In this lecture we will study the Nearest Neighbor Search problem. The input to the problem is a set of points. Given a query point, the goal is to find the point in the set that is closest to the query point. Naively, this requires $O(n)$ comparisons. In the setting where the number of points is really large, it is not computationally feasible to use the brute force algorithm. Therefore, our goal is to preprocess the input and create a data structure that can quickly answer queries.

The Nearest Neighbor Search problem has a a rich history and numerous applications [1]. It is an algorithmic primitive for finding all similar pairs, clustering problems on large datasets, closest pair problems in computational geometry, recommendation systems, spell checkers and more.

1 Nearest Neighbor Search

Problem 1. (Nearest Neighbor Search). Given as input points $\mathcal{P} = \{p_1, p_2, \dots p_n\}$ such that $p_i \in \mathbb{R}^d$, and a query point $q \in \mathbb{R}^d$, find point

$$p^* = \text{argmin}_{p_i} d(p_i, q)$$

i.e. the closest point to q in the set $\mathcal{P}$.

Trying to solve the Nearest Neighbor Search problem exactly for high dimensional data is a challenging task and known algorithms have an exponential dependence on dimension! These methods are not scalable in terms of dimension, a negative result which is sometimes called the "curse of dimensionality". Therefore, we relax our goal to approximate nearest neighbors.

In the Approximate Nearest Neighbor Search problem we instead find a point $p_i \in \mathcal{P}$ such that $d(p_i, q) \leq cd(p^*, q)$, where $c > 1$ is some constant. Formally,

Problem 2. (c-Approximate r-Near Neighbor). Given points $\mathcal{P} = \{p_1, p_2, \dots p_n\}$ and a query point $q \in \mathbb{R}^d$ such that

$$\min_{p_i \in \mathcal{P}} d(p_i, q) \leq r$$

output any point $p_j \in \mathcal{P}$ such that

$$d(p_j, q) \leq cr$$

i.e. if there exists a point p_j within distance r of the query point q, output any point p_i that is within distance cr of q.

At a high level, we want to design a function $W : \mathbb{R}^d \to \{0,1\}^k$ such that given $W(p_1)$ and $W(p_2)$ we can distinguish between p_1 and p_2 being close, i.e. $d(p_1, p_2) \leq r$, or p_1 and p_2 being far, i.e. $d(p_1, p_2) > cr$. Additionally, we want k to be small in order to store the data structure in small space. For concreteness, let $c = 1 + \epsilon$ and $k = O\left(\frac{\log(n)}{\epsilon^2}\right)$ for some $\epsilon > 0$.

[1] See more at `https://en.wikipedia.org/wiki/Nearest_neighbor_search`

Assuming we have access to a W that satisfies the above properties, we can construct a data structure for the c-Approximate r-Near Neighbor problem. Note, we can think of W as a $n \times k$ matrix. The first strategy we consider is a linear scan.

Linear Scan.

1. For all $p_i \in \mathcal{P}$, precompute $W(p_i)$.
2. Given a query point q, compute $W(q)$.
3. For all $p_i \in \mathcal{P}$, compare $W(p_i)$ and $W(q)$.

We observe that the space complexity and query time of the above algorithm is $O\left(\frac{n\log(n)}{\epsilon^2}\right)$, which is nearly linear in the input size. While we are okay with nearly linear space, we would want faster query time. This brings us to our next strategy called exhaustive storage.

Exhaustive Storage.

1. For each bit string $\sigma \in \{0,1\}^k$, construct a table A such that

$$A[\sigma] = \{p_i \in \mathcal{P} \mid d(W(p_i), \sigma) \leq (1+\epsilon)r\}$$

   i.e. $A[\sigma]$ stores the set of point p_i that are close to σ after applying W to p_i.

2. On query q, output any point in the set $A[W(q)]$.

First, we observe that the query time for the above algorithm is $O\left(\frac{d\log(n)}{\epsilon^2}\right)$, which is the time to compute $W(q)$. We now have a query time that is independent of n! Next, we observe that our space complexity is $O(2^k) = n^{O(1/\epsilon^2)}$ which is significantly larger than the Linear Scan approach. We show how to obtain the best of both worlds, i.e. near-linear space complexity and sub-linear query time.

2 Locality Sensitive Hashing

Next, we introduce an important algorithmic primitive in the design of Nearest Neighbor algorithms called Locality Sensitive Hashing.

Definition 1 *(informal.) A* locality sensitive hash function *is a random hash function $h : \mathbb{R}^d \to \{0,1\}^k$ (h drawn from a family $\mathcal{H}$) such that*

1. *If $d(q,p) \leq r$, then $\Pr[h(q) = h(p)] = P_1$ is "not-so-small", i.e. if p close to q, $h(q)$ and $h(p)$ collide with higher probability.*
2. *If $d(q,p) > cr$ then $\Pr[h(q) = h(p)] = P_2$ is "small", i.e. if q is far from p, $h(q)$ and $h(p)$ collide with lower probability.*

We will specify later what "small" and "not-so-small" actually mean. In general, $P_1 < P_2$ and we associate the following parameter with $\mathcal{H}$ to characterize this gap:

$$\rho = \frac{\log(1/P_1)}{\log(1/P_2)}.$$

If we had a locality sensitive hash function such that P_1 was "large," then we could simply compute the hash table of $\mathcal{P}$, A. Then on query q, simply compute $A[h(q)]$. Unfortunately, it is not possible to have P_1 high and P_2 low. Instead of using just one hash function for the entire input, we use $L = n^\rho$ hash functions for independent $h_1, \ldots, h_L \in \mathcal{H}$. (We will justify this choice of L later.) Note, since $\rho = \frac{\log 1/P_1}{\log 1/P_2} < 1$, the number of hash functions used is $n^\rho < n$.

2.1 LSH for Hamming Space

Next, we construct a LSH for Hamming Space, i.e. we consider points in $\{0,1\}^d$ with equipped with the distance metric $\text{Ham}(x,y) = |\{x_i \neq y_i\}|$ where $x, y \in \{0,1\}^d$ and x_i denotes the i-th coordinate of x. In other words, the Hamming Distance between two binary vectors x and y is given by the number of coordinates on which they differ. We then construct our hash family, $\{g : \{0,1\}^d \to \{0,1\}^k\}$, as follows. For $p \in \{0,1\}^d$, let

$$g(p) := (h_1(p), h_2(p), \ldots, h_k(p)),$$

where

$$h_i(p) := p_j \text{ for a random } j \in [d]$$

Next, we observe that for a given LSH, g, the probability of a collision is

$$\Pr[g(p) = g(q)] = \prod_{i=1}^{k} \Pr[h_i(p) = h_i(q)].$$

Next, we show that the parameter ρ for the hash functions g and h is identical.

Fact 2 $\rho_g = \rho_h$

Proof: Recall, if $d(q,p) \leq r$, $\Pr[h(q) = h(p)] = P_{1,h}$. If $d(q,p) > cr$, $\Pr[h(q) = h(p)] = P_{2,h}$. Similarly, if $d(q,p) \leq r$, $\Pr[g(q) = g(p)] = P_{1,g}$. If $d(q,p) > cr$, $\Pr[g(q) = g(p)] = P_{2,g}$. Observe,

$$\Pr[g(p) = g(q)] = \prod_{i=1}^{k} \Pr[h_i(p) = h_i(q)] \implies \begin{cases} P_{1,g} = P_{1,h}^k \\ P_{2,g} = P_{2,h}^k \end{cases}$$

since g consists of k independent copies of h. Then,

$$\rho_g = \frac{\log 1/P_{1,g}}{\log 1/P_{2,g}} = \frac{\log 1/P_{1,h}^k}{\log 1/P_{2,h}^k} = \frac{k \log 1/P_{1,h}}{k \log 1/P_{2,h}} = \rho_h$$

■

We then approximate ρ_g.

Claim 3 $\rho \approx \frac{1}{c}$

Proof: We first observe that for $p, q \in \{0,1\}^k$,

$$\forall i,\ \Pr[h_i(p) = h_i(q)] = 1 - \frac{\text{Ham}(p,q)}{d}.$$

For simplicity we assume $r \ll d$. This assumption is justified since we can always embed in a higher dimension, and the analysis goes through without the following approximation.

Consider the case where $\text{Ham}(p,q) \leq r$. Then,

$$\forall i,\ \Pr[h_i(p) = h_i(q)] = 1 - \frac{\text{Ham}(p,q)}{d} \geq 1 - \frac{r}{d} = P_{1,h}$$

Similarly, consider the case where $\text{Ham}(p,q) > cr$. We have,

$$\forall i,\ \Pr[h_i(p) = h_i(q)] = 1 - \frac{\text{Ham}(p,q)}{d} \leq 1 - \frac{cr}{d} = P_{2,h}$$

Using the taylor series approximation for $e^x = 1 + x + \frac{x^2}{2!} + \cdots$, we get the following approximation, up to an additive error of $O((cr/d)^2)$:

$$P_{1,h} = 1 - \frac{r}{d} \approx e^{-r/d}$$
$$P_{2,h} = 1 - \frac{cr}{d} \approx e^{-cr/d}$$

This implies

$$\rho_g = \frac{\log 1/P_{1,h}}{\log 1/P_{2,h}} \approx \frac{r/d}{cr/d} = \frac{1}{c}.$$

■

2.2 LSH for Nearest Neighbor Search

We now present an algorithm for Nearest Neighbor Search in Hamming Space via the above LSH construction. We will use the technique outlined earlier.

Data Structure: Given data points $\mathcal{P}$

1. Allocate L hash tables, $A_1, \ldots, A_L$ each with a fresh Hamming LSH g_i for $i \in [l]$.
2. Hash all points in $\mathcal{P}$ into tables $A_1, \ldots, A_L$.
3. Note, each hash table to have size n.

Query:

Given query q,

1. Compute $g_1(q), \ldots, g_L(q)$.
2. Check each table, $A_1[g_1(q)], \ldots, A_L[g_L(q)]$, for collisions.

3. For each collision $p \in \mathcal{P}$ under g_i, check if $d(p,q) \leq cr$. If so, output p. If none found, FAIL.

We now describe the setting of the parameters k and L above. Recall, for each hash table, we have a success probability of $P_{1,g} = P_{1,h}^k$. Since we have L tables, and we are hoping for a success probability of at least 0.5, we set $L = O(1/P_{1,h}^k)$. Also recall that the probability that we have a collision when p, q are far is $P_{2,g} = P_{2,h}^k$. We pick k such that $P_{2,h}^k = \frac{1}{n}$ (for reasons described below). Therefore, $k = \frac{\log(n)}{\log(1/P_{2,h})}$.

Observe, for the above choices of L and k, we have

$$\begin{aligned} P_{1,g} &= \Pr[g(p) = g(q) \mid d(p,q) \leq r] \\ &\geq P_{1,h}^k \\ &= P_{1,h}^{\frac{\log(n)}{\log(1/P_{2,h})}} \\ &= n^{\frac{-\log(1/P_{1,h})}{\log(1/P_{2,h})}} \\ &= n^{-\rho} \end{aligned} \tag{1}$$

and

$$\begin{aligned} P_{2,g} &= \Pr[g(p) = g(q) \mid d(p,q) \geq cr] \\ &\leq P_{2,h}^k \\ &= \frac{1}{n} \end{aligned} \tag{2}$$

Analysis.

Since we store L tables and for each table we hash the entire input set $\mathcal{P}$, we use a total space of $O(nL) = O(n^{1+\rho})$. Observe, we compute $g_i(q) \in \{0,1\}^k$, which requires $O(k)$ time. Repeating this for all $i \in [L]$ contributes $O(Lk)$ time. Next, we need to check for false positives. Each false positive requires $O(d)$ time to compute. Recall, the probability that two far points collide is $P_{2,g} = \frac{1}{n}$. In expectation, $n \cdot \frac{1}{n} = 1$ points collide in each of the table and we have to check $O(L)$ points for false positives, contributing a running time of $O(Ld)$. Therefore, we have a total query time of $O(Ld + Lk) = O(n^\rho d)$, where the last bound follows from $k \leq d$.

Finally, to argue correctness, we show that if there exists a point p^* such that $d(q,p^*) \leq r$, then the probability the above algorithm fails is bounded by at most $1/2$. Observe, the algorithm fails if p^* is not in $\{A_1[g_1(q)], A_2[g_2(q)], \ldots A_L[g_L(q)]\}$. Therefore,

$$\begin{aligned} \Pr\left[p^* \notin \{A_1[g_1(q)], A_2[g_2(q)], \ldots A_L[g_L(q)]\}\right] &= \prod_{i=1}^{L} \Pr[h_i(p^*) \neq h_i(q)] \\ &\leq \left(1 - \frac{1}{n^\rho}\right)^L \\ &= \left(1 - \frac{1}{n^\rho}\right)^{n^\rho} \\ &\leq \frac{1}{e} \end{aligned}$$

This completes our analysis.