# L2: Curse of Dimensionality: Nearest Neighbors

Consider $X \subset \mathbb{R}^d$.
Query point $q \in \mathbb{R}^d$.
*Nearest neighbor* is point $NN_X(q) = \arg\min_{x \in X} \|x - q\|$.

We can solve $NN_X(q)$ by calculating $\|x - q\|$ for all $x \in X$.

The challenge is to:

1. **Pre-process:** Build data structure $S$ in time that is not much longer than reading data $X$
2. **Query:** Solve $NN_X(q)$ using $S$, in time that is much faster than $O(nd)$.

**Rumor:**

One can construct examples $X \sim Unif(B_d(r = 1))$ so for each $x \in X$ the distance to all other data points are about the same. So the notion of nearest neighbor is not that meaningful.

These are mathematically true, but not reflected in data analysis. Real data rarely exhibits this behavior. If it does, it is (typically) not interesting for analysis – at least not the analysis where this matters.

## 1-dimensional Nearest Neighbor Search

First consider data $X \in \mathbb{R}^1$. *How do we preprocess X for efficient queries?*

Preprocess:

- Sort X.
- Build binary tree.
- For each internal node: store smallest value, at the root of subtree.
- $O(n \log n)$ time, $O(n)$ space.

NN Query:

- From root: check right-child if value smaller –> recurse left
  otherwise, recurse right
- At leaf: compare value to right-adjacent, return closest.
- $O(\log n)$ time.

Range query: interval I = [a,b]. Returns all $X \cap [a, b]$.

- $O(\log n + k)$ to return $k$ items.
- $O(\log n)$ to return number of items.

## d-dimensional Range Search

*How do generalize range search to 2 dimensions?*

**Range Tree:**

- Build binary tree on first coordinate
- Each internal node builds binary tree on second coordinate (over subtree data).
- Space $O(n \log n)$
- Query in $O(\log^2 n)$ time

- Save factor $\log n$ with "fractional cascading"

*How does Range Tree work in d dimensions?*

- recursively build binary tree on each coordinate.
- Space $O(n \log^d n) \rightarrow O(n \log^{d-1} n)$.
- Query $O(\log^d n) \rightarrow O(\log^{d-1} n)$.

*For what value of d is this useful?*
For $d = \log n / \log\log n$ then $\log^d n = n$

$\log^d n = n$
$\log(\log^d n) = \log n$
$d \log\log n = \log n$
$d = \log n / \log\log n$

## Balls and Halfspaces

if $x = NN_X(q)$, the need to verify that the interior of ball $B_r(q)$ centered at $q$ with radius $r = \|q - x\|$ is empty.
So *Ball Range Emptiness Query.*

In low dimensions, balls are harder than rectangles, since cannot use orthogonal decomposition.

**In high-dimensions, balls are essentially halfspaces.**
Assuming we are in regime where $d$ vs. $d + 1$ does not matter much.

Let $X \subset \mathbb{R}^d$, and consider ball $B$ (can be any radius, center).
Map to $X' \subset \mathbb{R}^{d+1}$ so $x = (x_1, x_2, ..., x_d) \in X$ then $x' = (x_1, x_2, ..., x_d, x_1^2 + x_2^2 + ... + x_d^2) \in \mathbb{R}^{d+1}$.
Now there exists a halfspace $H_B \subset \mathbb{R}^{d+1}$ that contains $x'$ if and only if $B$ contains $x$.

Moreover, consider $X \subset \mathbb{R}^d$, and consider halfspace $H \subset \mathbb{R}^d$.
For subset $S = X \cap H$, there exists a ball $B \subset \mathbb{R}^d$ so that $S = X \cap B$
–> start with a small ball $b$ with boundary tangent to boundary of $H$. Keep the point boundaries touch fixed, and move center of $b$ away from $H$.

## Halfspace Range Queries

For any binary split, a halfspace query may need to recurse on both sides!
*How can we get efficient $o(n)$ time queries?*

Willard [1982]: Partition Trees for d=2

- Split with 2 lines, into 4 regions (each approximately $n/4$ points)
- each query $H$ intersects at most 3 regions (other 1 region ia either *all in*, or *all out*).
- recurse on 3/4, so on roughly $(3/4)n$ points.
- query time $O(n^{\log_4 3}) \approx O(n^{0.792})$.
- and slight improvements with more cuts

Matousek, Chazelle [1989 - 1992]: optimal partition trees for general d

- query time $\tilde{O}(n^{1-1/\lfloor d/2 \rfloor})$
- space $O(n)$

So for balls in $\mathbb{R}^d$, invoke above for halfspaces in $\mathbb{R}^{d+1}$

- query time $\tilde{O}(n^{1-1/\lfloor d+1/2 \rfloor})$
- space $O(n)$

## kd Trees

Hierarchical range and NN searching, useful in practice.
Each node splits subtree's data in half along one coordinate at median. Cycles through coordinates.
Height $\log_2 n$; space $O(n)$; but queries more complicated.

### NN queries

- root to leaf in tree on query $q$, based on node-splits. Finds potential p = NN(q)
- backtracks, pruning subtrees which cannot improve upon p
- worst case $O(n)$ time, but fast in practice.

- allow (1+eps)-approximation (search to stop *much!* earlier)

### Data-adaptive splits

Instead of splitting on coordinates (which is fast), adapt to data. e.g.:

- 2-means clustering
- run PCA, split along top PC axis
- split in random direction (among pairs of data points)
- Ball tree: pick any data point $x$, find median distance $m$ to $x$, left tree is all points within $m$ of $x$, and right tree those further.

With these ideas, often extends to $d \approx 100$ on "real" data

## Space Filing Curves

Create 1-dimensional ordering along points in d=2,3,… dimensions.

To find nearest neighbor, use 1-dimensional techniques on this ordering $O(\log n)$ time.
Not always works, so repeat 3-7 times on randomly shifted curve structures…