6 Locality Sensitive Hashing

In the last few lectures we saw how to convert from a document full of words or characters to a set, and then to a matrix, and then to a k-dimensional vector. And from the final vector we could approximate the Jaccard distance between two documents.

However, now we face a new challenge. We have many many documents (say n documents) and we want to find the ones that are close. But we don't want to calculate $\binom{n}{2} \approx n^2/2$ distances to determine which ones are really similar.

In particular, consider we have $n = 1{,}000{,}000$ items and we want to ask two questions:

(Q1): Which items are similar?

(Q2): Given a query item, which others are similar to the query?

For **(Q1)** we don't want to check all roughly n^2 distance (no matter how fast each computation is), and for **(Q2)** we don't want to check all n items. In both cases we somehow want to figure out which ones might be close and the check those.

Consider n points in the plane $\mathbb{R}^2$. How can we quickly answer these questions:

- Hierarchical models (range trees, kd-trees, B-trees) don't work well in high dimensions. We will return to these in **L8**.
- Lay down a Grid: Close points should be in same grid cell. But some can always lay across the boundary (no matter how close). Some may be further than 1 grid cell, but still close. And in high dimensions, the number of neighboring grid cells grows exponentially. One option is to randomly shift (and rotate) and try again.

The second idea is close to a technique called *Locality Sensitive Hashing* (or LSH) which we will explore.

6.1 (Random) Hash Functions

The use LSH we need a good hash functions. There are two main types of hash functions. The second (which we define in the next section, and the rest of the lecture) is locality-preserving. But often one wants a hash functions which is the opposite.

A *random hash function* $h \in \mathcal{H}$ is one that maps from one set $\mathcal{A}$ to another $\mathcal{B}$ (usually $\mathcal{B} = [m]$) so that conditioned on the random choice $h \in \mathcal{H}$, the location $h(x) \in \mathcal{B}$ for any $x \in \mathcal{A}$ is equally likely. And for any two (or more strongly, but harder to achieve, *all*) $x \neq x' \in \mathcal{A}$ the $h(x)$ and $h(x')$ are *independent* (conditioned on the random choice of $h \in \mathcal{H}$). It is important to state that for a fixed $h \in \mathcal{H}$, the map $h(x)$ is *deterministic*, so no matter how many times we call h on argument x, it always returns the same result. The full independence requirement is often hard to achieve in theory, but in practice are often not a major problem. For the purposes of class, most built-in hash functions should work sufficiently well.

Assume we want to construct a hash function $h : \Sigma^k \to [m]$ where m is a power of two (so we can represent it as $\log_2 m$ bits) and Σ^k is a string of k of characters (lets say a string of numbers $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$). So $x \in \Sigma^k$ can also be interpreted as a k-digit number.

- **Modular Hashing:** $h(x) = x \mod m$
  This roughly evenly distributed numbers, but numbers that are both powers of m will *always* hash to the same location. We can use a random large prime $m' < m$. This will leave some bin always empty, but has less regularity.

- **Multiplicative Hashing:** $h_a(x) = \lfloor m \cdot \mathsf{frac}(x * a) \rfloor$
  a is a real number (it should be large with binary representation a good mix of 0s and 1s), $\mathsf{frac}(\cdot)$ takes the fractional part of a number, e.g. $\mathsf{frac}(15.234) = 0.234$, and $\lfloor \cdot \rfloor$ takes the integer part of a number, rounding down so $\lfloor 15.234 \rfloor = 15$. Can sometimes be more efficiently implemented as $(xa/2^q) \mod m$ where q is essentially replacing the $\mathsf{frac}(\cdot)$ operation and determining the number of bits precision.

- **Tabulation Hashing:** $h(x) = H_1[x_1] \oplus \ldots \oplus H_k[x_k]$
  Here $\oplus$ is an *exclusive or* and $H_i : \Sigma \to [m]$ is a map (stored in a fast cache) to a set of random bits, or another (say multiplicative) hash function. This can have surprisingly better *independence* properties, and a recent extension called *twisted tabulation hashing* has even better properties.

6.2 Properties of Locality Sensitive Hashing

We now shift back to the goal of constructing a *locality-preserving* hash function h with the following properties (think of a random grid). The locality needs to be with respect to a distance function $d(\cdot, \cdot)$. In particular, if h is $(\gamma, \phi, \alpha, \beta)$-*sensitive* with respect to d then it has the following properties:

- $\mathbf{Pr}[h(a) = h(b)] > \alpha$ if $d(a, b) < \gamma$
- $\mathbf{Pr}[h(a) = h(b)] < \beta$ if $d(a, b) > \phi$

For this to make sense we need $\alpha > \beta$ for $\gamma < \phi$. Ideally we want $\alpha - \beta$ to be large and $\phi - \gamma$ to be small. Then we can repeat this with more random hash functions to *amplify* the effect, according to a Chernoff-Hoeffding bound. This will effectively make $\alpha - \beta$ larger for any fixed $\phi - \gamma$ gap, and will work for all such $\phi - \gamma$ simultaneously.

6.2.1 Minhashing as LSH

Minhashing is an instance of LSH (and I think was the first such realized instance). Recall we had t hash functions $\{h_1, h_2, \ldots, h_t\}$ where each $h_i : [n] \to [n]$ (at random).

Documents:	D_1	D_2	D_3	D_4	$\ldots$	D_n
h_1	1	2	4	0	$\ldots$	1
h_2	2	0	1	3	$\ldots$	2
h_3	5	3	3	0	$\ldots$	1
$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$
h_t	1	2	3	0	$\ldots$	1

Where

$$\mathsf{JS}_t(D_1, D_2) = \mathbf{E}[(1/t) \text{ \# rows } h_i(D_1) = h_i(D_2)].$$

Then for some similarity threshold τ we can set $\tau = \gamma = \phi$ and the set $\alpha = \mathsf{JS}(a, b)$ and $\beta = 1 - \mathsf{JS}(a, b)$. *This scheme will work for any similarity such that* $s(a, b) = \mathbf{Pr}[h(a) = h(b)]$.

6.3 Banding for LSH

But how do we use LSH to answer questions **Q1** and **Q2**? If we only check the items that *always* fall into the same bin, then with more hash functions eventually almost nothing will be in the same bin (acts like a t-dimensional grid). If we items that *for any hash* fall into the same bin, then with more hash functions, eventually almost all items will be checked.

If the first approach is "Papa" bear's LSH, and the second approach is "Mama" bear's, then we will need a "Baby" bear approach. Baby bear likes *banding*.

Suppose we have a budget of t hash functions. We are going to use $b < t$ in Papa bears approach and $r = t/b$ in Mama bears approach (Baby bear takes after both of her parents after all $\ddot\smile$). Here b is the number of hashes in a band, and r is the number of rows of bands.

Recall that $s = \mathsf{JS}(D_1, D_2)$ is the probability that D_1 and D_2 have a hash collision. Then we have

$$\begin{aligned} s^r &= \text{ probability all hashes collide in 1 band} \\ (1 - s^r) &= \text{ probability not all collide in 1 band} \\ (1 - s^r)^b &= \text{ probability that in no bands do all hashes collide} \\ f(s) = 1 - (1 - s^r)^b &= \text{ probability all hashes collide in at least 1 band} \end{aligned}$$

We can plot f (as seen in Figure 6.3) as an S-curve where on the x-axis $s = \mathsf{JS}(D_1, D_2)$ and the y-axis represents the probability that the pair D_1, D_2 is a candidate to check the true distance. In this example we have $t = 15$ and $r = 3$ and $b = 5$.

$$\begin{aligned} f(0.1) &= 0.005 \\ f(0.2) &= 0.04 \\ f(0.3) &= 0.13 \\ f(0.4) &= 0.28 \\ f(0.5) &= 0.48 \\ f(0.6) &= 0.70 \\ f(0.7) &= 0.88 \\ f(0.8) &= 0.97 \\ f(0.9) &= 0.998 \end{aligned}$$

Choice of r and b. Usually there is a budget of t hash function one is willing to use. Perhaps it is $t = (2/\varepsilon^2)\ln(2/\delta)$. Then how does one divvy them up among r and b?

The threshold τ where f has the steepest slope is about $\tau \approx (1/b)^{1/r}$. So given a similarity s that we want to use as a cut-off (e.g. $s = \alpha = \beta$) we can solve for $r = t/b$ in $s = (t/r)^{1/r}$ to yield $r \approx \log_s(t)$.

If there is no budget on r and b, as they increase the S curve gets sharper.

6.4 LSH for Euclidean Distance

We so far operated on a specific distance and similarity for min hashing (more on other distances in L7). This extends to many other distances including, most famously the Euclidean distance between two vectors $v, u \in \mathbb{R}^k$ so $d_{\mathsf{E}}(u, v) = \|u - v\| = \sqrt{\sum_{i=1}^k (v_i - u_i)^2}$. This is most useful when k is quite large.

The hash h is defined as follows.

1. First take a random *unit vector* $u \in \mathbb{R}^d$. A *unit vector* u satisfies that $\|u\| = 1$, that is $d_{\mathsf{E}}(u, 0) = 1$. We will see how to generate a random unit vector later.

2. Project $a, b \in P \subset \mathbb{R}^k$ onto u:

$$a_u = \langle a, u \rangle = \sum_{i=1}^k a_i \cdot u_i$$

   This is *contractive* so $\|a_u - b_u\| \leq \|a - b\|$.

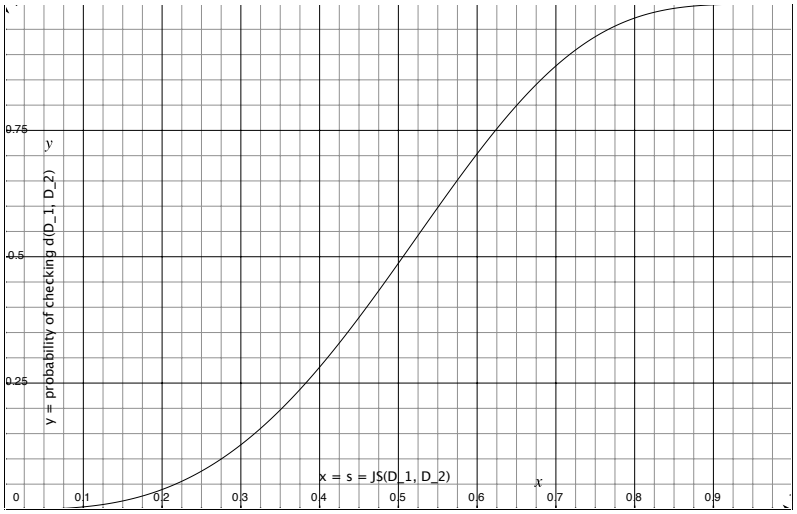

Figure 6.1: Probability that the distance $d(D_1, D_2)$ is checked in a LSH scheme with $r = 3$ bands with $b = 5$ hashes each, as a function of $s = \mathsf{JS}(D_1, D_2)$.

3. Create bins of size γ on u (now in $\mathbb{R}^1$). The index of the bin a falls into is $h(a)$.

If $\|a - b\| < \gamma/2$ then $\mathbf{Pr}[h(a) = h(b)] \geq 1/2$. If $\|a - b\| > 2\gamma = \phi$ then $\mathbf{Pr}[h(a) = h(b)] < 2/3$. So this is $(\gamma/2, 2\gamma, 1/2, 1/3)$-sensitive.

To see the second claim (with 2γ) we see that for a collision we need $\cos(a - b, u) < \pi/3$ (out of $[0, \pi]$). Otherwise $\|a - b\| > 2\|a_u - b_u\|$ and thus they must be in different bins.

We can also take $h(a) = \langle a, u \rangle \mod (t\gamma)$. For large enough t, the probability of collision in different bins is low enough that this can be effective.