CSE 312: Foundations of Computing II

Advanced Topics Session #2

Interesting Probabilistic Problems and Randomness in Computer Science

Lecturer: Alex Tsun

Date: April 11, 2017

0 Introduction

The applications of probability to computer science are growing, and it is increasingly important for computer scientists to be proficient in the language of probability. It could help you win on a game show, win at gambling, design efficient algorithms, and/or perform machine learning tasks! We discuss some paradoxes below, and applications of probability to hashing, and in particular, the bloom filter data structure.

1 Monty Hall Problem

The Monty Hall Problem is as follows: you are on a game show, and there are three doors for you to choose from. One of them has a car behind it, and the other two have goats behind them. You pick a door, say without loss of generality (WLOG) door #1, and then the host opens another door WLOG, door #3 which has a goat behind it. You are then given the option to switch doors to door #2 – should you switch doors?

The standard assumptions are that: the host will open a door that the contestant didn't choose, that he will always only reveal a goat, and you will always have a choice to switch doors.

What do you think? It seems reasonable to think that there is no point in switching because the probability that it's behind your door or the other door should be equal right? But the answer is in fact, that it is better to switch doors! Suppose WLOG that you choose door 1, let's do a case analysis:

	Door #1	Door #2	Door #3	Win if Stay	Win if Switch
Case 1	**Car**	Goat	Goat	**Car**	Goat
Case 2	Goat	**Car**	Goat	Goat	**Car**
Case 3	Goat	Goat	**Car**	Goat	**Car**

You can see that these three are all equally likely, and you will win the car in two of the three scenarios! So in fact, it **is better to switch**. You can think of it like this – when he opens one of the other doors, it is a goat, and the $1/3$ probability of that door being a car had to go somewhere. The question is – did it go evenly to the remaining two doors? This analysis shows that the $1/3$ probability "went" to the other unopened door, so in fact the probability that the car is behind your door is $1/3$ while the probability the car is behind the other becomes $\frac{1}{3}+\frac{1}{3}=\frac{2}{3}$. This is a an interesting paradox to think about!

To help with intuition, suppose instead there are 1,000,000 doors instead of 3, and again WLOG you picked door #1. If he showed you 9,999,998 of the others were goats, you'd be quick to switch!

2 Birthday Paradox

What is the probability that out of n people, none share the same birthday **as you**? Assume that there are only 365 possible birthdays, each equally probable. Let E be the event that none of the n people share the same birthday as you. Then,

$$|E| = 364^n$$

because each of them can choose one of the other 364 days. The sample space is the number of different ways they can have birthdays, which gives

$$|\Omega| = 365^n$$

So,

$$\mathbb{P}(E) = \frac{|E|}{|\Omega|} = \left(\frac{364}{365}\right)^n$$

For some values of n – if $n = 23$, this value is about 0.94, if $n = 90$, this value is about 0.78, and if $n = 253$, this value is just under 0.50.

Now, suppose we still have n people in the room, but now we ask what is the probability that no two of them share a birthday? Well, if $n > 365$, the probability is 0 by the pigeonhole principle. Otherwise, let F be the event that none of them share the same birthday. This is the number of n-permutations, so

$$|F| = P(365, n) = 365 \cdot 364 \cdot \ldots \cdot (365 - (n - 1))$$

and the sample size is the same: $|\Omega| = 365^n$.

Therefore,

$$\mathbb{P}(F) = \frac{|F|}{|\Omega|} = \frac{P(365, n)}{365^n} = \frac{365}{365} \cdot \frac{364}{365} \cdot \ldots \cdot \frac{365 - (n - 1)}{365}$$

For some values of n – if **$n = 23$, this value is just under 0.50**, if $n = 90$, this value is $< \frac{1}{162{,}000}$, and if $n = 150$, this value is $< \frac{1}{3{,}000{,}000{,}000{,}000{,}000}$.

Why are the results so different? It's because in the first analysis, we looked at the probability just no one shared a birthday with **you**, so we were considering n possible matches to your birthday. In the second analysis, we looked at $\binom{n}{2}$ pairs and saw what the probability of any of those pairs sharing a birthday was. It makes sense that it requires much fewer people to have a single match in the second case!

3 Hashing

One application of probability to computer science is in **hashing** – when we have a hash function h which maps elements x to some index in a hash table. You've learned in CSE 332 how to handle hash collisions, but we will analyze some results probabilistically, assuming that h hashes elements x uniformly at random to each bucket. That is, assume hash values are uniform and independent over $\{0,1,\dots,m-1\}$.

Suppose we have n items and m buckets (n elements into a hash table of size m), and each element is equally likely to be hashed to any of the m buckets by h. We can ask several questions.

Exercise: What is the probability that any two elements hash to the same bucket?
Solution: The answer is simply $\frac{1}{m}$. The first hashes to some bucket k, and the probability that the second also hashes to k is just $\frac{1}{m}$.

Exercise: Assume $m = n$. What is the probability that a particular bucket is empty after all n elements are put into the hash table? What happens to this quantity as $n \to \infty$?
Solution: The probability that a single item isn't hashed to a particular bucket is $\left(1-\frac{1}{n}\right)$. The probability that each of n items doesn't hash to this particular bucket is $\left(1-\frac{1}{n}\right)^n$, by independence. As $n \to \infty$, this quantity approaches $1/e$.

Exercise: What is the probability that a particular bucket has exactly k elements in it?
Solution: Out of the n items, we need k of them to hash to this bin. There are $\binom{n}{k}$ ways to choose which k. Once we've selected which k of them to hash to this bin, the other $n-k$ must not hash to this bin. This happens with probability $\left(\frac{1}{m}\right)^k\left(1-\frac{1}{m}\right)^{n-k}$. So the answer is $\binom{n}{k}\left(\frac{1}{m}\right)^k\left(1-\frac{1}{m}\right)^{n-k}$.

You can do a lot more analysis, and ask questions that we don't have the tools to answer yet. Some may include – what's the expected **load** of each bin? What is the expected **maximum load**?

4 Bloom Filters

Suppose we wanted to implement a **probabilistic data structure** that would only have the two operations:

- add(x): adds x to the data structure
- contains(x): returns whether or not this element was already added to the data structure

However, contains(x) would not be deterministic - if we return false, then x definitely isn't in the data structure. If we return true, then x "probably" is in the data structure. Therefore, we may get an incorrect answer with some probability.

We maintain this data structure using a **bit array** of length m – that is, an array of length m with only 0's and 1's. Initially, all elements of the array (call it t) are set to 0. Let X be the possible elements we can add to this data structure (e.g., integers, strings, etc). Suppose we have some hash function $h: X \to \{0,1,\dots,m-1\}$. Below, we describe the implementations of add(x) and contains(x).

```
add($x$):
        $t[h(x)] = 1$
contains($x$):
        return $t[h(x)] == 1$
```

You can see that these operations are both $O(1)$ time and the space required is $O(m)$. Hopefully you can see why there are "one-sided errors" – if we return false, then the element definitely is not in the data structure, because if it were, we would've set the hash value to 1. However, if we return true, the element may or may not be in the data structure, since some other element we added could've set the bit to 1. We'll make the same probabilistic assumptions we did in section 3 about hashing (uniform, independent).

Exercise: Suppose we have hashed n different elements $x_1, \dots, x_n$ into the data structure, and we now query it with contains(x). If x was one of the elements that we hashed (it was equal to x_i for some i), what is the probability that we return the wrong answer (false)? If x was not one of the elements that we hashed, what is the probability that we return the wrong answer (true)?
Solution: For the first case, the answer is 0 – if it were added, the bit in t at its hash value would be set to 1 and we would be guaranteed to return the correct answer. For the second case, the probability that one particular element didn't hash to the same index is $1-\frac{1}{m}$, and for this to happen all n times, we get $\left(1-\frac{1}{m}\right)^n$. This is the probability of being correct, so the probability of being wrong is $1-\left(1-\frac{1}{m}\right)^n$. For a fixed table size m, as $n \to \infty$, the probability of being wrong $\to 1$!

If we have just a single table, we may have a high probability of being wrong. How can we reduce this false positive rate (the probability of an error)?

The key to reduce error probability in probabilistic data structures or randomized algorithms is repetition! So now suppose we have k different independent hash functions $h_1, \dots, h_k: X \to \{0,1,\dots,m-1\}$, and k hash tables of size m each, $t_1, \dots, t_k$. We call this data structure a **bloom filter**. Here's how it implements things using this extra space and these extra hash functions:

```
add($x$):
        for $i = 1, \dots, k$
                $t_i[h_i(x)] = 1$
contains($x$):
        return $t_1[h_1(x)] == 1 \wedge t_2[h_2(x)] == 1 \wedge \dots \wedge t_k[h_k(x)] == 1$
```

This time, add and contains are now both $O(k)$ time and the space needed is $O(km)$. Notice that if contains(x) returns false, then at least one of the tables had the hash value set to 0, but if this element were in the bloom filter, it would've been set to 1 by add. If it returns true, the probability of an incorrect answer is **significantly** decreased. If it were to be a false positive, then all of the indices in the k tables must've been set to 1 by chance – by other elements.

Exercise: Suppose we have hashed n different elements $x_1, \dots, x_n$ into the bloom filter, and we now query it with contains(x). If x was not one of the elements that we hashed, what is the probability that we return the wrong answer (true)?
Solution: In the previous exercise, we determined that if we used only one table, the answer would be $1-\left(1-\frac{1}{m}\right)^n$. But the probability of being wrong is the probability that we are wrong in all of them, so the answer is $\left(1-\left(1-\frac{1}{m}\right)^n\right)^k$. Notice that $1-\left(1-\frac{1}{m}\right)^n < 1$, so by raising it to the k^{th} power, the error rate is decaying exponentially in k. So our error rate is lowered by having more hash functions!

You might be wondering, how does this error rate compare to if we had just used one table with km spaces in the bit vector instead? This way, contains and add would still be $\mathcal{O}(1)$. Let us assume $m = n$, and suppose n is large, for simplicity. If we had just used one larger table, we would have an error rate of:

$$1-\left(1-\frac{1}{km}\right)^n \leq e^{-\left(1-\frac{1}{km}\right)^n} \approx e^{-e^{-1/k}}$$

as opposed to the bloom filter's

$$\left(1-\left(1-\frac{1}{m}\right)^n\right)^k \leq e^{-\left(1-\frac{1}{m}\right)^n k} \approx e^{-ke^{-1}}$$

where we used the facts that $\left(1-\frac{b}{n}\right)^n \to e^{-b}$ and $1-x \leq e^{-x}$. Notice that, as $k \to \infty$ the error for the single table is bounded by only $1/e$ whereas the error for the bloom filter approaches 0, and does so very fast, even for moderate values of k. However, there is no free lunch unfortunately – this better error bound using a bloom filter will cost you $\mathcal{O}(k)$ for add and contains.

Analysis of the Bloom Filter:

- If we want to keep track of n elements with false positive probability $\leq \delta$, how large do we choose m and k?
- If $m \in \mathcal{O}(n)$ and $k \in \mathcal{O}\left(\log\frac{1}{\delta}\right)$, then $\mathbb{P}(error) \leq \delta$.

Applications of the bloom filter to computer science are endless! You may not think these are super useful (why would it be useful only to check whether something was in the structure?), but here are some applications from Wikipedia:

- Medium uses Bloom filters to avoid recommending articles a user has previously read
- The Google Chrome web browser used to use a Bloom filter to identify malicious URLs. Any URL was first checked against a local Bloom filter, and only if the Bloom filter returned a positive result was a full check of the URL performed (and the user warned, if that too returned a positive result).
- Google BigTable, Apache HBase and Apache Cassandra, and Postgresql use Bloom filters to reduce the disk lookups for non-existent rows or columns. Avoiding costly disk lookups considerably increases the performance of a database query operation.

5 Conclusion

You might think, why would we ever want to do something probabilistically, when we could do it deterministically? There are a myriad of reasons – it may not be feasible to run your algorithm because it is computationally intractable (**NP-Complete**), or you don't mind making errors once in a while if you get a huge performance boost. Also, we can improve the worst-case performance of some algorithms – worst-case analysis of quicksort yields $\mathcal{O}(n^2)$ if an **adversary** were to feed you the input that would make your algorithm run as slow as possible. Adding randomness to your pivot would yield an average case of $\mathcal{O}(n \log n)$ which has a lower constant factor than mergesort.

Probabilistic data structures and randomized algorithms are used to defend against adversaries, to approximate "hard" problems, and their probability of success can usually be boosted by repeatedly running the algorithm independently. It is crucial to understand probability in the design & analysis of (efficient) algorithms, and also for machine learning!