7 Approximate Nearest Neighbors

When we have a large data set with n items where n is large (think $n = 1{,}000{,}000$) then we discussed two types of questions we wanted to ask:

(Q1): Which items are similar?

(Q2): Given a query item, which others are similar to the query?

For **(Q1)** we don't want to check all roughly n^2 distances (no matter how fast each computation is), and for **(Q2)** we don't want to check all n items. In both cases we somehow want to figure out which ones might be close and the check only those.

We discussed LSH as a general technique to answer **(Q1)**, but it did not really answer how to solve **(Q2)** since it still needed to at least look at each element once.

The key to solving this problem is *pre-computation*. We will first build a data structure D_P on the data set P. Then we can ask D_P questions and expect fast answers. For instance, given a query object q we can ask for all points within distance r of q in P with notation $D_P(q, r) = \{p \in P \mid \mathtt{d}(p, q) \leq r\}$. Or we can ask for the nearest neighbor of q in P denoted $\phi_P(q) = \arg\min_{p \in P} \mathtt{d}(p, q)$.

With LSH, this preprocessing was to precompute all hash bins for each object in P. Then for a query point q, compute its hash bins, and check the exact distance to those that fall in all bands in at least one set of bands. But LSH is not the only way, and other techniques typically work better in "low" and "medium" dimensions.

7.1 High-dimensional Euclidean Data

We begin with a few examples of when you do actually have high-dimensional data.

7.1.1 Word Vectors

We have already seen how to map text documents to high-dimensional vectors. There were two ways; the bag-of-words approach directly generated a high-dimensional vector. It is typically normalized so that it is a unit vector, and so the cosine distance is most appropriate. If the vector v is normalized so $\|v\|_1 = 1$, then the Euclidean distance may be appropriate; although its more common to use information distances such as the KL-divergence, or the Hellinger distance. One could also use k-grams, and then use m min-hashes to generate an m-dimensional vector. But the values of each coordinate do not have meaning unless they match, so the Hamming distance is most appropriate.

Now we describe a newer approach which takes a very large corpus of text (say all of Wikipedia) and generates a vector representation for each word. And in this setting, the Euclidean distance is most appropriate. This is useful for many more detailed natural language applications. For instance synonyms typically have vectors close to each other, and in general all verbs are closer to each other than all nouns, or all adjectives. At a finer level, these embeddings usually reveal more structure like there is a direction that captures properties like gender (from man to woman, etc). Moreover, one can sometimes solve analogy questions. The most famous example is to take the representative vectors for king (v_{king}), queen (v_{queen}), and man (v_{man}), and then consider the vector manipulation

$$\hat{v} = v_{\mathsf{king}} - v_{\mathsf{queen}} + v_{\mathsf{man}}.$$

Impressively, the nearest vector to $\hat{v}$ is often v_{woman}, the vector representation of woman.

There are a few ways to create these embeddings, and the most common strategies ultimately, and a bit opaquely, use neural nets to find a representation. However, one of the most common approaches GloVe, starts with an understandable vector called a PPMI (positive pointwise mutual information) vector for each word. This is quite high-dimensional, often too high to easily work with, but the ultimate embeddings is usually into a 300-dimensional space $\mathbb{R}^{300}$. And for instance, the Euclidian representation is essential if the analogy vector algebra is expected to make sense.

PPMI vectors. We start with defining a fixed vocabulary we care about; lets say the $m = 100{,}000$ most common words. Then we define a window around each word in the text corpus (with a total of N words), lets say 3 words on either side. Then for each word, lets say apple, we keep a count of how often it appears n_{apple} and the number of times each other vocabulary word co-occurs with it within a window (i.e., continuous bag of words). For word i, define its number of cooccurrences with word j as $b_{i,j}$ Let these counts be a vector $b_i \in \mathbb{R}^m$ for word i with the jth coordinate as $b_{i,j}$.

Now define the probability of a word i as $p(i) = n_i/N$, and the co-occurence probability as $p(i,j) = b_{i,j}/N$. Then the pointwise mutual information of i and j as $\log \frac{p(i,j)}{p(i)p(j)}$. Ultimately we create a vector $v_i \in \mathbb{R}^m$ for each word, which we enforce each coordinate to be non-negative. The jth coordinate is then the PPMI between i and j, defined as

$$v_{i,j} = \max\left(0, \log \frac{p(i,j)}{p(i)p(j)}\right).$$

7.1.2 Images and SIFT Features

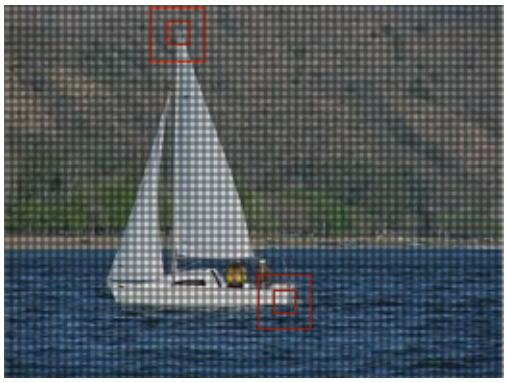
An image is represented as a list of pixels arranged in a grid. In deeper analysis (as we discuss below) which pixels are near each other is important. However, generically, we just need to make sure we correctly correspond pixels across images. Thus for say any 400×300 images, there are $400 \cdot 300 = 120{,}000$ pixels, and we can arrange them in an ordered list.

Then each pixel is either a single grey-value (there are typically 256 shades of grey, or are three *rgb*-values denoting the pixel's component in red, green, and blue. For simplicity we can assume black and white, otherwise there are just 3 times as many values. Now each pixel p_i corresponds to a value in $[0, 1]$, where the actual stored number, say $x = 52$ represents $52/256$. Then the entire image is a vector, where we can pretend it takes on continuous values, and it is a vector $p = (p_1, p_2, \ldots, p_{120{,}000}) \in \mathbb{R}^{120{,}000}$.

Such a representation is useful for a few tasks directly. It can easily allow one to understand the distribution of colors in images (e.g., blue ones are typically of the sky or ocean). If the images are pre-aligned (for faces, say mug shots, or restricted angle only profile pictures), then the corresponding pixels have corresponding meaning. Otherwise, there is often a pre-processing step to align images before the Euclidean analysis.

Another approach are more advanced features, which often are also high-dimensional and Euclidean. Of the hand-engineered ones, the SIFT feature are the most popular. However, deep learning techniques have surpassed these hand-engineered features in the last decade. In this setting, the deep neural net essentially transforms a picture, and at one of the final layers stores stores a vector of values to represent the image: that is again, there is an inherent high-dimensional representation. Perhaps unremarkable, if one examines the structure within the most popular deep nets for images, the representation a few layers in is not too different from what a SIFT feature represents.

Engineered (SIFT) features. A SIFT (scale-invariant feature transform) data point is 128-dimensional, and is often treated as if it is in $\mathbb{R}^{128}$. It is used to identify "interesting" features in images, and has been **enormously** popular and useful. There are others that perform similarly (e.g., SURF, BRISK) and sometimes it is useful to concatenate these to create even higher dimensional vectors.



N1	N2	N3
N8	X	N4
N7	N6	N5

Figure 7.1: (Left) An example images, with grid illustrating the notion that an image is just a set of pixels. There are actually many more pixels than drawn here. (Right) Shows a pixel X and its 8 neighboring pixels.

For SIFT features, we care about the neighborhood of each pixel. That is, each pixel has 8 neighbors as illustrated in Figure 7.1(Right). Then we measure the gradient in these 8 directions (a rough approximation of this can be seen as)

$$\langle N1 - X, N2 - X, N3 - X, N4 - X, N5 - X, N6 - X, N7 - X, N8 - X \rangle$$

and this corresponds with 8 dimensions. The gradient of images tends to be a much more reliable entity than the absolute pixel value. And the gradient is supposed to remove the *shift*.

Then rotate the neighbors so the first one has the maximum gradient value. For instance if this is $N6$, then set the gradient at location i to be $(i - 5) \mod 8$. The actual transform is a bit more subtle. This is supposed to remove the *rotation*.

Finally we need to remove the *scale*. We do this by considering a 4 x 4 grid of pixels around each pixel X. For each one of these 16 locations (at various scaled distance from X) we store one of the 8-valued *shift and rotation* invariant gradient vectors. The combination of these 16 vectors provides $8 \times 16 = 128$ values. And this set of values is the SIFT vector at X.

For an image, there are specialized ways to only keep the "interesting" ones which appear around corners. These are done through finding local maximum of the difference between Gaussian blurring at similar scales. Also depending on the scale used, this may suggest adjusting the scale of a "pixel" in the above construction.

7.2 Approximate Nearest Neighbors

We now answer question **(Q2)**. We will focus on a data set $P \in \mathbb{R}^d$ where $|P| = n$ is quite large (think millions). We will focus mainly on Euclidean distance $\mathbf{d}(p, q) = \|p - q\|_2$.

Given a query point $q \in \mathbb{R}^d$ then $\phi_P(q) = \arg\min_{p \in P} \mathbf{d}(p, q)$ is the *nearest neighbor of* q. For $d = 1$, this is possible with about $\log n$ time using a balanced binary tree on P (sorted on the one-dimensional value of each $p \in P$). For really large data sets, a B-tree works better with the cache. It splits each node of the tree into B pieces and each leaf has at most B elements. B is set as the block size of the cache.

7.2.1 Small Dimensions ($d = \{2, 3\}$)

For general d, this can be done exactly by building a *Voronoi diagram* on P; $\mathsf{Vor}(P)$, it is a decomposition of $\mathbb{R}^d$ into n cells, and each cell v_p is associated with one point $p \in P$ so that each $q \in v_p$ has $p = \arg\min_{p' \in P} \mathbf{d}(p, q)$. The *complexity* of $\mathsf{Vor}(P)$ is the numbed of boundary sections (d'-dimension facets for $0 \leq d' \leq d$) needed to describe all cells in $\mathsf{Vor}(P)$. The complexity of the Voronoi diagram can be as large as $\Theta(n^{\lceil d/2 \rceil})$. This means for $d = \{1, 2\}$ it is linear size, but it grows exponentially in size as d grows.

In many practical cases, the size of the Voronoi diagram may be closer to linear in $d = \{3, 4, 5\}$, but very rarely in even higher dimension.

So in low dimensions, $d = \{2, 3\}$ data structures (based on $\mathsf{Vor}(P)$) can be constructed to find the nearest neighbor in about $\log(n)$ time.

7.2.2 Approximate Nearest Neighbors

Often the exact nearest neighbors are unnecessary. Consider many points P very close to the boundary of a circle (or higher dimensional sphere), and a single query point q at the center of the circle (or sphere). Then all points are about the same distance away from q. Since the choice of distance is a modeling choice, it should often not matter which points is returned.

So we introduce a parameter $\varepsilon \in (0, 1]$, and we say a point p such that $\mathbf{d}(p, q) \leq (1 + \varepsilon)\mathbf{d}(\phi_P(q), q)$ is an *ε-approximate nearest neighbor of q*. There can be many such points, and finding one of these can be dramatically simpler and faster (especially in practice) than finding the exact nearest neighbor. This becomes essential in high dimensions.

7.2.3 Medium Dimensions ($d \in [3 - 12+]$)

In medium dimensions (usually between $d = 3$ and say $d = 12$, but miles may vary), a hierarchical spatial decomposition can be used to quickly find approximate nearest neighbors. Think of all points being in a box $B = [0, 1]^d$ (assume all points are in $[0, 1]^d$). In each level of the hierarchy, the box is divided into two (or more) smaller boxes. This hierarchy reaches the bottom (the leaf of the tree) when there is at most 1 point (or often more efficiently some constant number like 10 or 20 points) in the box.

These structures are queries with a point q as follows.

- First, find the leaf B_ℓ that contains q. Find $\tau_\ell = \phi_{P \cap B_\ell}(q)$. This provides an upper bound on the distance to the nearest neighbor. Specifically any box B_i such that

$$\min_{x \in B_i} \mathbf{d}(x, q) \geq \tau_\ell/(1 + \varepsilon) \tag{7.1}$$

  can be ignored.
- Second, walk up the hierarchy, visiting sibling boxes B_i to the current one visited B_ℓ. If B_i satisfies equation (7.1) then we continue up the hierarchy. Otherwise find $\tau_i = \phi_{P \cap B_i}(q)$ and if $\tau_i < \tau_\ell$, set $\tau_\ell \leftarrow \tau_i$ and move up the hierarchy. (Note that $\tau_i \phi_{P \cap B_i}(q)$ can be answered with an approximate query using the hierarchy structure.) Denote the parent node ℓ, as the new active node.
- Stop when the root is reached and $B_\ell = B$.

The key structures most often used are

- **kd-tree:** It divides a box, by alternatively splitting each dimensions. So if there are $d = 3$ dimensions, then the first split is on the x-dimension, the second level on the y-dimension, the third on the z-dimension, and then the fourth on the x-dimension again, and so on. The choice of the split is the median point along that dimension (the splits adapt to the data).

  This guarantees that it is balanced, so it has $\log_2 n$ levels, and is of size $2n$.

- **quad-tree:** It divides each box into 2^d axis-aligned rectangles around the geometric center of the box. Each box on the same level is the same size and shape. So the size of each box decreases each level, but not necessarily the size of the point set.

  Most algorithms with theoretical guarantees use some variant of the quad-tree. In particular a *compressed quad-tree* (where empty nodes and nodes where only one child is non-empty are removed)

can be shown to find the leaf containing q in $O(\log n)$ time and have size $O(n)$ and take $O(n \log n)$ to construct.

- **R-trees:** These structures divide a box B into two (or more) rectangles (that are possibly overlapping) that contains all $P \cap B$. These can work very well in practice if a good set of rectangles can be found at each level, but finding the best set of rectangles can be challenging. Can achieve searching bounds of about $2^d \log n$.

The dimension that these work in depend on how large of ε is permitted, and how much the data looks like it is actually in a lower dimension. R-trees and compressed quad trees adapt better than naive kd-trees.

7.2.4 High Dimensions ($d > 12$)

The problem in higher dimensions ($d > 12$) is that just about all distance look almost the same. In data randomly inside a cube or ball, most points are about the same distance apart!

In particular, these structures typically work with boxes since they are easier to compute with. But we really want all points within a ball. And as dimensions get larger, balls look less and less like boxes.

The volume of a unit ball (radius 1) in $\mathbb{R}^d$ is

$$\mathsf{vol}(B(d, \mathsf{rad} = 1)) = \frac{\pi^{d/2}}{\Gamma(d/2+1)}\mathsf{rad}^d \approx \frac{\pi^{d/2}}{(d/2)!}.$$

So it gets small as $d \to \infty$.

On the other hand, the volume of a unit cube (side length 2) in $\mathbb{R}^d$ is

$$\mathsf{vol}(C(d, \mathsf{rad} = 1)) = 2^d.$$

So it gets large (quickly goes to ∞) as $d \to \infty$.

As in rectilinear search we get everything in the box, when we want everything in the ball. And this becomes a **big** difference. So what can be done?

- **ANN:** is a library developed by David Mount and others. It basically pushes rectilinear kd-trees to the limit with various sampling and geometric approximate techniques. Reports are that this can scale to maybe $d = 20$, but depends largely on the niceness of the data.

- **LSH:** precompute the hash functions and placement of all $p \in P$ in the hashes. If the number of hashes is about the same as the dimension, then this is linear space. Reports of public code (by Alex Andoni) are up to 100s of dimensions. There are other variants that have reportedly surpassed this code, but its a good starting place.

- **random rotations kd-tree:** Instead of alternating by the dimensions (and on their axis), find a random dimension and split on that dimension. Purely random does not work all that well and there are several heuristics that work pretty well (and some have provable guarantees). One is, choose several random directions, see which one has best geometric split, and use that one. Another is to choose two random points, use that direction as the split direction (I am not sure I have seen this one in a publication - it may not have good worst case guarantees, but should have good average case guarantees).

- **clustering kd-trees:** On each node, construct a 2-means clustering (or any other fast clustering algorithm) and split the data set so each cluster is in one of the two subtrees. David Lowe (the inventor of SIFT) has an implementation of this that works very well for SIFT features in $\mathbb{R}^{128}$ (and seems to beat other variants up to that date ≈ 2007).

These last two work well when data is intrinsically in a lower dimension space. These techniques adapt to these data-dimensions, and then the behavior is similar to the regular (axis-aligned) kd-tree in the corresponding data dimension.

This is still an active and exciting research area!