25 Nearest Neighbor Algorithms: Voronoi Diagrams and k-d Trees

NEAREST NEIGHBOR ALGORITHMS

Exhaustive k-NN Alg.

Given query point q:
- Scan through all n training pts, computing distances to q.
- Maintain a max-heap with the k shortest distances seen so far.
  [Whenever you encounter a training point closer to q than the point at the top of the heap, you remove the heap-top point and insert the better point. Obviously you don't need a heap if $k = 1$ or even 5, but if $k = 99$ a heap will substantially speed up keeping track of the 99th-shortest distance.]

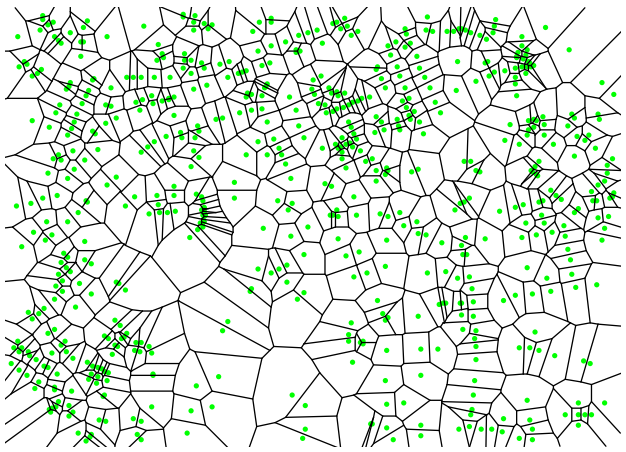
Time to train classifier: 0 [This is the only $O(0)$-time algorithm we'll learn this semester.]
Query time: $O(nd + n \log k)$
expected $O(nd + k \log n \log k)$ if random pt order
[It's a cute theoretical observation that you can slightly improve the expected running time by randomizing the point order so that only expected $O(k \log n)$ heap operations occur. But in practice I can't recommend it; you'll probably lose more from cache misses than you'll gain from fewer heap operations.]

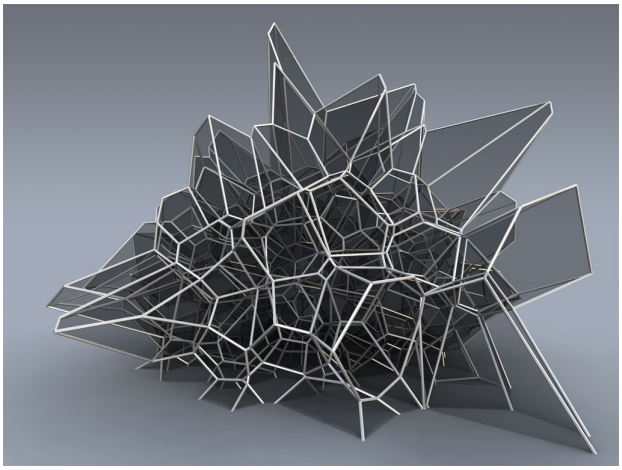
Can we preprocess training pts to obtain sublinear query time?

2–5 dimensions: Voronoi diagrams
Medium dim (up to ~ 30): k-d trees
Large dim: exhaustive k-NN, but can use PCA or random projection
locality sensitive hashing [still researchy, not widely adopted]

Voronoi Diagrams

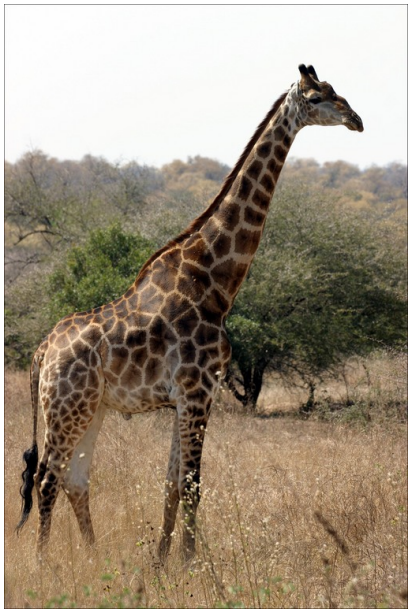
Let X be a point set. The Voronoi cell of $w \in X$ is
Vor $w = \{p \in \mathbb{R}^d : \|p - w\| \leq \|p - v\| \quad \forall v \in X\}$
[A Voronoi cell is always a convex polyhedron or polytope.]
The Voronoi diagram of X is the set of X's Voronoi cells.

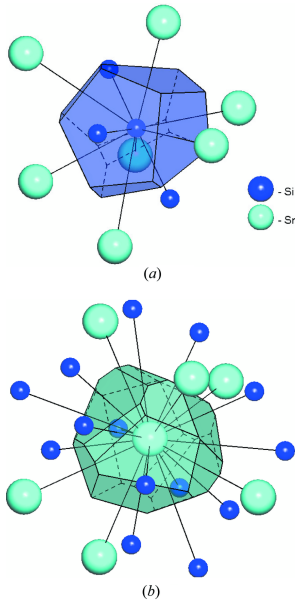


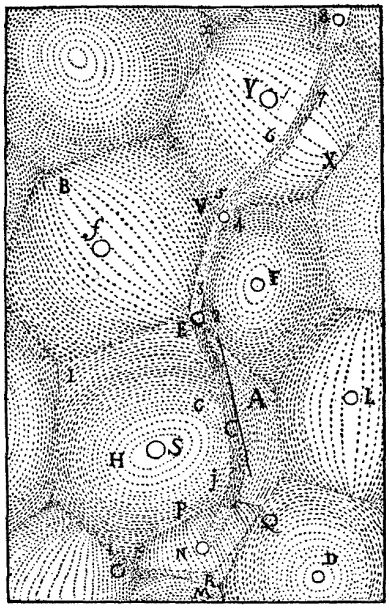


voro.pdf, vormcdonalds.jpg, voronoiGregorEichinger.jpg, saltflat3.jpg

[Voronoi diagrams sometimes arise in nature (salt flats, giraffe, crystallography).]







giraffe-1.jpg, srsi2.png (Vladislav Blatov), vortex.pdf (René Descartes)

[Perhaps the first frequent users of Voronoi cells were crystallographers, who call them Voronoi–Dirichlet polyhedra. Above we see how the polyhedra can clarify the crystal structure of the low-temperature phase of strontium silicide, α-$SrSi_2$.]

[Believe it or not, the first published Voronoi diagram dates back to 1644, in the book "Principia Philosophiae" by the mathematician and philosopher René Descartes. He claimed that the solar system consists of vortices. In each region, matter is revolving around one of the fixed stars (vortex.pdf). His physics was wrong, but his idea of dividing space into polyhedral regions has survived.]

Size (e.g., # of vertices) $\in O(n^{\lceil d/2 \rceil})$

[This upper bound is tight when d is a small constant. As d grows, the tightest asymptotic upper bound is somewhat smaller than this, but the complexity still grows exponentially with d.]

. . . but often in practice it is $O(n)$.

[Here I'm leaving out a "constant" that may grow exponentially with d.]

Point location: Given query point $q \in \mathbb{R}^d$, find the point $w \in X$ for which $q \in \text{Vor } w$.
[We need a second data structure that can perform this search on a Voronoi diagram efficiently.]

2D: $O(n \log n)$ time to compute V.d. and a trapezoidal map for pt location
$O(\log n)$ query time [because of the trapezoidal map]
[That's a pretty great running time compared to the linear query time of exhaustive search.]

dD: Use binary space partition tree (BSP tree) for pt location. [Unfortunately, it's difficult to characterize the running time of this strategy, although it is often logarithmic in 3–5 dimensions.]

1-NN only! [A standard Voronoi diagram supports only 1-nearest neighbor queries. If you want the k nearest neighbors, there is something called an order-k Voronoi diagram that has a cell for each possible k nearest neighbors. But nobody uses those, for two reasons. First, the size of an order-k Voronoi diagram is $\Theta(k^2 n)$ in 2D, and worse in higher dimensions. Second, there's no reliable software available to compute one.]

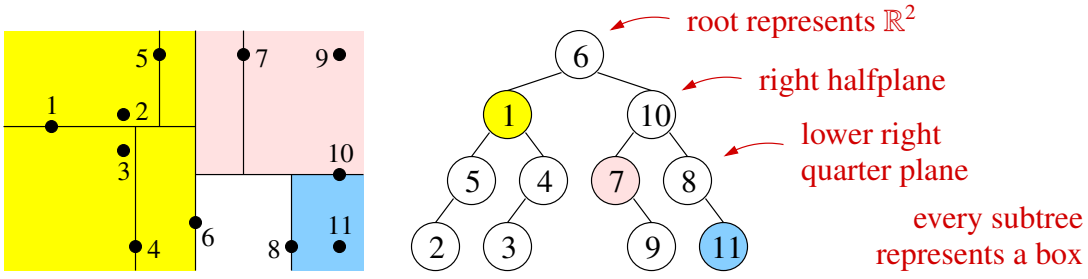
[There are also Voronoi diagrams for other distance metrics, like the ℓ_1 and ℓ_∞ norms.]

[Voronoi diagrams are good for 1-nearest neighbor queries in two dimensions, and maybe up to 5 dimensions, and they're a great concept for understanding the problem of nearest neighbor search. But k-d trees are much simpler, and probably faster in 6 or more dimensions.]

k-d Trees

"Decision trees" for NN search. [Just like in a decision tree, each treenode in a k-d tree represents a rectangular box in feature space, and we split a box by choosing a splitting feature and a splitting value. But we use different criteria for choosing splits.] Differences:

- Choose splitting feature w/greatest width: feature i in $\max_{i,j,k}(X_{ji} - X_{ki})$.
  [With nearest neighbor search, we don't care about the entropy. Instead, what we want is that if we draw a sphere around the query point, it won't intersect very many boxes of the decision tree. So it helps if the boxes are nearly cubical, rather than long and thin.]
  Cheap alternative: rotate through the features. [At depth 1 we split on the first feature, at depth 2 we split on the second feature, and so on. This builds the tree faster, by a factor of $O(d)$.]
- Choose splitting value: median point for feature i; OR midpoint $\frac{X_{ji}+X_{ki}}{2}$.
  Median guarantees $\lfloor \log_2 n \rfloor$ tree depth; $O(nd \log n)$ tree-building time.
  [...or just $O(n \log n)$ time if you rotate through the features. An alternative to the median is splitting at the box center, which improves the aspect ratios of the boxes, but it could unbalance your tree. A compromise strategy is to alternate between medians at odd depths and centers at even depths, which also guarantees an $O(\log n)$ depth.]
- Each internal node stores a training point. [...that lies in the node's box. Usually the splitting point.]
  [Some k-d tree implementations have points only at the leaves, but it's better to have points in internal nodes too, so when we search the tree, we often stop searching earlier.]



[Draw this by hand. kdtreestructure.pdf]

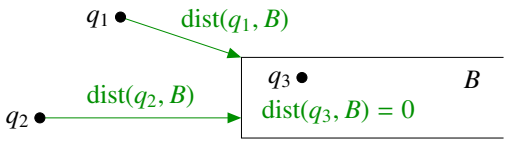
[Just like in a decision tree, each subtree represents an axis-aligned box in feature space. All the training points stored in a subtree are in that box.]

[Once the tree is built, the classification algorithm is very different from decision trees. Most importantly, you usually have to visit multiple leaves of the tree to find the nearest neighbor. To save time, we sometimes use an approximate nearest neighbor algorithm, instead of demanding the exact nearest neighbor.]

After tree is built (training), classify test pts:

Goal: given query pt q, find a training pt w such that $\|q - w\| \le (1 + \epsilon)\,\|q - u\|$,
where $\|\cdot\|$ is ℓ_p norm for some $p \in [1, \infty]$ [k-d trees are not limited to the Euclidean (ℓ_2) norm.]
& u is the nearest training pt in that norm.
$\epsilon = 0 \;\Rightarrow$ exact NN; $\epsilon > 0 \;\Rightarrow$ approximate NN.

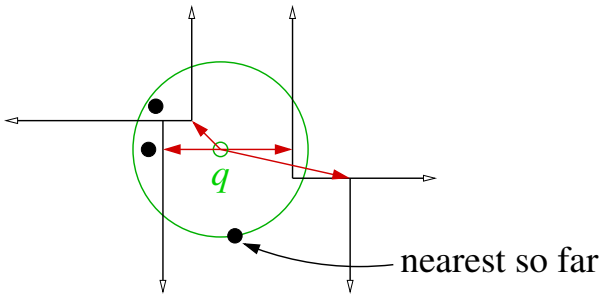
Each subtree represents a box $B = [s_1, t_1] \times [s_2, t_2] \times \cdots \times [s_d, t_d]$. [An s_i can be $-\infty$, and a t_i can be ∞.]
[Think of s as the lower left corner of the box, and t as the opposite corner.]
Think of B as an infinite point set.
The distance from q to B is $\text{dist}(q, B) = \min_{z \in B} \|q - z\|$. [This norm is the same norm we seek neighbors in.]



[Draw this by hand. dist.pdf] [Point-to-box distances.]

The minimizer's components are $z_i = \begin{cases} s_i, & q_i < s_i, \\ q_i, & q_i \in [s_i, t_i], \\ t_i, & q_i > t_i. \end{cases}$

Query alg. maintains:
- Nearest neighbor found so far (or k nearest). goes down ↓
- Binary min-heap of unexplored boxes/subtrees, keyed by distance from q. goes up ↑



[Draw this by hand. query.pdf] [A query in progress.]

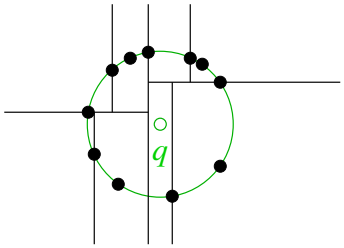
[We search the boxes nearest q first, hoping that we will never need to search most of the boxes or their associated subtrees. The binary heap makes it fast to find the box nearest q, because each box in the heap has a numerical key, the distance from q to the box. The search stops when the distance from q to the kth-nearest neighbor found so far $\le$ the distance from q to the nearest unexplored box (times $1 + \epsilon$). For example, in the figure above, the query will never visit the box at far lower right, because it doesn't intersect the circle. That's how we avoid searching most of the tree—when we're lucky.]

Alg. for 1-NN query. Interpret each B as both a box and a treenode.

$Q \leftarrow$ min-heap containing root node with key zero
$r \leftarrow \infty$
while Q not empty and $(1+\epsilon) \cdot \text{minkey}(Q) < r$
    $B \leftarrow \text{removemin}(Q)$
    $v \leftarrow B$'s training point
    if $\|q - v\| < r$ then { $w \leftarrow v$; $r \leftarrow \|q - v\|$ }
    $B', B'' \leftarrow$ child boxes of B
    if $(1+\epsilon) \cdot \text{dist}(q, B') < r$ then $\text{insert}(Q, B', \text{dist}(q, B'))$ [The key for B' is $\text{dist}(q, B')$]
    if $(1+\epsilon) \cdot \text{dist}(q, B'') < r$ then $\text{insert}(Q, B'', \text{dist}(q, B''))$
return w

For k-NN, replace "r" and "w" with a max-heap holding the k nearest neighbors.

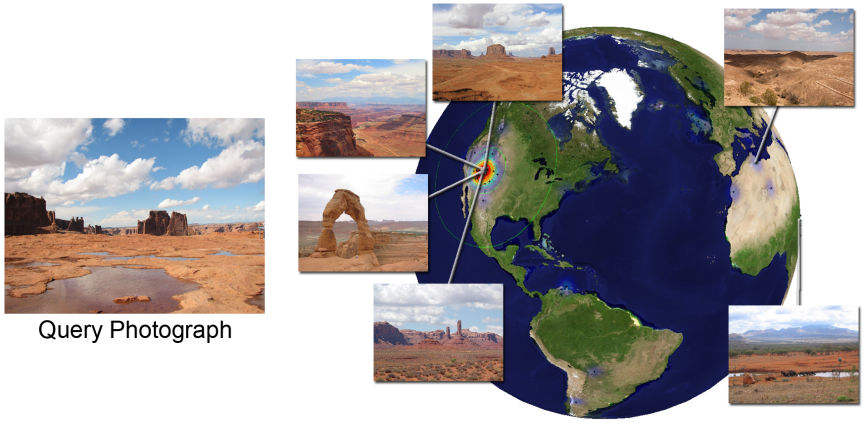
Why ϵ-approximate NN?



[Draw this by hand. kdtreeproblem.pdf] [A worst-case exact NN query.]

[In the worst case, we may have to visit every node in the k-d tree to find the exact nearest neighbor. In that case, the k-d tree is slower than simple exhaustive search. This is an example where an *approximate* nearest neighbor search can be much faster. In practice, settling for an approximate nearest neighbor sometimes improves the speed by a factor of 10 or even 100, because you don't need to look at most of the tree to do a query. This is especially true in high dimensions—in a high-dimensional space, the nearest point often isn't much closer than a *lot* of other points.]

[I want to emphasize the fact that exhaustive nearest neighbor search really is one of the first classifiers you should try in practice, even if it seems too simple. So here's an example of a research paper that uses a 120-nearest neighbor classifier to solve a problem.]



im2gpspress.pdf

[In 2008, James Hays and our own Prof. Alexei Efros wrote a paper on geolocalization, where the goal is to take a query photograph and determine where on earth the photo was taken. Their training data was 6 million GPS-tagged photos downloaded from Flickr. The bottom line is that by using 120-nearest neighbors, they came within 64 km of the correct location about 50% of the time. That was good for the time, but I think that in 2025, you could do much better with the data available to us today.]

RELATED CLASSES [if you like machine learning, consider these courses in 2024–25]

CS 180/280A (fall): Computer Vision/Photography
CS 182/282A (fall): Deep Neural Networks
EECS 183 (fall?): Natural Language Processing
CS 185/285 (fall?): Deep Reinforcement Learning
CS 194-196/294-196 (fall): Agentic AI (D. Song)
CS C281A (fall): Statistical Learning Theory [C281A is the most direct continuation of CS 189/289A.]
EECS 127 (both), 227AT (both): Numerical Optimization [a core part of ML]
[It's hard to overemphasize the importance of numerical optimization to machine learning, as well as other CS fields like graphics, theory, and scientific computing.]
EECS 126 (both): Random Processes [Markov chains, expectation maximization, PageRank]
EE C106A/B (fall/spring): Intro to Robotics [dynamics, control, sensing]
Math 110 (both): Linear Algebra [but the real gold is in Math 221]
Math 221 (fall): Matrix Computations [how to solve linear systems, compute SVDs, eigenvectors, etc.]
CS C281B (spring): Learning & Decision Making
CS C267 (spring): Scientific Computing [parallelization, practical matrix algebra, some graph partitioning]
CS C280 (spring): Computer Vision (Efros, Kanazawa)
CS 294-162 (fall): ML Systems (Gonzalez/Stoica/Zaharia)
CS 294-286 (fall): Machine Learning in Social Settings (Chang)
NEU 100A (fall): Cellular and Molecular Neurobiology
VS 265 (?): Neural Computation