princeton univ. F'18 cos 521: Advanced Algorithm Design

Lecture 12: Nearest Neighbor Search and Locality Sensitive Hashing

Lecturer: *Christopher Musco*

1 Nearest Neighbor Search

Today we will discuss a widely solved problems involving high dimensional data points: nearest neighbor search (NNS). In NNS, we are given:

1. A database of points $X = \{x_1, \ldots x_n\} \subset U \subseteq \mathbb{R}^d$ (e.g. U might be $\{0,1\}^d$).
2. A distance metric $d(a,b)$ for points $a, b \in U$.[1]

Our goal is to:

1. Construct a data structure for X (we're allowed pre-processing time and extra space).
2. Given any query point $q \in U$, find the nearest point in our database, $\arg\min_{x \in X} d(q,x)$.

The most important complexity measures for this lecture are **space complexity** and **query time** – i.e. how much auxiliary space do we use on top of what's required to store X and how fast can we find the nearest neighbor for a given query q. In general there will be a tradeoff here: larger space data structures allow for faster query time. Naively, if we just store our points in $O(dn)$ space, our query time is linear in n (we need to scan through our entire database). Our goal will be to find a solution with $o(n)$ query time.

For example, when $d = 1$ the NNS problem can be solved using a binary search tree, which uses space $O(n \log n)$ and query time $O(\log n)$. Typically, however, we are interested in much higher dimensional data.

Curse of dimensionality

Binary search trees can be extended to d dimensions via kd trees, or other space partitioning data structures, but the query time of these data structures grows exponentially in d! Unfortunately, this is a generic phenomenon. Beating the naive linear scan is very hard when d is even moderately large (e.g. when $d > \log n$).

In particular, all known data structures that beat $O(dn)$ query time require $O(2^d)$ space. It is even possible to develop some formal lower bounds based on complexity assumptions like the "Strong Exponential Time Hypothesis".[2]

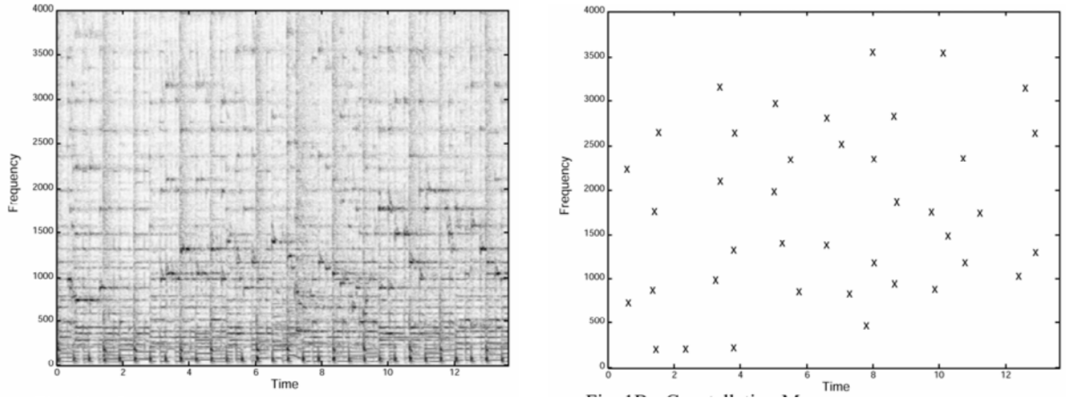
This tradeoff is sometimes referred to as the "curse of dimensionality" and it effects a whole host of other geometric problems (clustering, physical simulation, etc.)

[1] We will assume that d is an actual metric with all the usual properties: $d(a,b) = d(b,a)$, $d(a,b) \geq 0$, $d(a,b) = 0 \leftrightarrow a = b$, $d(a,b) + d(b,c) \geq d(a,c)$.

[2] See e.g. [1] and references.

Example applications

Music search and identification services like Shazam are often built on top of efficient nearest neighbor search algorithms for high dimensional vectors. These algorithms seek to identify a song from an audio recording of a short clip from the song. To do so, they extract a feature vector $q \in \mathbb{N}^d$ from the short audio clip. Typically, this involves constructing a "spectrogram" for the clip, which is a sort of windowed and shifted Fourier transform. The spectrogram tracks frequency components in the audio clip versus time. After identifying "peaks" in the spectrogram (see Figure 1), q is constructed by recording the distance + time offsets of nearby peaks, which is a feature invariant to time shifts. A nearest neighbor search is used to compare q against a huge database of already processed clips from known songs. This is done efficiently using the same ideas discussed in the rest of the lecture.[3]



(a) Spectrogram extracted from clip of an audio file.

(b) Post-processed spectrogram which can be used to construct an audio "fingerprint" $q \in \mathbb{N}^d$ for identifying a song clip.

Figure 1: Images from the paper outlining Shazam's approach to audio retrieval [2]. Once features are extracted from an audio clip, a hashing based algorithm is used to perform approximate nearest neighbor search in a database of known songs.

Very similar ideas are also used in detecting early warning signs of earthquakes and other seismic events [3]. The spectrogram of a seismogram is computed in real-time and quickly searched against a database of past spectrograms that surrounded major seismic events to see if any appear similar. Again, this is a very high-dimensional search problem, and is solved using fast hashing methods.

2 Approximate Near Neighbor

Fortunately, we can break the curse of dimensionality if we are willing to tolerate some level of approximation.

[3] Shazam's original approach was outlined in [2].

Problem 1 (Approximate Nearest Neighbor (ANN))**.** *For a given query point q, find some point $p \in X$ such that, for some approximation factor $c > 1$,*

$$d(q,p) < c \cdot \min_{x \in X} d(x,q).$$

To solve Problem 1 we actually consider an even simpler problem:

Problem 2 ((r_1, r_2) - Point Location in Equal Balls (PLEB))**.** *For given radii r_1, r_2 with $r_1 \leq r_2$, if there is at least one point $p \in X$ with $d(q,p) \leq r_1$, return any p with $d(q,p) < r_2$. On the other hand, if there is no point $p \in X$ with $d(q,p) < r_2$, output FAIL.*

Note: If there is a point with radius $d(q,p) \leq r_2$, but no point with $r_1 < d(q,p)$, we don't require anything about what the algorithm returns – it can return either FAIL or a point.

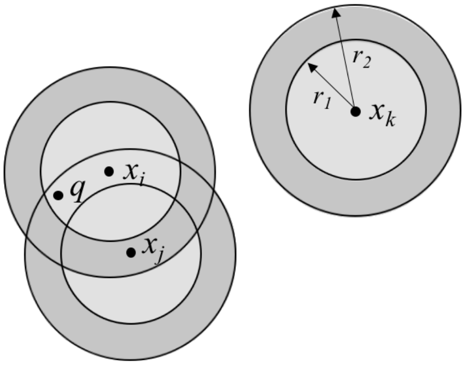


Figure 2: A solution to the (r_1, r_2)-PLEB problem for query point q could correctly return either x_i or x_j for this instance.

Claim 3. *Let d_{max} be the maximum distance between two points in X and let d_{min} be the minimum distance between distinct points. For approximation factor $c > 1$, (r, cr)-PLEB can be solved in space S and query time T for any r, than Problem 1 (ANN) can be solved with accuracy c^2 in $O(S \log_c(cd_{max}/(c-1)d_{min}))$ space and $O(T \log\log_c(cd_{max}/(c-1)d_{min}))$ time.*

Note that if $c = 1+\epsilon$ for some small ϵ, then $\log_c(cd_{max}/(c-1)d_{min}) = O\left(\frac{\log(cd_{max}/\epsilon d_{min})}{\epsilon}\right)$. This dependence on $1/\epsilon$ could be costly, but in practice it's much more common to set c to be a constant, in which case the logarithmic terms are small.

Proof. We construct a data structure for (r, cr)-PLEB for:

$$r = \left\{ \frac{d_{min}}{2c},\ \frac{d_{min}}{2},\ c\frac{d_{min}}{2},\ c^2\frac{d_{min}}{2},\ \ldots,\ \frac{d_{max}}{c-1} \right\}.$$

We then use binary search to find the minimum r such that PLEB returns a point p. Let r^* be this minimum radius. , As long as r^* does not equal the first value in our list (i.e. it does not equal $\frac{d_{min}}{2c}$), we have:

- $d(q,p) < cr^*$ (by definition of PLEB)

- There is no $x \in X$ with $d(q,x) \leq \frac{1}{c} r^*$.

It follows that $d(q,p) < c^2 \min_{x \in X} d(q,p)$ as desired.

We are left to handle two edge case: when $r^* = \frac{d_{min}}{2c}$ and when the (r, cr)-PLEB data structure returns fail for *all* radii in our list.

If $r^* = \frac{d_{min}}{2c}$, then it must be that the point p returned has $d(q,p) < \frac{d_{min}}{2}$. There can be no other points $p' \in X$ with $d(q,p') < \frac{d_{min}}{2}$ because, otherwise, by triangle inequality, we would have $d(p,p') < d_{min}$. It follows that the point p returned is q's exact nearest neighbor – i.e. $d(q,p) = \min_{x \in X} d(q,p) < c^2 \min_{x \in X} d(q,p)$.

On the other hand, if we never succeed in returning a point, and in particular even $(\frac{d_{max}}{c-1}, c\frac{d_{max}}{c-1})$-PLEB fails, then it must be that $z = \min_{p \in X} d(q,p) \geq \frac{d_{max}}{c-1}$. From this claim, and triangle inequality, it follows that, for any $p \in X$,

$$d(q,p) \leq z + d_{max} \leq z + (c-1)z \leq cz.$$

In other words, if none of our PLEB data structures succeed then q must be so far from X that we can simply return any point $x \in X$ and obtain a c approximate solution. □

3 PLEB via Locality Sensitive Hashing

So we have successfully reduced to the approximate near neighbor problem to a simpler query problem (Problem 2) that only considers a single "distance scale" at a time. To solve this problem we will use a technique called "locality sensitive hashing" (LSH), which was introduced by Indyk and Motwani in 1998 [4] and has been very influential, both in theory and in practical implementations of high-dimensional nearest neighbor search.

Definition 1 (Locality Sensitive Hash Family)**.** *For distances r_1, r_2 with $r_1 < r_2$, a family of hash functions $\mathcal{H} : U \to S$ is (r_1, r_2, p_1, p_2)-locality sensitive if for any $x, y \in U$:*

1. *If $d(x,y) \leq r_1$ then $\Pr_{h \in \mathcal{H}}[h(x) = h(y)] \geq p_1$*
2. *If $d(x,y) \geq r_2$ then $\Pr_{h \in \mathcal{H}}[h(x) = h(y)] \leq p_2$.*

For our purposes, we always have $p_2 < p_1$. I.e. if two points are close together, they have a strictly higher probability of hashing to the same bucket than two points that are far apart. This property is ultimately what allows us to perform efficient near neighbor search.

Example 1. *Suppose $U = \{0,1\}^d$ and our distance metric for $x, y \in U$ is $d(x,y) = \sum_{i=1}^{d} |x_i - y_i|$ – i.e. the number of entries that differ between x and y. This is also known as the the hamming distance. It is closely related to Jaccard similarity, which was discussed in Lecture 4. There exists a locality sensitive hash family for this metric with parameters:*

$$(r_1, r_2, p_1, p_2) = (r, cr, 1 - \frac{r}{d}, 1 - \frac{cr}{d})$$

In particular, we select a random hash function h by choosing i uniformly at random from $\{1, \ldots d\}$ and simply set:

$$h(x) = x_i.$$

So our LSH family is the set of d hash functions that maps a bit vector x to any single bit in the vector. In this example the output space of our hash function is $S = \{0, 1\}$. This is typical: natural locality sensitive hash functions for most distances have $|S|$ very small.

Theorem 4 (Indyk, Motwani, 1998)**.** *Given a (r_1, r_2, p_1, p_2)-locality sensitive hash family, (r_1, r_2)-PLEB can be solved with constant probability using:*

- *Space: $O\left(dn + n^{1+\gamma}\right)$,*
- *Query time: $O\left(n^{\gamma}\right)$ hash function evaluations and metric computations, $d(\cdot, \cdot)$.*

where $\gamma = \frac{\log(1/p_1)}{\log(1/p_2)}$.

γ is smaller when either p_1 is bigger or p_2 is smaller. Intuitively, the more we can "spread out" these probabilities, the better our locality sensitive hash function, and the more efficient our near neighbor search.

For the hamming distance example we have:

$$\frac{\log(1/p_1)}{\log(1/p_2)} = \frac{-\log(1 - r/d)}{-\log(1 - cr/d)} \leq 1/c.$$

The last inequality follows from noting that $(1 - r/d)^c \geq (1 - cr/d)$ for $c \geq 1$.

Accordingly, for the hamming distance we are able to construct a data structure that uses space $O(n^{1+1/c})$ and answers queries in $n^{1/c}$ time. This can be a significant improvement over a linear scan! When $c = 2$, our query complexity drops to $O(\sqrt{n})$. For the song identification application discussed earlier, Shazam indexes roughly 1.5 million songs. If each song is ≈ 4 minutes long and is chopped up into sections of 10 seconds, this means that $n = |X| \approx 36$ million. A linear scan would be infeasible.

However, in this case, $\sqrt{n} = 6,000$, a much more manageable complexity bound. What's more, if $c = 4$, we get a bound depending on $n^{1/4} < 80$.

Proof of Theorem 4. The result applies to the following procedure for using a locality sensitive hash family for near neighbor search:

Preprocessing:

Set $k = \log n / \log(1/p_2)$ and $\ell = 2n^{\gamma}$.

Construct a new hash family $\mathcal{G} : U \to S^k$ by concatenating k random hash functions from our locality sensitive family $\mathcal{H}$. I.e. $g \in \mathcal{G}$ is selected so that:

$$g(x) = [h_1(x), h_2(x), \ldots, h_k(x)] \text{ where } h_1, \ldots, h_k \text{ are drawn independently from } \mathcal{H}.$$

In the case of hamming distance and many other hash functions $S = \{0, 1\}$, so $g(x)$ is simply a bit string of length k.

Select ℓ hash functions $g_1, \ldots, g_\ell$ from $\mathcal{G}$. For each hash function g_j, i.e. each $j \in 1, \ldots, \ell$, construct a *separate hash table* T_j. Each T_j has a bucket for each element in S^k.[4]

For each $x \in X$, store x in bucket $g_1(x)$ in table T_1, in bucket $g_2(x)$ in table T_2, etc. Discard any empty buckets (and keep track of the indices of non-empty buckets using e.g.

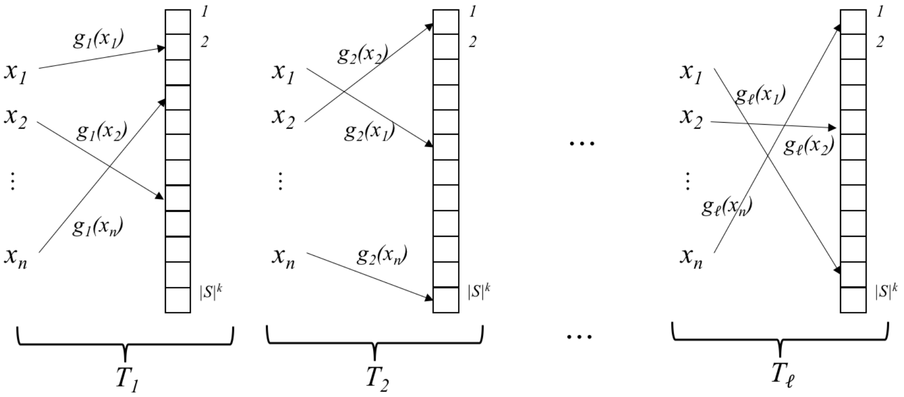


Figure 3: We use our locality sensitive hash function to build hash functions $g_1, \ldots, g_\ell$ which map each database point x_i to ℓ hash tables using ℓ independent hash functions.

a binary search tree or hash map). Our total space complexity is bounded by $O(n)$ for each table, for a total of $O(n) \cdot \ell = O(n^{1+\gamma})$.

Query:

Given a query point q, traverse through the elements in buckets $T_1(g_1(q)), \ldots, T_\ell(g_\ell(q))$ and return the first x with $d(x,q) \leq r_2$. If no such x is found before traversing $4\ell = O(n^\gamma)$ elements, don't return anything. This guarantees that our total query complexity is bounded by $O(n^\gamma)$.

Analysis:

Intuitively, because each g_j was constructed using a set of locality sensitive hash functions, $g_j(x)$ is more likely to collide with $g_j(q)$ when x is close to q. Accordingly, we view the contents of buckets $T_1(g_1(q)), \ldots, T_\ell(g_\ell(q))$ as "candidate near neighbors". We check all of these candidates by brute force until we find a point x that is actually close to q.

Formally, it suffices to prove that:

1. For any x with $d(x,q) \leq r_1$, with probability 3/4, $g_j(x) = g_j(q)$ for some $j \in \{1, \ldots, \ell\}$.
2. With probability 3/4, there are no more than 4ℓ pairs (x,j) with $d(x,q) \geq r_2$ and $g_j(x) = g_j(q)$ for some j.

Part 1 guarantees that, with probability 3/4, at least one of our "candidate near neighbors" in $T_1(g_1(q)) \cup \ldots \cup T_\ell(g_\ell(q))$ actually satisfies $d(x,q) \leq r_1$, and could thus be returned as a solution to the PLEB problem. We should find this point when traversing $T_1(g_1(q)), \ldots, T_\ell(g_\ell(q))$, unless two possible things happen. First, when traversing our candidate buckets, we might find some other point x' with $d(q,x') \leq r_2$ and return that point before we even consider x. This is fine: we still solve the problem.

Alternatively, it might be that we consider 4ℓ points, all of which are $> r_2$ away from q, and thus we would have to terminate our query before considering x and before any solution to PLEB is found. This would be a bad outcome.

[4] Note: this approach is slightly different than the approach discussed in class: it simplifies the algorithm while achieving the same complexity

Part 2 of our claim guarantees that this bad outcome only happens with probability $\leq 1/4$: with probability $3/4$ there aren't more than 4ℓ "bad" points in our candidate buckets, so as long as Part 1 holds, we will certainly find some x with $d(q,x) \leq r_2$ before traversing 4ℓ candidates.

To prove Part 1, we note that, for any particular x with $d(x,q) \leq r_1$ and any particular j, the probability that $g_j(x) = g_j(q)$ is at least:

$$\Pr[g_j(x) = g_j(q)] \geq p_1^k = e^{\log(p_1)\log(n)/\log(1/p_2)} = n^{-\gamma}.$$

Since each g_j is chosen independently, it follows that:

$$\Pr[g_j(x) = g_j(q) \text{ for some } j \in 1,\ldots,\ell] \geq 1 - (1 - n^{-\gamma})^{\ell} \geq 3/4$$

as long as $\ell = 2\gamma$. Here we're using the inequality $(1 - \frac{1}{z})^z \leq 1/e$ for any $z \geq 1$. So Part 1 of the claim holds with probability at least $3/4$.

To prove Part 2, we let Y be a random variable denoting the number of pairs (x,j) with $d(x,q) \geq r_2$ and $g_j(x) = g_j(q)$.

$$\mathbb{E}Y \leq n \cdot \ell \cdot \Pr[g_j(x) = g_j(q) \text{ for a fixed } x, j] \leq n\ell \cdot p_2^k = \ell.$$

The last step follows from our choice of $k = \log(n)/\log(1/p_2) = \log(1/n)/\log(p_2)$. Then, by Markov's inequality, Y does not exceed 4ℓ with probability greater than $1/4$.

We conclude that, by a union bound over Parts 1 and 2 both holding, our data structure and query routine solves the PLEB problem with probability $1/2$. We can repeat the construction $\log(1/\delta)$ times to succeed with probability $1-\delta$, incurring a space and runtime overhead of just $\log(1/\delta)$. □

4 Other examples of locality sensitive hash functions

The locality sensitive hashing framework makes it easy to develop ANN algorithms for different metrics: it reduces the problem to finding hash functions that are more likely to collide for points that are close in your metric. It is often possible to find such hash functions by *quantizing* the output of dimensionality reduction algorithms.

For example consider points in $\mathbb{R}^d$ with $d(x,y) = \|x-y\|_2$. I.e. we want to solve ANN for the standard Euclidean norm. The JL lemma gives a way of generating short vectors $f(x)$ and $f(y)$ with $\|f(x) - f(y)\|_2 \approx \|x-y\|_2$. Imagine rounding each entry of $f(x)$ and $f(y)$ so that the entire vector can be represented by a small number of bits. These bits form a hash value and, if $\|f(x) - f(y)\|_2$ is small (i.e. $\|x-y\|_2$ is small) then the hash value for x is more likely to collide with the hash value for y.

Formally, we consider a scheme based on rounding a single entry of $f(x)$. To simplify the analysis, assume that all points in X and all query points lie on the unit sphere: i.e. $\|x\|_2 = 1$ for all $x \in X$ and $\|q\|_2 = 1$. Let:

$h(x) = \text{sign}(\langle g, x\rangle)$ were g has entries $g_i \sim \mathcal{N}(0,1)$ (each is a standard normal random variable).

Since g is spherically symmetric, this hash function is equivalent to drawing a random hyperplane through the origin and setting $h(x) = 1$ for all points above the hyperplane, and $h(x) = -1$ otherwise (see Figure 4).

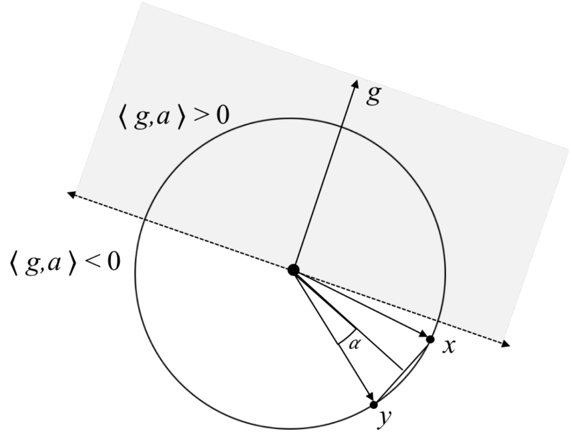


Figure 4: Our Euclidean locality sensitive hash function draws a random Gaussian vector g. Consider the hyperplane orthogonal to g and passing through $\{0\}^d$. All points above the hyperplane are hashed to 1 and all points below are hashed to -1.

From the figure above, it's not too hard to see that the probability that two points x, y with distance $\|x-y\|_2 =$ end up on the same side of the hyperplane is:

$$1 - \frac{2}{\pi}\alpha$$

where $\alpha = \sin^{-1}(r/2)$. So for $r_1 = r$ and $r_2 = cr$ we have:

$$\gamma = \frac{\log\left(1 - \frac{2}{\pi}\sin^{-1}(r/2)\right)}{\log\left(1 - \frac{2}{\pi}\sin^{-1}(cr/2)\right)} \leq \frac{1}{c}.$$

Overall, each hash function g in our LSH scheme is a bit vector constructed by taking the sign of each entry in Πx, where Π is a random Gaussian matrix. It's possible to speedup hashing time, and thus query time, by using a structured matrix Π that supports fast matrix-vector multiplication (like the randomized Hadamard matrices discussed in Lecture 10). However, doing so is not known to give the same theoretical guarantees.

Finally, we note that the $1/c$ bound matches our bound for hamming distance, but is actually sub-optimal. A more involved construction can achieve $\gamma \approx 1/c^2$ for Euclidean distance [5]. This can be a signficant improvement for relatively large c (e.g. for $c \geq 1/2$, query time reduces by a square root factor). There has been work on practical hashing schemes that achieve the $1/c^2$ improvement while also preserving the simplicity (and speed) of the random hyperplane hashing method described here [6].

References

[1] Aviad Rubinstein. Hardness of approximate nearest neighbor search. Symposium on Theory of Computing (STOC), 2018.

[2] Avery Li-Chun Wang. An Industrial Strength Audio Search Algorithm. 4th International ISMIR Conference on Music Information Retrieval. 2003.

[3] Clara E. Yoon, Ossian O'Reilly, Karianne J. Bergen, and Gregory C. Beroza. Earthquake detection through computationally efficient similarity search. Science Advances, 1(11). 2015.

[4] Piotr Indyk and Rajeev Motwani. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. Symposium on Theory of Computing (STOC), 1998.

[5] Alexandr Andoni and Piotr Indyk. Near-Optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions. IEEE Symposium Symposium on Foundations of Computer Science (FOCS), 2006.

[6] Alexandr Andoni, Piotr Indyk, Thijs Laarhoven, Ilya Razenshteyn and Ludwig Schmidt. Practical and Optimal LSH for Angular Distance. Alexandr Andoni, Piotr Indyk Neural Information Processing Systems (NIPS), 2015.