5 Approximate Nearest Neighbors + HNSW

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.

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 $D_P(q, r) = \{p \in P \mid \mathbf{d}(p, q) \leq r\}$. Or we can ask for the nearest neighbor of q in P denoted $\phi_P(q) = \arg\min_{p \in P} \mathbf{d}(p, q)$.

5.1 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.

5.1.1 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.

5.1.2 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.

5.1.3 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; and all can be checked brute force) 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

$$\min_{x \in B_i} \mathbf{d}(x,q) \geq \tau_\ell/(1+\varepsilon) \tag{5.1}$$

  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 (5.1) then we continue up the hierarchy. Otherwise find $\tau_i = \phi_{P \cap B_i}(q)$ [the closest distance (and point) in this other box B_i] 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 itself be answered *recursively* 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 $P \cap B$. They can adapt to the clusters in the data. 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 $2^d \log n$, but lacks strong gaurantees of quad-trees or even kd-trees.

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.

5.1.4 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:** `https://www.cs.umd.edu/~mount/ANN/` is a library pushes rectilinear kd-trees and quadtrees 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.
- **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 – the pioneering way to embed image patches in $\mathbb{R}^{128}$) has an implementation of this that works very well in $\mathbb{R}^{128}$ (and seems to beat other variants up to that date $\approx$ 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.

Locality Sensitive Hashing: This precomputes numerous (roughly say $n^{1/7}$) hash functions and the placement of all $p \in P$ in the hashes. Unlike random hash functions, these have collisions between items more likely if $p_1, p_2 \in P$ are close. If the number of hashes is about the same as the dimension, then this is linear space. The FALCONN library `https://github.com/FALCONN-LIB/FALCONN` is an implementation that follows best theoretical guarantees and is well-engineered.

LSH requires a set of t hash functions that are grouped in bands of b hash functions each; which can each be stored in a single super-hash table. We then need to probe $r = t/b$ of these to see if any find collisions. Thus pre-processing requires storing each of $|P| = n$ data points in r (super)-hash tables, and a query takes $O(r)$ time. To find nearest neighbors we can start with a very selective set-up but unlikely to find collisions; then gradually adjust it to be less selective (small threshold of similarity) until we find collisions. Among these will be the (approximate nearest neighbor).

How does the approximation factor affect this? If the margin between collides with high probability, and does not collide with high probability is large around a threshold, we may stop when we find a collision, but it is not the true nearest-neighbor. The number of hash tables needed (and storage and query time $O(r)$) depends on the approximation factor. These are rigorously proven, but the query time is not nearly logarithmic; rather it is roughly $O(n^\rho)$ for some value ρ that depends on the approximation factor. For a 2 approximation, then we have $\rho = 1/7$. Note that for $n = 10$ billion $= 10,000,000$, then $n^{1/7} = 10$. The storage cost is then roughly $O(n^{1+\rho})$. For the approximate nearest neighbor problem on Euclidean vectors (or for Cosine/Angular distance) with no control of the dimension (e.g., $d = 2$) these trade-offs are asymptotically tight.

LSH is theoretically optimal in high-dimensions from a space/runtime/accuracy perspective, and some libraries have been very-well engineered. However, despite valient efforts, they seem to work less well in practice compared to the next-described *graph-based* approaches.

Hierarchical/Neighborhood Graph Search: An old idea has recently become very popular, and well-engineered with exciting new solutions. Preprocess P by connecting nodes together in a graph by edges (e.g., connect each $p \in P$ to its k-nearest neighbors in P). Then on a search query $q \in \mathbb{R}^d$, we follow the simple *graph gradient descent* approach:

- 1. Choose arbitrary $p'_0 \in P$ and $i = 0$
- 2. Consider neighborhood set $N(p'_i) = \{p_1, p_2, \ldots, p_k\}$
- 3. Let $p'_{i+1} = \arg\min_{p_j \in N(p'_i)} \|q - p_j\|$
- 4. If $\|q - p'_{i+1}\| \geq \|q - p'_i\|$, then stop and return p'_i.
- 5. Else, go to Step 2.

This has been made practical and effective with a couple of innovations.

Beam Search: Instead of just maintaining the best candidate at all times, maintain a set of $K > 1$ candidate vertices at each step in the search. This is often called *beam search*. When a new best candidate is explored, we consider all neighbors $N(p'_i)$ against the maintained set of size K, and out of all neighbor $N(p'_i)$ and all maintained candidates, the best K are retained. If the best candidate produces now additional candidates (it reaches a potential dead end, or local minimum), then the second best candidate has its neighbors considered. This continues until all K candidates have now neighbors which improve the candidate set.

This makes the process much more robust for getting stuck in local minimum. A small candidate set $K = 5$ can be used without much overhead, and it dramatically improves the results.

HNSW: Second, instead of just the k-nearest neighbor (k-NN) graph, it tries to maintain a **hierarchy** over nodes, so the starting point is connected diversely across the graph, and then those next nodes connected also broadly, but less so than the top-level, and so on a few levels down (sort of like kd-trees and quad-trees, but less formally structured). Then there is still a k-NN graph at the bottom. This allows for fast movement early in the search, and then the same sort of local convergence at the end.

The first point checked is at the top-level of the hierarchy, and only connects to diverse over points. These nodes then themselves connect to fairly divrese data, but typically not much further away than the closest from the first layer. The progression continues recursively, say $\log n$ levels deep. Then, on a search, we aways start with the first layer, and find the nearest point (or set of K points) on the first layer. Only then do we consider the second layer.

This structure allows for fast search (typically a constant number of steps per layer), and typically ensures it expand to the right area very quickly before zeroing in.

One can think of it as a generalized Skip List data structure.

Implementations: Common libraries are

- Hierarchical Navigable Small Worlds (HNSW: `https://github.com/nmslib/hnswlib`)
- DiskANN (`https://www.microsoft.com/en-us/research/project/project-akupara-approx`
- FAISS (`https://faiss.ai`) uses quantization (a vector compression trick) on top of this to make each step fast on a GPU.
- Pinecode (a market leader in the ANN industry) has a nice write-up on these ideas (`https://www.pinecone.io/learn/series/faiss/`).

This is still an active area with many common practical improvements, but a lack of mathematical understanding of when and why these methods (beyond LSH) work so well.