COMS 4995-2: Advanced Algorithms (Spring'20) April 23, 2020

Lecture 24: Nearest Neighbor Search (high-dimensional)

Instructor: *Alex Andoni* Scribes: *Alper Cakan, Alexandre Lamy*

1 Nearest Neighbor Search Overview

In this lecture we study the Nearest Neighbor Search (NNS) problem. Formally the problem is defined as follows:

Preprocess: a set P of points.
Query: given a query point q, find $p^* \in P$ with the smallest distance to q.

This problem has a lot of practical importance. It is a primitive for calculating all similar pairs, as well as for various clustering problems. In practice, NNS is used for finding pairs of similar images in a large image set, for signal processing, for bioinformatics, etc.

1.1 Easy versions of NNS

Exact match
If instead of looking for the nearest neighbor, we simply want to look for an exact match, then the problem reduces to the Dictionary problem (which we studied at the beginning of the class). Any solution to the Dictionary problem (such as a hash table) would then work. Thus we could get $O(n)$ preprocessing time and space, and $O(1)$ query time.

1D case
In the 1D case, i.e. when the dataset P is a subset of $\mathbb{R}$, the NNS problem can be solved by a simple sort and binary search. Specifically we can sort P in the preprocessing step and then do a binary search for q during queries. This gives a $O(n)$ space and $O(\log n)$ query time algorithm.

1.2 High Dimensional Exact NNS

Now let us consider the case where $P \subseteq \mathbb{R}^d$ (one should be thinking of both a large n and d, something like $n = 1,000,000,000$ and $d = 400$). We start by showing two naive solutions to the NNS problem. Ideally we would like constant or logarithmic query time with small space requirement.

No indexing
The first solution is two just store the raw unprocessed data and then, for each query, calculate the distances to each of the n points and return the point with the smallest distance. This solution use only $O(nd)$ space (which is optimal) but also has $O(nd)$ query time, which is much too high. The issue here is that, in a sense, the data is "underprepared".

Full indexing

The second naive solution is to store the answers to all possible queries (i.e. for every point $q \in \mathbb{R}^d$ store the closest point in P). Afterwards, we can answer any query in $O(d)$ time by doing a single lookup. However, this requires 2^d space which is untractable for large d (say $d >> \log n$). In contrast to the previous solution, here the data is "overprepared".

Other solutions
With the above results in mind, we have to wonder whether it is possible to do better than the first (no indexing) solution while keeping the space requirement reasonable. For example, we can ask whether there exists a solution using $O(n^2)$ space and $O(n^{.99})$ time (or really $O(n^{1-\epsilon})$ for any ϵ). Unfortunately, [Williams'04] showed that if this was possible (for large d), it would refute a strong version of $P \neq NP$ conjecture.

2 Approximate Near Neighbor Search

Given the difficulty of the exact NNS problem, we try to look at a relaxed version instead. Formally, the Approximate Near Neighbor Search (ANNS) problem is as follows (for parameters r and c):

Preprocess: a set P of points.
Query: given a query point q, if there exists $p \in P$ such that the distance between p and q is smaller than r, then the algorithm should return a point $p' \in P$ such that the distance between p' and q is smaller than cr.

Intuitively, there are two relaxations that differentiate ANNS from NNS. First, the problem asks for a near neighbor (as characterized by r) rather than the nearest neighbor. This difference is actually relatively unimportant because it is possible to use a near neighbor search algorithm to solve the nearest neighbor search problem. The second difference is that we allow a gap (as characterized by c) so that points at a distance between r and cr from q can be viewed either as similar or as dissimilar. This is the "approximate" part of the new problem. Note that the problem should get easier as c gets larger, so we expect running time and space cost to be inversely related to c.

3 Locality Sensitive Hashing

One approach to ANNS is to use *locality sensitive hashing*. Ideally, we want *similar* points to hash to the same value. More formally, a map g on R^d such that for any q, p, p'

- $\|q - p\| \leq r \implies g(q) = g(p)$
- $\|q - p'\| > cr \implies g(q) \neq g(p')$

However, deterministic version is too restrictive. Instead consider the randomized version. For any q, p, p'

- $\|q - p\| \leq r \implies P_1 = Pr[g(q) = g(p)]$ is not too low
- $\|q - p'\| > cr \implies P_2 = Pr[g(q) = g(p')]$ is low

Note that the probabilities are over random choice of g from a family, not over the points q, p, p'.

One way to construct such g is using several hash tables. In fact, use n^ρ hash tables where $\rho = \frac{\log(1/P_1)}{\log(1/P_2)}$.

Hash function g is usually a concatenation of *primitive* functions. That is, $g(p) = \langle h_1(p), h_2(p), \ldots, h_k(p)\rangle$ For example, consider the Hamming metric over $\{0,1\}^d$ and define $h(p) = p_j$ where j is random (*picked* before seeing the data). Then, $g(p)$ is just a projection on k random bits and

$$\begin{aligned} Pr[h(p) = h(q)] &= 1 - \frac{Ham(p,q)}{d} \\ P_1 &= 1 - \frac{r}{d} \approx e^{-r/d} \\ P_2 &= 1 - \frac{cr}{d} \approx e^{-cr/d} \end{aligned}$$

Hence,

$$\rho = \frac{\log(1/P_1)}{\log(1/P_2)} = \frac{r/d}{cr/d} = \frac{1}{c}$$

Remember that we are actually interested in ρ of g, but since the h that comprise g are independent, we just get a factor of k in both nominator and denominator of ρ of g, and they cancel out.

Now let's construct the actual algorithm that uses g.
Data structure Just $L = n^\rho$ hash tables. Each table uses a fresh random function and we hash all dataset points to the table.

$$g_i(p) = \langle h_{i,1}(p), h_{i,2}(p), \ldots, h_{i,k}(p)\rangle \text{ for } i \in [L]$$

Query Check for collisions in each table until we see a point within distance cr
Guarantees Space is $O(n^{1+\rho})$ since we have n^ρ hash tables. Query time is $O(n^p \cdot (k+d)) = O(n^p \cdot d)$. Finally, the probability of success is 50%.

Let's choose the parameters L and k. Remember that P_2^k is the probability of collision of *far* pairs and P_1^k is the probability of collision of *close* pairs. Therefore, set $k = \frac{1}{n}$ so that we get $\geq nP_2^{1/n} = 1$ false positives in expectation. Then, by definition $P_1^k = (P_2^\rho)^k = \frac{1}{n^\rho} = \frac{1}{L}$. Hence, L repetitions suffice.
Now that we have set the parameters, let's move onto correctness. First of all, remember that we don't care about the case when there is no r-near neighbor. So, let p^* be an r-near neighbor. The algorithm fails when p^* is not in any of the tables $g_i(q)$. So,

$$Pr[\text{algorithm fails}] = (1 - P_1^k)^L = (1 - \frac{1}{n^\rho})^{(n^\rho)} \leq \frac{1}{e}$$

Finally, the runtime. It is dominated by hash function evaluations ($O(L \cdot k)$) and distance computations to the points in buckets. For the distance computations, we only need to worry about far points (distance $> cr$). In a single hash table, we have that the probability a far point collides $\geq P_2^k = \frac{1}{n}$. Hence, the expected number of far points in a single bucket is $n\frac{1}{n} = 1$. Going up to L hash tables, we get L far points in expectation. Summing everything up, we get $O(Lk) + O(Ld) = O(n^\rho d)$ in expectation.

[Datar-Indyk-Immorlica-Mirrokni'04] and [A.-Indyk'06] shows that for the Euclidean space, we can do $n^{1+\rho}$ space and n^{ρ} time with $\rho \approx \frac{1}{c^2}$. Getting something better than this requires data-dependence ([Motwani-Naor-Panigrahy'06, O'Donell-Wu-Zhou'11] for the data-independent lower bound, [A-Indyk-Nguyen-Razenshteyn'14], [A-Razenshteyn'15] for data-dependent construction).