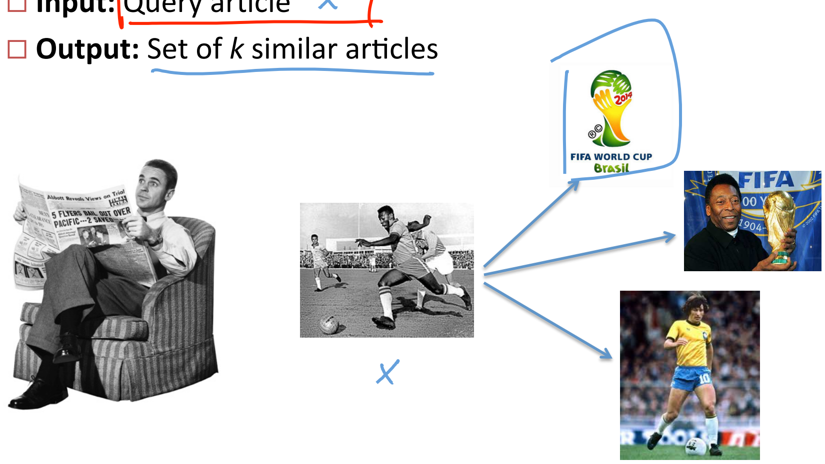
Case Study 2: Document Retrieval

Task Description: Finding Similar Documents

Machine Learning for Big Data
CSE547/STAT548, University of Washington
Emily Fox
April 14th, 2015

Task 1: Find Similar Documents

- **To begin…**
  - **Input:** Query article x
  - **Output:** Set of *k* similar articles



k-Nearest Neighbor

- Articles $X = \{x^1, \dots, x^N\}, \quad x^i \in \mathbb{R}^d$
- Query: $x \in \mathbb{R}^d$
- *k*-NN
  - Goal: find k articles in X closest to x
  - Formulation:

$$X^{NN} = \{x^{NN_1}, \dots, x^{NN_k}\} \subseteq X$$

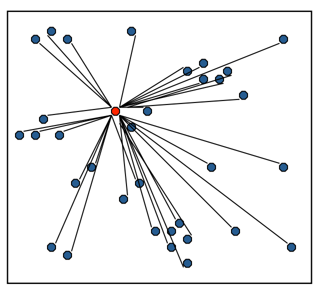
$$\text{s.t. } \forall x^i \in X \setminus X^{NN}$$

$$d(x^i, x) \geq \max_{x^{NN_i} \in X^{NN}} d(x^{NN_i}, x)$$

Issues with Search Techniques

- Naïve approach:
  Brute force search
  - Given a query point x
  - Scan through each point x^i
  - O(*N*) distance computations per 1-NN query!
  - O(*N*log*k*) per *k*-NN query!

    ↑ keep priority queue of top k + inserting into queue is log k

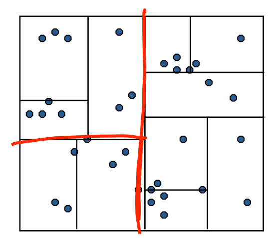


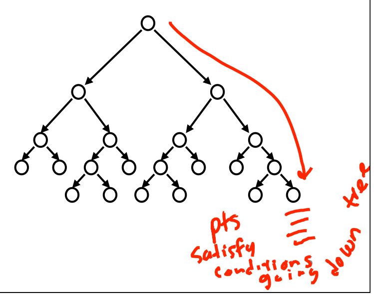
33 Distance Computations

- What if *N* is huge??? (and many queries)

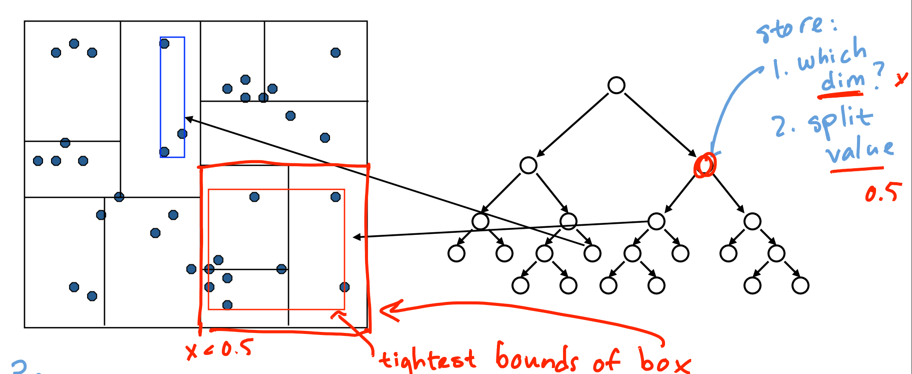
KD-Trees

- Smarter approach: ***kd-trees***
  - Structured organization of documents
    - Recursively partitions points into axis aligned boxes.
  - Enables more efficient pruning of search space
    - Examine nearby points first.
    - Ignore any points that are further than the nearest point found so far.
- ***kd-trees*** work "well" in "low-medium" dimensions d
  - We'll get back to this…





KD-Tree Construction



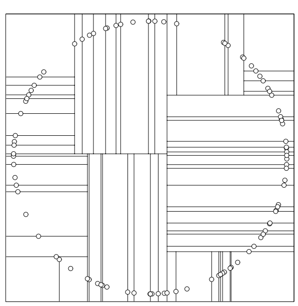
3. Keep one additional piece of information at each node:
   - The (tight) bounds of the points at or below this node.

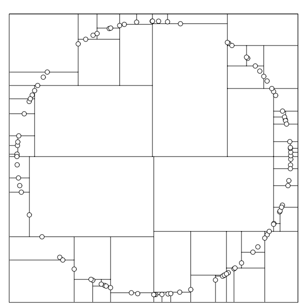
KD-Tree Construction

- Use heuristics to make splitting decisions:
- Which dimension do we split along?
  widest (or alternate)
- Which value do we split at?
  median of chosen split dim (or center)
- When do we stop?
  fewer than m pt ~~left~~ left
  or
  box hits minimum width

 7

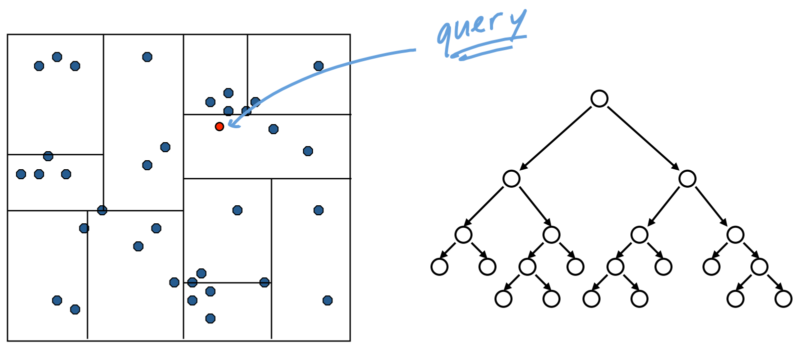
Many heuristics…


median heuristic


center-of-range heuristic

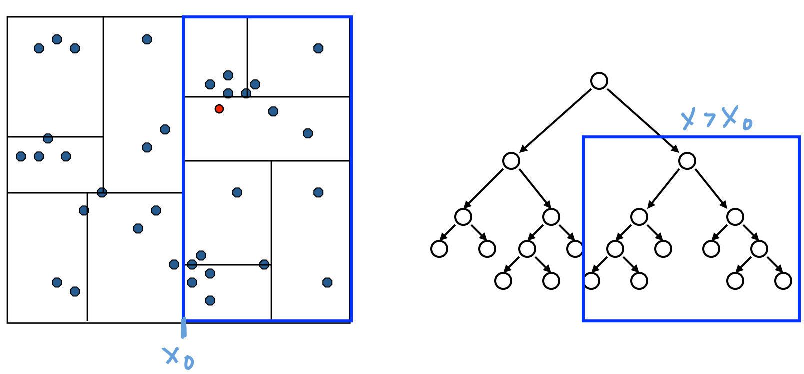
 8

Nearest Neighbor with KD Trees



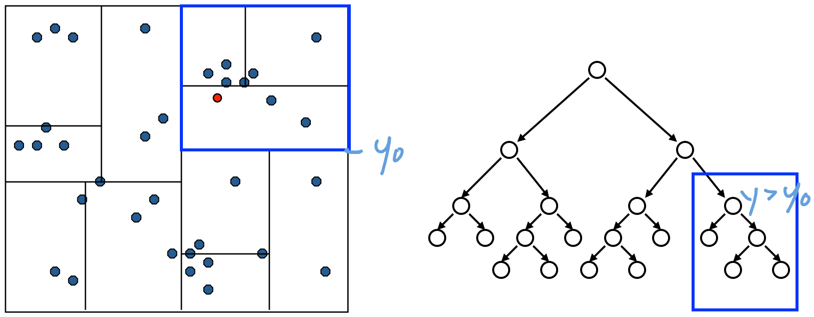
- Traverse the tree looking for the nearest neighbor of the query point.

Nearest Neighbor with KD Trees



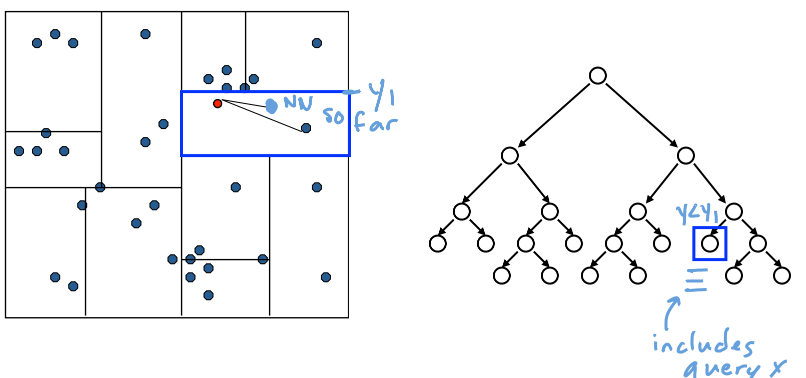
- <u>Examine nearby points first</u>:
  - Explore branch of tree closest to the query point first.

Nearest Neighbor with KD Trees



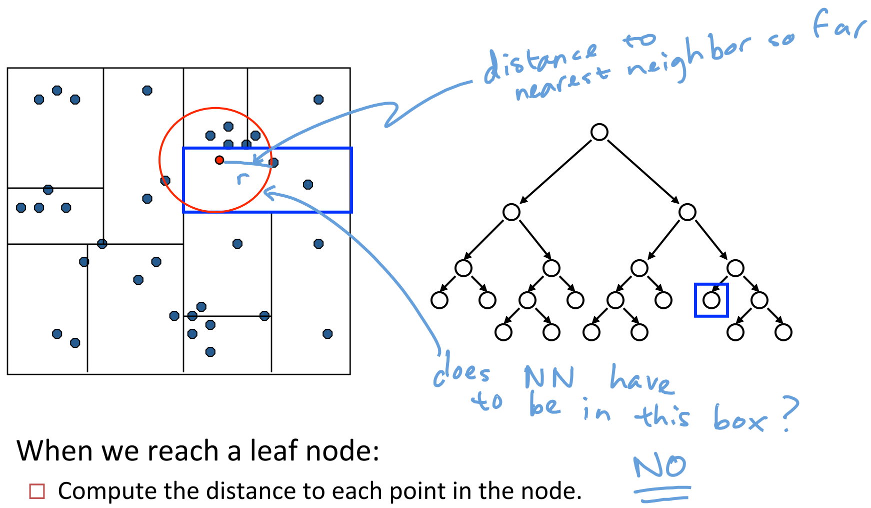
- Examine nearby points first:
  - Explore branch of tree closest to the query point first.

Nearest Neighbor with KD Trees



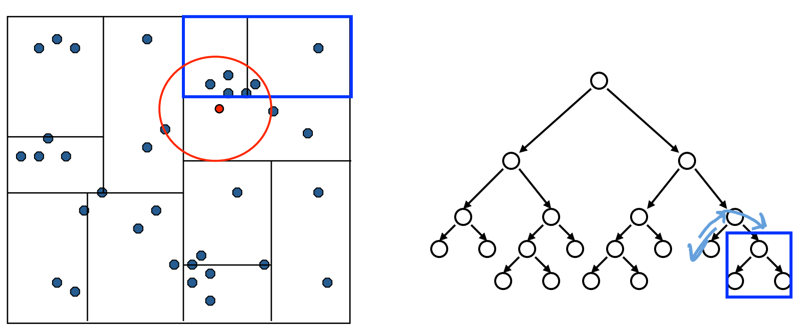
- When we reach a leaf node:
  - Compute the distance to each point in the node.

Nearest Neighbor with KD Trees



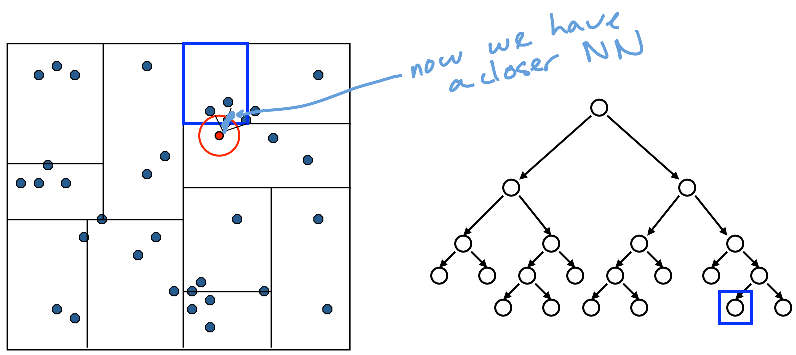
- When we reach a leaf node:
  - Compute the distance to each point in the node.

Nearest Neighbor with KD Trees



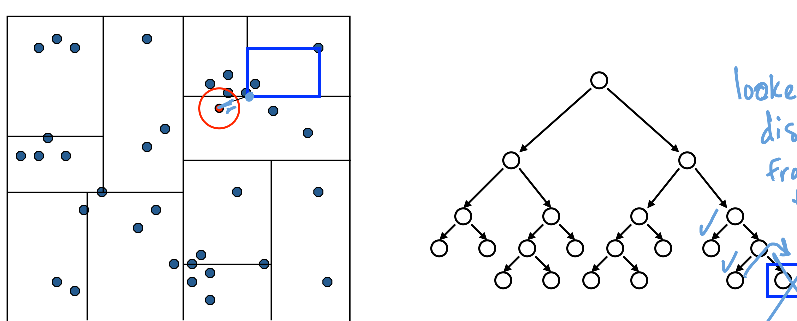
- Then backtrack and try the other branch at each node visited

Nearest Neighbor with KD Trees



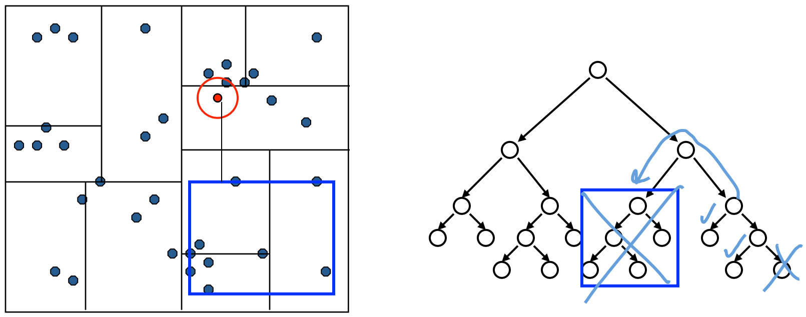
- Each time a new closest node is found, update the distance bound

Nearest Neighbor with KD Trees



- Using the distance bound and bounding box of each node:
  - Prune parts of the tree that could NOT include the nearest neighbor

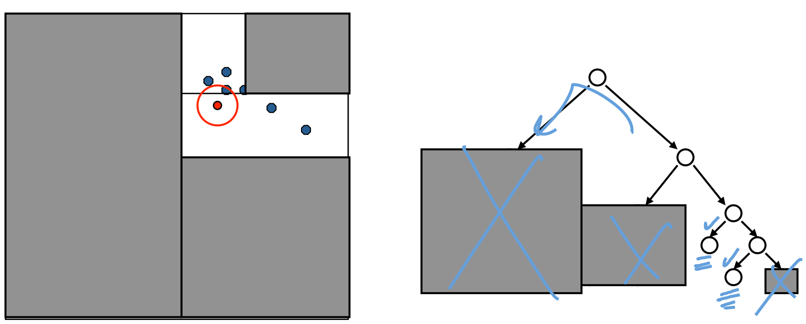
Nearest Neighbor with KD Trees



- Using the distance bound and bounding box of each node:
  - Prune parts of the tree that could NOT include the nearest neighbor

 17

Nearest Neighbor with KD Trees



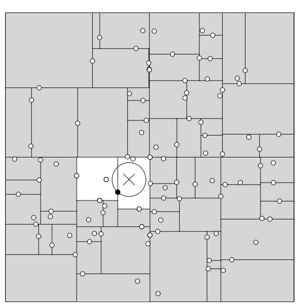
- Using the distance bound and bounding box of each node:
  - Prune parts of the tree that could NOT include the nearest neighbor

 18

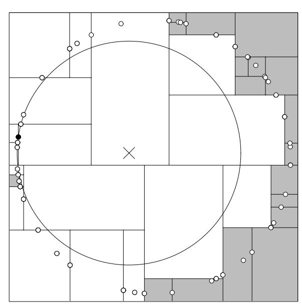
Complexity

- For (nearly) balanced, binary trees...
- Construction
  - Size: $2N-1 \rightarrow O(N)$
  - Depth: $O(\log N)$
  - Median + send points left right: $O(N)$ at every tree level (smart)
  - Construction time: $O(N \log N)$
- 1-NN query
  - Traverse down tree to starting point: $O(\log N)$
  - Maximum backtrack and traverse: $O(N)$ worst case
  - Complexity range: $O(\log N) \rightarrow O(N)$
- Under some assumptions on distribution of points, we get O(logN) but exponential in *d* (see citations in reading)

Complexity



pruned many (closer to $O(\log N)$)



pruned few (closer to $O(N)$)

Complexity for *N* Queries

- Ask for nearest neighbor to each document

  N queries

- Brute force 1-NN: $O(N^2)$

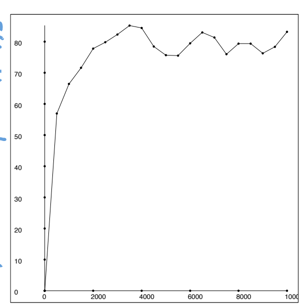
- kd-trees: $O(N \log N) \rightarrow O(N^2)$

  potentially large savings

Inspections vs. *N* and *d*

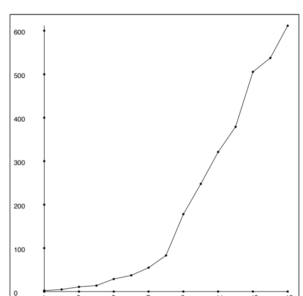
log N

# of inspections



N

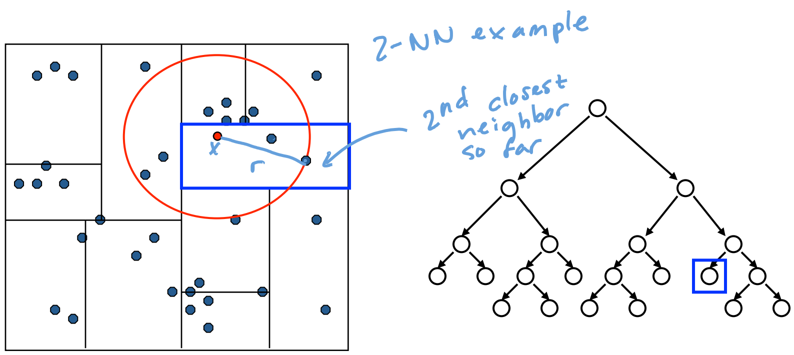
exponential



d

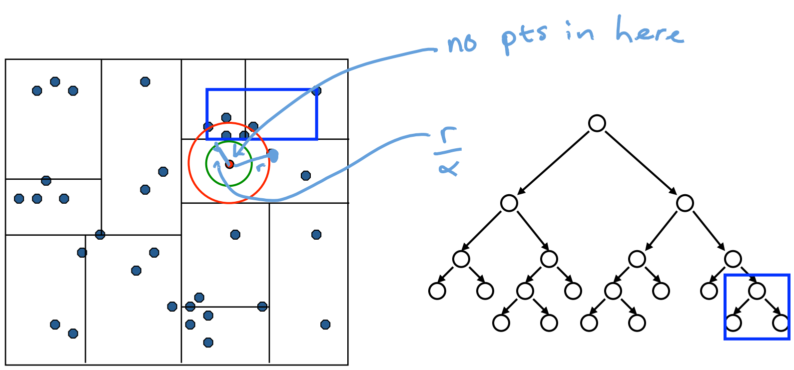
$(x^i \in \mathbb{R}^d)$

K-NN with KD Trees



- Exactly the same algorithm, but maintain distance as distance to furthest of current *k* nearest neighbors
- Complexity is: $O(k \log N)$

Approximate K-NN with KD Trees



- **Before:** Prune when distance to bounding box > r
- **Now:** Prune when distance to bounding box > $\frac{r}{\alpha}$ $\alpha > 1$
- Will prune more than allowed, but can guarantee that if we return a neighbor at distance r, then there is no neighbor closer than r/α.
- In practice this bound is loose…Can be closer to optimal.
- Saves lots of search time at little cost in quality of nearest neighbor.

Wrapping Up – Important Points

kd-trees

- Tons of variants
  - On construction of trees (heuristics for splitting, stopping, representing branches…)
  - Other representational data structures for fast NN search (e.g., ball trees,…)

Nearest Neighbor Search

- Distance metric and data representation are crucial to answer returned

For both…

- High dimensional spaces are hard! large d
  - Number of kd-tree searches can be exponential in dimension
    - Rule of thumb… $N >> 2^d$… Typically useless.
  - Distances are sensitive to irrelevant features
    - Most dimensions are just noise → Everything equidistant (i.e., everything is far away)
    - Need technique to learn what features are important for your task

What you need to know

- Document retrieval task
  - Document representation (bag of words)
  - tf-idf
- Nearest neighbor search
  - Formulation
  - Different distance metrics and sensitivity to choice
  - Challenges with large N
- kd-trees for nearest neighbor search
  - Construction of tree
  - NN search algorithm using tree
  - Complexity of construction and query
  - Challenges with large d

Acknowledgment

- This lecture contains some material from Andrew Moore's excellent collection of ML tutorials:
  - http://www.cs.cmu.edu/~awm/tutorials
- In particular, see:
  - http://grist.caltech.edu/sc4devo/.../files/sc4devo_scalable_datamining.ppt

Case Study 2: Document Retrieval

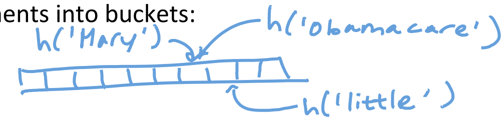
Locality-Sensitive Hashing Random Projections for NN Search

Machine Learning for Big Data
CSE547/STAT548, University of Washington
Emily Fox
April 14th, 2015

Using Hashing to Find Neighbors

- KD-trees are cool, but…
  - Non-trivial to implement efficiently
  - Problems with high-dimensional data
- Approximate neighbor finding…
  - Don't find exact neighbor, but that's OK for many apps, especially with Big Data
- What if we could use hash functions: $h: \mathcal{X} \rightarrow \{1, \dots, m\}$
  - Hash elements into buckets:

$T =$



typical version where we keep list of words in every bin

  - Look for neighbors that fall in same bucket as x:

$h(x) = i$, for all $y \in T[h(x)=i]$ look for neighbors there

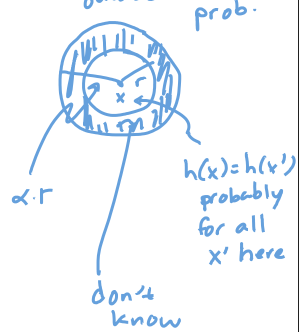
- But, by design…

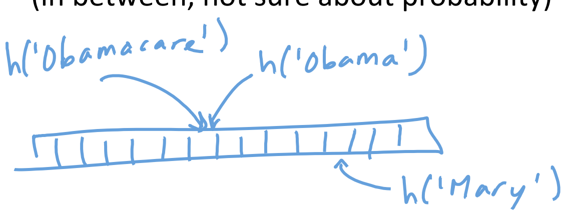
$P(h(x)=h(x')) = \frac{1}{m} \quad \forall x'$

even if $d(x,x')$ is low $\not\Rightarrow h(x) \approx h(x')$

Locality Sensitive Hashing (LSH)

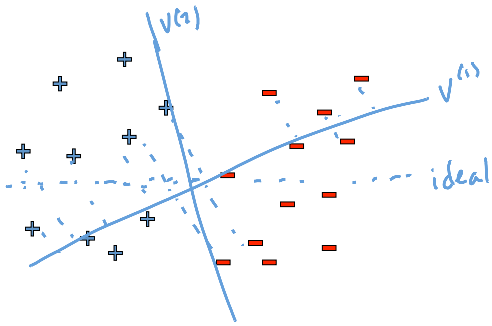
- A LSH function *h* satisfies (for example), for some similarity function *d*, for *r*>0, *α*>1:
  - $d(x,x') \le r$, then $P(h(x)=h(x'))$ is high
  - $d(x,x') > \alpha.r$, then $P(h(x)=h(x'))$ is low
  - (in between, not sure about probability)



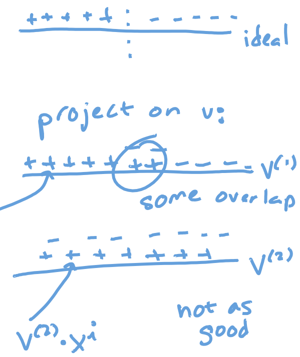


now look for pts hashing to same bin

Random Projection Illustration



$v^{(1)}\cdot x^i$



- Pick a random vector *v*:
  - Independent Gaussian coordinates

    $v_i \overset{iid}{\sim} N(0,1)$ define d-dim vector $[v_1 \ldots v_d]$, each $\sim N(0,1)$
- Preserves separability for most vectors
  - Gets better with more random vectors

    $v^{(1)} = [v_1^{(1)} \ldots v_d^{(1)}]$

 31

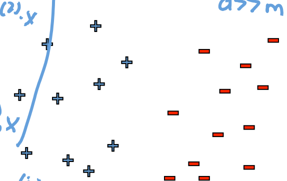
Multiple Random Projections: Approximating Dot Products

Projection $d \to m$ assuming $d \gg m$

- Pick m random vectors $v^{(i)}$:
  - Independent Gaussian coordinates

    projection vectors $v_j^{(i)} \overset{iid}{\sim} N(0,1)$

    $\phi(x) = [v^{(1)}\cdot x, v^{(2)}\cdot x, \ldots, v^{(m)}\cdot x]$



- Approximate dot products:
  - Cheaper, e.g., learn in smaller *m* dimensional space

    $x\cdot y \approx \frac{1}{m}\phi(x)\cdot\phi(y) = \frac{1}{m}\sum_{i=1}^{m}(v^{(i)}\cdot x)\cdot(v^{(i)}\cdot y)$ ← see HW
- Only need logarithmic number of dimensions!
  - N data points, approximate dot-product within ε>0:

$$m = \mathcal{O}\left(\frac{\log N}{\epsilon^2}\right)$$

If I have big data N → very large, but only need log N random vecs.

- But all sparsity is lost

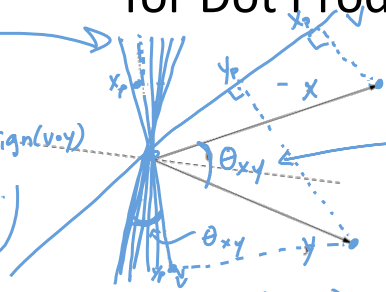
  $v^{(i)}$ are dense wp 1 ⇒ $v^{(i)}\cdot x \neq 0$ wp 1 } even if x is sparse, $\phi(x)$ is not sparse

 32

LSH Example: Sparser Random Projection for Dot Products

STEP 1: Transform to binary vec.

Only vectors here lead to sign(v·x) ≠ sign(v·y)
(happens w.p. $\frac{\theta_{xy}}{\pi}$)



close to 0

- Pick random vectors $v^{(i)} \sim N(0, I)$ ← m in total
- Simple 0/1 projection: $\phi_i(x) = \begin{cases} 1 & \text{if } \mathrm{sign}(v^{(i)} \cdot x) \geq 0 \\ 0 & \text{if } \mathrm{sign}(v^{(i)} \cdot x) < 0 \end{cases}$
- Now, each vector is approximated by a bit-vector
  $\phi(x) = \underbrace{(0, 0, 1, 0, 1, 1, 1, 0)}_{m\text{-dim}}$
- Dot-product approximation:

$$\frac{x \cdot y}{\|x\| \, \|y\|} = \cos\theta_{xy} \approx \cos\left(\frac{\pi}{m} \mathrm{HammDist}(\phi(x), \phi(y))\right)$$

LSH for Approximate Neighbor Finding

STEP 2: Use $\phi(x)$ as index in hash table

- Very similar elements fall in exactly same bin:

x → $T[\phi(x)]$ turn bit vector into integer

T

$\phi(x)$: m-dim

2^m bins

- And, nearby bins are also nearby:

  in terms of Hamming dist.

- Simple neighbor finding with LSH:
  - For bins b of increasing hamming distance to $\phi(x)$:
    - Look for neighbors of x in bin b

  $\forall y$ in $T[b]$, compute $d(x,y)$
  keep closest
  - Stop when run out of time
- Pick m such that $N/2^m$ is "smallish"
  (in practice)

$S(x,y) = \cos\theta_{xy}$ ← similarity, not dist.
"cosine similarity"
$= \cos\left(\frac{\pi}{m} \mathrm{Hamm}(\phi(x), \phi(y))\right)$

$S(x,y)$ high
$\Rightarrow \cos\theta_{xy} \approx 1$
$\Rightarrow \theta_{xy} \approx 0$
$\Rightarrow \mathrm{HammDist} \approx 0$
$\Rightarrow x, y$ same bin

similar: $x \rightarrow \phi(x) = (0,1,1,0)$
$y \rightarrow \phi(y) = (0,1,1,0)$ or $(1,1,1,0)$
but not $(1,0,0,1)$

Hash Kernels: Even Sparser LSH for Learning

ASIDE: formalizing hash kernels as random proj.

- Two big problems with random projections:
  - Data is sparse, but random projection can be a lot less sparse
  - You have to sample m huge random projection vectors
    - And, we still have the problem with new dimensions, e.g., new words
- **Hash Kernels**: Very simple, but powerful idea: combine sketching for learning with random projections
- Pick 2 hash functions:
  - h : Just like in Count-Min hashing $h: \mathcal{X} \to \{1,\dots,m\}$
  - ξ : Sign hash function $\xi: \mathcal{X} \to \{+1,-1\}$
    - Removes the bias found in Count-Min hashing (see homework)
- Define a "kernel", a projection ϕ for x: $h[j]: \xi(j)\cdot X_j$

$\phi(x) =$

$$\phi_i(x) = \sum_{j:h(j)=i} \xi(j)\cdot X_j$$

Hash Kernels, Random Projections and Sparsity

$$\phi_i(\mathbf{x}) = \sum_{j:h(j)=i} \xi(j)\mathbf{x}_j$$

- Hash Kernel as a random projection:

Mary, had, a, little, lamb

$X = (0\ 0\ 0\ 1\ 0\ 0\ 0\ 1\ 1\ 0\ 0 \dots 2\ 0\ 0\ 0\ 2)$

$\phi(x) =$ -1 $+1$ $1\ 2 \dots m$

$h(\text{'Mary'}) = 7$
$h(\text{'little'}) = 7$
$\xi(\text{'Mary'}) = -1$
$\xi(\text{'little'}) = +1$

- What is the random projection vector for coordinate i of ϕ_i:

what is $v^{(i)}$ s.t. $\phi_i(x) = v^{(i)} \cdot x$?

$v^{(i)}$ mostly 0, non-zero $\forall j : h(j) = i$ { $v_j^{(i)} \in \{+1,-1\}$ determined by $\xi(j)$

m in total

- Implicitly define projection by h and ξ, so no need to compute apriori and automatically deals with new dimensions
- Sparsity of ϕ, if x has s non-zero coordinates:

How many times does x_j "show up" in $\phi(x)$? Once! at $h[j]$

$\Rightarrow$ If sparsity of $X = S \geq$ sparsity of $\phi(x)$ ~ has fewer non-zero entries

What you need to know

- Locality-Sensitive Hashing (LSH): nearby points hash to the same or nearby bins
- LSH uses random projections
  - Only $O(\log N/\varepsilon^2)$ vectors needed
  - But vectors and results are not sparse
- Use LSH for nearest neighbors by mapping elements into bins
  - Bin index is defined by bit vector from LSH
  - Find nearest neighbors by going through bins
- Hash kernels:
  - Sparse representation for feature vectors
  - Very simple, use two hash functions
    - Can even use one hash function, and take least significant bit to define ξ
  - Quickly generate projection $\phi(x)$
  - Learn in projected space