Lecture 7: Nearest Neighbours and dimensionality reduction

Last week, we focused on hash tables and Bloom filters, which enable us to solve the *dictionary problem*:

> Given a dataset S, subset of a very large "universe" $\mathcal{X}$, how do we quickly check if a new element $x \in \mathcal{X}$ is in the dataset?

This week, we will focus on a different, but related question, which can be seen as a "relaxed" version of the dictionary problem. Namely, we will not ask whether a query x is *in* the dataset, but instead we will ask to return a point $y \in S$ that is *close* to x – ideally, the closest. You can see the applications of this question: (1) given a noisy version x of an image, find the closest picture stored in the dataset (i.e., a "denoised" version of x); (2) given a point in a high-dimensional space, cluster it by finding a close center; (3) given an assignment submitted this semester, find the most similar in the database of assignments from previous years... those are only a few examples.

To even start, we need to define what "close" means: that is, we need a notion of *distance* between elements of the universe $\mathcal{X}$. This can be weakened a little, but here we will assume $\mathcal{X}$ is a metric space, and comes with a metric[34]

$$\text{dist}\colon \mathcal{X} \times \mathcal{X} \to \mathbb{R}_+ .$$

To give an example, think of a few metrics you most likely know or have encountered before:

1. The *Manhattan distance* on $\mathcal{X} = \mathbb{R}^d$ (a.k.a. the ℓ_1 distance, or taxicab distance): $\text{dist}(x,y) = \sum_{i=1}^{d} |x_i - y_i| = \|x-y\|_1$
2. The *Euclidean distance* on $\mathcal{X} = \mathbb{R}^d$ (a.k.a. the ℓ_2 distance): $\text{dist}(x,y) = \sqrt{\sum_{i=1}^{d} (x_i - y_i)^2} = \|x-y\|_2$
3. The *Hamming distance* on $\mathcal{X} = \{0,1\}^d$: $\text{dist}(x,y) = \sum_{i=1}^{d} \mathbb{1}_{x_i \neq y_i}$

Any meaningful others?

With this in hand, we can define the problem we want to solve, the *Nearest Neighbour Problem:*

Nearest Neighbour (NN)

[34] That is, dist(,) is non-negative, reflexive:

$$\text{dist}(x,y) = 0 \Leftrightarrow x = y$$

symmetric:

$$\text{dist}(x,y) = \text{dist}(y,x),$$

and satisfies the triangle inequality:

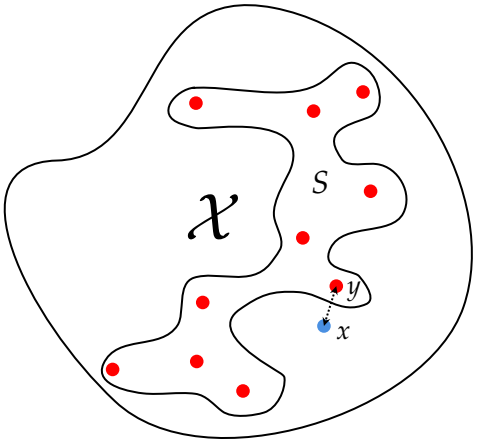
$$\text{dist}(x,y) \leq \text{dist}(x,z) + \text{dist}(z,y)$$

for all $x, y, z \in \mathcal{X}$.

Given a dataset S, subset of a very large metric space $(\mathcal{X}, \text{dist})$, how to, given a new element $x \in \mathcal{X}$, output an element $y \in S$ that minimises $\text{dist}(x, y)$?

As before, we will let $n = |S|$ denote the current size of the dataset (which can change as we insert or remove elements), but instead of denoting by m the size of the universe $\mathcal{X}$, we will instead focus on high-dimensional universes such as $\{0,1\}^d$ or $\mathbb{R}^d$, and denote by d the *dimension* of the universe.

So, for $\mathcal{X} = \{0,1\}^d$, we have $m = 2^d$. But for $\mathcal{X} = \mathbb{R}^d$, $m = \infty$.



We will also for simplicity restrict ourselves to the *offline* setting, where the dataset S is not changing over time: instead, we are given all n elements of S at once, and can spend some time preprocessing them to create our data structure. We then only need to support the QUERY operation:

QUERY(x): given an element $x \in \mathcal{X}$, return an element $y \in S$ minimising $\text{dist}(x, y)$, that is, $\text{dist}(x, y) = \min_{y' \in S} \text{dist}(x, y')$.

The two main complexity measures we will seek to minimise for our data structure are:

- *Space complexity:* we would like our data structure to take as little space as possible, ideally $O(nd)$;
- *Query time:* we want queries to be *fast*, ideally in time *sublinear in* n (and "reasonable" in d). For instance, $\text{poly}(d) \cdot O(n^{0.99})$ would not be bad.

Here d takes the role of $\log m$ from the previous lecture. Can you see why?

(if possible, we will also try to keep the *preprocessing time* under control too, which is the time complexity of creating the data structure from the n elements of S). The rationale for seeking $o(n)$, reasonable-in-d query time is because while we think of the regime where the data is high-dimensional ($d \gg 1$), we also focus on the

regime where the dataset is *huge*: $n \gg d$. As a rule of thumb, you should keep in mind

$$1 \ll d \ll n \ll 2^d$$

"Why?" Note that for the case of $\mathcal{X} = \{0,1\}^d$ for instance, n can never be larger than 2^d. And (peeking ahead), we will see the JL lemma, which in some sense guarantees that even in Euclidean space, d can be "made" as small as $O(\log n)$.

In what follows, we will also assume that we can compute distances efficiently: specifically, that $\mathrm{dist}(x,y)$ can be computed in time $O(d)$ for any two $x, y \in \mathcal{X}$; and that storing an element of $\mathcal{X}$ takes space $O(d)$.

Let's not get into the details of how to actually store a value $x \in \mathbb{R}^d$ on a computer.

Baseline. So, what can we do? The good news is that we can relatively easily achieve one of the two requirements. Namely:

- There is a (deterministic) data structure for the Nearest Neighbour problem using space $O(nd)$, and query time $O(nd)$;
- There is a (deterministic) data structure for the Nearest Neighbour problem using space $O(2^d)$, and query time $O(2^d)$.

We will see them in the tutorial.

The bad news is that this is essentially all that we know. That is, *every* data structure (even probabilistic) we know for Nearest Neighbour has either space or query complexity $\Omega\Big(\min(2^d, nd)\Big)$ in the worst case.

Relax (the problem)! Faced with this, a natural response is to shrug and give up. Another natural response, a few minutes later usually, is to try and modify the problem we were aiming to solve, to see if a weaker, "relaxed" variant could be enough (and easier). This is the basis for the *Approximate Nearest Neighbour* question: instead of asking for a point $y^* \in S$ *closest* to the query $x \in \mathcal{X}$, we only ask for a point $y \in S$ that is *not much further from* x *than* y^*, *i.e.*, is "good enough."

Approximate Nearest Neighbour (ANN)

> Given a dataset S, subset of a very large metric space $(\mathcal{X}, \mathrm{dist})$, how to, given a new element $x \in \mathcal{X}$, output an element $y \in S$ that is within a constant factor of $\min_{y' \in S} \mathrm{dist}(x,y')$?

Now, our data structure is parameterised by a value $C > 1$ (fixed at the creation of the data structure), and must support queries of this type:

> QUERY(x): given an element $x \in \mathcal{X}$, return an element $y \in S$ sort-of-minimising $\mathrm{dist}(x,y)$, that is, $\mathrm{dist}(x,y) \leq C \cdot \min_{y' \in S} \mathrm{dist}(x,y')$.

(If we were to set $C = 1$, then we would be back to the Nearest Neighbour problem.)

Dimensionality reduction: the Johnson–Lindenstrauss lemma

The first tool we will see is specific to Euclidean space ($\mathcal{X} = \mathbb{R}^d$, $\operatorname{dist}(x,y) = \|x-y\|_2$), but has applications going beyond Approximate Nearest Neighbours: it is a *dimensionality reduction technique* which gives a way to map points in $\mathbb{R}^d$ to points in $\mathbb{R}^k$, for $k \ll d$, while nearly preserving all their pairwise distances. That is, the Johnson–Lindenstrauss lemma gives an efficient, probabilistic mapping

$$\Phi\colon \mathbb{R}^d \to \mathbb{R}^k$$

where $k = O(\log(1/\delta)/\varepsilon^2)$ such that, for any two fixed $x, y \in \mathbb{R}^d$,

$$\|\Phi(x) - \Phi(y)\|_2 = (1 \pm \varepsilon)\|x-y\|_2$$

with probability at least $1-\delta$. This seems quite magical: for instance, taking $\varepsilon = 0.01$ and $\delta = 1/100$, this gives a mapping from the d-dimensional Euclidean space (d is huge!) to a *constant*-dimensional space which preserves the distance between any two points of your choosing, up to a factor 1.01!

What is even better is that this Φ is not some insanely complicated, cumbersome to describe and impossible to implement random function. It is a random *linear* mapping, obtained by just picking a random matrix $M \in \mathbb{R}^{k\times d}$ with independent random Gaussian coefficients, and setting $\Phi(x) = Mx$. [35,36]

Theorem 36 (Distributional JL Lemma). *Fix any $\varepsilon, \delta \in (0, 1/2)$, and set*

$$k = \Theta\left(\frac{\log(1/\delta)}{\varepsilon^2}\right).$$

Consider the random matrix $M \in \mathbb{R}^{k\times d}$ obtained by drawing each entry M_{ij} independently from the Gaussian distribution $\mathcal{N}(0, 1/k)$. Then, for any fixed $u \in \mathbb{R}^d$, we have

$$\Pr_M[\,(1-\varepsilon)\|u\|_2 \le \|Mu\|_2 \le (1+\varepsilon)\|u\|_2\,] \ge 1-\delta\,.$$

We will not prove the theorem in this class, but a few remarks are in order: first, M can be created in time $O(kd)$ time (assuming sampling from $\mathcal{N}(0,1)$ in constant time). Second, for any $x \in \mathbb{R}^d$ the projection Mx can be computed in time $O(kd)$ as well.[37] Third, and this is quite important for us, this implies the following corollary, which we *will* prove and is what is commonly known as "the JL Lemma."

Corollary 36.1 (JL Lemma). *Fix any $\varepsilon \in (0, 1/2)$ and $n \ge 2$, and set*

$$k = \Theta\left(\frac{\log n}{\varepsilon^2}\right).$$

Consider the random matrix $M \in \mathbb{R}^{k\times d}$ defined in Theorem 36. Then, for any fixed set $T \subseteq \mathbb{R}^d$ of n elements, we have

$$\Pr_M[\,\forall x, y \in T,\ (1-\varepsilon)\|x-y\|_2 \le \|Mx - My\|_2 \le (1+\varepsilon)\|x-y\|_2\,] \ge \frac{9}{10}\,.$$

[35] William B. Johnson and Joram Lindenstrauss. Extensions of Lipschitz mappings into a Hilbert space. In *Conference in modern analysis and probability (New Haven, Conn., 1982)*, volume 26 of *Contemp. Math.*, pages 189–206. Amer. Math. Soc., Providence, RI, 1984

[36] Even, even better: subsequent work has shown how to replace "random Gaussian coefficients" by even simpler random coefficients (*e.g.*, in $\{-1, 0, 1\}$ or $\{-1, 1\}$, then scaled) while preserving the same guarantees.

[37] There are some improvements, such as the *Fast JL Transform*, to do this even faster.

Proof. Invoke Theorem 36 with $\delta = \frac{1}{10\binom{n}{2}} = \Theta\left(\frac{1}{n^2}\right)$. Take a union bound over all $\binom{n}{2}$ pairs of distinct $x, y \in T$, applying the theorem to $u := x - y$. □

What this corollary promises us is that, up to a small distortion in the $\binom{n}{2}$ pairwise Euclidean distances of our elements, we can replace our d-dimensional Euclidean space by a much more manageable $O(\log n)$-dimensional Euclidean space, losing essentially nothing in the process.

Application to ANN. We can use the JL Lemma to get a somewhat non-trivial improvement over our "baseline for Nearest Neighbour" in the case of Euclidean space, for *Approximate* Nearest Neighbour problem. Specifically, apply Corollary 36.1 with $n + 1$ and small $\varepsilon > 0$ of our choosing, and get the conclusion for the set $T = S \cup \{x\} \subseteq \mathbb{R}^d$ (which is fixed, even though we do not know the query x in advance). This allows us can solve the ANN problem, using the baseline approach but in $\mathbb{R}^k$, no longer $\mathbb{R}^d$:

Lemma 36.1. *For every $\varepsilon \in (0, 1/2$, there is a (probabilistic) data structure for the Approximate Nearest Neighbour problem with $C = 1 + \varepsilon$, using space $O(\frac{n \log n}{\varepsilon^2})$, and query time $O(\frac{n \log n}{\varepsilon^2})$, where the output to each query is correct with probability at least $9/10$.*

This is not getting all the way there (we still have a near-linear dependence on n in the query time!), but it is *better.*

Locality-Sensitive Hashing

In view of what we have seen about hashing, another appealing idea would be to design some type of (family of) hash function $h \colon \mathcal{X} \to \mathcal{Y}$ which somehow "preserves distances": if two elements x, x' are close, then are hashed into nearby buckets, and if they are far h sends them into very different buckets. In a sense, this is what the JL Lemma does for Euclidean distance: can we generalise this to other notions of distances than ℓ_2, and have a better control on the size ($\approx$ dimension) of the hashing space $\mathcal{Y}$?

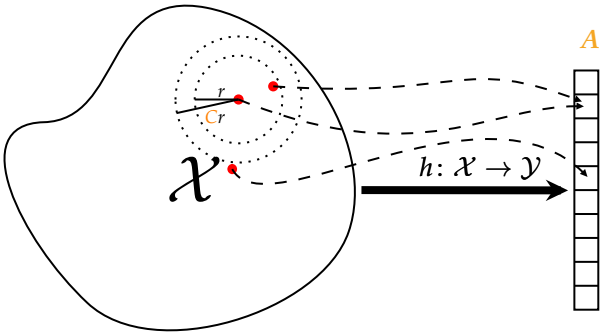
Locality-Sensitivity Hashing does exactly that, or, at least, sort of. It was introduced, for the case of Hamming distance, in an influential paper by Gionis, Indyk, and Motwani [38]: here is the formal definition.

[38] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity search in high dimensions via hashing. In *VLDB*, pages 518–529. Morgan Kaufmann, 1999

Definition 36.1. Let $0 \leq q < p \leq 1$, $r > 0$, $C > 1$, and $(\mathcal{X}, \mathrm{dist})$ be a metric space. Then a family of functions $\mathcal{H}$ from $\mathcal{X}$ to $\mathcal{Y}$ is a (r, C, p, q)-*Locality Sensitive Hash family* (LSH) if, for every $x, x' \in \mathcal{X}$,

- If $\mathrm{dist}(x, x') \leq r$, then $\Pr_{h \sim \mathcal{H}}[h(x) = h(x')] \geq p$;
- If $\mathrm{dist}(x, x') \geq Cr$, then $\Pr_{h \sim \mathcal{H}}[h(x) = h(x')] \leq q$;

and we say $\rho := \frac{\log(1/p)}{\log(1/q)} < 1$ is the *sensitivity parameter* of $\mathcal{H}$.



We will soon see the rationale behind defining this quantity ρ: in the meantime, as usual, a few observations:

- An LSH family provides a way to control, at a given "scale" r, the collisions probabilities: sure, you *will* have collisions, but points close to each other (closer than this parameter r) are more likely to collide than those much farther apart. This is not as strong as we would like (ideally, we would have wanted the guarantee to be true simultaneously for *all* $r > 0$), but this will be good enough.
- We would like p to be as large as possible, and q as small as possible: or, put differently, ρ to be as large as we can achieve. This gap is what will allow us to distinguish "far" from "close" with good enough probability: if p, q are almost equal, this makes our task harder.
- LSH families do exist (trivially): one could take $\mathcal{Y} = \mathcal{X}$ and $\mathcal{H}$ to be the single function mapping x to itself. This is not interesting at all (it does not save space, time, or anything really), but at least it shows this definition is not impossible to satisfy. What remains is to get more interesting LSH families, with small $\mathcal{Y}$.

As a start, we will show how to, given an LSH family for a specific scale $r > 0$, solve a "'baby version" of our Approximate Nearest Neighbour question: that is, we will give a data structure that, after preprocessing, supports the following query, for fixed values of $r > 0$ and $C > 1$.

QUERY$_r(x)$: given an element $x \in \mathcal{X}$, return an element $y \in S$, or $\perp$, such that:

- If there exists $y^* \in S$ such that $\text{dist}(x, y^*) \leq r$, then, with probability at least $9/10$, QUERY$_r(x)$ returns an element $y \in S$ such that $\text{dist}(x, y^*) \leq C \cdot r$;
- If $\text{dist}(x, y) > C \cdot r$ for *every* $y \in S$, then, with probability 1, QUERY$_r(x)$ returns $\perp$.
- Otherwise, any output in $S \cup \{\perp\}$ is allowed.

To solve this "baby ANN version," we will use... hash tables. But in addition to a standard, run-of-the-mill good hashing family for the hash tables, we will also use an (r, C, p, q)-LSH family, which we assume is given to us (we will later show how to design such an LSH family): in what is below, k and ℓ are integers, whose values we will carefully choose after analysing the guarantees of our data structure. We first need the following fact, which allows us to "tune" the parameters p, q of an LSH family.

Lemma 36.2. *Suppose $\mathcal{H}$ from $\mathcal{X}$ to $\mathcal{Y}$ is an (r, C, p, q)-LSH family, and fix any integer $\ell \geq 1$. For any $g_1, \dots, g_\ell \in \mathcal{H}$, define the function $g : \mathcal{X} \to \mathcal{Y}^\ell$ by*

$$g(x) = (g_1(x), \dots, g_\ell(x)) \in \mathcal{Y}^\ell$$

Then, the resulting family $\mathcal{H}^{(\ell)} := \{(g_1, \dots, g_\ell) : \mathcal{X} \to \mathcal{Y}^\ell\}$ is an (r, C, p^ℓ, q^ℓ)-LSH family of size $|\mathcal{H}|^\ell$ (and as a result has the same sensitivity parameter ρ).

Proof. "Proof by writing it down." □

Intuitively, this gives us some flexibility when designing our data structure: we do not get to choose p, q, and would like both (1) p to be large (better chances of finding close elements, and so smaller probability of failure) *and* (2) q to be small (fewer "spurious" collisions introduced, which will mean better query complexity in our final hash table-based data structure). This parameter ℓ allows us to control (2) (as q^ℓ can be made as small as we want), at the price of making (1) worse as well (p^ℓ goes down too); fortunately, we will soon introduce our other parameter k which will give us control over (1), and so we will be able to achieve both (1) *and* (2) by carefully balancing k and ℓ.

"What is the point of this ℓ?"

Let us now actually describe our data structure:

For a fixed, given r (and some $C > 1$), let $\mathcal{H}$ be an (r, C, p, q)-LSH family. Let $g_1, \dots, g_k : \mathcal{X} \to \mathcal{Y}^\ell$ be hash functions chosen independently from $\mathcal{H}^{(\ell)}$. Build k hash tables $(A_1, h_1), \dots, (A_k, h_k)$ with separate chaining, where $h_1, \dots, h_k : \mathcal{Y} \to \mathcal{Z}$ are k independent "standard" hash functions from a suitable hashing family.

- PREPROCESS(S):
  for all $x \in S$ **do** ▷ Insert all n elements in all k hash tables, using their LSH hashing as keys
    for all $1 \leq t \leq k$ **do**
      $A_t[h_t(g_t(x))]$.INSERT(x)

- QUERY$_r(x)$:
  for all $1 \leq t \leq k$ **do**
    $L_t \leftarrow A_t[h_t(g_t(x))]$ ▷ List of elements of S colliding with $g_t(x)$ in the t-th hash table

```
        for all y ∈ L_t do
            if dist(x, y) ≤ C · r then
                return y                          ▷ Found one!
    return ⊥                                      ▷ Did not find any.
```

Note that we use both the locality-sensitive hash functions g_t *and* the "regular" hash function h_t: this is to save space, as if we only used the former we would have the "locality sensitive" part, but none of the good guarantees from usual hash functions (small number of hash buckets, along with small collision probability) which allowed us last lecture to argue hash tables were space-efficient. That is, we use two levels of hashing, with two different "types" of hash functions:

"Why do we use two types of hash functions?"

- the first level uses the LSH function g_t to hash elements in a "locality-sensitive way" (we *want* close elements to collide, but far elements to go to distinct buckets): this is the main conceptual idea. However, these at-most-n resulting elements $S'_t := \{g_t(x)\}_{x \in S}$ still live in a very big space (*i.e.*, $\mathcal{Y}^\ell$), so storing them naively would not be space-efficient; and so,
- the second level uses the "usual" hash functions h_t to hash this set of "LSH hashes" S'_t in a "standard hash table way" (we want as few collisions as possible), to save space.

We will establish the following guarantees for this data structure, assuming each hash function from $\mathcal{H}$ can be evaluated in time T and each hash table A_t uses space $O(nd)$:

We can improve the $O(knd)$ space complexity to $O(kn \log n + nd)$, by only keeping the index of each element $x \in S$ in the data structure along with a separate array containing all of them, to look up where the i-th element actually is.

Theorem 37. *The data structure described above for the "baby version of ANN" has space complexity $O(knd + k\ell d)$, expected query complexity $O(k\ell T + knd q^\ell)$, and satisfies the correctness requirements as long as $(1 - p^\ell)^k \leq \frac{1}{10}$.*

Proof. The space complexity follows from that of the k hash tables, each of which using space $O(nd)$ (we also have to account for the storage of the k "ℓ-fold" LSH hash functions, which we assume can be done with $\ell \cdot O(d)$ bits each).

The expected query time comes from (1) evaluating (up to) k hash functions $g_1(x), \ldots, g_k(x)$, for a total time $O(k\ell T)$; (2) for each $1 \leq t \leq k$, checking the distance to x of every point in the list L_t until we find one that is at distance less than Cr. So we only have to count the expectation number of *false positives* which collide but are far, since a true positive y will be a success, and causes the function to immediately stop and return y. Each distance computation can be done in time $O(d)$, and by the LSH guarantee we have on expectation at most

$$n \cdot q^\ell$$

such false positives (points y such that $\text{dist}(x, y) \geq C \cdot r$ but with $g_t(x) = g_t(y)$); to this, we must add an additional expected $O(1)$

"standard hash collisions," just from the usual guarantees of good hash tables. So that gives us expected query time at most

$$O(k\ell T) + k \cdot \Big(n \cdot q^\ell + O(1) \Big) \cdot O(d) = O(k\ell T + knd q^\ell)$$

assuming, for the last part, that $nq^\ell = \Omega(1)$.[39]

[39] In particular, there is no point in setting ℓ such that $nq^\ell \ll 1$, as the $O(1)$ term would then dominate.

For the correctness, first observe that the second item is immediate from definition of the algorithm: since Line 4 checks the distance is at most $C \cdot r$, if every point in S is at distance greater than $C \cdot r$ from x then the algorithm will always output $\perp$. The first item is trickier: assume there is a point y^* such that $\mathrm{dist}(x, y^*) \le r$. Then the probability that *none* of the k LSH hash functions "collide" is at most

$$\Pr[\,\forall t \in [k],\ g_t(x) \neq g_t(y^*)\,] \le (1 - p^\ell)^k \le \frac{1}{10}\,.$$

This is *the* bad event: if it does not happen, then at least one of the k hash tables will map x and y^* to the same bucket, and so at least one y will pass the test in Line 4 (at the very least, y^* would: another y might be returned earlier). □

This is encouraging, but we have a *lot* of parameters there. *How should we set k and ℓ?*

- From the second term in the expected query complexity, by setting
$$\ell := \frac{\log n}{\log(1/q)} \tag{54}$$
we get $q^\ell = 1/n$, and so the expected query time becomes $O(k\ell T + kd)$.

- This gives us $p^\ell = 2^{-\ell \log(1/p)} = n^{-\rho}$, and to satisfy the correctness condition
$$(1 - p^\ell)^k \le \frac{1}{10}$$
it is then sufficient (and necessary) to set
$$k = O(n^\rho) \tag{55}$$

"A wild sensitivity parameter ρ appears"

Check it!

This finally gives us the following (recalling that $\log n \ll d$ to simplify the query complexity):

Corollary 37.1. *With the settings of k, ℓ from Eqs. (54) and (55) and assuming $q, T = \Theta(1)$, the data structure for the "baby version of ANN" has space complexity $O(n^{1+\rho} d)$ and expected query complexity $O(n^\rho d)$.*

This is quite good: the space is only "mildly worse" than the necessary nd, and query complexity is *sublinear* in n, since $\rho < 1$!

But... this was just a "baby" version of our problem: this did not solve the ANN question itself! Thankfully, there is a 'simple" reduction from this version (where r is "hardcoded") to the general case, using, essentially, a binary search over r:

Theorem 38. *Suppose that, for every $0 < r \leq d$, we have a data structure for the "baby" version of ANN with scale parameter r and parameter $C > 0$, with space complexity $S(n,d)$ and expected query complexity $T(n,d)$. Then there is a data structure for ANN with parameter $2C$, space complexity $O(S(n,d)\log d)$ and expected query complexity $O(T(n,d)\log d)$.*

Proof. See tutorial. □

At this point, we know what to do if we are *given* an LSH family for the metric space $(\mathcal{X}, \text{dist})$ we care about – and the smaller the parameter ρ of that LSH family is, the better the query times and space complexity we will obtain are. Which brings a very natural question:

> Are there good LSH families for the metric spaces we care about?

We will give two examples, showing that the answer is (mostly) "yes."

Example: Hamming space

The first example is that of the *Hamming space of dimension d*, which, again, is just a fancy way of saying "the universe of d-bit strings where the distance between two $x, x' \in \mathcal{X} = \{0,1\}^d$ is the number of bits in which they differ."

What will be the LSH family? The simplest thing one could think of: $\mathcal{H}$ will just be the set of d functions

$$h_i \colon \{0,1\}^d \to \{0,1\}, \qquad 1 \leq i \leq d$$

where $h_i(x) = x_i$ just "hashes" a string x to its i-bit. That's all! One can then verify that, for every $1 \leq r \leq d$ and $C > 1$,

If $Cr > d$ the guarantee is trivially true.

- If $\text{dist}(x,x') \leq r$, then $\Pr_{h\sim\mathcal{H}}[h(x) = h(x')] \geq 1 - \frac{r}{d}$;
- If $\text{dist}(x,x') \geq Cr$, then $\Pr_{h\sim\mathcal{H}}[h(x) = h(x')] \leq 1 - \frac{Cr}{d}$;

and so this $\mathcal{H}$ is an (r, C, p, q)-LSH family for $p := 1 - \frac{r}{d}, q := 1 - \frac{Cr}{d}$, with size $|\mathcal{H}| = d$ and sensitivity

$$\rho = \frac{\log(1 - \frac{r}{d})}{\log\left(1 - \frac{Cr}{d}\right)} \approx \frac{1}{C}. \tag{56}$$

(as a side note, it has been shown that $\rho = \Theta(1/C)$ is the best one can get for the Hamming space.)

Example: Euclidean space (ℓ_2 distance)

What about the Euclidean space? Recycling is good, so we may want to use some of the ideas behind the JL Lemma to design a

good LSH Family. Fortunately, this is possible: pick a random vector $g \sim \mathcal{N}(0, I_d)$, and set

$$h_g(x) = \text{sign}(\langle g, x \rangle) \in \{-1, 1\} \tag{57}$$

(that is, pick a random gaussian vector g, and take the sign of the inner product between g and x). One can show[40] that, for every $r > 0$ and $C > 1$, this gives an LSH family with sensitivity parameter

$$\rho \leq \frac{1}{C}. \tag{58}$$

Interestingly, this is not optimal! For the Euclidean space, more involved constructions are able to obtain LSH families with sensitivity parameter $\rho = O(1/C^2)$.

[40] And you will in the tutorial.