25 Faster Nearest Neighbors: Voronoi Diagrams and k-d Trees

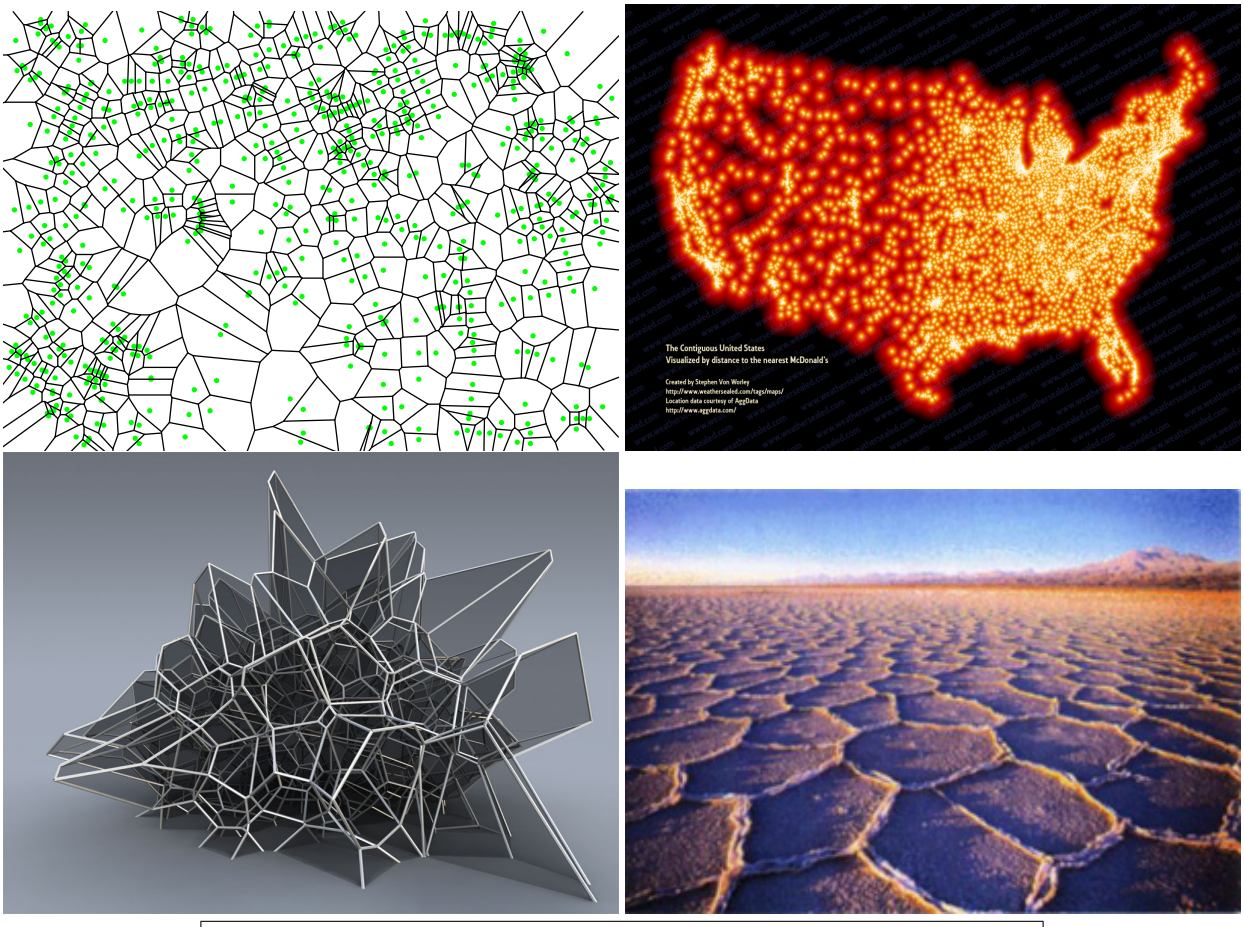
SPEEDING UP NEAREST NEIGHBOR CLASSIFIERS

Can we preprocess training pts to obtain sublinear query time?

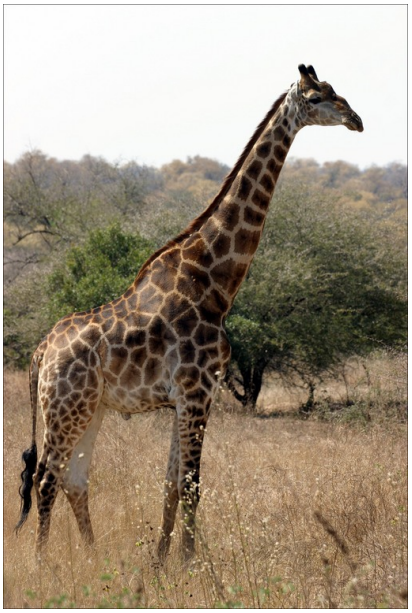
2–5 dimensions: Voronoi diagrams
Medium dim (up to ~ 30): k-d trees
Larger dim: locality sensitive hashing [still researchy, not widely adopted]
Largest dim: exhaustive k-NN, but can use PCA or random projection [or another dimensionality reduction method]

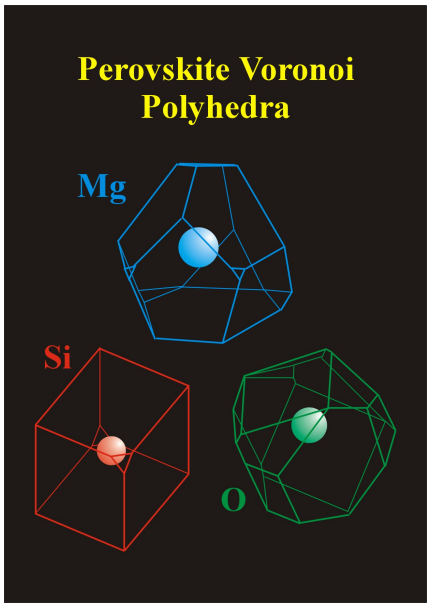
Voronoi Diagrams

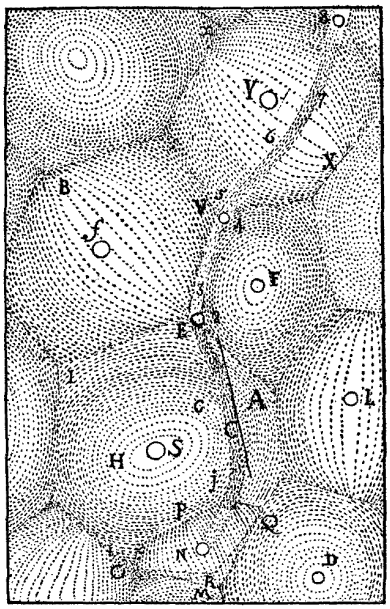
Let P be a point set. The Voronoi cell of $w \in P$ is
$\text{Vor } w = \{p \in \mathbb{R}^d : |pw| \leq |pv| \quad \forall v \in P\}$
[A Voronoi cell is always a convex polyhedron or polytope.]
The Voronoi diagram of P is the set of P's Voronoi cells.



voro.pdf, vormcdonalds.jpg, voronoiGregorEichinger.jpg, saltflat-1.jpg
[Voronoi diagrams sometimes arise in nature (salt flats, giraffe, crystallography).]







giraffe-1.jpg, perovskite.jpg, vortex.pdf

[Believe it or not, the first published Voronoi diagram dates back to 1644, in the book "Principia Philosophiae" by the famous 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, find the point $w \in P$ for which $q \in \text{Vor } w$.

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 likely to be reasonably fast 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 $O(k^2 n)$ in 2D, and worse in higher dimensions. Second, there's no software available to compute one.]

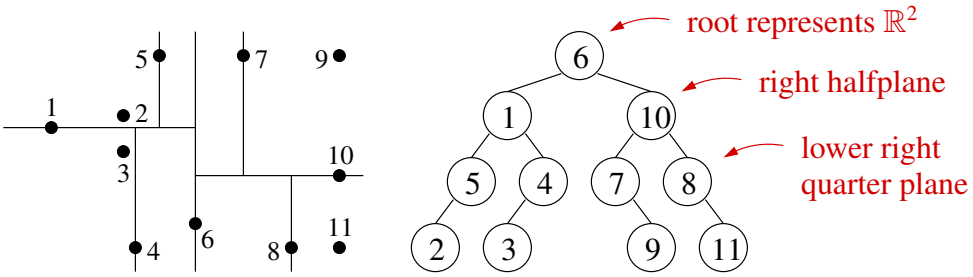
[There are also Voronoi diagrams for other distance metrics, like the L_1 and L_∞ norms.]

[Voronoi diagrams are good for 1-nearest neighbor in 2 or 3 dimensions, maybe 4 or 5, but for anything beyond that, k-d trees are much simpler and probably faster.]

k-d Trees

"Decision trees" for NN search. Differences: [compared to decision trees]

- 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. [We split on the first feature at depth 1, the second feature at depth 2, and so on. This builds the tree faster, by a factor of $O(d)$.]
- Choose splitting value: median point for feature i, or $\frac{X_{ji}+X_{ki}}{2}$.
  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. By contrast, splitting near the center does more to improve the aspect ratios of the boxes, but it could unbalance your tree. You can alternate between medians at odd depths and centers at even depths, which also guarantees an $O(\log n)$ depth.]
- Each internal node stores a sample point. [. . . that lies in the node's box. Usually the splitting point.]
  [Some k-d tree implementation have points only at the leaves, but it's usually better to have points in internal nodes too, so when we search the tree, we might stop searching earlier.]

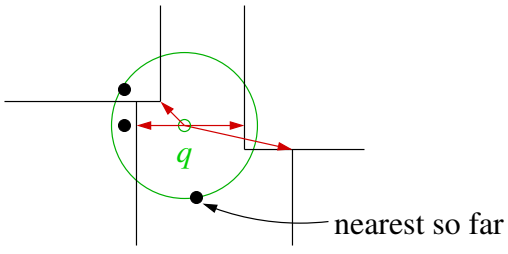


[Draw this by hand. kdtreestructure.pdf]

Goal: given query pt q, find a sample pt w such that $|qw| \leq (1 + \epsilon)|qs|$, where s is the closest sample pt.
$\epsilon = 0 \Rightarrow$ exact NN; $\epsilon > 0 \Rightarrow$ approximate NN.

The alg. maintains:

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



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

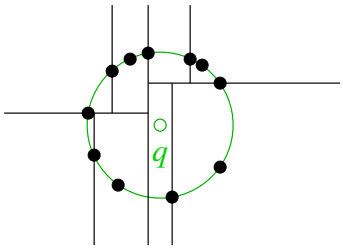
[Each subtree represents an axis-aligned box. The query tries to avoid searching most of the subtrees by searching the boxes close to q first. We measure the distance from q to a box and use it as a key for the subtree in the heap. The search stops when the distance to the kth-nearest neighbor found so far $\leq$ the distance to the nearest unexplored box (divided by $1 + \epsilon$). For example, in the figure above, the query never visits the box at far upper left or the box at far lower right, because those boxes don't intersect the circle.]

$Q \leftarrow$ heap containing root node with key zero
$r \leftarrow \infty$
while Q not empty and minkey(Q) $< \frac{r}{1+\epsilon}$
    $B \leftarrow$ removemin(Q)
    $w \leftarrow B$'s sample point
    $r \leftarrow \min\{r, |qw|\}$ [Optimization: store square of r instead.]
    $B', B'' \leftarrow$ child boxes of B
    if dist$(q, B') < \frac{r}{1+\epsilon}$ then insert$(Q, B', \text{dist}(q, B'))$ [The key for B' is dist(q, B')]
    if dist$(q, B'') < \frac{r}{1+\epsilon}$ then insert$(Q, B'', \text{dist}(q, B''))$
return point that determined r

For k-NN, replace "r" with a max-heap holding the k nearest neighbors
[. . . just like in the exhaustive search algorithm I discussed last lecture.]

Works with any L_p norm for $p \in [1, \infty]$.
[k-d trees are not limited to the Euclidean (L_2) norm.]

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 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—remember that in high-dimensional space, the nearest point often isn't much closer than a lot of other points.]

Software:
- ANN (David Mount & Sunil Arya, U. Maryland)
- FLANN (Marius Muja & David Lowe, U. British Columbia)
- GeRaF (Georgios Samaras, U. Athens) [random forests!]

Example: im2gps

[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 modern research paper that uses 1-NN and 120-NN search to solve a problem.]

Paper by James Hays and [our own] Prof. Alexei Efros.
[Goal: given a query photograph, determine where on the planet the photo was taken. Called geolocalization. They evaluated both 1-NN and 120-NN with a complex set of features. What they did not do, however, is treat each photograph as one long vector. That's okay for tiny digits, but too expensive for millions of travel photographs. Instead, they reduced each photo to a small descriptor made up of a variety of features that extract the essence of each photo.]
[Show slides (im2gps.pdf). Sorry, images not included here. http://graphics.cs.cmu.edu/projects/im2gps/]

[Features, in rough order from most effective to least:

1. GIST: A compact "image descriptor" based on oriented edge detection (Gabor filters) + histograms.
2. Textons: A histogram of textures, created after assembling a dictionary of common textures.
3. A shrunk 16×16 image.
4. A color histogram.
5. Another histogram of edges, this one based on the Canny edge detector, invented by our own Prof. John Canny.
6. A geometric descriptor that's particularly good for identifying ground, sky, and vertical lines.]

[Bottom line: With 120-NN, their most sophisticated implementation came within 64 km of the correct location about 50% of the time.]

RELATED CLASSES

[If you like machine learning and you'll still be here next year, here are some courses you might want to take.]

CS C281A (spring): Statistical Learning Theory [C281A is the most direct continuation of CS 189/289A.]
EE 127 (spring), EE 227BT (fall): 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.]
EE 126 (both): Random Processes [Markov chains, expectation maximization, PageRank]
EE C106A/B (fall/spring): Intro to Robotics [dynamics, control, sensing]
Math 110: Linear Algebra [but the real gold is in Math 221]
Math 221: Matrix Computations [how to solve linear systems, compute SVDs, eigenvectors, etc.]
CS 194-26 (fall): Computational Photography (Efros)
CS 294-43 (fall): Visual Object and Activity Recognition (Efros/Darrell)
CS 294-112 (fall): Deep Reinforcement Learning (Levine)
CS 298-115 (fall): Algorithmic Human-Robot Interaction (Dragan)
CS 298-131 (fall): Special Topics in Deep Learning (Song/Darrell)
VS 265 (?): Neural Computation
CS C280 (?): Computer Vision
CS C267 (?): Scientific Computing [parallelization, practical matrix algebra, some graph partitioning]