

DSC 190

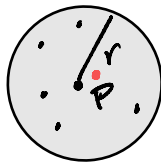
DATA STRUCTURES & ALGORITHMS

Today's Lecture

Approximate Nearest Neighbor

- ▶ **Given:** a set of n points in \mathbb{R}^d and query point p .
- ▶ **Compute:** (approximate) nearest neighbor of p .
- ▶ Last time: k-d trees do not scale well with d .

Today



- ▶ Slightly different problem: given a distance r and query p .
- ▶ Return (approximately) all of the points within distance r of p .
- ▶ Can use to compute ANN.

Today

- ▶ We'll introduce **locality sensitive hashing**.
- ▶ An important idea.
- ▶ We'll see similar themes in remainder of course.

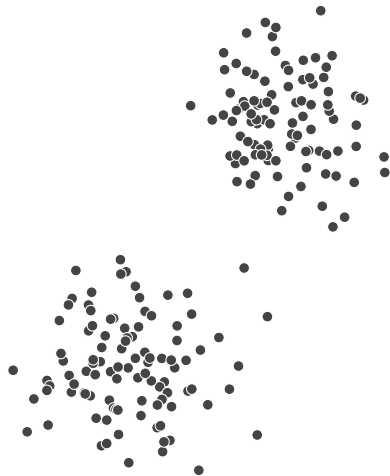
DSC 190

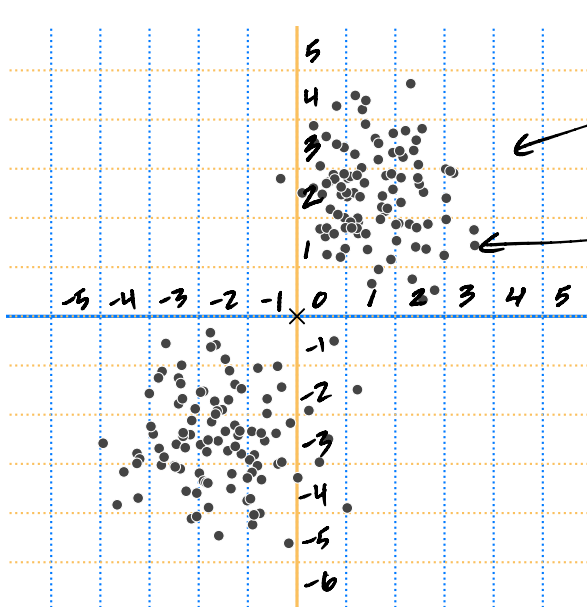
DATA STRUCTURES & ALGORITHMS

Implementing a NN Grid

Grids

- ▶ Given input point p , want quick way to find nearby points.
- ▶ Idea: divide space into cells using grid.
- ▶ Find cell containing p , search it.
- ▶ How would we implement this?





cell id

$(4, 3)$

$(3.6, 1.5)$

\downarrow
 $(\lfloor 3.6 \rfloor, \lfloor 1.5 \rfloor)$

\downarrow
 $(3, 1)$

Grid Cells

- ▶ Each point (x, y) given cell id: $(\lfloor x \rfloor, \lfloor y \rfloor)$
 - ▶ Example: $(1.2, 6.7)$ given cell id $(1, 6)$.
- ▶ Store (x, y) in dictionary with cell id as key.
 - ▶ Discretization allows multiple points in same cell.
 - ▶ Store collisions in list.
- ▶ Generalizes naturally to d -dimensions.

```
class NNGrid:
```

```
    def __init__(self, width):  
        self.width = width  
        self.cells = {}
```

```
    def cell_id(self, p):  
        p = np.asarray(p)  
        cell_id = np.floor(p / self.width).astype(int)  
        return tuple(cell_id)
```

```
    def insert(self, p):  
        """Insert p into the grid."""  
        cell_id = self.cell_id(p)  
        if cell_id not in self.cells:  
            self.cells[cell_id] = []  
        self.cells[cell_id].append(p)
```

```
    ...
```

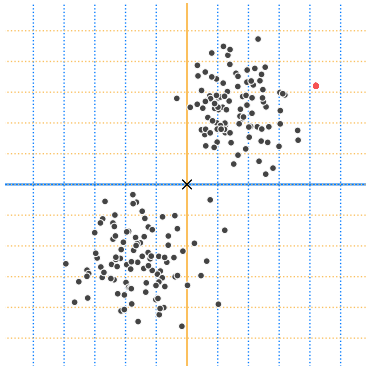
...

```
def points_in_cell(self, p):
    cell_id = self.cell_id(p)
    if cell_id not in self.cells:
        return []
    points_in_cell = self.cells[cell_id]
    # turn into an array
    return np.vstack(points_in_cell)

def query(self, p):
    return brute_force_nn(self.points_in_cell(p), p)
```

Note

- ▶ This may **fail** – NN could be in different cell.



Problems

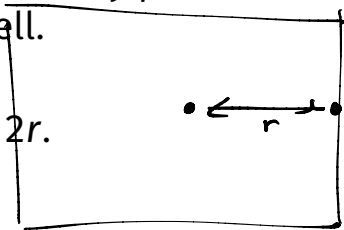
- ▶ In d dimensions, cell id has d entries.

$$\text{cell-id}(p) = (\lfloor x_1/w \rfloor, \lfloor x_2/w \rfloor, \dots, \lfloor x_d/w \rfloor)$$

- ▶ All entries must be **exactly** the same for two points to have same cell id.
- ▶ This is **very unlikely**. Most cells are empty or contain one point.

High-Dimensional Cuboids

- ▶ One “fix”: increase cell width parameter.
- ▶ Suppose we want to ensure any points within distance r are in same cell.
- ▶ Then cell width must be $2r$.



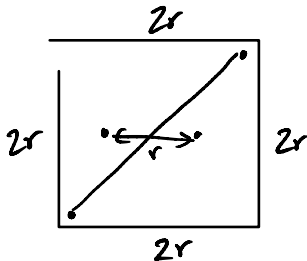
High-Dimensional Cuboids

- ▶ But a d -dimensional cuboid of width $2r$ can contain points at distance $2\sqrt{d}r$ from one another!

$$d = 1000$$

$$2\sqrt{1000}r \approx 60r$$

- ▶ For even modest r , the whole data set is in one cell.



$$\begin{aligned} & \sqrt{4r^2 + 4r^2} \\ &= \sqrt{8r^2} \\ &= 2\sqrt{2}r \end{aligned}$$

Main Idea

Dividing into a grid of cuboids fails in high dimensions. Either the cells are empty, or contain everything, depending on the width!

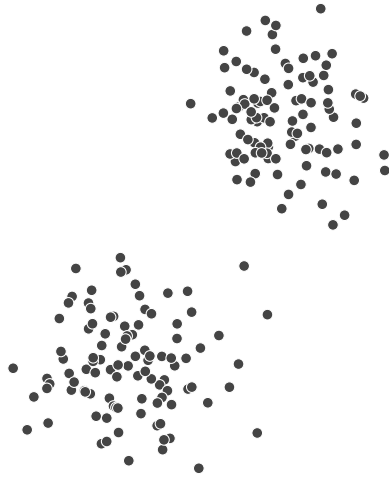
DSC 190

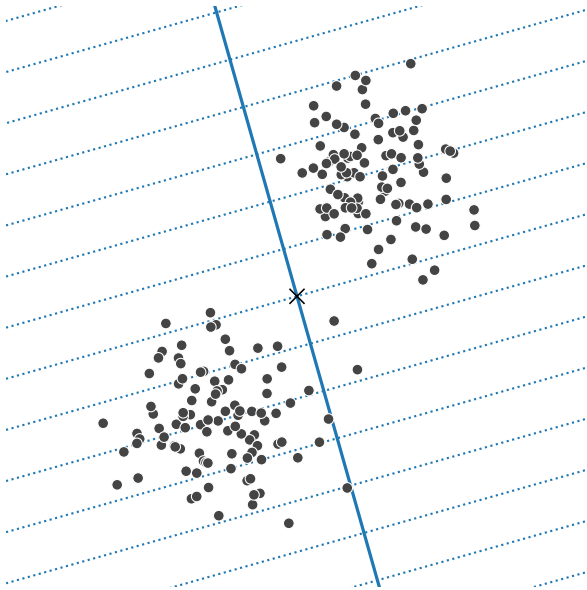
DATA STRUCTURES & ALGORITHMS

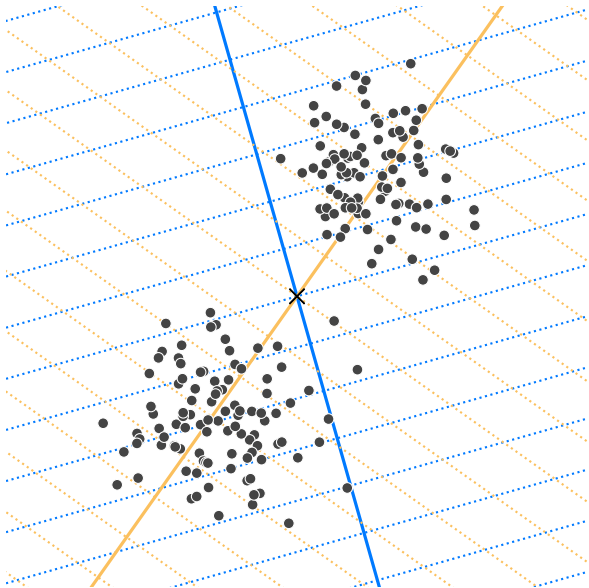
A Randomized “Grid”

A Randomized “Grid”

- ▶ Idea: Instead of axis-aligned grid, divide into cells using $k \ll d$ random directions.

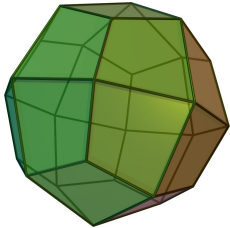






Cell Shape

- ▶ Cells are no longer d -dimensional cuboids.
- ▶ They are random k -dimensional **polytopes**.



Question

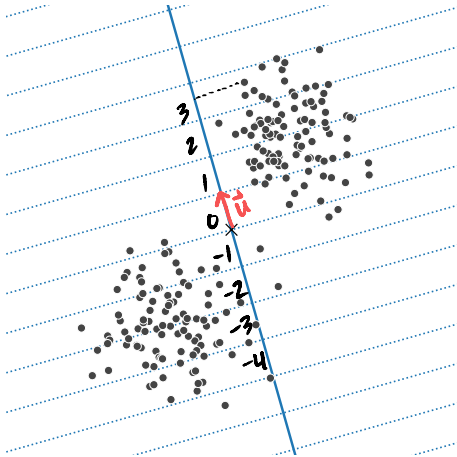
- ▶ Why is this better? We'll see in the next sections.

Projection

- ▶ How do we determine which cell a point lies in?

$$\vec{x} = (x, y) \quad (\lfloor x/w \rfloor, \lfloor y/w \rfloor)$$

$$\left\lfloor \frac{\vec{x} \cdot \vec{u}}{w} \right\rfloor$$



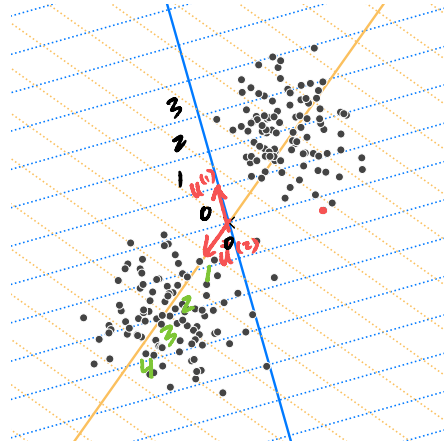
Cell IDs

- ▶ Pick k random unit vectors, $\vec{u}^{(1)}, \dots, \vec{u}^{(k)} \in \mathbb{R}^d$.
- ▶ Pick a width parameter, w .
- ▶ Given any point \vec{p} , its cell id is¹:

$$\text{cell-id}(\vec{p}) = \left(\left\lfloor \frac{\vec{u}^{(1)} \cdot \vec{p}}{w} \right\rfloor, \left\lfloor \frac{\vec{u}^{(2)} \cdot \vec{p}}{w} \right\rfloor, \dots, \left\lfloor \frac{\vec{u}^{(k)} \cdot \vec{p}}{w} \right\rfloor, \right)$$

¹use same width and unit vectors for all points

Example



$(-1, -2)$

Quick Cell-ID Calculation

- ▶ Place $\vec{u}^{(1)}, \dots, \vec{u}^{(k)}$ into a matrix:

$$U = \begin{pmatrix} \leftarrow & (\vec{u}^{(1)})^T & \rightarrow \\ \leftarrow & (\vec{u}^{(2)})^T & \rightarrow \\ \vdots & \vdots & \vdots \\ \leftarrow & (\vec{u}^{(k)})^T & \rightarrow \end{pmatrix} \begin{array}{c} \updownarrow \\ k \end{array}$$

The diagram shows a matrix U with k rows and d columns. The columns are labeled with \leftarrow , $(\vec{u}^{(i)})^T$, and \rightarrow . A double-headed arrow above the matrix indicates the width d . A vertical double-headed arrow to the right of the matrix indicates the height k .

- ▶ Then $\text{cell-id}(\vec{p}) = \text{entrywise-floor}(U\vec{p}/w)$

Generating Random Unit Vectors

```
def gaussian_projection_matrix(k, d):  
    X = np.random.normal(size=(k, d))  
    U = X / np.linalg.norm(X, axis=1)[:,None]  
    return U
```

```
class NNProjectionGrid:
```

```
    def __init__(self, projection_matrix, width):  
        self.width = width  
        self.projection_matrix = projection_matrix  
        self.cells = {}
```

```
    def cell_id(self, p):  
        projection = self.projection_matrix @ p  
        cell_id = np.floor(projection / self.width)  
        return tuple(cell_id.astype(int))
```

```
# insert, query, points_in_cell same as for NNGrid
```

But wait...

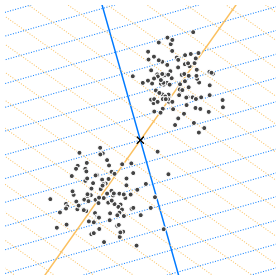
 P^k P^d

- ▶ In high dimensions, still **very unlikely** for cell to contain >1 point.

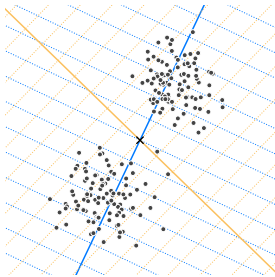
$$\text{cell-id} = \frac{_}{\downarrow}, \frac{_}{\downarrow}, \frac{_}{\downarrow}, \dots, \frac{_}{\downarrow}$$

- ▶ Idea: **banding**. Try, try again.
- ▶ Build multiple NNProjectionGrids with different random projections.
- ▶ Find `points_in_cell` for each, pool them together.

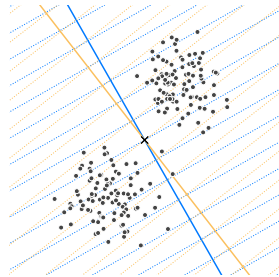
Multiple Random Projections



U_1



U_2



U_3

Locality Sensitive Hashing

- ▶ This idea (multiple random projections) is an example of **Locality Sensitive Hashing** (LSH).
- ▶ We'll explore it more in the next section.


```
class LocalitySensitiveHashing:

    def __init__(self, l, k, d, w):
        self.randomized_grids = []
        for i in range(l):
            U = gaussian_projection_matrix(k, d)
            randomized_grid = NNProjectionGrid(U, w)
            self.randomized_grids.append(randomized_grid)

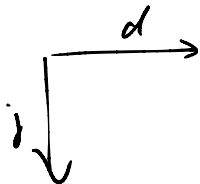
    def insert(self, p):
        for randomized_grid in self.randomized_grids:
            randomized_grid.insert(p)

    ...
```

...

```
def query_close(self, p):
    nearby = []
    for randomized_grid in self.randomized_grids:
        points_in_cell = randomized_grid.points_in_cell(p)
        nearby.append(points_in_cell)
    return np.vstack(nearby)

def query_nn(self, point):
    results = self.query_close(point)
    pool = np.vstack([r for r in results])
    if len(pool) == 0:
        raise ValueError('No points nearby.')
    return brute_force_nn(pool, point)
```



Parameters

- ▶ τ : number of randomized “grids”
- ▶ k : number of random directions in each “grid”
- ▶ w : bin width

Tuning Parameters

- ▶ Choose so that `.query_close` returns a small # of points.
- ▶ If # is very small (or zero), either:
 - ▶ increase w or ℓ
 - ▶ decrease k

Note

- ▶ This is an approximate NN technique!
- ▶ May not find **the** NN.
- ▶ May not return **anything!**

DSC 190

DATA STRUCTURES & ALGORITHMS

Theory of Locality Sensitive Hashing

Why does LSH work?

- ▶ Two approaches to understanding LSH.
- ▶ **1) Hashing view.**
- ▶ 2) Dimensionality reduction view.

Standard Hashing

- ▶ A **hash function** $h : \mathcal{X} \rightarrow \mathbb{Z}$ takes in an object from \mathcal{X} and returns a bucket number.



Standard Hashing

- ▶ **Collision**: two different objects have same hash.
- ▶ Usually, collisions are **bad**.
- ▶ Want similar things to have very different hashes.

Locality Sensitive Hashing

- ▶ But in NN search, we want “close” items to be in the **same bucket** (have same hash).
- ▶ “Far” items should be in **different buckets** (have different hash).

Locality Sensitive Hashing

- ▶ Let r be a distance we consider “close”.
- ▶ Let cr (with $c > 1$) be a distance we consider “far”.
- ▶ Suppose H is a **family** of hash functions.

LSH Family

- ▶ H is an **LSH family** if when h is randomly drawn from H :

$$\mathbb{P}(h(x) = h(y)) \geq p_1 \quad \text{when } d(x, y) \leq r$$

$$\mathbb{P}(h(x) = h(y)) \leq p_2 \quad \text{when } d(x, y) \geq cr$$

where $p_1 > p_2$.

Main Idea

If x and y are close, the probability that they hash to the **same** bin is not too small. If they are far, the probability is not too large.

Example: Random Projections

- ▶ We have seen one LSH family: random projections followed by binning.
- ▶ H has infinitely-many hash functions, one for each direction \vec{u} :

$$h_{\vec{u}}(\vec{p}) = \left\lfloor \frac{\vec{u} \cdot \vec{p}}{w} \right\rfloor,$$

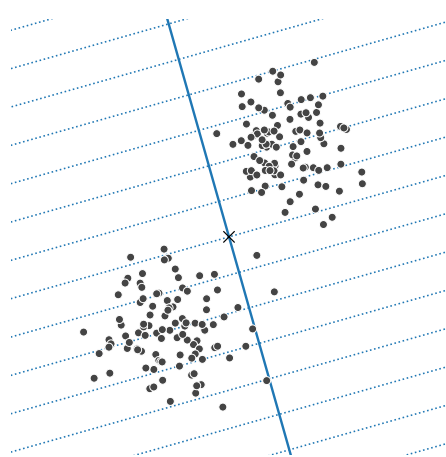
Example: Random Projections

- ▶ Suppose a random hash function h is chosen.
- ▶ Claim:

$$\mathbb{P}(h(x) = h(y)) \geq \frac{1}{2} \quad \text{when } d(x, y) \leq w/2$$

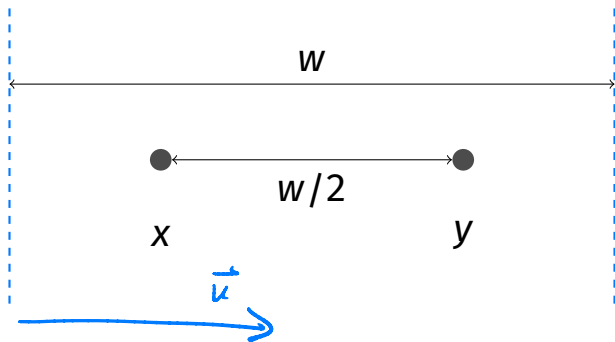
$$\mathbb{P}(h(x) = h(y)) \leq \frac{1}{3} \quad \text{when } d(x, y) \geq 2w$$

Intuition



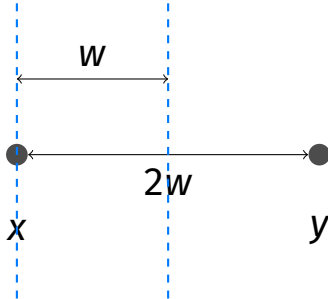
Proof: Close

- ▶ In worst case, grid is orthogonal to line between points.



Proof: Far

- ▶ Only possible if grid is close to parallel.

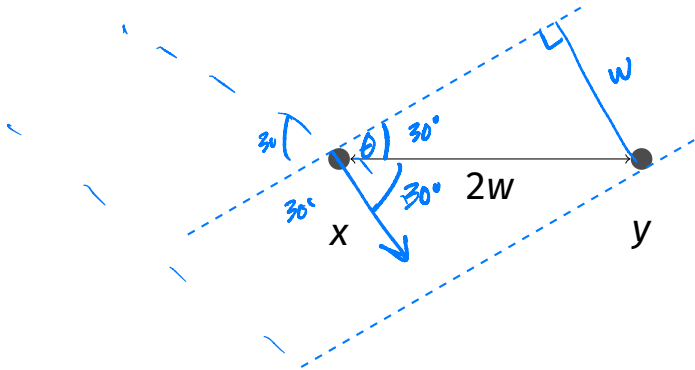


Proof: Far

$$\theta = \sin^{-1} \frac{1}{2} = 30^\circ$$

- ▶ Angle must be below 30° .

$$\frac{1}{3}$$



Amplification

- ▶ Lots of points have same hash.
- ▶ To be more selective, randomly select k hash functions for cell id.

$$\text{cell-id}(x) = (h_1(x), h_2(x), \dots, h_k(x))$$

Example: Random Projections

- ▶ In case of random projections.

$$\text{cell-id}(\vec{p}) = \left(\underbrace{\left\lfloor \frac{\vec{u}^{(1)} \cdot \vec{p}}{w} \right\rfloor}_{h_1}, \underbrace{\left\lfloor \frac{\vec{u}^{(2)} \cdot \vec{p}}{w} \right\rfloor}_{h_2}, \dots, \underbrace{\left\lfloor \frac{\vec{u}^{(k)} \cdot \vec{p}}{w} \right\rfloor}_{h_k} \right)$$

Collision Probability

- ▶ Remember:

$$P(h(x) = h(y)) \geq p_1 \text{ if close.}$$

$$P(h(x) = h(y)) \leq p_2 \text{ if far.}$$

$$h_1(x) = h_1(y) \cong P_1$$

$$h_2(x) = h_2(y) \cong P_1$$

$$\vdots$$

- ▶ Collision occurs if $h_i(x) = h_i(y) \forall i \in \{1, \dots, k\}$.
- ▶ Probability of collision...
 - ▶ if close: $\geq p_1^k$
 - ▶ if far: $\leq p_2^k$

Choosing k

- ▶ Want prob. of far points colliding to be small.
- ▶ Say, $1/n$.
- ▶ Set $p_2^k = 1/n$. Then

$$k = \log_{p_2} \frac{1}{n} = \frac{\log n}{\log 1/p_2}$$

Main Idea

We can use $k = \Theta(\log n)$ hash functions.

Main Idea

When using random projections as hash functions, we can use $k = \Theta(\log n)$ directions. This is usually much less than d .

But wait...

- ▶ Probability of close points colliding is p_1^k .
- ▶ Let $p_1 = p_2^\rho$. We'll have $\rho < 1$, since $p_2 < p_1$.
- ▶ Since $p_2^k = \frac{1}{n}$, we have $p_1^k = \frac{1}{n^\rho}$.
- ▶ This is **very small**.

Banding

- ▶ Before: one set of k hash functions.
- ▶ With **banding**: keep ℓ sets (**bands**) of k hash functions.
- ▶ To query NN of p , find points that are in the same cell as p in *any* of the bands.

Banding

- ▶ Probability of at least one match:

$$\underbrace{\frac{1}{n^{\rho}}}_{\text{collision in band 1}} + \underbrace{\frac{1}{n^{\rho}}}_{\text{collision in band 2}} + \dots + \underbrace{\frac{1}{n^{\rho}}}_{\text{collision in band } \ell} = \frac{\ell}{n^{\rho}}$$

- ▶ Want this to be ≈ 1 , so:

$$\ell = n^{\rho}$$

Main Idea

We should set the number of bands to be n^ρ . ρ depends on c , and is usually not small. For random projections, $\rho \approx .63$.

Analysis

- ▶ How efficient is LSH?
- ▶ Worst case, everything hashes to same bin: $O(n)$.
- ▶ In practice, much better.
- ▶ Requires **a lot** of memory. $\Theta(\ell n)$.

Other Distances

- ▶ LSH works for many different similarity measures.
- ▶ Random projections are for Euclidean distances.
- ▶ But other hashing approaches work for cosine distance, Jaccard distance, etc.

DSC 190

DATA STRUCTURES & ALGORITHMS

The Johnson-Lindenstrauss Lemma

Why does LSH work?

- ▶ Two approaches to understanding LSH.
- ▶ 1) Hashing view.
- ▶ **2) Dimensionality reduction view.**



Main Idea

The **Johnson-Lindenstrauss Lemma** says that, given n points in \mathbb{R}^d , you can reduce the dimensionality to $k \approx \log n$ while still preserving relative distances by randomly projecting onto a set of k unit vectors.

Claim

The **Johnson-Lindenstrauss Lemma** (Informal). Let X be a set of n points in \mathbb{R}^d . Let U be a matrix whose $k = O(\log(n)/\epsilon^2)$ rows are Gaussian random vectors in \mathbb{R}^d . Then for every $\vec{x}, \vec{y} \in X$,

$$\|\vec{x} - \vec{y}\| \leq (1 \pm \epsilon) \|U\vec{x} - U\vec{y}\|$$

LSH and J-L

- ▶ In LSH, we use $k = O(\log n)$ hash functions.
- ▶ If these hash functions are random projections, the J-L lemma tells that distances are largely preserved.

A Different View of LSH

- ▶ Given $p \in \mathbb{R}^d$, randomly project to \mathbb{R}^k with $k \approx \log n$.
- ▶ Let new coordinates be (y_1, y_2, \dots, y_k) .
- ▶ Use standard grid to assign cell id.

Main Idea

LSH (for Euclidean distances) (without banding) can be viewed as dimensionality reduction by random projections, followed by binning into a standard grid.

DSC 190

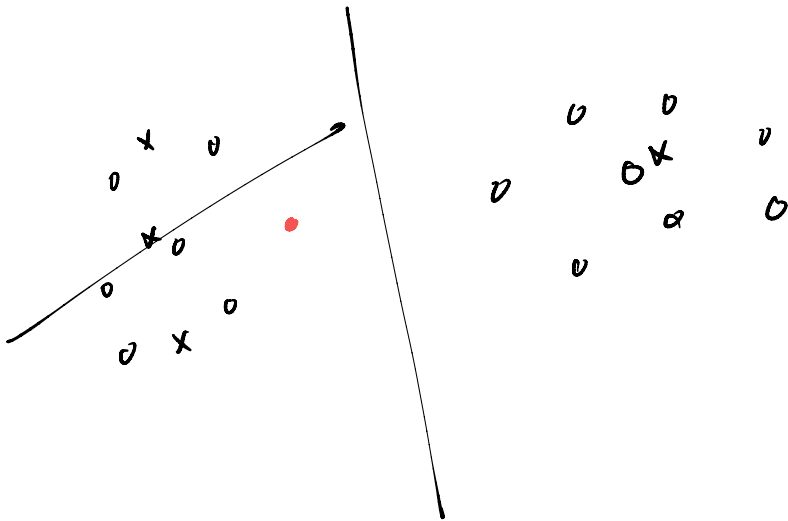
DATA STRUCTURES & ALGORITHMS

NN in Practice

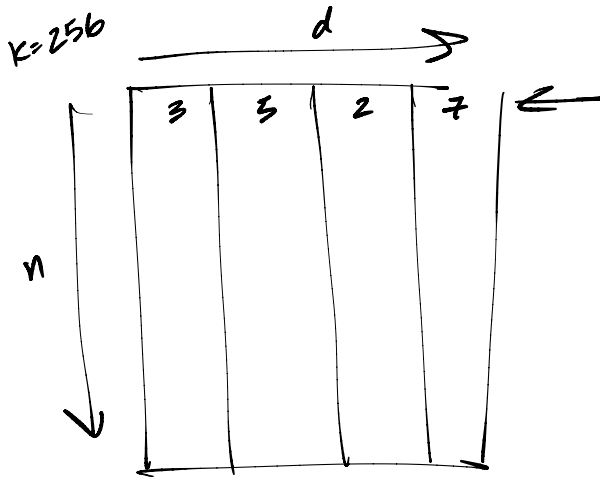
In Practice

- ▶ LSH is an important idea.
- ▶ Good performance in practice.
- ▶ But heuristic approaches are often faster.
- ▶ `faiss` and `annoy`, among others.

Hierarchical k-Means



Product Quantization



Navigable Small Worlds

