solution involves looking at probabilities (LIKE ALWAYS)

Min sketch involves hashing every item in the stream onto a continuous range from 0 to 1,
\( h(i) \rightarrow [0, 1] \), but only storing the item with the minimum valued hash

Our expectation that the next value, ie the \((n + 1)\)th item, is the minimum hashed value
is equal to: \( E(\hat{n}) = 1/(n + 1) \)

Therefore \((1/\hat{n}) + 1 \approx n\)

1.3 Fajold-Martin

The Fajold-Martin Algorithm is similar to min-sketch except we hash each unique element in
the stream an additional time with a function \( g(i) \sim \{1, -1\} \) so that we only keep the minimum
of \( g(h(i)) \) for all values in the stream \( i \). If we see \( g \) give a 1 we ignore the value, if we see \( g \) give
a -1 then we check it.

The trick here is that we only need range \( m/2 \) to find the number of unique elements. If
we add another function on top that performs the same as \( g \), then we only need a range of
\( m/4 \). We can apply the concept recursively to create count unique elements with \( \log(m) \) of the
range.

Q: Can we make this \( O(\log(\log(m))) \) ?
What if instead of storing the lowest hash, we store the least significant bit of every hash?

**Algorithm:**

Let \( \min(k \geq 0 | \text{bit}(y,k) \neq 0) \)

Init bitmap of length \( L \)

for each \( i \):
  calculate \( k = (\text{hash}(i)) \)
  bitmap\([k]\) = 1

Let \( R \) = smallest index \( k \) st. bitmap\([k]\) = 0

return \( 2^R / \Phi \), \( \Phi \approx 0.77 \)

Since the smallest element of the bit map will be accessed \( n/2 \) times, we can see that as you progress the bit map the probability goes down exponentially such that we actually only work with \( O(\log(\log(m))) \) range.

### 1.4 Summation of \( c_i^2 \) Within a Stream

Q: What if we want to know the summation of \( c_i^2 \) within a stream?

We can break down this by first considering it as a step in a more complex version of the problem. If instead we look to solve \( \sum c_i^2 \), we can start by solving \( r = 1 \). This is simple, every time that we see a new item we simply add one to a count.

For count sketch we know that the variance is \( O(\sum c_i^2 / n) \) when \( E(\hat{c}_i) = c_i \)

But instead, if we simply use one counter and one bin, we can calculate the squared count in \( O(\sum^2) \)

### 1.5 Finding Frequent Items

Q: How can we find the \( R \) most frequently occurring items in the stream?

- Every time that you see an item add it to the sketch of size \( R \), but if the item is already in the sketch increase the count by 1.
- Once the sketch is full check to see if the next item is in the sketch, if it’s not subtract all the counts in the sketch by one, otherwise add 1 to its count.
- If any item’s count goes down to zero remove it from it from the list.
- If there is an empty spot in the list add the next item into the list.

This method runs in \( O(1/\epsilon) \) where \( \epsilon \) is the number of times that a number is decremented to zero and swapped.

\[ c_i - \epsilon \sum \leq \hat{c}_i \leq c_i \rightarrow O(1/\epsilon) \]

\[ c_i \leq \hat{c}_i \leq c_i + \epsilon \sum \rightarrow O((1/\epsilon) \log(1/\delta)) \]

**References**