CS 388R: Randomized Algorithms, Fall 2019 December 5th, 2019

Lecture 28: Locality Sensitive Hashing

Prof. Eric Price *Scribe: Joshua Cook*

NOTE: THESE NOTES HAVE NOT BEEN EDITED OR CHECKED FOR CORRECTNESS

1 Overview

In previous lectures we discussed hash functions and how they can be used in hash functions. We discussed how we can use n-wise independence to efficiently run hashing algorithms through specific efficiently computable and representable hash families.

In this lecture we discuss locality sensitive hashing for computing approximate nearest neighbors. Locality sensitive hashing is another way to use hashing. It uses hashing for high dimensional computational geometry. In particular, we use it solve an approximate version of the nearest neighbors problem.

1.1 Nearest neighbors problem

The nearest neighbors problem takes n points, and for another point tries to give the closest point in the n original points. Specifically, we want to construct a data structure to answer "nearest neighbor" queries fast, but approximately.

So let our set of points be $Y = \{p_1, ..., p_n\} \subseteq X$ (think of $X = \mathbb{R}^d, d >> 1$). We want to construct a data structure so that given a query to some point p in X, find i such that $\|p - p_i\|$ is minimized. But actually getting the min is tricky, so we relax to:

If $\min_i \|p - p_i\| = r$, we find j such that $\|p - p_j\| \leq cr$ for some approximation factor c.

This is the nearest neighbor problem, but locality sensitive hashing is best at solving 'near' neighbor queries.

1.2 "Near" neighbors

Near neighbor queries are like nearest neighbor, except we are told the radius r at the beginning. That is r such that $\min_i \|p - p_i\| \leq r$ find some $j \in [n]$ such that $\|p - p_j\| \leq cr$ for constant factor c. More generally, an r-near query with constant factor c will return a j such that $\|p - p_j\| \leq cr$ if there is some i so that $\min_i \|p - p_i\| \leq r$.

We can use r-near queries to solve nearest neighbor by checking with $r = 1, 2, 4, 8, ..., R$.

We only need to give output if there is a point within r, even if something is within cr. But it may always output any point within cr, whether there are points within r or not.

We only consider points that are binary strings $X = \{0,1\}^d$ with hamming distance metric. For other metrics there are slightly different approaches, but general methods apply for other spaces and other metrics.

2 Different Algorithm Runtimes

We are trying to optimize:

1. Time per query
2. Space to store data structure.

We are ignoring the initialization overhead, though it is usually very close to the space requirements. Then the run time of a few algorithms are:

Algorithm	Time	Space
Naive Exhaustive search	nd	nd
JL Exhaustive search	$d + \frac{n\log(n)}{\epsilon^2}$	$\frac{n\log(n)}{\epsilon^2}$
Precompute Answers	d	$2^d \log(n)$
JL Precompute	d	$n^{1/\epsilon^2}\log(n)$
LSH	dn^ρ	$n^{1+\rho}$
LSH (Hamming)	$d + n^{1/c}(k + dn^{1-c})$	$n^{1+1/c}$

Naive algorithm: Store everything in a list and search with exhaustive search. Takes nd space and nd lookup time. If d is very large, we can approximate using a smaller space with Johnson–Lindenstrauss (JL) dimensionality reduction. Then space becomes $n\frac{\log(n)}{\epsilon^2}$. Time per query is just d time for embedding plus $\frac{n}{\epsilon^2}\log(n)$ for comparison.

Precompute: we can precompute every answer and store it in $2^d \log(n)$ space and then get d time lookup. This is good if $d = \log(n)$. Otherwise we can again use JL again to get a d time lookup with polynomial space in n (but exponential space in ϵ!).

Locality Sensitive Hashing (LSH) gives another trade off. Gives lookup time n^ρ and space $n^{1+\rho}$ for $\rho = \frac{1}{c}$. These notes are all for hamming or L_1 distance, but for L_2 can get down to $\rho = \frac{1}{c^2}$. Thus we will use ρ instead of $\frac{1}{c}$. Similarly there is an even tighter bound for the L_2 norm than the specific hamming bound given above.

3 Algorithm Outline

What we would like is that nearby points tend to hash to the same bucket, and distant hashes tend to hash to different buckets. That is $h : X \to U$ where U are the cells of our hash table. When we look at $\mathbb{P}[h(x) = h(y)]$, we want it to decrease as $\|x - y\|$ increases.

Specifically, we want a p_1 and p_2 so that:

$$\forall x, y : \|x - y\| \leq r \implies \mathbb{P}[h(x) = h(y)] \geq p_1$$
$$\|x - y\| \geq cr \implies \mathbb{P}[h(x) = h(y)] \leq p_2$$

If we have this, then we will get LSH with parameter $\rho = \log(1/p_1)/\log(1/p_2)$. This will end up being the ρ in LSH above and improves as p_1 increases and p_2 decreases.

3.1 Intuition on Technique Limits

For rest of the notes, we will construct locality sensitive hashing, but first we will look at the limits of this method.

Let us analyze points that are just in a line all separated by distance r. Lets call then $x, z_1, ..., z_{c-1}, y$. Then the probability $h(x) = h(y)$ which is at most p_2 is bounded below by the probability all the z hash to the same thing. Since we are only looking for *intuition*, let us assume all these collisions are independent (even though we know this is probably wrong). Then we see that:

$$\begin{aligned} p_2 \geq \mathbb{P}[h(x) = h(y)] & \\ & \geq \mathbb{P}[h(x) = h(z_1) = h(z_2) = ...h(z_{c-1}) = h(y)] \\ & \sim \mathbb{P}[h(x) = h(z_1) \cdot] \, \mathbb{P}[h(z_1) = h(z_2)] \cdot ... \cdot \mathbb{P}[h(z_{c-1}) = h(y)] \\ & \geq p_1 \cdot p_1 \cdot ... \cdot p_1 \\ & = p_1^c \end{aligned}$$

So we don't expect to get $p_2 \geq p_1^c$. Now, this is **NOT** a proof that such a bound is optimal, but such a bound is a reasonable goal. Indeed, as stated above, there are improvements that can be made over this in the L_2 space. But this is indeed what we get for hamming distance of strings.

4 Algorithim

4.1 Constructing the Locality Sensitive Hash

To construct our locality sensitive hash, we first make a locality sensitive hash with an appropriate ρ. Then we repeat to decrease p_2 and get the false positive rate small.

1. Let h output one coordinate of its input: i. That is, for some random i, $h(x) = x_i$. Then $\mathbb{P}[h(x) = h(y)] = 1 - \frac{\|x-y\|}{d}$. Thus $p_1 = 1 - \frac{r}{d}$ and $p_2 = 1 - \frac{cr}{d}$. Then

$$\rho = \log(1/p_1)/\log(1/p_2) \simeq \frac{1}{c}$$

This comes from $\log(1 - \epsilon) \simeq -\epsilon + O(\epsilon^2)$, or from $\log(1/p_1)/\log(1/p_2) = \frac{1}{\log_{p_1}(p_2)}$.

2. Now we want to get $p_1 = \frac{1}{n}$ and $p_2 \leq \frac{1}{n^c}$. Let

$$g(x) = (h_1(x), h_2(x), ..., h_k(x)) \text{ for independent h from step 1}$$

Now we are choosing k random coordinates of x and using that as our hash function. Then

$$\mathbb{P}[g(x) = g(y)] = \left(1 - \frac{\|x-y\|}{d}\right)^k$$

Then $p_{1,g} = p_{1,h}^k$ and $p_{2,g} = p_{2,h}^k$. Now we want to set k so that $p_{1,g} = \frac{1}{n}$, so set $k = \frac{\log(n)}{-\log(1-r/d)}$. Then

$$\begin{aligned} p_{1,g} =& p_{1,h}^k \\ =& (1 - \frac{r}{d})^{\frac{\log(n)}{-\log(1-r/d)}} \\ =& (1 - \frac{r}{d})^{-\log_{1-r/d}(n)} \\ =& \frac{1}{n} \end{aligned} \qquad \begin{aligned} p_{2,g} =& p_{1,h}^k \\ =& (1 - \frac{cr}{d})^{\frac{\log(n)}{-\log(1-r/d)}} \\ \leq& (1 - \frac{r}{d})^{-c\log_{1-r/d}(n)} \\ =& \frac{1}{n^c} \end{aligned}$$

4.2 Solving Near Neighbors with LSH

Now we take this small false positive rate from above and look for hash collisions. This will give false positives much more rarely than it gives true positives. Then we will repeat many times so that we give true positives very often but false negatives still rarely.

1. LSH uses g as a hash function, which outputs to $\{0,1\}^k$. We store each point into a hash table using g. We will later lookup close points using this hash, hoping that the hash stores them together. Using a linked list hash table, this will use $2^k + n = O(n)$ space to store the hash table.

   To query q, we will hash q and look through the linked list until we find some p_i within distance cr to q. The time this will take is the amount of time to hash plus the amount of time it takes to compare distances times the number of collisions of far points in this bucket plus one if something within cr is in this bucket.

   The expected time to hash is just k, the number of expected collisions is at most $\frac{n}{n^c}$ and a length comparison takes time d. So the expected time is $k + \frac{d}{n^{c-1}}$ if there are no matches and d larger if there are. We can pessimistically bound this by $O(d)$.

   We fail if every point within r does not collide, which if there is a point close, is chance at most $1 - p_1 = 1 - 1/n^\rho$. So we have space O(n) with success probability $\frac{1}{n^\rho}$.

2. To get the LSH parameters above, we repeat n^ρ times to get constant factor probability of success. Can do $\log(n)$ more to get high probability. Then the space is just $n^{1+\rho}$ and time dn^ρ.

   If we analyze a tiny bit closer, we actually see that lookup time from step 1 was only $O(d)$ if we find a match. Otherwise it is expected to be much smaller. Further, a match will only happen once and then we will immediately return. Thus if we analyze closely, we can find a slightly better expected lookup time of $O(d + n^\rho k + dn^{\rho+1-c})$.

4.3 LSH Recap

So LSH just starts with a naive hash function with good ρ, but high false positive. Then we apply many of these to get a very small false positive rate, and a larger but still small true positive rate. Then we try looking up with this new hash function to find a near neighbor with a small positive rate, but a much smaller false positive rate. To make finding a match likely, repeat this hash lookup with many different hashes. If there is a good answer, this algorithm will find an okay answer. In particular this algorithm works well if ρ is significantly less than 1.