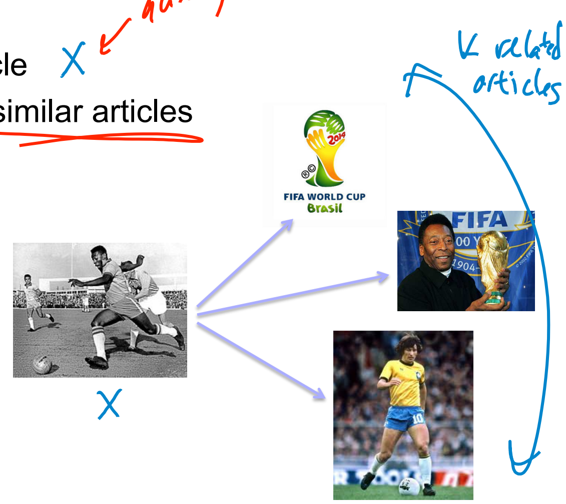
Case Study 2: Document Retrieval

Task Description: Finding Similar Documents

Machine Learning for Big Data
CSE547/STAT548, University of Washington
Emily Fox
January 21st, 2014

Task 1: Find Similar Documents

- **To begin…**
  - **Input:** Query article X ← query
  - **Output:** Set of k similar articles



k-Nearest Neighbor

- Articles $X = \{x^1, \dots, x^N\}, \quad x^i \in \mathbb{R}^d$

  ← all articles in the world

- Query: $x \in \mathbb{R}^d$

- k-NN
  - Goal: find k articles in X closest to x
  - Formulation:

    k articles

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

    $$\text{s.t. } \forall x^i \in X \setminus X^{NN}$$ ← "not in nearest neighbor set"

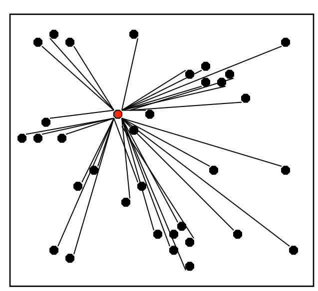
    $$d(x^i, x) \geq \max_{x^{NN_i} \in X^{NN}} d(x^{NN_i}, x)$$

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(Nlog$k$) per k-NN query!

    ↑ keep priority queue of top k

    + inserting into queue log k



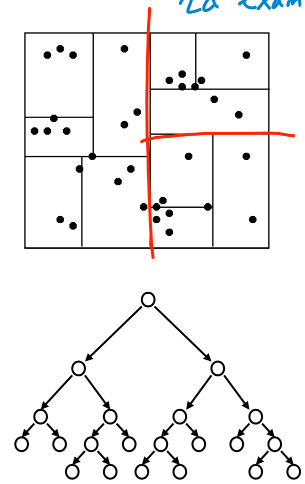
33 Distance Computations

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

KD-Trees

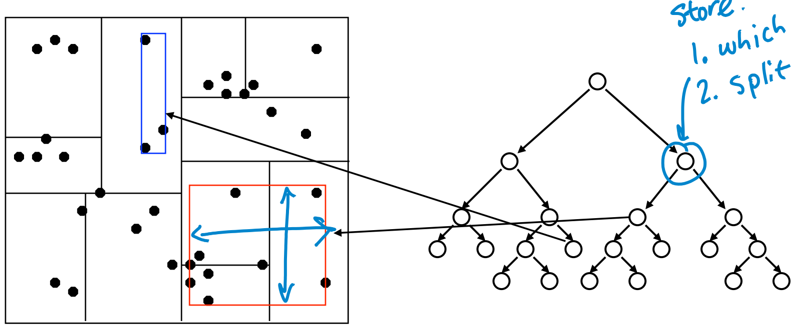
2d examples

- 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…



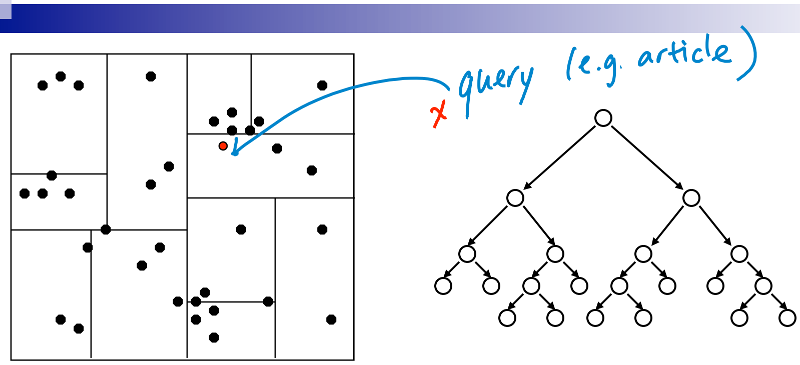
KD-Tree Construction

store:
1. which dim? d_j
2. split value v



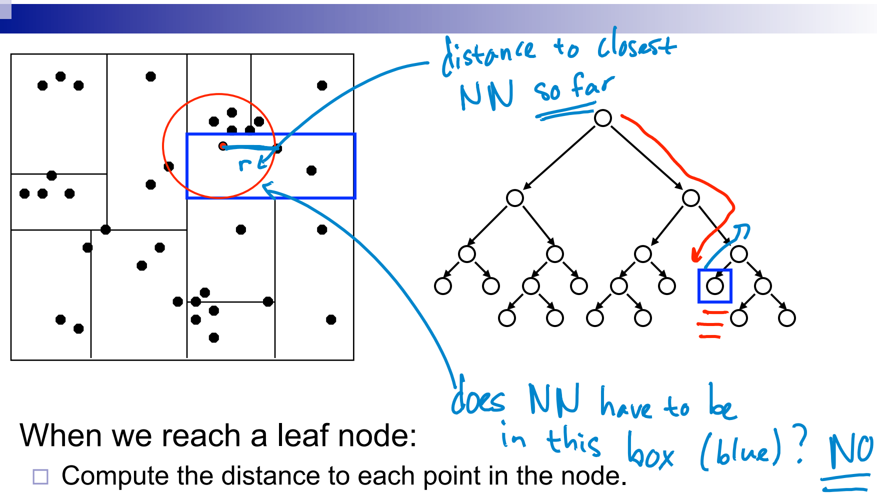
3.
- Keep one additional piece of information at each node:
  - The (tight) bounds of the points at or below this node.

Nearest Neighbor with KD Trees



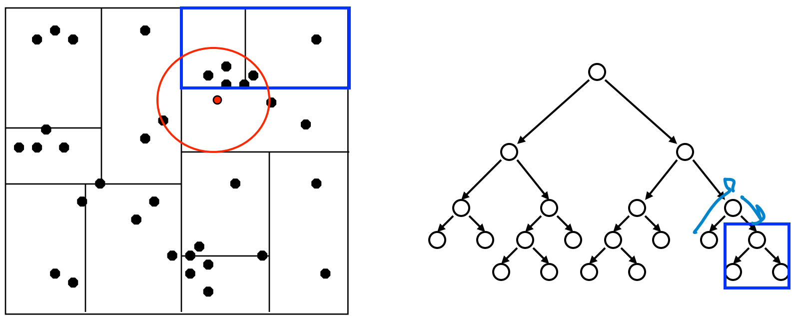
- Traverse the tree looking for the nearest neighbor of the query point.

Nearest Neighbor with KD Trees



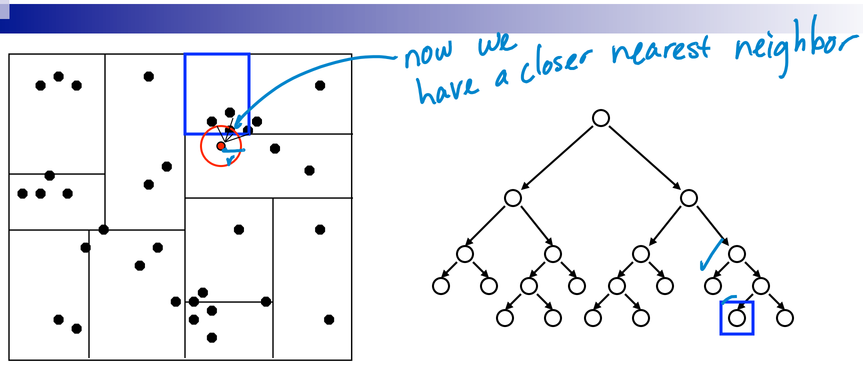
- When we reach a leaf node:
  - Compute the distance to each point in the node.

Nearest Neighbor with KD Trees



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

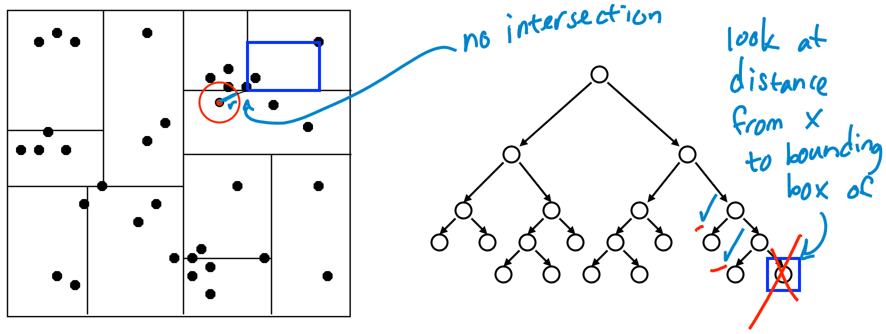
Nearest Neighbor with KD Trees



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

Dist. from x to closest pt of bounding box < r, so search for NN

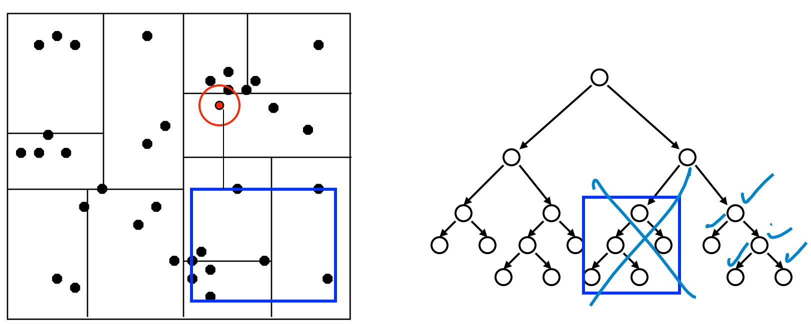
Nearest Neighbor with KD Trees



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

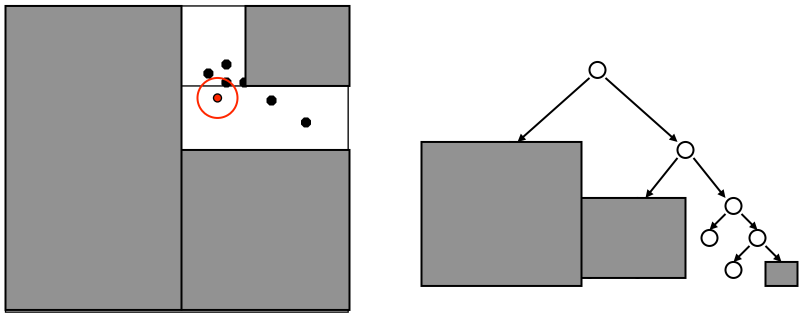
No article in this box could be the NN.

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

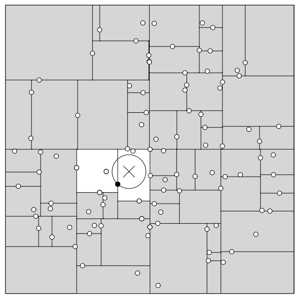


- Using the distance bound and bounding box of each node:
  - Prune parts of the tree that could NOT include the nearest neighbor

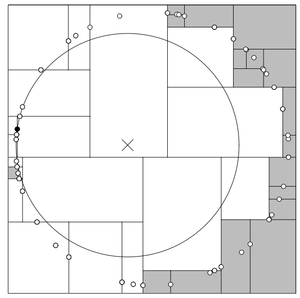
Complexity

- 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 level of tree (smart)
  - Construction time: $O(N \log N)$
- 1-NN query
  - Traverse down tree to starting point: $O(\log N)$ go to leaf node
  - 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(log$N$) but exponential in d (see citations in reading) $x \in \mathbb{R}^d$

Complexity



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



pruned only a few
$O(N)$

Complexity for N Queries

- Ask for nearest neighbor to each document

  N queries

- Brute force 1-NN:

  $O(N^2)$

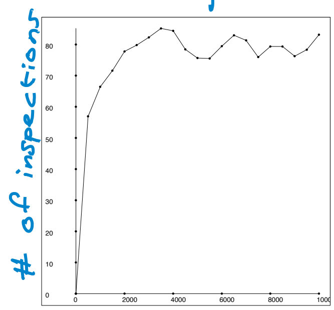
- kd-trees:

  $O(N \log N) \rightarrow O(N^2)$

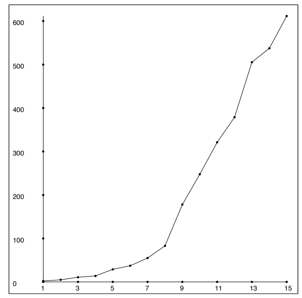
  potentially large savings!

Inspections vs. *N* and *d*

log N   exponential in d



N



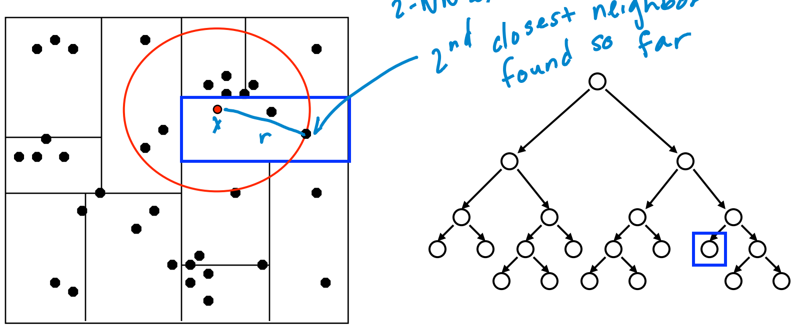
d

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

K-NN with KD Trees

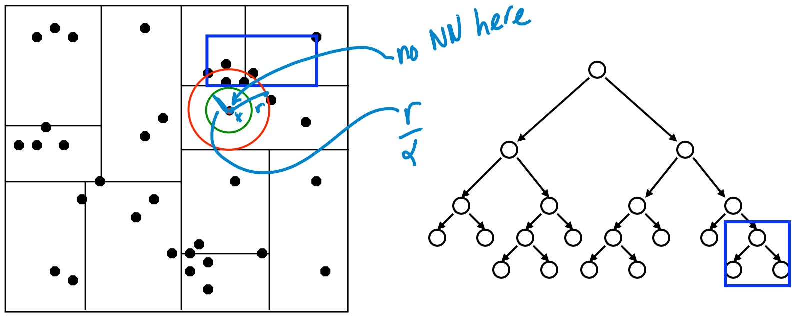
2-NN example

2nd closest neighbor found so far



- 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 > r/α for α>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! ← important
  - 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

Locality-Sensitive Hashing Random Projections for NN Search

Machine Learning for Big Data
CSE547/STAT548, University of Washington

Emily Fox

January 21st, 2014

Using Hashing to Find Neighbors

- KD-trees are cool, but…
  - Non-trivial to implement efficiently ←
  - Problems with high-dimensional data ← d large
- 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: $h: \mathcal{X} \to \{1, \ldots, m\}$
  - Hash elements into buckets:

    h('Mary')   h('Obamacare')

    T :

    h('lamb')

    typical hash table where we keep list 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

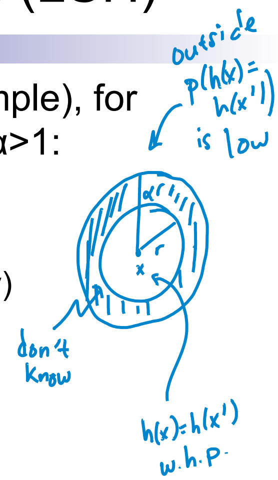
- But, by design…

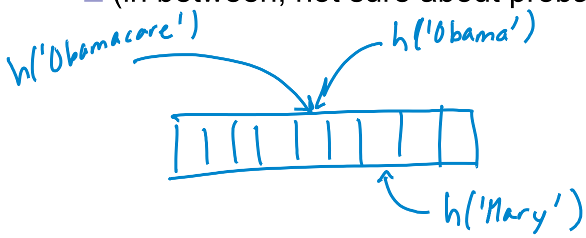
$$P(h(x) = h(x')) = \frac{1}{m} \quad \forall x'$$

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

Locality Sensitive Hashing (LSH)

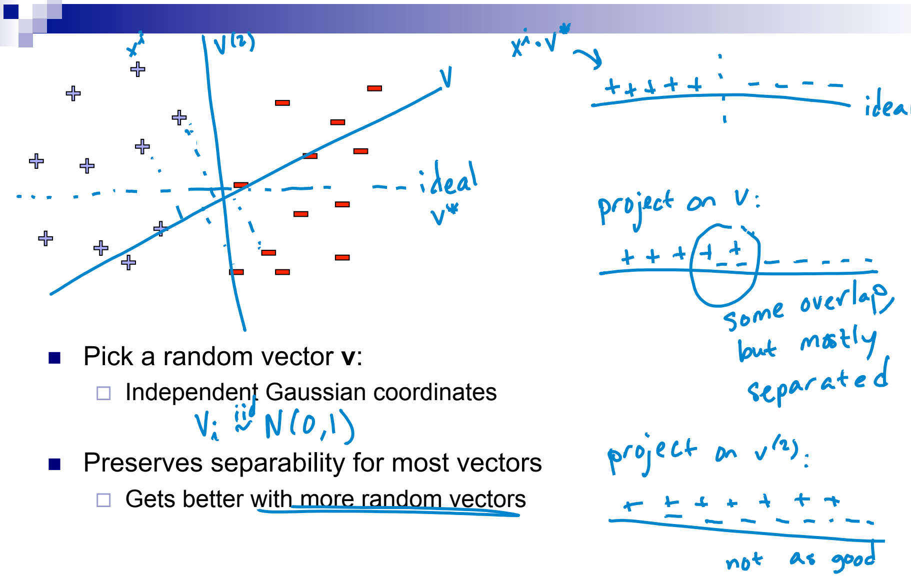
- A LSH function h satisfies (for example), for some ~~similarity~~ distance function d, for $r>0$, $\alpha>1$:
  - $d(\mathbf{x},\mathbf{x}') \le r$, then $P(h(\mathbf{x})=h(\mathbf{x}'))$ is high
  - $d(\mathbf{x},\mathbf{x}') > \alpha.r$, then $P(h(\mathbf{x})=h(\mathbf{x}'))$ is low
  - (in between,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





now look for pts hashing to same bin

Random Projection Illustration



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

    $v_i \overset{iid}{\sim} N(0,1)$
- Preserves separability for most vectors
  - Gets better with more random vectors

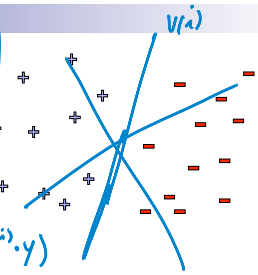
Multiple Random Projections: Approximating Dot Products

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

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

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

$\phi(x) = \begin{pmatrix} v^{(1)} \cdot x \\ v^{(2)} \cdot x \\ \vdots \\ v^{(m)} \cdot x \end{pmatrix}$



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

$x \cdot y \approx \frac{1}{m} \phi(x) \cdot \phi(y) = \frac{1}{m} \sum_{i=1}^{m} (v^{(i)} \cdot x) \cdot (v^{(i)} \cdot y)$

- Only need logarithmic number of dimensions!
  - *N* data points, approximate dot-product within ε>0:

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

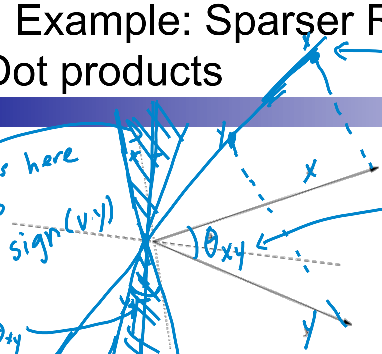
If I have big data, N → very large, but only need log N rand. vectors

- 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 (w.h.p.)

LSH Example: Sparser Random Projection for Dot products



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

then, w.h.p. sign(v·x) = sign(v·y)

- Pick random vectors $\mathbf{v}^{(i)} \sim N(0, I)$
- Simple 0/1 projection: $\phi_i(x) = \begin{cases} 1 & \text{if } \operatorname{sign}(v^{(i)} \cdot x) \geq 0 \\ 0 & \text{if } \operatorname{sign}(v^{(i)} \cdot x) < 0 \end{cases}$
- Now, each vector is approximated by a bit-vector

$\phi(x) = (0, 0, 1, 0, 1, 1, 1, 0)$

- Dot-product approximation:

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

LSH for Approximate Neighbor Finding

- Very similar elements fall in exactly same bin:

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

  T: $\quad$ 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 h(**x**):
    - Look for neighbors of **x** in bin *b*

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

$S(x,y) = \cos \theta_{xy}$ "cosine similarity"

$\approx \cos\left(\frac{\pi}{m} \text{ Hamm Dist}(\phi(x), \phi(y))\right)$

$S(x,y)$ high

$\Rightarrow \cos \theta_{xy} \approx 1$

$\Rightarrow \theta_{xy} \approx 0$

$\Rightarrow$ Hamm Dist ≈ 0

$\Rightarrow$ x, y in same bin

Hash Kernels: Even Sparser LSH for Learning

ASIDE: How do hash kernels relate to random projections?

- 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

  $v^{(i)}$ dense vectors w/ 40M entries
- **Hash Kernels**: Very simple, but powerful idea: combine sketching for learning with random projections
- Pick 2 hash functions:
  - *h* : Just like in Min-Count hashing $\quad h: \mathcal{X} \rightarrow \{1, \ldots, m\}$
  - ξ : Sign hash function $\quad \xi: \mathcal{X} \rightarrow \{+1, -1\}$
    - Removes the bias found in Min-Count hashing (see homework)
- Define a "kernel", a projection ϕ for **x**:

  For each non-zero x_j, add to bin $h[j]$: $\xi(j) \cdot x_j$

  $\phi(x) =$

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

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

$X = (0,0,0,1,0,0,1,1,0,0,0,2,0,0,2)$

Saw before that hash kernels preserve dot prod.

$\phi(x) =$ +1

$h('a') = 7 \quad \xi('a') = -1$

$h('lamb') = 7 \quad \xi('lamb') = +1$

- Random projection vector for coordinate *i* of ϕ_i:

$v^{(i)}$ mostly 0, non-zero $\forall j : h(j) = i$

here, $v_j^{(i)} \in \{+1, -1\}$

$v^{(i)} = (0,0,0,1,0,-1,-1,0,0)$

m in total

- Implicitly define projection by h and ξ, so no need to compute apriori and automatically deal with new dimensions
- Sparsity of ϕ, if $\mathbf{x}$ has *s* non-zero coordinates:

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

$\Rightarrow$ sparsity of $X = S \geq$ sparsity of $\phi(x)$

# of non-zeros

What you need to know

- Locality-Sensitive Hashing (LSH): nearby points hash to the same or nearby bins in terms of Hamming dist.
- LSH use random projections
  - Only O(log N/ε^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 function
    - Can even use one hash function, and take least significant bit to define ξ
  - Quickly generate projection $\phi(\mathbf{x})$
  - Learn in projected space