CSCI 1052 Problem Set 1

January 10, 2024

Submission Instructions

Please upload your solutions by **5pm Friday January 12, 2023.**

- You are encouraged to discuss ideas and work with your classmates. However, you **must acknowledge** your collaborators at the top of each solution on which you collaborated with others and you **must write** your solutions independently.
- Your solutions to theory questions must be typeset in LaTeX or markdown. I strongly recommend uploading the source LaTeX (found here) to Overleaf for editing.
- I recommend that you write your solutions to coding question in a Jupyter notebook using Google Colab.
- You should submit your solutions as a **single PDF** via the assignment on Canvas.

Problem 1 (from January 4)

In class, we calculated the number of duplicates in a sample of size m as

$$D = \sum_{i=1}^{m} \sum_{j=i+1}^{m} D_{i,j}$$

where $D_{i,j}$ is an indicator random variable that the ith and jth sampled items are the same. In expectation when the samples are drawn uniformly at random, we saw that

$$\mathbb{E}[D] = \frac{m(m-1)}{2n}.$$

Part 1

In practice, we know m the number of samples we've taken and D the number of duplicates in the sample. Using these quantities, suggest a method for estimating n the set size inspired by our expression for $\mathbb{E}[D]$.

Part 2

Implement your method from Part 1 to estimate the number of unique articles in the English Wikipedia. You can access "random" articles (see the discussion here) by visiting the link: `https://en.wikipedia.org/wiki/Special:Random`.

According to the article here, English Wikipedia has 6.7 million articles. How does that compare to your estimate? How does the fact that the random article feature doesn't perfectly return random articles bias your estimate?

In Python, you can get a random URL by running the following code:

```
import requests
response = requests.get("https://en.wikipedia.org/wiki/Special:Random")
random_url = response.url
```

In my experiments, it took 36 minutes to get 5000 random articles. At this rate, it would take 33.5 days to get 6.7 million articles.

In your solution, include all relevant results and calculations in addition to a discussion of how accurate you think your estimate is.

Problem 2 (from January 8)

We showed that Count-Min can estimate the frequency of any item in a stream of n items up to additive error $\frac{1}{m}n$ using $O(m)$ space. In practice it is often observed that this bound is pessimistic: the algorithm performs better than expected. In this problem, you will establish one reason why.

For any positive integer m, let $f_1, \ldots, f_m$ be the frequencies of the m most frequent items in our stream. Let $C = n - \sum_{i=1}^{m} f_i$. In general, we can have that $C \ll n$. For example, it has been observed that up to 95% of YouTube video views come from just 1% of videos. Prove that using $O(m)$ space, Count-Min actually returns an estimate $\tilde{f}(v)$ to $f(v)$ for any item v satisfying:

$$f(v) \leq \tilde{f}(v) \leq f(v) + \frac{2}{m}C \tag{1}$$

with 9/10 probability. This is strictly better than the $\frac{1}{m}n$ error bound shown in class.

Part 1

Explain why it suffices to show Equation 1 holds with any constant probability $c > 0$.

Then write the estimate $\tilde{f}(v) = A[h(v)]$ as the true frequency $f(v)$ and two error terms; the first error term is for the m most frequent items that could collide with v and the second error term is for the remaining items that could collide with v.

Part 2

Prove that the first error term is 0 with constant probability; that is, there is a constant probability that none of the m most frequent items collide with v.

Hint: Use one of these inequalities:

$$\frac{1}{e} \geq \left(1 - \frac{1}{m}\right)^m \geq \frac{1}{2e}.$$

You can check these inequalities for yourself on Desmos.

Following the analysis in class, prove that the second error term is at most $\frac{2}{m}C$ with constant probability.

Show how both results together imply that Equation 1 holds with a constant probability.

Problem 3 (from January 9)

One of the most important factors in controlling diseases like COVID-19 is testing. Before at-home kits became available, testing was expensive and slow. One way to make it cheaper was to test patients in *groups*. The biological samples from multiple patients (e.g., multiple nose swabs) are combined into a single test tube and tested for COVID-19 all at once. If the test comes back negative, we know everyone in the group is negative. If the test comes back positive, we do not know which patients in the group actually had COVID-19, so further testing would be necessary. There's a trade-off here, but it turns out that, overall, group testing can save on the total number of tests run.

Part 1

Consider the following deterministic "two-level" testing scheme. We divide a population of n individuals to be tested into C groups of the same size. We then test each of these groups. For any group that comes back positive, we retest all members of the group individually. Show that there is a choice for C such that, if k individuals in the population have COVID-19, we can find all of those individuals with $\leq 2\sqrt{nk}$ tests. You can assume k is known in advance (often it can be estimated accurately from the positive rate of prior tests). This is already an improvement on the naive n tests when $k < 25\% \cdot n$.

Part 2

We can use randomness to do better. Consider the following scheme: Collect $q = \log_2(10n)$ nose swabs from each individual (I know... not pleasant). Then, repeat the following process q times: randomly partition our set of n individuals into $C = 2k$ groups, and test each group in aggregate. Once this process is complete, report that an individual "has COVID" if the group they were part of tested positive all q times. Report that an individual "is clear" if *any* of the groups they were part of tested negative. Show that, with probability 9/10, this scheme finds all truly positive patients and reports no false positives. Thus, we only require $2k * \log_2(10n) = O(k \log n)$ tests!

Problem 4 (from January 10)

In modern systems, hashing is often used to distribute data items or computational tasks to a collection of servers. What happens when a server is added or removed from a system? Most hash functions, including those discussed in class, are tailored to the number of servers, n, and would change completely if n changes. This would require rehashing and moving all of our m data items, an expensive operation.

Here we consider an approach to avoid this problem. Assume we have access to a completely random hash function that maps any value x to a real value $h(x) \in [0, 1]$. Use the hash function to map *both* data items and servers randomly to $[0, 1]$. Each data item is stored on the first server to its right on the number line (with wrap around – i.e. a job hashed below 1 but above all serves is assigned to the first server after 0). When a new server is added to the system, we hash it to $[0, 1]$ and move data items accordingly.

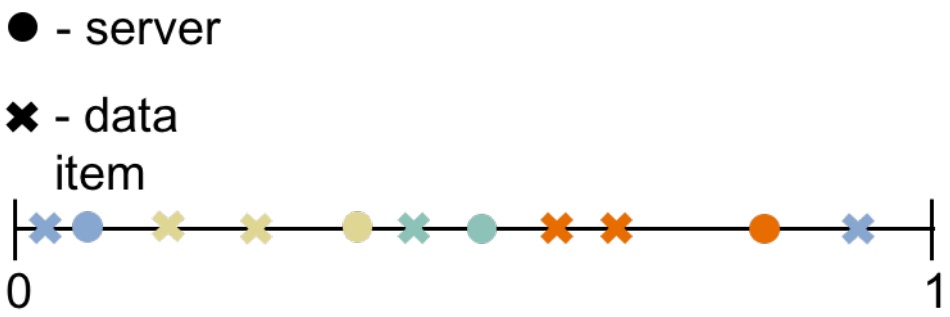

Figure 1: Each data item is stored on the server with matching color.

Part 1

Suppose we have n servers initially. When a new server is added to the system, what is the expected number of data items that need to be relocated?

Part 2

Show that, with probability $> 9/10$, no server "owns" more than an

$$\frac{\log_e(10n)}{n}$$

fraction of the interval $[0, 1]$. **Hint:** This can be proven without a concentration bound.