CS166 Problem Set 5: Randomized Data Structures

Luis A. Perez

Problem One: Final Details on Count Sketches (3 Points)

In our analysis of count sketches from lecture, we made the following simplification when determining the variance of our estimate:

$$\text{Var}\left[\sum_{j\neq i}\mathbf{a}_j s(x_i)s(x_j)X_j\right] = \sum_{j\neq i}\text{Var}\left[\mathbf{a}_j s(x_i)s(x_j)X_j\right]$$

In general, the variance of a sum of random variables is not the same as the sum of their variances. That only works in the case where all those random variables are ***pairwise uncorrelated***, as you saw on Problem Set Zero.

Prove that any two terms in the above summation are uncorrelated under the assumption that both s and h are drawn uniformly and independently from separate 2-independent families of hash functions.

As a refresher, two random variables X and Y are uncorrelated if $\text{E}[XY] = \text{E}[X]\,\text{E}[Y]$.

Solution: We begin by defining the j-th term in the sum as:

$$T_j = \mathbf{a}_j s(x_i)s(x_j)X_j$$

What we wish to show is that $T_j \perp T_k$ for $j \neq q$ (distinct terms are independent). We can do this rather directly by showing that $\text{E}[T_jT_k] = \text{E}[T_j]\,\text{E}[T_j]$. We know from lecture that $\text{E}[T_j] = 0$ (however, for completeness, we present the proof below):

$$\begin{aligned}
\text{E}[T_j] &= \text{E}[\mathbf{a}_j s(x_i)s(x_j)X_j] && \text{(Definition of } T_j\text{)}\\
&= \mathbf{a}_j\text{E}[s(x_i)s(x_j)X_j] && (\mathbf{a}_j \text{ is not a random variable})\\
&= \mathbf{a}_j\text{E}[s(x_i)s(x_j)]\,\text{E}[X_j] && \text{(indendence of } h \text{ and } s\text{)}\\
\mathbf{a}_j\text{E}[s(x_i)]\,\text{E}[s(x_j)]\,\text{E}[X_j] && (s \text{ is pairwise independent, and } i \neq j)\\
&= 0\cdot 0\cdot \mathbf{a}_j\text{E}[X_j] = 0 && (\text{E}[s(x_i)] = 0 \text{ since it is uniform over } \{-1,1\})
\end{aligned}$$

From the above, we can conclude that $\text{E}[T_j]\,\text{E}[T_k] = 0$. As such, all we need to show is that $\text{E}[T_jT_k] = 0$. We do that below:

$$
\begin{aligned}
\mathrm{E}\left[T_j T_k\right] &= \mathrm{E}\left[\mathbf{a}_j \mathbf{a}_k s(x_i)^2 s(x_j) s_(x_k) X_j X_k\right] && \text{(Definition)} \\
&= \mathbf{a}_j \mathbf{a}_k \mathrm{E}\left[s(x_i)^2 s(x_j) s_(x_k) X_j X_k\right] && (\mathbf{a}_j, \mathbf{a}_k \text{ are not random variables}) \\
&= \mathbf{a}_j \mathbf{a}_k \mathrm{E}\left[s(x_i)^2 s(x_j) s_(x_k)\right] \mathrm{E}\left[X_j X_k\right] && \text{(independence of } h \text{ and } s) \\
&= \mathbf{a}_j \mathbf{a}_k \mathrm{E}\left[s(x_j) s_(x_k)\right] \mathrm{E}\left[X_j X_k\right] && (s(x_i)^2 = 1 \text{ since the range is } \{-1, 1\}) \\
&= \mathbf{a}_j \mathbf{a}_k \mathrm{E}\left[s(x_j)\right] \mathrm{E}\left[s_(x_k)\right] \mathrm{E}\left[X_j X_k\right] && (s \text{ is pairwise independent and } j \neq k) \\
&= 0 \cdot 0 \mathbf{a}_j \mathbf{a}_k \mathrm{E}\left[X_j X_k\right]
\end{aligned}
$$

The above shows that $\mathrm{E}\left[T_j T_k\right] = \mathrm{E}\left[T_j\right] \mathrm{E}\left[T_k\right]$, therefore we conclude that two distinct terms in the given summation are uncorrelated.

Problem Two: Cardinality Estimation (12 Points)

A ***cardinality estimator*** is a data structure that supports the following two operations:

- ds.`see`(x), which records that the value x has been seen; and
- ds.`estimate`(), which returns an estimate of the number of ***distinct*** values we've seen.

Imagine that we are given a data stream consisting of elements drawn from some universe $\mathscr{U}$. Not all elements of $\mathscr{U}$ will necessarily be present in the stream, and some elements of $\mathscr{U}$ may appear multiple times. We'll denote the number of unique elements in the stream as $\boldsymbol{F_0}$.

Here's an initial data structure for cardinality estimation. We'll begin by choosing a hash function h uniformly at random from a family of 2-independent hash functions $\mathscr{H}$, where every function in $\mathscr{H}$ maps $\mathscr{U}$ to the open interval of real numbers $(0, 1)$.

Our data structure works by hashing the elements it `see`s using h and doing some internal bookkeeping to keep track of the kth-smallest unique hash code seen so far. The fact that we're tracking *unique* hash codes is important; we'd like it to be the case that if we call `see`(x) multiple times, it has the same effect as just calling `see`(x) a single time. (The fancy term for this is that the `see` operation is ***idempotent***.) We'll implement `estimate`() by returning the value $\hat{F} = \frac{k}{h_k}$, where h_k denotes the kth smallest hash code seen.

i. Explain, intuitively, why $\hat{F}$ is likely to be close to the number of distinct elements.

> **Solution:** Intuitively, we expect our distribution of *unique* hash-codes to be uniform over the interval $(0, 1)$. Taking this intuition further, and to clarify with an example, if we had one hash code, it's expected value would be $\frac{1}{2}$. Taking the inverse of this, we would get 2 as the guess for the number of unique values see.
>
> Continuing with this example, if we had 2 unique hash codes, we would expect the values to be $\frac{1}{3}, \frac{2}{3}$, with an estimate of 3.
>
> We have hopefully developed sufficient intuition through example to now approach the general case. If we have n unique hash codes, we would expect the values to be uniformly distributed such that they divide the unit interval equally. As such, we'd expect them to be located at $\frac{1}{n+1}, \frac{2}{n+1}, \cdots, \frac{n}{n+1}$. We can then see that all of our estimates will be $n + 1$, which for large n is approximately n (eg, the number of unique hash codes).

Let $\varepsilon \in (0, 1)$ be some accuracy parameter that's provided to us.

ii. Prove that $\Pr\left[\hat{F} > (1 + \varepsilon)F_0\right] \leq \frac{2}{k\varepsilon^2}$. This shows that by tuning k, we can make it unlikely that we overestimate the true value of F_0.

As a hint, use the techniques we covered in class: use indicator variables and some sort of concentration inequality. What has to happen for the estimate $\hat{F}$ to be too large? As a reminder, your hash function is only assumed to be 2-independent, so you can't assume it behaves like a truly random function and can only use the properties of 2-independent hash functions.

Solution: We follow the hint. First, we need to consider what it means for our estimate to be too large. More precisely, what exactly does it mean for the event $\hat{F} > (1+\varepsilon)F_0$ to occur. We can immediately expand this out in-terms of the randomness involved by noting:

$$\hat{F} > (1+\varepsilon)F_0 \implies \frac{k}{h_k} > (1+\varepsilon)F_0 \implies h_k < \frac{k}{(1+\varepsilon)F_0}$$

Given how our algorithm works, this bad event can occur only if at least k distinct elements from our universe $\mathscr{U}$ were hashed to a value smaller than $\frac{k}{(1+\varepsilon)F_0}$. To see this, suppose this does not occur. Then this means that we saw at most k *distinct* elements whose hash codes were smaller $\frac{k}{(1+\varepsilon)F_0}$, implying that the k-th smallest hash-code is actually greater than $\frac{k}{(1+\varepsilon)F_0}$.

With the above insight, let use define B_i be an indicator random variable correspondong to the i-th distinct element, x_i, `seen` by our data-structure. We define it as follows:

$$B_i = \begin{cases} 1 & h(x_i) < \frac{k}{1+\varepsilon F_0} \\ 0 & \text{otherwise} \end{cases}$$

Then we can define $B = \sum_i B_i$, which intuitively just represents the number of distinct elements seen which hashed to a value smaller than $\frac{k}{(1+\varepsilon)F_0}$. We can now re-state our original problem as:

$$\begin{aligned} \Pr\left[\hat{F} > (1+\varepsilon)F_0\right] &= \Pr\left[h_k < \frac{k}{(1+\varepsilon)F_0}\right] \\ &\leq \Pr\left[B \geq k\right] \end{aligned}$$

We note that the above is starting to resemble some concentration inequalities. As

such, we compute some propertie sof B. We begin by computing $\mathrm{E}[B]$.

$$\begin{aligned}
\mathrm{E}[B] &= \mathrm{E}\left[\sum_i B_i\right] \\
&= \sum_i \mathrm{E}[B_i] && \text{(Linearlity of expectation)} \\
&= \sum_i \Pr\left[h(x_i) < \frac{k}{(1+\varepsilon)F_0}\right] \\
&< \sum_i \frac{k}{(1+\varepsilon)F_0} && \text{(By distribution property of } h\text{)} \\
&= \frac{k}{1+\varepsilon}\frac{\sum_1 i}{F_0} \\
&= \frac{k}{1+\varepsilon} && \text{(We're summing over distinct } x_i\text{, so we have } \textstyle\sum_i 1 = F_0\text{)} \\
\implies \mathrm{E}[B] &< \frac{k}{1+\varepsilon}
\end{aligned}$$

Next, we compute $\mathrm{Var}[B]$.

$$\begin{aligned}
\mathrm{Var}[B] &= \mathrm{Var}\left[\sum_i B_i\right] \\
&= \sum_i \mathrm{Var}[B_i] && (B_i \text{ are pairwise independent due to properties of } h(x_i)) \\
&= \sum_i \mathrm{E}\left[B_i^2\right] - \mathrm{E}[B_i]^2 && \text{(Def. of variance)} \\
&< \sum_i \mathrm{E}\left[B_i^2\right] \\
&= \sum_i \mathrm{E}[B_i] && \text{(Properties of indicators)} \\
&< \sum_i \frac{k}{(1+\varepsilon)F_0} && \text{(See prev. results)} \\
&= \frac{k}{1+\varepsilon} && \text{(See prev. results)}
\end{aligned}$$

Using the above results and putting everything together, we now have:

$$\begin{aligned}
\Pr\left[\hat{F} > (1+\varepsilon)F_0\right] &\leq \Pr\left[B \geq k\right] && \text{(See prev. results)} \\
&= \Pr\left[B - \mathrm{E}\left[B\right] \geq k - \mathrm{E}\left[B\right]\right] \\
&= \Pr\left[B - \mathrm{E}\left[B\right] \geq k(1 - \frac{1}{1+\varepsilon})\right] \\
&= \Pr\left[\left|B - \mathrm{E}\left[B\right]\right| \geq \frac{k\varepsilon}{1+\varepsilon}\right] && \text{(Trivial upper-bound)} \\
&< \frac{k}{1+\varepsilon}\frac{(1+\varepsilon)^2}{k^2\varepsilon^2} && \text{(Applying Chebyshev's Inequality)} \\
&= \frac{1+\varepsilon}{k\varepsilon^2} && \text{(Simplify)} \\
&\leq \frac{2}{k\varepsilon^2} && (\varepsilon < 1)
\end{aligned}$$

As such, we can safely conclude that:

$$\Pr\left[\hat{F} > (1+\varepsilon)F_0\right] < \frac{2}{k\varepsilon^2}$$

which means our estimate won't be too large, with high-probability which can be tweaked by changing k.

Using a proof analogous to the one you did in part (ii) of this problem, we can also prove that

$$\Pr\left[\frac{k}{h_k} < (1-\varepsilon)F_0\right] \leq \frac{2}{k\varepsilon^2}.$$

The proof is very similar to the one you did in part (ii), so we won't ask you to write this one up. However, these two bounds collectively imply that by tuning k, you can make it fairly likely that you get an estimate within $\pm\varepsilon F_0$ of the true value! All that's left to do now is to tune our confidence in our answer.

iii. Using the above data structure as a starting point, design a cardinality estimator with tunable parameters $\varepsilon \in (0,1)$ and $\delta \in (0,1)$ such that:

- `see`(x) takes time $O\left(\log \varepsilon^{-1} \cdot \log \delta^{-1}\right)$;
- `estimate`() takes time $O\left(\log \delta^{-1}\right)$, and if we let C denote the estimate returned this way, then
$$\Pr\left[|C - F_0| > \varepsilon F_0\right] < \delta; \text{and}$$
- the total space usage is $\Theta\left(\varepsilon^{-2} \log \delta^{-1}\right)$.

Solution:
Overview

As an overview, our data structure is actually relatively simple. We run $d = 24 \ln \delta^{-1}$ indepent copies of the data structure described above, each with it's own independently randomly chosen hash function. For each of the data structures, we use $k = \frac{12}{\varepsilon^2}$. Each *ds*.`see` leads to a call to `see` for the d sub-structures. We merge the results of `estimate` by simply returning the median reported value.

Data Structure Details

We begin by specifying in more detail how the *small* data structure from the previous part works.

The Small Data Structure

The *small* data structure maintains a max-heap, H, of the smallest $k = \frac{12}{\varepsilon^2}$ values seen. Duplicate inserts into the max-heap are a no-op (eg, no duplicate values allowed).

- On a `see`(x) operation, if $|H| < k$, we insert $h(x)$ into H. Otherwise, we compare the root of H to $h(x)$. If the value is smaller, the root is removed and the new $h(x)$ is inserted into the max-heap.
- On a `estimate` call, if $|H| < k$, simply return $|H|$. Otherwise, we look at the root value h_k and return $\hat{F} = \frac{k}{h_k}$ as previously described.

The Primary Data Structure

With the above details clarified, we now design the *primary* data structure, *ds*. To begin, instantiate $d = 24 \ln \delta^{-1}$ independent versions of the *small* data structure described above, label them D_i.

- *ds*.`see`(x): Simply perform D_i.`see`(x) for all i we've instantiated (there are d of them).
- *ds*.`estimate`(): Perform D_i.`estimate` and obtain d estimates. Compute the median value of these d estimates, and return this median.

The above describes in full detail the exten of the data structure.

Approximate Correcness: Let C be the estimate produced by our data structure. We wish to show that:

$$\Pr\left[|C - F_0| > \varepsilon F_0\right] < \delta \implies \Pr\left[|C - F_0| \leq \varepsilon\right] \geq 1 - \delta$$

We first tbegin by discussing the correcness of the *small* data structure.

The Small Data Structure

We wish to show that the small data structure returns the estimate $\hat{F} = \frac{k}{h_k}$, where h_k is the k-th smallest unique hash code. This primarily follows from the correctness of H, the max-heap which we maintain. By induction on the number of operations, we can see that for $|H| \geq k$, the root values is the k-th smallest unique hash code. As such, when $|H| \geq k$, the structure correctly return $\hat{F} = \frac{k}{h_k}$. We note that the structure return $|H|$ when $|H| < k$, which is trivially the correct cardinality.

Finally, from part *ii* above, we note that the described *small* data structure provides the following probabilistic guarantee for $k = \frac{12}{\varepsilon^2}$

$$\begin{aligned}
\Pr\left[|\hat{F} - F_0| > \varepsilon F_0\right] &= \Pr\left[C < (1-\varepsilon)F_0\right] + \Pr\left[C > (1+\varepsilon)F_0\right] \\
&\qquad \text{(Definition of absolute value)} \\
&< \frac{2}{k\varepsilon^2} + \frac{2}{k\varepsilon^2} \qquad \text{(Results from part } ii\text{)} \\
&= \frac{4}{k\varepsilon^2} \\
&= \frac{1}{3}
\end{aligned}$$

THe Primary Data Structure

With the above established, we now proceed to show correcness of the *primary data structure*. The *primary* data structure doesn't actually perform much work, with the correcness following almost immediately from the *small* data structures.

The only interesting part is proving the probabilstic bounds, which we do now.

For *ds*.`estimate`(), we output the median of the $d = 24 \ln \delta^{-1}$ smaller data structures. As such, the only way to to output an estimate C such that $|C - F_0| > \varepsilon F_0$ is if more than half of the *small* data structures output an estimate that is more that εF_0 from the true value.

We let X be the random variable which counts the number of *small* data structures which output a "bad" answer. From the previous analysis, we know that each data structure has a failure probability of $\frac{1}{3}$. We also know that each data structure is independent, by construction. As such, we can upper-bound X with a binomial variable $\text{Binom}(d, \frac{1}{3})$. Note that this satisfies the requirements for applying the Chernoff bound, so we have:

$$\begin{aligned}
\Pr\left[X > \frac{d}{2}\right] &\leq e^{\frac{-d(\frac{1}{2}-\frac{1}{3})^2}{\frac{2}{3}}} \\
&= e^{-\frac{d}{24}} \qquad \text{(Simplify)} \\
&= e^{-\ln \delta^{-1}} \qquad \text{(Substitute } d = 24 \ln \delta^{-1}\text{)} \\
&= \delta
\end{aligned}$$

From the above, we can see that our success probability is $1 - \delta$, as desired. Therefore, our data structure has the probabilistic guarantees we were looking for.

Run-time and space complexity

We now proceed to analyze the run-time and space complexity of each of our data structure.

- *ds*.`see`(x): We must perform `see` on each of d sub-structures. Each operation takes time $O(\log k)$ (inserting into a heap), giving a total run-time for this operation of $O(d \log k) = O(\log \delta^{-1} \log \varepsilon^{-2}) = O(\log \delta^{-1} \log \varepsilon^{-1})$
- *ds*.`estimate`(): We must perform `estimate` on each of d sub-structures. Each such operation takes $O(1)$ (just getting the max out of a heap), so the total run-time is $O(d) = O(\log \delta^{-1})$.

As for the space taken by the data structure, we note that we have d sub-structures (with constant overhead). Each of the sub-structures stores k values in its heap, for space usage of $\Theta(kd) = \Theta(\varepsilon^{-2} \log \delta^{-1})$.

This complete our analysis.

As a reminder, $\log \varepsilon^{-p} = \Theta(\log \varepsilon^{-1})$ for any constant p.

There are a number of really beautiful approaches out there for building cardinality estimators. Check out the impressive HyperLogLog estimator for an example of a totally different approach to solving this problem that's used widely in practice!

Problem Three: Hashing IRL (10 Points)

This one is mostly, but not all, coding! In addition to the brief writeup below, make sure to submit your code on `myth`.

i. The theory predicts that linear probing and cuckoo hashing degenerate rapidly beyond a certain load factor. How accurate is that in practice?

> **Solution:** The statement appears to be fairly accurate for linear probing. Across all tested hash functions ($2, 3, 5$-Independent, Identity, Jenkins, and Tabulation), the insert and hit average times scale relatively well with load factors up to 0.9, but degenerately extremely rapidly (taking $\tilde{5}$x longer) for 0.99 load factor. The same thing occurs for the miss average time, but it appears to occur at a lower loadfactor (factors greater than 0.7 already exhibit rapid degeneration).
>
> As for cuckoo hashing, miss and hit rates don't exhibit any degenerate behavior (as expected, these are always constant). Surprisingly, insert timings seem to exibit severely degenerate times only for the 2-independent polynomial hash, increasing rapidly after a 0.45 load factor. The other hash functions do increase in time (after a 0.45 load), but do so slowly (the difference in average insert from 0.2 load to 0.499 is approximately 2x).

ii. How does second-choice hashing compare to chained hashing across the range of load factors? Why do you think that is?

> **Solution:** Chained-hashing is faster than second-choice hashing across the board for the range of load factors, across inserts, hits, and misses.
>
> While this appears surprising at first, it makes sense. The largest tested load-factor is 3.0, which really means that the expected length of the longest chain is $\Theta(\log n / \log \log n)$ nodes, compared to $\Theta(\log \log n)$ for second-choice hashing. While in theory this would give a large advantage to second-choice hashing, the values of n used in the test-harness is small enough where the difference between these two values is not meaninful. Instead, the cost is primarily governed by the constant factors. As such, second-choice hashing is at a disadvantage.

iii. How does Robin Hood hashing compare to linear probing across the range of load factors? Why do you think that is?

> **Solution:** We see that across the range of load-factors tested, Robin hood hashing tends to have similar performance as linear probing for inserts and hits. This makes sense, since the insert process is almost idential for both hashing techniques, with Robin hood hashing having a slightly larger constant factor (which is also reflected in our results, but only if analyzed closely, and is the most evident with the Tabulation Hash at high load-factors).
>
> However, Robin hood hashing shines when it comes to misses, especially at high load

factors such as 0.9 and 0.99. This is almost surely due to the ability to perform early stopping when an element is not in the table. While in linear probing, a load-factors nearing 1 lead a long scan of a large number of elements in the table, with Robin hoodo hashing, this scan can stop early (as soon as it finds an element closer to home). This is clearly evident in our timing results, since Robin hood. hashing is on average 100 times faster than linear probing for 0.99 load-factor.

iv. In theory, cuckoo hashing requires much stronger classes of hash functions than the other types of hash tables we've covered. Do you see this in practice?

Solution: Yes, we do see this in practice. Most other hashing techniques aren't affected much by the choice of hash function. However, for cuckoo hashing, it's very clear that a 2-Independent Polynomial hash function actually leads to very poor insert performance, even for relatively low load factors (eg, $0.4, 0.45$).

However, we see that in practice, all other hash functions perform relatively well for cuckoo hashing (only 2-independent is a problem).

v. In theory, cuckoo hashing's performance rapidly degrades as the load factor approaches $\alpha = \frac{1}{2}$. Do you see that in practice?

Solution: This is true in practice only for the 2-Independent Polynomial hash. For all others, this is not true. We see that for all good hash functions, the average insert time for a load factor of 0.499 is around $2x$ slower than the insert for a load factor of 0.2.