Main topics are locality sensitive hashing.

Prelude

- This is the last real lecture of the semester. We’ll start presentations next week. I’m looking forward to seeing what you’ve been working on!

Approximate Nearest-Neighbor

- We’ll continue our discussion of higher dimensional inputs and arbitrary metrics by focusing on a specific problem defined for any metric.
- In the approximate nearest-neighbor problem, you’re given a metric \((X, d)\) and a subset \(S\) of \(X\) with \(n\) elements. You want to preprocess \(S\) to answer approximate nearest-neighbor queries denoted as \(\text{NN}(q, r, c)\).
  - If there is some \(x\) in \(S\) such that \(d(q, x) \leq r\), then report some \(y\) in \(S\) such that \(d(q, y) \leq cr\).
  - If there is no \(x\) in \(S\) such that \(d(q, x) \leq cr\), then report failure.
  - Otherwise, report some \(x\) in \(S\) such that \(d(q, x) \leq cr\) or fail. Either is fine.
- So this is similar to our approximate decision procedure for set cover and hitting set.
- When \(c = 1\), then it’s just a query for whether the nearest neighbor to \(q\) is within distance \(r\). Like set cover and hitting set, we can binary search for an approximately smallest value to find an approximately nearest neighbor to any query point \(q\).
- When \(r\) is large, then \(\text{NN}(q, r, c)\) is good for distinguishing between \(q\) being really far from the data set and \(q\) being close enough. It turns out this problem is much easier to solve than finding \(q\)’s nearest neighbor exactly.
- Now, if \(X = \mathbb{R}^d\) for some small constant \(d\), you can answer \(\text{NN}(q, r, (1 + \text{eps}))\) for any constant \(\text{eps} > 0\) in \(O(\log n)\) time using a somewhat involved data structure called a BBD-tree that takes only \(O(n \log n)\) space.
- But like many things we saw this semester, the hidden constants increase exponentially in \(d\).

Locality Sensitive Hashing

- Today, we’ll look at a different approach called locality sensitive hashing that works fine in high dimensions, although the guarantees aren’t as good.
- The main idea is to randomly choose from a large collection of hash functions specific to the metric you care about. If two elements are close to one another, then hopefully they hash to the same values.
- We call a probability distribution \(H\) over different hash functions a hash family.
Formally, given a parameter $c > 1$, probabilities $p_1 > p_2$, and a distance $r \geq 0$, a hash family $H$ is $(r, cr, p_1, p_2)$-Locality Sensitive (LSH) if for all $q$ in $X$, $x$, $y$ in $S$

- If $d(x, q) \leq r$, then $\Pr[h(x) = h(q)] \geq p_1$, and
- If $d(y, q) \geq cr$, then $\Pr[h(y) = h(q)] \leq p_2$.

So we hope that $p_2$ is much smaller than $p_1$.

This is pretty different from cryptographic hashing where you’d hope two items have completely different hash values if they differ at all, but naming is one of the two hardest problems in computer science.

**Hamming Distance**

- Given two $m$-dimensional bit vectors $x$ and $y$, their Hamming distance $d(x, y)$ is the number of positions at which they disagree.
- So 0010 and 0100 have Hamming distance 2.
- Let $H$ be the hash family where each $h$ in $H$ is assigned a different coordinate so that $h(x)$ is $x$'s value at that coordinate. Hash functions are chosen uniformly at random.
- We get $\Pr[h(x) = h(y)] = 1 - d(x, y) / m$, because there are $d(x, y)$ choices out of $m$ for the coordinate that give us a different value for $h(x)$ and $h(y)$.
- As we would hope, $x$ and $y$ are more likely to hash to the same value if their Hamming distance is small.
- So in this case, $H$ is $(r, cr, p_1, p_2)$-LSH for
  - $p_1 = 1 - r / m$ and
  - $p_2 = 1 - cr / m$.

**Jaccard Distance**

- Let $U$ be some universe of elements. Given two subsets $S_1$ and $S_2$ of $U$, the Jaccard similarity coefficient $J(S_1, S_2) = |S_1 \text{ intersect } S_2| / |S_1 \text{ union } S_2|$. This is not a metric.
- But the Jaccard distance $d(S_1, S_2) = 1 - J(S_1, S_2)$ is a metric.
- To do approximate nearest neighbors for the Jaccard distance, let each $h$ in $H$ be a different permutation $p_i$ of $H$. $h(S) =$ the first element $p_i$ in $S$. Again, we choose an $h$ uniformly at random.
- So, $\Pr[h(S_1) = h(S_2)] = J(S_1, S_2) = 1 - d(S_1, S_2)$.
- This $H$ is also LSH.
  - $p_1 = 1 - r$ and
  - $p_2 = 1 - cr$.

**Angular Distance**

- Given two vectors $x$ and $y$ in some $\mathbb{R}^m$, then angle between them is $d(x, y) = \cos^{-1}(\langle x$
dot y) / (||x|| ||y||).

- Now, for each h in H choose a different unit vector u. h(x) = sign(x dot u).
- In other words, h(x) = 1 if x makes an acute angle with u and h(x) = -1 if the angle is obtuse.
- Pr[h(x) = h(y)] = 1 - d(x, y) / pi.

- Again, H is LSH. For any r in [0, pi] and c > 1 with cr ≤ pi,
  - p_1 = 1 - r / pi and
  - p_2 = 1 - cr / pi.

### The LSH Algorithm

- So we have all these nice LSH hash families. How do we use them?
- Say H is (r, c, p_1, p_2)-LSH. We want to build a data structure for NN(q, r, c).
- Fix two parameters k and ell. We’ll figure out what they should be later.
- For each i, j with 1 ≤ i ≤ ell and 1 ≤ j ≤ k, pull h_{i j} independently from hash family (distribution) H. These will stay fixed for the life of our data structure.
- Now, for each x in our set of n elements S, and for each 1 ≤ i ≤ ell, store x in bucket g_i(x) = <h_{i 1}(x), h_{i 2}(x), …, h_{i k}(x)>. So that’s ell buckets for x, each bucket indexed by a k-dimensional hash function.
- The data structure will store just the buckets that actually contain some element x along with their elements.
- Now, for a query q, we compute g_1(q), g_2(q), …, g_ell(q). We look at each bucket in order, and check the elements of S within each bucket. When checking an element x, we return it if d(q, x) ≤ cr. We return failure if we run out of buckets or check more than 4 ell elements.
- So then the analysis depends upon the following: Suppose there is a point x^* in S such that d(q, x^*) ≤ r. Then with constant then probability,
  1. there are at most 4 ell elements in S with d(x, q) > cr such that for some i, g_i(x) = g_i(q).
  2. The algorithm will never check more than 4 ell elements. With constant probability, there will be something good to check, that element x^*. And the algorithm won’t give up too early doing bad checks for 4 ell elements that are inappropriate to return.
- At this point we need to pick values for k and ell that make the data structure useful.
- The running time of a query is O(ell k). You need to find those ell buckets, and each computed by evaluating k hash functions.
- Space usage is O(n ell), though, since you store each point of S in ell different buckets.
- So, we want to pick k and ell as small as possible so that queries have a constant probability of success.
- Let rho = ln(p_1) / ln(p_2) = log_{p_2} p_1. In each of the three cases, rho = 1/c.
For Hamming distance, \( \rho = \frac{\ln(p_1)}{\ln(p_2)} = \frac{r}{d} / \frac{cr}{d} = 1 / c \). The other cases are almost identical.

**Theorem:** Let \( \ell = n^\rho \) and \( k = \log n / \log(1 / p_2) = - \log_{(p_2)} n \). Properties 1 and 2 both hold with constant probability.

**Proof for 2.**
- Consider \( x' \) in \( S \) where \( d(x', q) > cr \). The LSH property implies \( g_i(x') = g_i(q) \leq p_2^k \) for all \( i \), because we’d need to agree with all \( k \) hash functions.
  - \( p_2^k = p_2^{-\log_{p_2} n} = 1/n \)
  - So for a fixed \( i \), the expected number of \( x' \) that map to the same bucket as \( q \) is \( 1/n \times n = 1 \).
  - And therefore, the expected total number of false positives is \( \ell \times 1 = \ell \).
  - I think I mentioned something called Markov’s inequality earlier in the semester. The probability that there are more than 4 \( \ell \) false positives is at most \( \ell / (4 \ell) = 1/4 \).

**Proof for 1.**
- \( \Pr[g_i(x^*) \neq g_i(q)] \)
  - \( \leq 1 - p_1^k \)
  - \( = 1 - p_1^{-\log_{p_2} n} \)
  - \( = 1 - n^{-\log_{p_2} p_1} \)
  - \( = 1 - 1 / n^\rho \)
  - But since we chose \( \ell = n^\rho \), \( \Pr[g_i(x^*) \neq g_i(q) \) for all \( i \)]
    - \( \leq (1 - 1 / n^\rho)^n \)
    - \( \leq 1/e. \)
- The probability that both hold is at least \( 1 - 1/e - 1/4 \geq 1/3 \).
- With those settings you get results like the following: If \( c = 2 \), then you get do queries that succeed with constant probability in about \( \sim O(\sqrt{n}) \) time each (ignoring logs) while using only \( O(n^{1.5}) \) space for the data structure. That’s less than linear time and less than quadratic space.
- If you want to boost the probability of success, just build data structures with their own set of hash functions. \( O(\log n) \) of them is enough to get really really go probability of success for each query.