8
Searching for Near Neighbors

Notes by Anupam Gupta

In this chapter we consider the setting where we are given a set of points in a metric space, and we want to create a data structure to find return points in the data set that are close to given query. This kind of data structure is not only at the heart of all search engines today, it underlies many other applications too. We discuss some of the many ways to solve this problem, some using geometry and others using hashing.

Formally, the setup for the near-neighbor (NN) problem is the following:

1. Fix a metric space (X, d).
2. Given a data set $S \subseteq X$ with n data points, we can preprocess it to create a data structure of "small" size, in "small" time.
3. Now, queries $q_1, q_2, \ldots \in X$ arrive over time, and for each query q we must return a point in S that is "close" to q.

For instance, the points of the metric space may be the set of all documents in our corpus, and we may have our favorite notion of distance between them (say the Jaccard distance, which you will see later in the lecture). Or we may consider the set of all strings over the alphabet $\{A, C, G, T\}$ with the edit distance between them.

8.1 Bringing Structure into Raw Data

If we are dealing with unstructured data, say a collection of documents, there may be very little structure to it. Typically the first step is to embed it into some kind of geometric space, which we can then use for our next steps. As an example, suppose we have a collection of text documents: how can we give it more geometric structure?

1. One approach is to view the documents as ASCII strings, with edit distance measuring the proximity between them. It is natural, and

useful for some contexts (say genome fragments) but difficult to interpret in others—how can we use this to perform web searches, for instance?

2. A different approach would be to view these ASCII strings as vectors and use Euclidean distance between them. Again, difficult to interpret. If we take the same document and swap the other of two paragraphs, the distance may change by a lot, but the meaning may change very little.

3. A different way is to view the document as a *bag of words*: represent it as a multiset containing the words in it (along with their multiplicity). This itself can be represented as a N-dimensional vector of numbers, where N is the number of words in our language. For example, the phrase "Time flies like an arrow, fruit flies like a banana" would be represented as

For a ballpark estimate, English roughly has $N = 100,000$ words.

$$(\ldots, \underbrace{1}_{\text{time}}, \ldots, \underbrace{1}_{\text{banana}}, \ldots \underbrace{2}_{\text{flies}}, \ldots, \underbrace{2}_{\text{like}}, \ldots \underbrace{1}_{\text{arrow}}, \ldots).$$

   Typically we drop all the *stop words* like articles and other common but relatively uninformative words from it.

4. There are some natural problems with such a bag-of-words approach. Taking two copies of the same document can double the vector magnitude, but not increase the information. Also, even if stop-words are dropped, other less meaningful words can form most of the vector mass. One simple fix to both is the TF/IDF (total frequency/inverse document frequency) mapping. In this each word count (for the word w, say) is first divided by the total number of words in the document (the *total frequency* part) and then multiplied by

$$\log\left(\frac{\#\text{ documents in corpus}}{\#\text{ documents containing } w}\right),$$

   the (log) inverse document frequency.

Figure 8.1: Shingling text: these shingles consist of three words.

5. This entire approach ignores the structure of the sentence, which can be very important in most languages, like in English. To handle this, one approach is to use *shingling*. In this, a document is broken not into a bag of words, but into a bag of shingles, where each shingle consists of a sequence of some k words. (And we can use the re-normalization ideas used above.) Of course, this gives an embedding into N^k dimensional space. These resulting vectors are very sparse, and hence the sets of interest in most near-neighbor problems are relatively small (say of size 10^6 to

10^9, a million to a billion vectors) which lie in some very high-dimensional space, and each of which is very sparse (with relatively few entries being non-zero).

This idea of shingling can also be done for images, to turn them into vectors that be searched for.

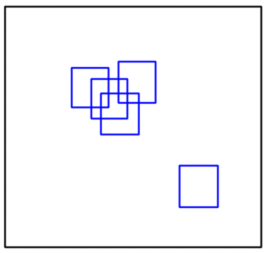

Figure 8.2: Shingling images: say shingles of size 10×10.

8.2 The (Geometric) ε-Near-Neighbors Problem

Henceforth, we will assume that our dataset S lie in $\mathbb{R}^k$ for some dimension k. The distance $d(x, y)$ between two points x, y is measured either with respect to the standard Euclidean distance

$$\|x - y\|_2 := \sqrt{\sum_{i=1}^{k} (x_i - y_i)^2},$$

or else they are measured in the ℓ_1 or Manhattan taxicab distance

$$\|x - y\|_1 := \sum_{i=1}^{k} |x_i - y_i|.$$

Let us formally define ***the ε-near-neighbors problem*** as the following: given dataset S of size $|S| = n$, and a parameter $\varepsilon > 0$, preprocess S so that given any query point $q \in \mathbb{R}^k$, we can quickly return an "answer" point $a \in S$ such that

$$d(q, a) \leq (1 + \varepsilon) \min_{x \in S} d(q, x).$$

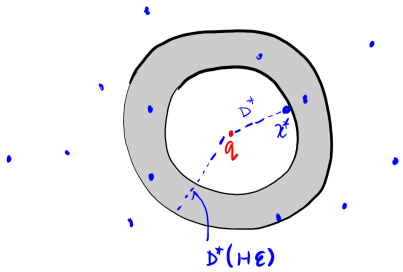

Figure 8.3: The ε-NN problem. Returning any point within the gray region (including x^*) is a valid solution.

The measures of goodness will be (a) the *preprocessing time and space* and (b) the *query time*. There is a natural tension between these two measures, since the more we compute during the preprocessing, the less we may have to do at query-time.

8.2.1 Strawman #1: Linear Search

The simplest approach is to not do any preprocessing at all. Then given a query point $q \in \mathbb{R}^k$, we just run over the n points, and return the closest point to it.

1. There is no preprocessing, and the space usage is $O(nk)$ numbers—the k coordinates for each of n points.
2. The query time is also $O(nk)$.

Clearly the space usage and preprocessing time is excellent, and we get the absolute nearest point to q. But the query time is nothing to write home about. Of course, it's a start: let's see if we can do better with some more work.

8.2.2 *Strawman #2: k-d trees*

This data structure should be thought of as a natural k-dimensional analog of imposing a binary-tree structure over a line. The idea is to subdivide space into boxes with fewer and fewer points (or regions in general), until each box contains a single point.

This was developed by Jon Bentley, who was on the CMU faculty in the 1970s-80s.

Specifically, we can assume we start inside a large enough box. For each box B that contains more than one point, we choose one of the k dimensions, say i, find the median of the i^{th}-coordinates of the points inside B, and split B into two boxes using an axis-parallel hyperplane at this median. (See the figure to the right.)

The choice of the splitting coordinate at each step is up to the user: it can be done in a round-robin fashion, or based on the spread of the points.

The two children of B now have half the number of points that B has, so the depth of the tree is $\log_2 n$. Each box in the data structure now stores the points within it, and also the identity of the points with the maximum and minimum coordinates along each of the k coordinate axes. (These will be needed for the following query procedure.) Overall this requires only $O(nk \log n)$ bits of space.

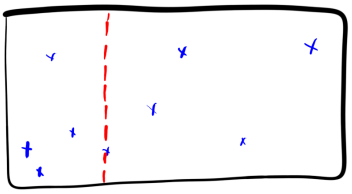

Figure 8.4: Partitioning the box through the median of one of the coordinates.

Now given a query q point, we start at the root box and go down the tree to a leaf box by choosing the child containing q at each step. Let x be the point in this leaf box: this is our initial guess for the nearest neighbor. Of course, x may not be the closest point, so we try to refine our guess as we go back up the tree. Suppose a box B has two children B_ℓ and B_r obtained by splitting along the first coordinate (with B_ℓ to the left of B_r), and we have just come returned from B_ℓ with a current guess of x. If the box B_r contains a closer point y, then the first coordinate of y must satisfy $y_1 \leq q_1 + d(q, x)$. Hence, we can ignore B_r if the minimum first coordinate value among its points is larger than $q_1 + d(q, x)$, else we try to find a closer point to q within it. This query can spend $\Omega(n)$ time in the worst-case, but empirically it seems to do well on real-world data sets. (The rule of thumb is that the query time tends to be closer to $\approx 2^k \log n$ on most "well-behaved" data sets.) This is fine for small k, but even for $k = 100$ this kind of runtime can be prohibitive, and is often called the ***curse of dimensionality***!

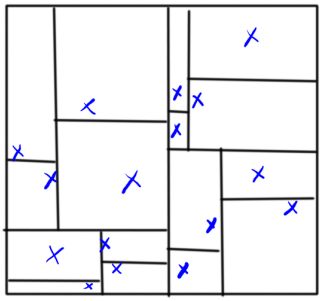

Figure 8.5: The recursive partitioning.

Moreover, there are there are variants of k-d trees (e.g., the *random projection trees* of Dasgupta and Freund) that can give guarantees based on the "intrinsic dimension" of the data.

This is another phrase coined by Richard Bellman.

8.3 *Optional: A Provably Good Variant of k-d Trees*

It is possible to alter k-d trees to give a provable worst-case bound of approximately $O(1/\varepsilon)^k$ on the query time. Note that this is still exponential in the dimension, and hence is relevant only for small values of k—but at least we are ensured that we do not spend $\Omega(n)$ time in the worst case.

The idea will again be to split the data repeatedly, but now we split the side-lengths of the box in half, instead of splitting the num-

ber of points in half. To make this precise, some notation will be useful: let $d_{\max}(S)$ be the largest distance between any two points in S, and let $d_{\min}(S)$ be the smallest non-zero distance. We define the *aspect ratio* of the point set S to be

$$\Delta(S) := \frac{d_{\max}(S)}{d_{\min}(S)}.$$

Here is the construction algorithm for the k-d tree:

1. By translating the points, assume that all points lie in the box $[0, d_{\max}]^k$. Now break this box into 2^k smaller boxes of half the side-length. Let these 2^k boxes be children of the original bounding box, and hence siblings of each other. For each resulting box that has at more than one point in it, break it into 2^k boxes of half its side-length. And repeat this process. Note that each time we perform this operation, the side length halves.

2. For each of the resulting boxes that contains at least one point, choose any point within it as its *representative*. For simplicity, ensure that the representative of any box is also the representative of its box among the children boxes.

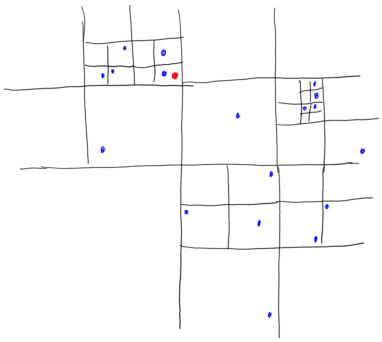

Figure 8.6: Partitioning space in smaller boxes.

Since each leaf of the tree contains a distinct point from S, each point in S is a representative of the leaf box that contains it. But as we go up the tree, the set of representatives becomes smaller, until there is a single representative at the root.

Before we move on, let's bound the total space usage of this data structure. Since we shatter a box only when it has multiple points, the side-length of the leaf boxes must be at least $\frac{d_{\min}}{2\sqrt{k}}$. (To see why, think about why its parent box was split into smaller boxes!) The side-length starts off at $d_{\max}$ and halves each time, so the height of this tree is at most

$$H := \log\left(\frac{d_{\max}}{d_{\min}/(2\sqrt{k})}\right) = \log\Delta(S) + \log k + O(1).$$

Moreover, each level contains at most n points as representatives, so the total space usage is at most $O(nk\,H)$ bits. This is more than the naive approach only by the (small-ish) logarithmic factor, which is great.

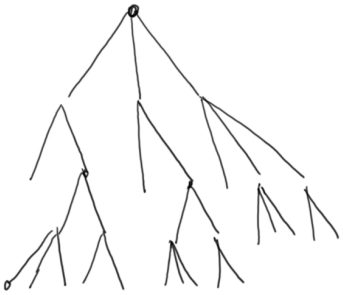

Figure 8.7: The tree structure that arises from the partitioning.

8.3.1 *The Query Procedure and Query Time*

Here's the query procedure at a high level: we keep track of a list L of ℓ close points seen thus far, where ℓ is a parameter we will choose soon. As we go down the tree, we update this list, and when we

reach the leaves, we output the closest point among the ℓ points still remaining in L.

The idea is simple: this list L starts off with the representative point of the root box, which is a weak guess for the near neighbor. Now in each round, we descend one level of the tree. Suppose at height h we have some set L of points in hand. These are the representatives of some boxes at this height. We look at the 2^k children boxes (at height $h-1$) of each of these boxes, and their representatives. Among these $\leq |L|2^k$ representatives, we keep the $\leq \ell$ points closest to q, and continue. When we reach the leaves, we return the best point among the those in L. Here's the formal procedure:

Algorithm 14: Provably Good k-d Tree

```
14.1 L ← { the representative point for the root box }.
14.2 for height h ← H ... 1 do
14.3 |  L' ← ∅
14.4 |  for points x ∈ L do
14.5 |  |  B ← box containing x at height h
14.6 |  |  L' ← L' ∪ { representatives of all 2^k children boxes of B}
14.7 |  L ← the ℓ points in L' that are closest to q
14.8 return a ← arg min{d(q, x) | x ∈ L}.
```

The query time is at most $H\ell 2^k$ by construction, since we spend $\ell 2^k$ time per level. The question is: how large do we need to make H to ensure that the output is a $(1+\varepsilon)$-approximate near-neighbor? It is possible to prove the following theorem, which we omit for now.

Theorem 8.1. *Setting $\ell := (4/\varepsilon)^k$ ensures that the answer is always a $(1+\varepsilon)$-approximate near-neighbor. This makes the query time*

$$H\ell 2^k = O(1/\varepsilon)^k \log \Delta(S).$$

Note the exponential dependence in the dimension k, and moreover the (logarithmic) dependence on the aspect ratio of the data set. One can combine the ideas from the basic k-d trees and this one, to alternate splitting according to (a) the number of points in the box and (b) the side-length of the containing box. This can reduce the dependence to $O(1/\varepsilon)^k \log n$, which still falls prey to the curse of dimensionality. In the next section, we see a hashing-based approach that avoids this exponential dependence, at the cost of a large amount of preprocessing.

Work of Krauthgamer and Lee, among others, give variants of this construction that can reduce the dependence further to be exponential only in the "intrinsic dimension" of the data set, but the exponential behavior remains.

8.4 Fixing a Distance Scale

Let's consider a slightly restricted version of the ε-NN problem: given a data set S *and a fixed distance* r, build the data structure NN_r to quickly answer queries of the form: *Given query point* q*, return a point* $a \in S$ *such that* $d(q,a) \approx r$*, if one exists*. Formally, we want the following guarantees from our data structure:

1. If there exists a point $x \in S$ with $d(q,x) \leq r$, it must return a point a with $d(q,a) \leq (1+\varepsilon)r$,
2. If every point $x \in S$ has $d(q,x) > r$, it can either return a point a with $d(q,a) \leq (1+\varepsilon)r$, or say `No`.

We call this problem ***the*** (ε, r)***-near-neighbor***, or the (ε, r)-NN problem.

The next theorem shows how to solve the original ε-NN problem using this "fixed scale" version, using the idea of *doubling search*. The idea itself is very simple but versatile: *solve the problem for all powers of* $(1+\varepsilon)$ *in some relevant range of values, and choose the best answer*. One needs a bit of care to get the details right, but the idea is just that simple.

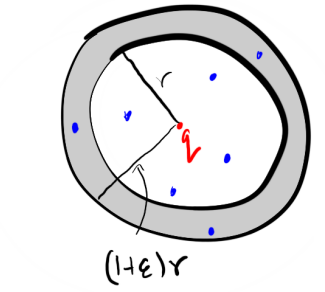

Figure 8.8: The (ε, r)-NN problem. Returning any point within the inner circle or the outer circle is is a valid solution.

Theorem 8.2. *For any* $\varepsilon \in (0,1)$ *and dataset* S *of* n *points, a solution to* (ε, r)*-NN using* $A = A(n,\varepsilon)$ *preprocessing time and space and* $Q = Q(n,\varepsilon)$ *query time can be used to give a solution to* (3ε)*-NN with* $O\big(A \frac{\log \Delta(S)}{\varepsilon}\big)$ *preprocessing time and space and* $O\big(Q \frac{\log \Delta(S)}{\varepsilon}\big)$ *query time.*

Proof. The combined data structure is the following: let $d_{\min}$ and $d_{\max}$ be the smallest non-zero distance and largest distance between points in S. Then for each value of r of the form

$$\frac{d_{\min}}{4}, \frac{d_{\min}}{2}(1+\varepsilon), \frac{d_{\min}}{4}(1+\varepsilon)^2, \cdots, \frac{d_{\min}}{4}(1+\varepsilon)^i, \cdots$$

until the value of r exceeds $\frac{2d_{\max}}{\varepsilon}$, we maintain a data structure for the associated (ε, r)-NN problem. When a query q arrives, we present it to each of these data structures and return the best of all the answers. Since there are

$$\log_{(1+\varepsilon)} \frac{2d_{\max}/\varepsilon}{d_{\min}/4} \approx O\Big(\frac{\log \Delta(S)}{\varepsilon}\Big)$$

values of the exponent i that satisfy the conditions above, the bounds on the preprocessing time/space and query time follow.

It remains to show that the answer satisfies the requirements of the (3ε)-NN problem. Indeed, suppose the closest point in S to query q is at distance D. Suppose D lies between $\frac{r}{1+\varepsilon}$ and r for one of the

We use that

$$\log_{1+\varepsilon} X = \frac{\ln X}{\ln(1+\varepsilon)}.$$

Now $\ln(1+\varepsilon) \in [\varepsilon/2, \varepsilon]$ for $\varepsilon \in [0,1]$, so

$$\log_{1+\varepsilon} X = \Theta\Big(\frac{\ln X}{\varepsilon}\Big).$$

choices of r we used above. Then the associated (ε, r)-NN data structure will return a point at distance at most

$$(1+\varepsilon)r \leq (1+\varepsilon)^2 D \leq (1+3\varepsilon)D$$

using $\varepsilon \leq 1$ here.

Edge cases here, flesh these out. Else if D is larger than the largest value of r in our collection, then any point in S an ε-NN. Or if D is smaller than $\frac{d_{\min}/4}{1+\varepsilon}$, then the answer for the $r = d_{\min}/4$ data structure must be the exact closest point. □

8.5 *Locality Sensitive Hashing*

The idea of regular hashing is this: if $x \neq y$, we want the probability of collisions to be tiny. Formally,

$$\begin{aligned} x = y \quad &\implies \quad \Pr[h(x) = h(y)] = 1 \\ x \neq y \quad &\implies \quad \Pr[h(x) = h(y)] \approx 0. \end{aligned}$$

The first line is trivial, but we're saying it explicitly to draw the parallels with LSH. The idea of ***locality sensitive hashing (LSH)*** is a more nuanced version of the above, one that takes distances into account. We want:

$$\begin{aligned} d(x,y) \text{ close} \quad &\implies \quad \Pr[h(x) = h(y)] \geq \text{large} \\ d(x,y) \text{ far} \quad &\implies \quad \Pr[h(x) = h(y)] \leq \text{small}. \end{aligned}$$

So it's not just equality and inequality any more, we care about the collision probabilities of keys that are close and far. In words,

- far items fall in same bin with low probability, and
- close items fall in the same bin with high probability.

A bit more formally and quantitatively: given a values r, ε, we want

$$\begin{aligned} d(x,y) \leq r \quad &\implies \quad \Pr[h(x) = h(y)] \geq \text{large } p_{\text{close}} \\ d(x,y) \geq (1+\varepsilon)r \quad &\implies \quad \Pr[h(x) = h(y)] \leq \text{small } p_{\text{far}}. \end{aligned}$$

For these constructions to be useful, $p_{\text{close}} \approx 1$ and $p_{\text{far}} \approx 0$, the former ensuring few false negatives, and the latter ensuring few false positives.

8.5.1 *The Plan*

Here's how almost all the constructions for LSH will proceed:

1. First, we get a very weak result, a hashing result where the gap between p_{close} and p_{far} is very tiny. Say

$$p_{\text{far}} \approx p_{\text{close}} - \varepsilon.$$

2. Next, we "amplify" this gap. To reduce the probability of far items colliding, we perform a process we call *parallel repetition*. Given a hash family H, consider the hash family H^t which is obtained by choosing t hash functions independently, and using all these hash values in parallel. Namely, the new hash family is $H' := H^k$, and the new hash function h' maps

$$h'(x) = \big(h_1(x), h_2(x), \ldots, h_t(x)\big),$$

where each of the h_i's are independently chosen. Since colliding in h' requires us to collide in all the t coordinates, this drives the probability of far collisions from p_{far} to $(p_{\text{far}})^t$. We choose t to set this to $1/n$, say.

The terms *parallel repetition* and *serial repetition* are typically not used in the context of LSH. But we find it useful: giving the technique an evocative name makes it easier to remember.

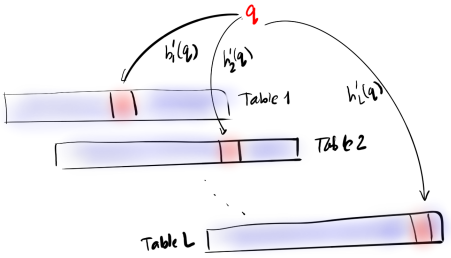

Figure 8.9: Serial repetition.

3. Unfortunately, the probability of close collisions has also dropped, from p_{close} to $(p_{\text{close}})^t$. So we "amplify" this as well, using what we call *serial repetition*. We simply take L independent copies of the data structure. By linearity of expectation, if x, y are close, they will collide in an expected $L \cdot (p_{\text{close}})^t$ number of these copies. If we set this value close to 1, then the probability of collisions in none of these L copies is small, by Markov's inequality.

And that's it.

Note that steps (2) and (3) are completely generic. They show a "boosting"-type result—given an LSH with a tiny gap between p_{close} and p_{far}, we can mechanically boost the gap. Of course, a smaller initial gap means t and hence L will eventually be large, which will cost us in both space usage and query times.

8.5.2 LSH for Hamming Metrics

The above discussion was abstract, so let us see this in the context of LSH for Hamming distances. Suppose we have points in the set $\{0,1\}^k$—i.e., k-bit strings—equipped with the Hamming metric. And suppose we want to distinguish distance $\le r$ from $\ge 2r$, which means $\varepsilon = 1$. How would we do this? We follow the recipe above.

The *Hamming* metric counts the number of bit positions where the two bit strings differ.

1. First, here's a simple LSH. Pick a uniformly random bit position $i \in \{1, 2, \ldots, k\}$, and let $h(x) = x_i$. Note that

$$d(x,y) \le r \quad \Longrightarrow \quad \Pr[h(x) = h(y)] \ge 1 - \frac{r}{k} =: p_{\text{close}}$$
$$d(x,y) \ge 2r \quad \Longrightarrow \quad \Pr[h(x) = h(y)] \le 1 - \frac{2r}{k} =: p_{\text{far}}.$$

2. Next, we do parallel repetition (which corresponds to choosing t bit-positions, with repetition. We want to set t so that $(p_{\text{far}})^t \leq 1/n$. Since
$$\left(1 - \frac{2r}{k}\right)^t \leq e^{-(2r/k)\cdot t},$$
we can set
$$^{2r}/_{k}t = \ln n \implies t = \frac{k}{2r} \ln n.$$

3. Of course, now the close collision probability has become
$$\left(1 - \frac{r}{k}\right)^t \approx e^{-(r/k)\cdot t} = e^{-\frac{1}{2}\ln n} = \frac{1}{\sqrt{n}}.$$
So we do "serial repetition": we make $L = \sqrt{n}\ln n$ independent copies of the data structure. Then the probability of a close pair colliding in at least one of these is
$$1 - \left(1 - \frac{1}{\sqrt{n}}\right)^L \approx 1 - e^{-L/n} = 1 - \frac{1}{n}.$$

This is great. We started with a very weak LSH scheme, and by using parallel and serial repetition, we get a construction where (a) the probability of a far pair colliding in any fixed table is $\frac{1}{n}$, and (b) the probability of a close pair colliding in at least one table is at least $1 - \frac{1}{n}$. We used $L \approx \sqrt{n}$ tables, each obtained by projecting down to $\approx (k/r)\log n$ random bits.

8.5.3 (ε, r)-NN using Locality Sensitive Hashing

To complete the story, let's just recap how this solves the (ε, r)-NN for Hamming metrics, and also discuss the space and query time for the construction. The construction is the following:

1. Maintain L different tables: in each table, choose random t coordinates of the bit-vector (with replacement, say). For each of the $\{0,1\}^t$ possible values in these t positions, keep track of the points in S that have those values. E.g., if the t positions are $4, 5, 19, 21$, then the entry in the table corresponding to 0110 will contain a list of all the elements $x \in S$ that have
$$x_4 = 0, x_5 = 1, x_{19} = 1, x_{21} = 0.$$
Naïvely we would maintain a table of size 2^t. Since $t \gg \log n$, this is much bigger than n, and so is very wasteful. How should we reduce the space usage? Conventional hashing! Using a standard dictionary data structure, we can store this information using only $O(n)$ space per table. (Check that you see how, else please ask!) So that's a total of $O(Ln)$ space, and a similar preprocessing time.

2. When a query q arrives, for each of the L tables we look up whether any of the data points in S agree with q on the chosen t coordinates. In the example above, this requires looking up the table location $q_4q_5q_{19}q_{21}$ and returning the first point among these at distance at most $(1+\varepsilon)r$.

   Now for the probability of success: if there exists a point x with $d(q,x) \le r$, our choice of t and L means $h'(x) = h'(q)$ for at least one of the tables, with probability $1 - 1/n$, so the probability of ***false negatives*** is small. Moreover, any points x with $d(q,x) > (1+\varepsilon)r$ (i.e., *false positives*) have a $1/n$ change of colliding with q in any fixed table, so we need to look at only $O(1)$ items per table before succeeding. Hence, this takes $O(1)$ expected number of distance computations per table, and hence $O(Lk)$ time overall.

   We can summarize this discussion as a theorem:

Theorem 8.3. *The LSH data structure for Hamming metrics given above uses $O(n^{1.5}k \operatorname{poly}\log n)$ space and preprocessing time, has $O(\sqrt{n}k)$ expected runtime, and is correct with probability at least $1 - 1/n$ on any query.*

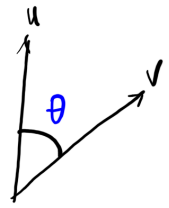

Figure 8.10: A random hyperplane through the origin separates the two vectors with probabiltiy θ/π.

8.5.4 LSH for Vectors in Euclidean Space

If we have a different metric, we can replace the simple basic LSH (which was projecting onto a single bit for the Hamming metric case) by a "good" LSH for that metric space. For instance, given unit vectors in Euclidean space, one commonly used metric is the *cosine distance*: the distance between vectors u, v is the cosine of the angle between them. A little thought shows that this is nothing but the inner product

$$\langle u, v \rangle .$$

One good LSH for this distance is to pick a uniformly random half-space

$$Z := \{x \mid \langle a, x \rangle \ge 0\}$$

through the origin, and define $h(v) = 1$ if $v \in Z$, and 0 otherwise. One can check that

$$\Pr[h(u) = h(v)] = \theta/\pi.$$

8.5.5 LSH for Sets via the Jaccard Distance

We can use the LSH idea even for unstructured documents, without doing the steps in §8.1 to map them to vectors. Given two sets A, B, the Jaccard distance between them is

$$d_{\text{Jacc}}(A,B) := 1 - \frac{|A \cap B|}{|A \cup B|}.$$

This distance takes values between 0 and 1, and can be checked to be a metric.

An LSH for this metric is the famed *minhash* hash family: here we pick a uniformly random permutation of all the elements of the universe, and map a set A to the element in A that comes *first* in this permutation. Note that

$$h(A) = h(B)$$

when then have the same first element. This happens with probability exactly

$$\frac{|A \cap B|}{|A \cup B|} = 1 - d_{\text{Jacc}}(A, B).$$

Again, we can take this basic hash, and then use parallel and serial repetition to get an excellent data structure out of it.

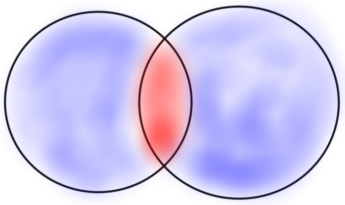

Figure 8.11: The Jaccard *similarity* of the two sets is the size of their intersection, to that of their union. The distance is one minus their similarity.

8.6 Takeaways

Some takeaways to keep in mind:

1. We saw the classic k-d tree data structure, which is based on spatial partitioning. It is easy, popular, and works pretty well. However, it suffers from the *curse of dimensionality*: the space and time to search grows exponentially in the dimension of the data.
2. We then saw some approaches to extend hashing from unstructured settings ("two elements map to the same place (with high probability) if and only if they are identical) to the settings with an underlying metric space: "*two elements map to the same place (with high probability) if and only if they are similar/close*". This does not have the same "curse of dimensionality" problems, and hence has tremendous consequences for searching in these metric spaces.
3. The idea was very clean: first reduce the problem to one distance scale. That is, just figure out whether the closest point is at most D or at least $(1 + \varepsilon)D$. And then use binary search or doubling search.
4. For a fixed distance scale, the LSH idea is to come up with a hash function with a (slightly) higher probability of two close items colliding than for two far items colliding. Then use parallel repetition to reduce the probability of false positives. Finally, use serial repetition to reduce the probability of false negatives.
5. We can use this idea for many different metric spaces: Hamming, vectors, Jaccard, Euclidean, and many others. Please check out other resources for details on the things we did not cover, if you are interested. LSH is a popular and widely used approach by now, and works well both in theory and in practice.

We can view LSH as a special-purpose "dimension reduction" technique. In the coming chapters, we will see some general approaches to reduce the dimension of a dataset, some based on randomness and others on linear algebra.