Introduction to Locality-Sensitive Hashes

Tyler Neylon

145.2018

Locality-sensitive hashes are techniques that dramatically speed up search or de-duplication operations on data. They can be used, for example, to filter out duplicates of scraped web pages at an impressive speed, or to perform near-constant-time lookups of nearby points from a geospatial data set.

When you think about hash functions, you might think about *hash tables*, which is perhaps the most common use case. As a reminder, the hash functions used in a hash table are designed to map a data structure to an integer that can be used to look in a particular *bucket* within the hash table to retrieve (or delete) that element. Common containers with string keys like JavaScript object attributes and Python dictionaries are based on hash tables. Although they might not *guarantee* constant-time lookups, in practice they effectively provide them. Hash functions used for hash tables are called *universal hash functions*. [CHECK]

There are a number of other classes of hash functions as well. For example the SHA1 cryptographic hash function is designed to be *difficult to reverse*, which is useful if you want to store someone's password as a hashed value. [CHECK] Another security-oriented hash function is CHECK, which is actually designed to be *expensive to compute*, as this can deter malicious ne'er-do-wells from easily building large lookup tables to be able to reverse a hash on more likely input values. Hash functions like these are called *secure hash functions*. [CHECK]

Here are what all these various hash functions have in common: * They map a wide variety of input data types to discrete values. * In practice, we care about whether or not two (or more) input values map to the same output (hashed) value.

Locality-sensitive hash (LSH) functions are specifically designed so that collisions of the hash value are *more likely* given two input values that are *close together*. Just as there are different implementations of secure hash functions for different use cases, there are different implementations of LSH functions for different data types and for different definitions of being *close together*. In this post, I'll give a brief overview of the key ideas, and take a look at a toy example based on *random projections* of vectors into lower-dimensional spaces.

1 An example

It will probably be much easier to grasp the main idea with an example. (The "toy example" for random projections will come later. This is like a mini-toy example.)

Suppose you have a million people from across the United States all standing in a huge room. It's your job to get people who live close together to stand together in groups. Imagine how much time it would take to walk up to each person, ask for their street address, map that to a lat/long pair, then write some code to find reasonable geographic clusters, and walk up to every person again and tell them their cluster number. It's a disaster.

Here's a much better way to solve this problem: Write every U.S. zip code on poster boards, and hang those from the ceiling. Then announce to everyone to go stand under the zip code where they live.

Voila! That's much easier, right? The main idea here is also the main idea behind locality-sensitive hashes. We're taking an arbitrary data type (a person, who we could of as a ton of data including their street address), and mapping that data into a set of discrete values (zip codes) such that people who live close together probably hash to the same value. In other words, the clusters are very likely to be groups of neighbors.

The distinction between walking sequentially up to each person versus parallelizing the work by asking everyone to find their own way to their zip code was not an accident. Besides avoiding whatever clustering algorithm you'd have to run on lat/long coordinates, another advantage of this hashing approach is that it's extremely friendly to parallel processing. Despite caring about *relationships* within your data, you can still split up the data any way you like and compute the hashes in a fully parallelized fashion.

Another property of this example is that it is *approximate* in the sense that some people may live across the street from each other, but happen to cross a zip code line, in which case they would not be clustered together here. As we'll see below, it's also possible for data points to be clustered together even when they're very far apart, although a well-designed LSH can at least give some mathematical evidence that this will be a rare event, and some implementations manage to guarantee certain bad cases (such as clustering of very far points or non-clustering of very close points) never happen.

2 Hashing points via projection

Let's start with an incredibly simple mathematical function that we can treat as an LSH. Define $f : \mathbb{R}^2 \to \mathbb{Z}$ for a point $x \in \mathbb{R}^2$ by

$$f(x) := \lfloor x_1 \rfloor;$$

that is $f(x)$ is the largest integer a for which $a \leq x_1$. (For example, $f((3.2, -1.2)) = 3$.)

Let's suppose we choose points at random by uniformly sampling from the origin-centered circle $\mathcal{C}$ with radius 3:

$$\mathcal{C} := \{(x, y) : x^2 + y^2 \leq 3^2\}.$$

If we want to find which of our points in $\mathcal{C}$ are close together, we can estimate this relationship by clustering together points a and $b \in \mathcal{C}$ iff (if and only if) $f(a) = f(b)$. It will be handy to introduce the notation $a \sim b$ to indicate that a and b are in the same cluster. With that notation, we can write our current hash setup as

$$a \sim b \iff h_1(a) = h_1(b).$$

Here's an example of such a clustering:

IMAGE

You can immediately see that some points are far apart yet clustered, while others are relatively close yet unclustered. There's also a sense that this particular hash function h_1 was arbitrarily chosen to focus on the x-axis. What would have happened with the same data if we had used instead $h_2(x) := \lfloor x_2 \rfloor$? Here's that image:

IMAGE

While neither clustering alone is amazing, things start to work better if we use both of them simultaneously. That is, we can redefine our clustering via

$$a \sim b \iff h_1(a) = h_1(b) \text{ and } h_2(a) = h_2(b). \tag{1}$$

Our same example points look like this under the new clustering rule:

IMAGE

2.1 Randomizing our hashes

So far we've defined deterministic hash functions. Let's change that by choosing a random rotation matrix U (a rotation around the origin) along with a random offset $b \in [0, 1)$. Given such a random U and b, we could define a new hash function via

$$h(x) := \lfloor (Ux)_1 + b \rfloor,$$

where I'm using the notation $(vec)_1$ to indicate the first coordinate of the vector value vec (that is, the notation $(Ux)_1$ means the first coordinate of the vector Ux).

It may seem a tad arbitrary to use only the first coordinate here rather than any other, but the fact that we're taking a random rotation first means that we have the same set of possibilities, with the same probability distribution, as we would when pulling out any other single coordinate value.

The advantage of using randomized hash functions is that any theoretical properties we want to discuss will apply without having to worry about pathologically weird data. Conceptually, if we were using deterministic hash functions, then someone could choose the worst-case data for our hash function, and we'd be stuck with that poor performance (for example, choosing maximally-far apart points that are still clustered together by our h_1 function above). By using randomly chosen hash functions, we can ensure that any average-case behavior of our hash functions applies equally well to *all data*. This same perspective is useful for hash tables in the form of *universal hashing*; if randomized hash functions are a new idea for you, I recommend checking out Wikipedia's universal hashing page.

Let's revisit the example points we used above, but now apply some randomized hash functions. In figure CHECK, points are clustered iff both of their hash values (from $h_1()$ and $h_2()$) collide. We'll use that same idea, but this time choose four rotations $U_1, \ldots, U_4$ as well as four offsets $b_1, \ldots, b_4$ to define $h_1(), \ldots, h_4()$ via

$$h_i(x) := \lfloor (U_i x)_1 + b_i \rfloor.$$

Here's the resulting clustering:

IMAGE

It's not obvious that we actually want to use all four of our hash functions. The issue is that our clusters have become quite small. There are a couple ways to address this. One is to simply increase the scale of the hash functions; for example:

$$\tilde{h}_i(x) := \left\lfloor \left(U_i \frac{x}{s} \right)_1 + b_i \right\rfloor,$$

where s is a scale factor (larger s values will result in larger clusters).

However, there is something a bit more nuanced we can look at, which is to allow some adaptability in terms of *how many hash collisions we require*. In

other words, suppose we have k total hash functions (just above, we had $k = 4$). Instead of insisting that all k hash values must match, we could look at cases where some numer $j \leq k$ of them matches. To state this mathematically, we would be rewriting equation (1) as

$$a \sim b \iff \#\{i : h_i(a) = h_i(b)\} \geq j.$$

---

3 References