# Lectures 12 & 13: Locality Sensitive Hashing

## More on NN Searching

Recall the problem setup from previous lectures. We are given a set $S = \{\alpha, \cdots, n\} \subset \mathbb{R}^d$ of points in some high-dimensional space $\mathbb{R}^d$, equipped with some metric (think of distance function) $\rho : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}_{\geqslant 0}$. Given any point $x \in \mathbb{R}^d$, the nearest neighbor of $x$ with respect to $S$ is the $p_i \in S$ whose distance to $x$ is smallest, i.e.,

$$N(s) \coloneqq \operatorname*{argmin}_{p \in S} \rho(x, p).$$

Note the nearest neighbor may not be unique — if this is the case, any of the $p \in S$ that achieves this minimum will do. The previous lecture talked about **approximate nearest neighbor** (ANN), in which one wants to find a point $q \in S$ such that

$$\rho(x, q) \leqslant (1 + \epsilon)\rho(x, N(x));$$

in other words a potentially slightly sub-optimal candidate for being near $x$. We showed that using **quad trees** where one iteratively partitions the space into hypercubes of half the side length, an $(1 + \epsilon)$-ANN can attain a query time of $\mathcal{O}(\log n + 1/\epsilon^d)$ using a data structure whose size is only $\mathcal{O}(n)$.
However, this method quickly degrades in high dimensions due to the *curse of dimensionality* (which by itself is a very interesting phenomenon that's worth looking more into). This motivates hashing-based methods which provide an alternate approach to reduce this adverse effect of high dimension. To this end, we consider the following decision problem instead:

- Does there exist a point $p \in S$ that is within distance $r$ from our query point $q$?
- Equivalently, consider the ball $B(q, r) = \{x \in \mathbb{R}^d : \rho(x, q) \leqslant r\}$. Are $B(q, r)$ and $S$ disjoint? By combining this data structure with binary search, we can answer an NN query.

Because exact NN searching can be expensive in high dimensions, in our lecture, we focus on the latter, the approximate versions of this problem. This give rise to the **approximate ball emptiness** (ABE) query, in which we are given a query $q$, a radius $r$, an approximation factor $\epsilon$, and we wish to determine the output of the ANN query up to some slackness (think room of error) allowed by $\epsilon$. The requirements of our algorithms are

- If $B(q, r) \cap S \neq \varnothing$, always return YES — our algorithm confidently detects points in $S$ that are *very* close to $q$;
- If $B(q, (1 + \epsilon)r) \cap S = \varnothing$, always return NO — our algorithm confidently claims $q$ is quite isolated; and
- Otherwise, allow either answer — this is a "gray zone" where borderline cases may appear, but this is allowed by our $\epsilon$-slackness.

### Using Hashing for ABE Queries

To perform hashing-based ABE queries, we first fix a family $\mathcal{H} = \{h_1, \cdots, h_r\}$ hash functions, where each maps points in $\mathbb{R}^d$ to discrete buckets. We choose $h \sim \mathcal{H}$ randomly. The intuition is that *if $x$ and $y$ are close in $\mathbb{R}^d$, then they should collide (i.e. $h(x) = h(y)$) with relatively high probability*. Such family $\mathcal{H}$ leads to a **Monte Carlo** method for deciding whether a query point $q$ is near some point in $S$ — if there exists a close point $p \in S$ then they ($p$ and $q$) likely share some hash buckets, whereas if all points in $S$ are $(1+\epsilon)r$ away, collisions become unlikely, and we have more confident in saying "NO" in this case.

To boost the probability of correctness, we will maintain multiple independent hash tables. Specifically, we randomly select $s$ hash functions from $\mathcal{H}$ and, with abuse of notation, call them $h_1, \cdots, h_s$. Correspondingly, we maintain hash tables $T_1, \cdots, T_s$. The insert and deletion process follow naturally — nothing fancy here.

```
1 Function Insert(q):
2     for $i = 1, \cdots, s$ do
3         Insert($q$, $T_i[h_i(q)]$);
4 Function Query(q):
5     for $i = 1, \cdots, s$ do
6         for all $p \in T_i[h_i(q)]$ do
7             if $\rho(q,p) \leqslant (1+\epsilon)r$: return $p$
```

## Locality-Sensitive Hashing

Recall we want $\mathcal{H}$ to have the property where it tends to map close elements to the same bucket. Formally, $\mathcal{H}$ is called **$(r, R, \alpha, \beta)$-sensitive** if

(i) For all inputs $x, y$, $\mathbb{P}_{h\sim\mathcal{H}}[h(x) = h(y) \mid \rho(x,y) \leqslant r] \geqslant \alpha$, and

(ii) For all inputs $x, y$, $\mathbb{P}_{h\sim\mathcal{H}}[h(x) = h(y) \mid \rho(x,y) \geqslant R] \leqslant \beta$.

By convention we assume $r < R$. Property (i) controls the probability of correctness of our approximate query: the probability of obtaining a correct answer is $\geqslant \alpha$. Property (ii) controls the expected time of the query procedure: the expected query time is $\mathcal{O}(\beta n)$.

Let us consider a simple example: **Hamming distance**. Let $S \subset \{0,1\}^d$ be a collection of binary strings of length $d$. The Hamming distance $\rho$ between two binary strings is the number of bits that differ. Let $\mathcal{H} = \{h_1, \cdots, h_d\}$ where $h_i(x) : S \to \{0,1\}$ maps $x$ to the $i^{\text{th}}$ bit of $x$.
First suppose $\rho(x,y) \leqslant r$. That is, $x$ and $y$ differ by no more than $r$ bits. How do we calculate $\mathbb{P}[h_i(x) = h_i(y) \mid \rho(x,y) \leqslant r]$? Well, $h_i(x) = h_i(y)$ if and only if the $i^{\text{th}}$ bits of $x$ and $y$ agree. Given that no more than $r$ out of $d$ bits differ, if we randomly draw $h_i$ from $\mathcal{H}$, this event happens with probability $\geqslant 1 - r/d$:

$$\mathbb{P}_{h_i \in \mathcal{H}}[h_i(x) = h_i(y) \mid \rho(x,y) \leqslant r] \geqslant 1 - \frac{r}{d}.$$

Likewise, we now assume $\rho(x,y) \geqslant (1+\epsilon)r$. This means at least $(1+\epsilon)r$ bits out of all $d$ bits differ. Then, the probability that we pick an index at which $x$ and $y$ agree is at most $1 - (1+\epsilon)r/d$, so

$$\mathbb{P}_{h_i\sim\mathcal{H}}[h_i(x) = h_i(y) \mid \rho(x,y) \geqslant (1+\epsilon)r] \leqslant 1 - (\frac{1+\epsilon)r}{d}.$$

Combining the two inequalities, we see $\mathcal{H}$ is $(r, (1+\epsilon)r, 1 - r/d, 1 - (1+\epsilon)r/d)$-sensitive.

## LSH with Boosting

Ideally, we want the $\alpha, \beta$ values to be far apart, as this indicates that the distance thresholds $r$ and $(1+\epsilon)r$ significantly alter the behavior of the hash functions. Looking at our Hamming distance hashing, it seems like $\alpha = 1 - r/d$ and $\beta = 1-(1+\epsilon)r/d$ are, however, not so different. This is somewhat unsurprising, for after all we are only choosing one random bit for hashing, and certainly, one single bit does not disclose too much information about the rest of the string. To boost the gap, we upgrade our hash functions to independently take $k$ random bits at a same time and make two strings $x, y$ collide if and only if all $k$ chosen bits match between $x, y$. Formally, we consider

$$\mathcal{G} = \mathcal{H}^k = \{(h_1, \cdots, h_k) \mid h_1, \cdots, h_k \in \mathcal{H}\}.$$

It follows that

$$\mathbb{P}_{g\in\mathcal{G}}[g(x) = g(y)] = \mathbb{P}_{g\in\mathcal{G}}[h_i(x) = h_i(y) \text{ for all } 1 \leqslant i \leqslant k] = \prod_{i=1}^{k} \mathbb{P}[h_i(x) = h_i(y)].$$

The rest of the analysis will then show that $\mathcal{G}$ is $(r, (1+\epsilon)r, \alpha^k, \beta^k)$-sensitive, given that our original $\mathcal{H}$ is $(r, (1+\epsilon)r, \alpha, \beta)$-sensitive. Now our "probability gap" increases from $\alpha/\beta$ to $(\alpha/\beta)^k$, great.
One caveat: the success probability has decreased from $\alpha$ to $\alpha^k$. To fix this, we make maintain multiple hash tables and search in all of them, and see if a collision happens in at least one table.

$$\begin{aligned}\mathbb{P}[y \text{ collides with } x \text{ in at least one table } T_i \mid \rho(x,y) \leqslant r] &= 1 - \mathbb{P}(\text{no collision in any table} \mid \rho(x,y) \leqslant r) \\ &= 1 - (1-\alpha^k)^s \sim 1 - \exp(-\alpha^k \cdot s).\end{aligned}$$

As the number of tables $s$ increases, this quantity quickly approaches $1$, so everything is fine.

A quick **recap of how to deal with LSH**:

- First find *some* parameters $\alpha, \beta$ under which $\mathcal{H}$ is $(r, (1+\epsilon)r, \alpha, \beta)$-sensitive.
- To boost the ratio $\alpha/\beta$, consider concatenating $k$ hash functions into one: this new hashing scheme is $(r, (1+\epsilon)r, \alpha^k, \beta^k)$-sensitive.
- To ensure high success probability, maintain multiple hash tables.

As for query runtime complexity:

- The expected number of items $y$ with $\rho(x,y) \geqslant (1+\epsilon)r$ in one cell is bounded by $\beta^k \cdot n$.
- The time to compute each $g(y)$ takes $k$ steps, one for each individual hash function $h_i$.
- There are $s$ tables, and for each of them we need to perform the two steps above.

Therefore, the overall query time takes $s(k + \beta^k \cdot n)$.
A common goal is to make the success probability $\geqslant 3/4$. Using the approximation $1 - \exp(-\alpha^k \cdot s)$, we choose $s = 4/\alpha^k$. Since the query time is $s(k + \beta^k \cdot n)$, a natural choice is to make $k = \beta^k \cdot n$: this gives $k = \mathcal{O}(\log_{1/\beta}(n))$.
With some arithmetic, we can arrive at the following collection of performance guarantees (you don't need to memorize these for the midterm):

By choosing $s = 4/\alpha^k$ and $k = \mathcal{O}(\log_{1/\beta}(n))$ and defining $\phi = 1/(1+\epsilon)$, we obtain a (boosted) Hamming distance LSH scheme with high probability of success, whose query time is $sk = n^{\phi}\log n = n^{1/(1+\epsilon)}\log n$, requires space $ns = n^{1+\phi} = n^{1+1/(1+\epsilon)}$, and preprocessing time $nks = n^{1+\phi}\log n = n^{1+1/(1+\epsilon)}\log n$.

## LSH on $\ell^1$ Metric

Given $x, y \in \mathbb{R}^d$, the $\ell^1$ distance between them is defined as $\rho(x,y) = \sum_{i=1}^{d}|x_i - y_i|$. We first consider the 1-dimensional case, in which the $\ell^1$ metric reduces to simply absolute difference $|x-y|$.

Fix a positive integer parameter $\Delta$. For simplicity we consider only integers. Following the intuition of locality-sensitive hashing, we group integers into consecutive chunks of size $\Delta$ and map each chunk to a distinct value. Formally, we partition $\mathbb{Z}_{\geqslant 0}$ into buckets via the following functions:

$$h_a(x) := \left\lfloor \frac{x+a}{\Delta} \right\rfloor \qquad \text{for } a \in [\Delta] = \{0, \cdots, \Delta - 1\}.$$

Then $\mathcal{H} = \{h_a \mid a \in [\Delta]\}$ forms a family of hash functions. Essentially, these hash functions $h_a$'s are "translated" copies of each other, each with a different "offset." To calculate $\mathbb{P}[h_a(x) \neq h_a(y) \mid \rho(x,y) \leqslant r]$, we see that there must be a bucket boundary that lies in between $x$ and $y$ so that $h_a(x), h_a(y)$ map to different buckets. Now, for any interval of length $\Delta$, each of the $\Delta$ integers are equally likely to be a bucket boundary of a randomly chosen $h_a \in \mathcal{H}$. If we consider any such interval containing $[x,y]$, we see that with probability $|x-y|/\Delta \leqslant r/\Delta$, this randomly chosen hash function has bucket boundaries that separate $x$ from $y$. That is, $\mathbb{P}[h_a(x) \neq h_a(y) \mid \rho(x,y) \leqslant r] \leqslant r/\Delta$, and so

$$\mathbb{P}_{h_a \sim \mathcal{H}}[h_a(x) = h_a(y) \mid \rho(x,y) \leqslant r] \geqslant 1 - r/\Delta.$$

Likewise,

$$\mathbb{P}_{h_a \sim \mathcal{H}}[h_a(x) = h_a(y) \mid \rho(x,y) \geqslant (1+\epsilon)r] \leqslant 1 - \frac{(1+\epsilon)r}{\Delta}.$$

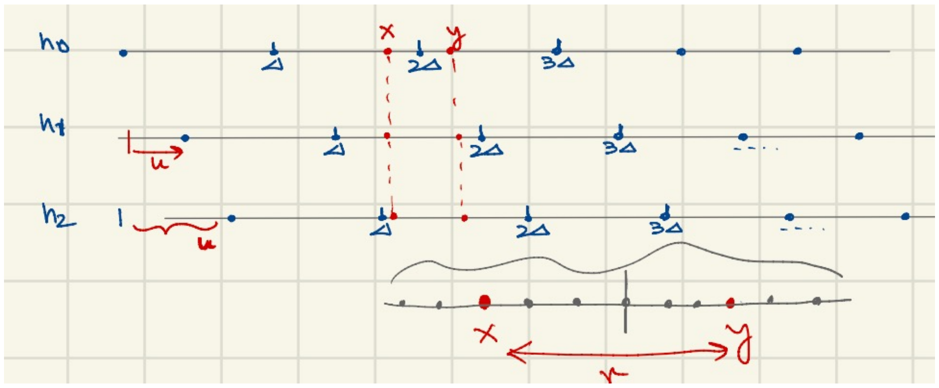

Figure 1: The blue ticks are bucket boundaries of various hash functions. $h(x) \neq h(y)$ iff there exists one such tick between $x$ and $y$. For each hash function, the ticks are all integers that share a certain common remainder when divided by $\Delta$.

More generally, in $\mathbb{R}^1$ we may consider **randomly shifted** intervals, where each chunk is now an interval of length $\Delta$ (instead of $\Delta$ consecutive integers). The only difference is that $\mathcal{H}$ now contains a continuum of hash functions instead of discrete ones for each remainder mod $\Delta$.
Instead of requiring $x, y \in \mathbb{Z}$ and $a \in \{1, \cdots, \Delta\}$, we now allow $x, y$ to be any real number, and $a \in [0, \Delta)$ any real number as well. Thus, we have an infinite (in fact uncountable) collection $\mathcal{H} = \{h_a : 0 \leqslant a < \Delta\}$, where

$$h_a(x) = \left\lfloor \frac{x+a}{\Delta} \right\rfloor \tag{*}$$

but in a more general sense, allowing remainder for fractional values. For example, if $x = 99.75$, $a = 3$, and $\Delta = 10$, then $h_a(x) = \lfloor (99.75 + 3)/10 \rfloor = 10$. Another way to characterize these hash functions is that they *map contiguous chunks of length $\Delta$ into the same bucket.*
The rest of the analysis stays the same — if $h(x) \neq h(y)$ then a bucket boundary of $h$ must be between $x$ and $y$. Now, in a continuum of $[0, \Delta)$, only a certain segment of total length $|y - x|$ can make this come true. The rest of the argument follows.

Finally, we may consider higher-dimensional analogues: **randomly shifted grids** in $\mathbb{R}^d$. Each "unit" is called a **cell**. In 1-D these are the intervals with length $\Delta$ as described above; in 2-D they would be squares with side length $\Delta$; in 3-D, cubes, and so on. We still want to create hash functions that *map elements within the same cell to the same bucket*. This is not hard. Let $a = (a_1, \cdots, a_d) \in [0, \Delta)^d$ be the random offset (think of this as the remainder in the 1-D case). We can define a 1-D function $h_{a,i}$ as in (*) for each component $1 \leqslant i \leqslant d$, and let $h_a$ be the concatenation of $(h_{a,1}, \cdots, h_{a,d})$. Formally, $h_a : \mathbb{R}^d \to [0, \Delta)^d$ can be expressed as

$$h_a(x_1, \cdots, x_d) = (h_{a,1}(x_1), \cdots, h_{a,d}(x_d)) \qquad \text{where } h_{a,i}(x_i) = (x_i + a_i) \pmod{\Delta}.$$

It follows that in order for two $d$-dimensional vectors to be hashed into the same bucket, all $d$ component-wise hash functions $h_{a,i}$ must agree. The rest of the calculation is analogous, except with an extra power of $d$ involved.