Randomised Algorithms

How many statisticians does it take to screw in a light bulb?

One. But then you're only 95% confident it's *been* changed.

Most algorithms are *deterministic*. If given the same input, they will return the same output. Every time.

A randomised algorithm will return different outputs for the same input.

That might seem weird: if there is one optimum value for a problem, then shouldn't I always return that?

Well, in most practical optimisation problems we actually want to find a *good* solution. The *best* would be icing, but sometimes that would require almost infinitely more computation to find (that isn't hyperbola).

There are very many examples of randomised algorithms:

- You will see some applied to optimisation in this course.
- Statistical sampling is an almost too obvious case.
- The general class of MCMC for all sorts of stuff.
- Some neural network algorithms start from a random set of weights.

I want to cover some examples you might not normally see in a maths course.

Key example: random hashes

Hash Functions

Imagine my data objects $x \in \Omega$ (the set Ω is sometimes called the *universe*, and the objects the *keys*). The objects can be completely unstructured (or not).

A hash function $h : \Omega \to [m] = \{0, \ldots, m-1\}$. That is, it is a function that maps the objects x to the set $\{0, \ldots, m-1\}$ denoted by $[m]$. We often call the elements of the set *bins* Usually the size of the potential input universe $|\Omega| \gg m$.

Most hash functions are many-to-one. Many hash functions are *surjective* on non-trivial data sets, that is, for every $i \in [m]$ there is at least one potential input element $x \in \Omega$ such that $h(x) = i$. But hash functions are not, in general, invertable. That is, you can't use $h(x)$ to work out x.

A *collision* occurs if $h(x) = h(y)$ for $x \neq y$. That is, two different inputs are mapped to the same hash value.

Hash functions often occur in families H, sometimes indexed by a parameter, $e.g.$, we might write $H = \{h_i\}_{i=1}^{n}$. The index set is often assumed to be very large for cryptography applications, but we don't need that here.

Desirable Properties of Hash Functions

Hash functions have a number of desirable properties depending on application.

- In most cases it is important that they be either fast, or very, very fast.
- We would like the hash function to spread out the data roughly evenly over $[m]$ to avoid collisions as much as possible.

Random Hash Functions

Input data is not really random, so often we use families of random functions, particularly in order to define properties in a formal sense.

In this context, uniformity can be defined by

$$P_{h\in H}\big(h(x) = i)\big) = \tfrac{1}{m},$$

for $x \in \Omega$ and $i \in [m]$. Remember that x is not random here, the function h is. Uniformity is appealing, but can be achieved by stupid hash functions such as the constant $h_i(x) = i$.

A family of hash functions is called *universal* if it satisfies the property that for all $x, y \in \Omega$ with $x \neq y$

$ P_{h \in H}\big(h(x) = h(y) \big) \leq \frac{1}{m}.$

That says that two different inputs collide with a probability at most $1/m$ when we draw a random hash function from H.

Universality might be hard to achieve, particularly with a fast hash, so we often allow near-universailty, by putting a constant factor (say of 2) into the above.

Examples

1. **Linear algebra:** if the inputs are vectors $\mathbf{x} \in \{0,1\}^n$ (0-1 vectors) then we can perform hashing through matrix multiplication by an $m \times n$ 0-1 matrix. We can construct a family of hashes by taking random 0-1 matrices. This example is illustrative but rarely used, because it only works for very restrictive data. More generally we would like hashes that map unstructured data.
2. **Modular arithmetic:** Take $h_i(x) = x + i \mod m$. Or do some multiplication and addition first to get

   $$h_{a,b}(x) = ((ax + b) \mod p) \mod m,$$

   for some given (fixed) prime number p. Chose the right p and set of a and b you get a universal hash family. But this only works for input integers.

2. **Recursive hashes:** You can take any input of a fixed length and convert it first to an integer as a block, and then use modular arithmetic to convert it. But what do you do with arbitrary length data like natural language text? You could pad the text out to a fixed length, but that might have to be really large. You can instead treat the string like a polynomial and use a recursive hash, which is really Horner's method for computing polynomials, but in general this looks like:

$$h_a(\text{'xyf'}) = a \oplus h_x(\text{'yf'}).$$

This is commonly called a *rolling hash*.

Recursion is a general strategy for dealing with fast data – the cost for each element is a small constant.

In many ways such hash function are linked to pseudo-random number generation as well.

The above aren't stupidly slow, but there are faster hashes. If we can use only base-2 arithmetic (multiplication and division by 2) we can make these much faster, for instance.

Applications

A core use for hashes in data science is to enable fast indexing of a data set. That is, a fast way to look up, or add data to a larger repository of potentially unstructured data. A "hash table" is a standard data structure used in most modern programming languages to create associative arrays (also called hashes, and dictionaries). Naively you just store x in a fixed array at location $h(x)$, but it is a little more complicated than that.

But they have many other uses. A key one is in cryptography, but I won't talk about that here. What I will talk about is some cute ways to summarise data very, very quickly to deal with fast data.

Calculating Similarity of Two Giant Sets

Imagine two very, very large sets A and B. A common measure of similarity (roughly the opposite of distance) between sets is the *Jaccard Index*

$$J(A,B) = \frac{A \cap B}{A \cup B}.$$

Imagine, for instance, that A and B are the sets of words that appear in two documents. If the two are identical, then $J(A,B) = 1$. If they are completely different, then $J(A,B) = 0$. Mostly likely they are somewhere in between.

A naive algorithm would check, for every element $a \in A$ if that element also appeared in B. That would take $O(|A| \times |B|)$ to do the dumb way. We could use hash tables to speed it up. But there is a fast approximate approach called MinHash.

The **MinHash** algorithm provides a fast means calculate an approximation to $J(A,B)$. Its usually described in terms of random permutations in order to prove its properties, but the standard implementation is as follows

Or at least my best guess is:

Start with sets S_j to compare, with union Ω and one main hash $g : \Omega \to \{1, \ldots, N\}$. Take k random hash functions $\{h_1, \ldots, h_k\}$ and initialise k values $\{c_1, \ldots, c_k\}$ to ∞. We create a signature of a set as follows:

1. We start by mapping the elements of the sets we want to compare to numbers $\{1, \ldots, N\}$ using a hash f.
2. Use a hash table $g(S)$ to index all of the elements of S
3. For all $x \in \Omega$ (or for $i = 1$ to N)

   If $g(x) \in g(S)$ then take $i = g(x)$

   For $j = 1$ to k

   If $h_j(i) < c_j$ then $c_j \leftarrow h_j(i)$

The signature of S_j is the vector $\mathbf{m}(S_j) = \mathbf{c}$.

The Jaccard index is computed by taking

$$J(A,B) = \tfrac{1}{k} \sum_{i=1}^{k} I\big(m_j(A) = m_j(B)\big),$$

where $I(\cdot)$ is an indicator function. Intuitively this is because the probability that $m_j(A) = m_j(B)$ for any given random hash h_j is proportional to the Jaccard index of the two sets.

The accuracy of the approach depends on k, but the computation is proportional to k, so there is a tradeoff.

There are very many algorithms in this space, showing how hashes can be used (particularly groups of random hashes) to construct various approximate summaries of a data set.

- **Bloom filters** let you quickly test if an element $x \in S$. False positives are possible, but false negatives are not.
- The **Count Min** sketch lets you count (approximately) the number of times an element appears in a set of data.
- We can use hashes (in *feature hashing* or the *hashing trick*) to vectorise a set of features, $i.e.$, put them into tensor form.

They can often be applied to rapidly growing data sets.

To do (1 of the the following)

In Python (using PyTorch or not)

1. Write a rolling hash function for images. Use it to hash the MNIST images into 6 bins. Check the uniformity of your hash on this data.
2. Implement MinHash. Generate two largish random sets of integers from 1-N (avoid the first hashing step), and then compute their Jaccard index both directly and using MinHash.

   See `jaccard.ipynb`

Terminology

Lookup the following:

- FPRAS
- Monte-Carlo vs Los-Vegas Algorithms
- PAC learning
- VC-dimension

There are many similarity indexes that are similar to Jaccard, *e.g.,* the Dice coefficient.

Links

- https://brilliant.org/wiki/randomized-algorithms-overview/
- http://theory.stanford.edu/people/pragh/amstalk.pdf
- https://datascience.stackexchange.com/questions/32557/what-is-the-exact-definition-of-vc-dimension
- https://www.cs.utah.edu/~zhe/pdf/lec-13-pac-definition-upload.pdf
- https://www.cs.princeton.edu/courses/archive/spr08/cos511/scribe_notes/0211.pdf
- https://jeremykun.com/2014/01/02/probably-approximately-correct-a-formal-theory-of-learning/
- https://courses.engr.illinois.edu/cs473/sp2017/notes/05-hashing.pdf
- https://ageek.dev/polynomial-rolling-hash
- https://www.cs.cmu.edu/afs/cs/project/pscico-guyb/realworld/www/slidesS14/minhash.pdf
- https://florian.github.io/count-min-sketch/
- http://dimacs.rutgers.edu/~graham/pubs/papers/cmencyc.pdf
- https://en.wikipedia.org/wiki/Feature_hashing