Nearest Neighbor Search:

Retrieving Documents

Emily Fox & Carlos Guestrin

Machine Learning Specialization

University of Washington

Retrieving documents of interest

Document retrieval

- Currently reading article you like

Document retrieval

- Currently reading article you like
- **Goal**: Want to find similar article

Document retrieval

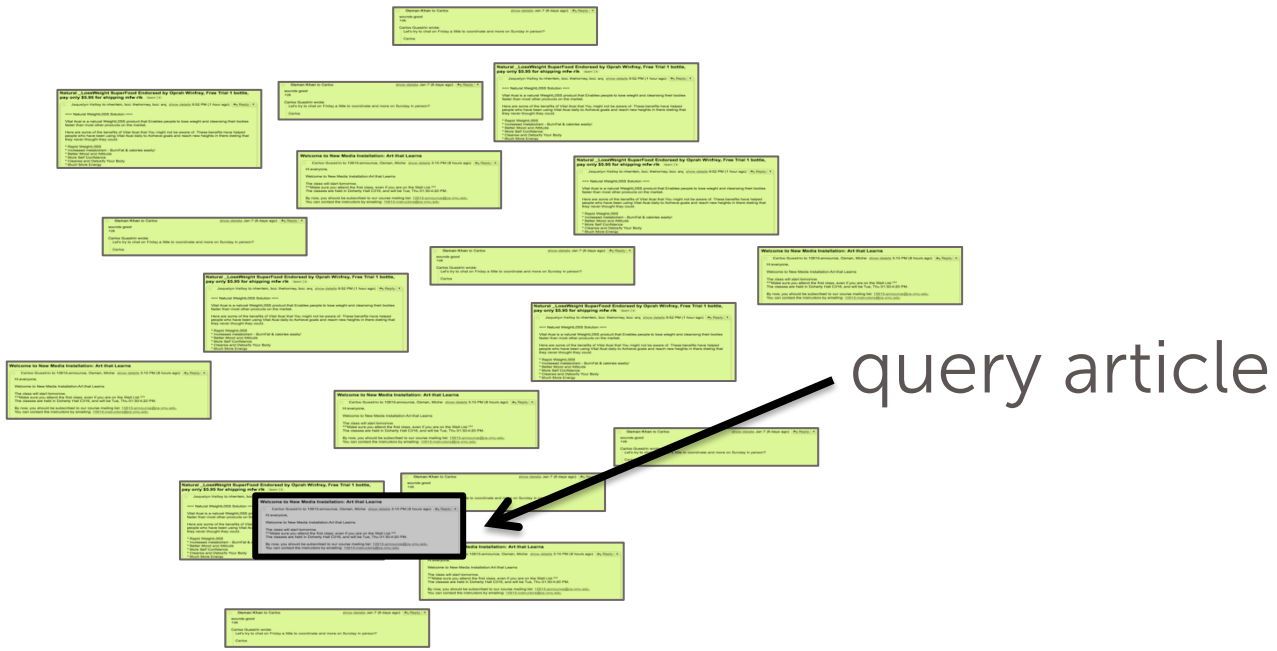
Challenges

- How do we measure similarity?
- How do we search over articles?

Retrieval as k-nearest neighbor search

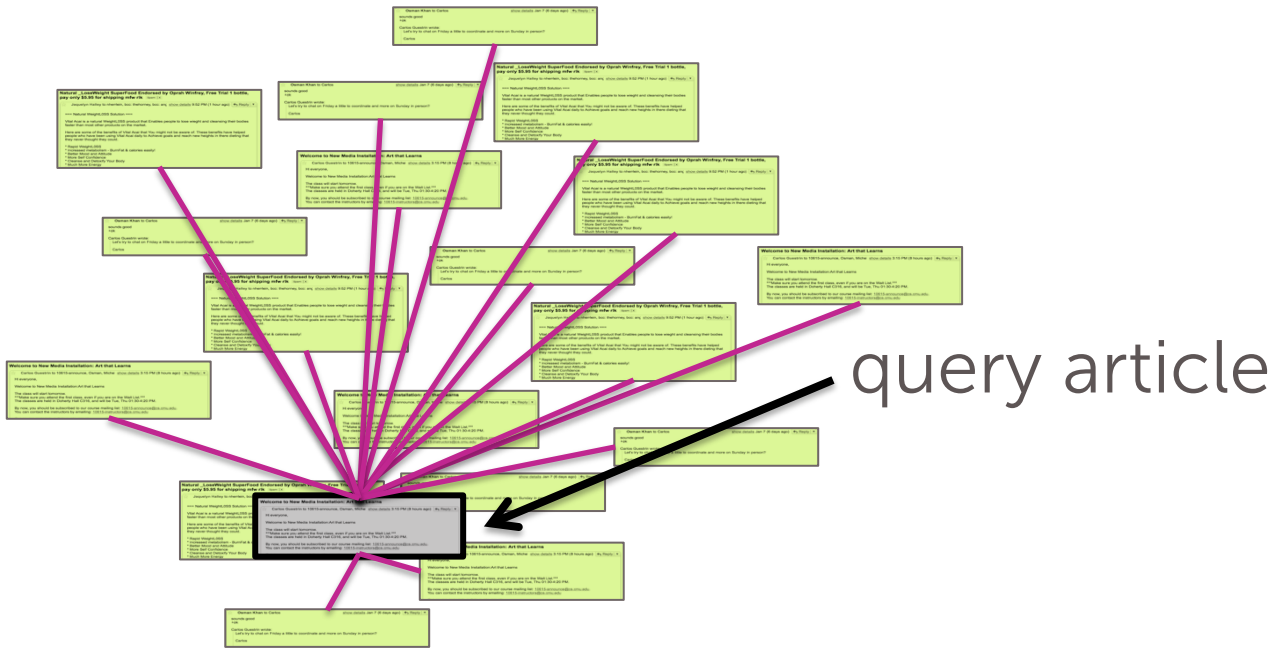
1-NN search for retrieval

Space of all articles,
organized by similarity of text



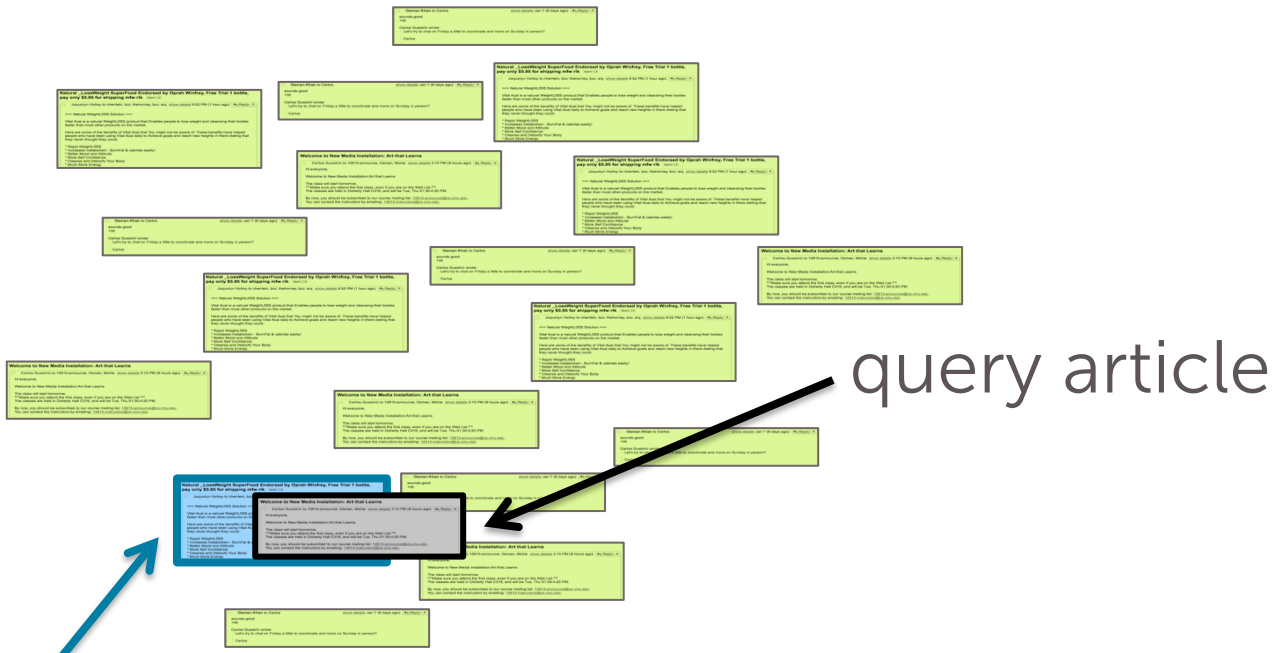
Compute distances to all docs

Space of all articles,
organized by similarity of text



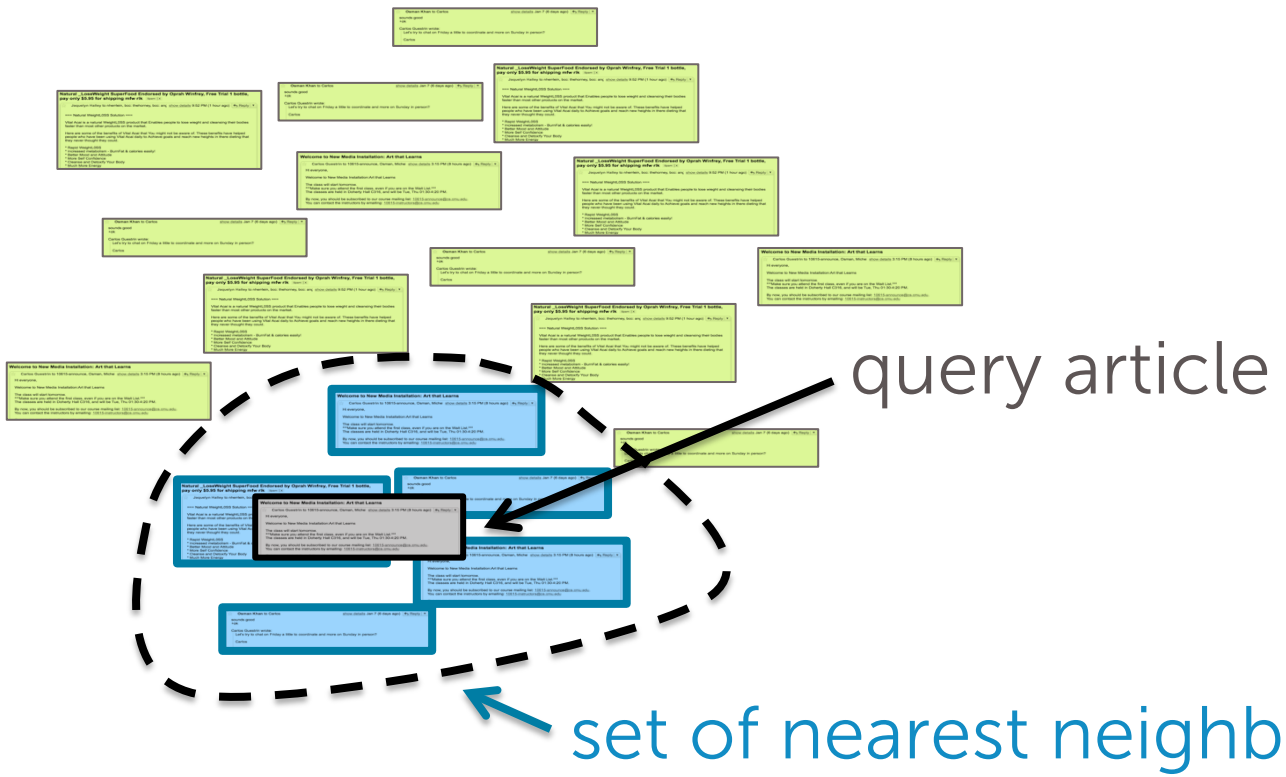
Retrieve “nearest neighbor”

Space of all articles,
organized by similarity of text



Or set of nearest neighbors

Space of all articles,
organized by similarity of text



1-NN algorithm

1 – Nearest neighbor

- **Input:** Query article  : $\mathbf{x}_q$

  Corpus of documents (N docs)

   : $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N$

- **Output:** *Most* similar article  $\leftarrow x^{NN}$

Formally: $x^{NN} = \min_{x_i} \text{distance}(x_q, x_i)$

1-NN algorithm

closest document

Initialize **Dist2NN** = ∞,  = ∅

For i=1,2,…,N

query document

x_q

x_i

Compute: δ = **distance(**  ,  **)**

document i from corpus

If δ < **Dist2NN**

x_i

set  = 

set **Dist2NN** = δ

Return most similar document 

x^{NN}

closest document in corpus to query article

k-NN algorithm

k – Nearest neighbor

- **Input:** Query article  : $\mathbf{x}_q$

  Corpus of documents

   : $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_N$

- **Output:** *List of k* similar articles

Formally:

$X^{NN} = \{X^{NN_1}, \ldots, X^{NN_k}\}$

for all x_i not in X^{NN}, $\text{distance}(x_i, x_q) \geq \max_{X^{NN_j}, j=1..k} \text{distance}(X^{NN_j}, x_q)$

k-NN algorithm

sort first **k documents** by distance to query doc

Initialize **Dist2kNN** = sort($\delta_1,\ldots,\delta_k$) ← list of sorted distances

=  rt(,… )  ← list of sorted docs

For i=k+1,…,N

query doc

Compute: δ = **distance(**  ,  **)**

If δ < **Dist2kNN**[k] ← distance to k^{th} NN (furthest NN in set)

find j such that δ > **Dist2kNN**[j-1] but δ < **Dist2kNN**[j]

remove furthest house and shift queue:

 1:k] =  -1]

inserting new article

Dist2kNN[j+1:k] = **Dist2kNN**[j:k-1]

set **Dist2kNN**[j] = δ and  [j] = 

closest k docs to query doc

Return k most similar articles 

Critical elements of NN search

Item (e.g., doc) representation

$\mathbf{x}_q \leftarrow$ 

Measure of distance between items:

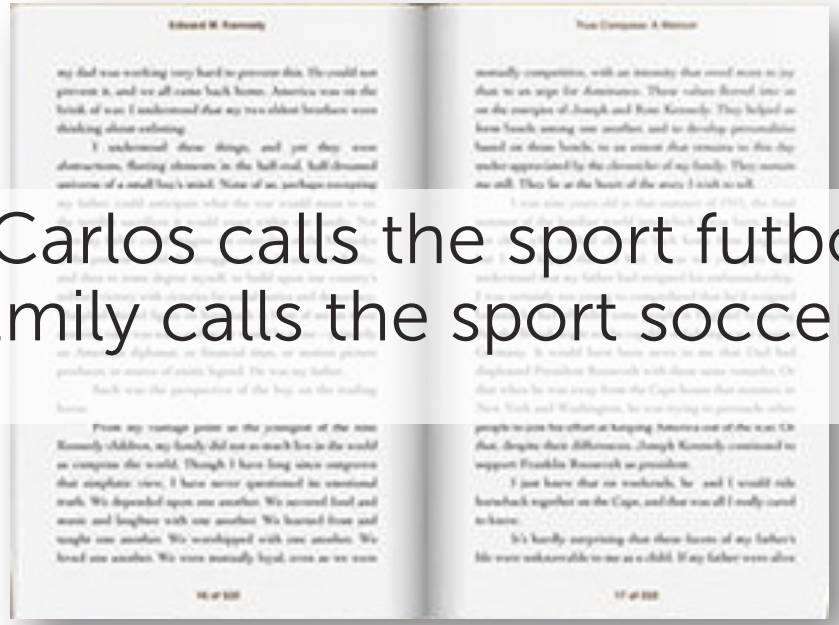
$\delta = \text{distance}(\mathbf{x}_i, \mathbf{x}_q)$

Document representation

Word count document representation

Bag of words model

- Ignore order of words
- Count # of instances of each word in vocabulary



"Carlos calls the sport futbol. Emily calls the sport soccer."

Issues with word counts – Rare words

Common words in doc: "the", "player", "field", "goal"

Dominate rare words like: "futbol", "Messi"

TF-IDF document representation

Emphasizes important words

– Appears frequently in document (common locally)

– Appears rarely in corpus (rare globally)

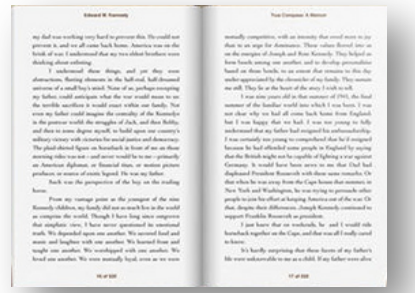
TF-IDF document representation

Emphasizes important words

– Appears frequently in document (common locally)

Term frequency = word counts

– Appears rarely in corpus (rare globally)



TF-IDF document representation

Emphasizes important words

– Appears frequently in document (common locally)

Term frequency = word counts

– Appears rarely in corpus (rare globally)

Inverse doc freq. = $\log \frac{\#\text{ docs}}{1 + \#\text{ docs using word}}$

TF-IDF document representation

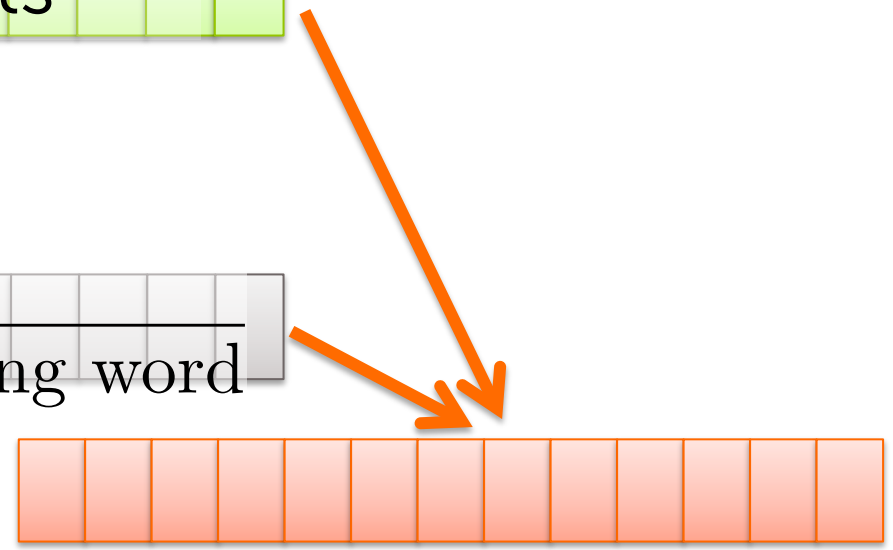
Emphasizes important words

– Appears frequently in document (common locally)

Term frequency = word counts

– Appears rarely in corpus (rare globally)

Inverse doc freq. = $\log \frac{\#\text{ docs}}{1 + \#\text{ docs using word}}$



Trade off: local frequency vs. global rarity

Distance metrics

Distance metrics: Defining notion of "closest"

In 1D, just Euclidean distance:

$$\text{distance}(x_i, x_q) = |x_i - x_q|$$

In multiple dimensions:

- can define many interesting distance functions
- most straightforwardly, might want to weight different dimensions differently

Weighting different features

Reasons:

– Some features are more relevant than others

 # bedrooms
 # bathrooms
sq.ft. living
sq.ft. lot
floors
year built
year renovated
waterfront

Weighting different features

Reasons:

– Some features are more relevant than others

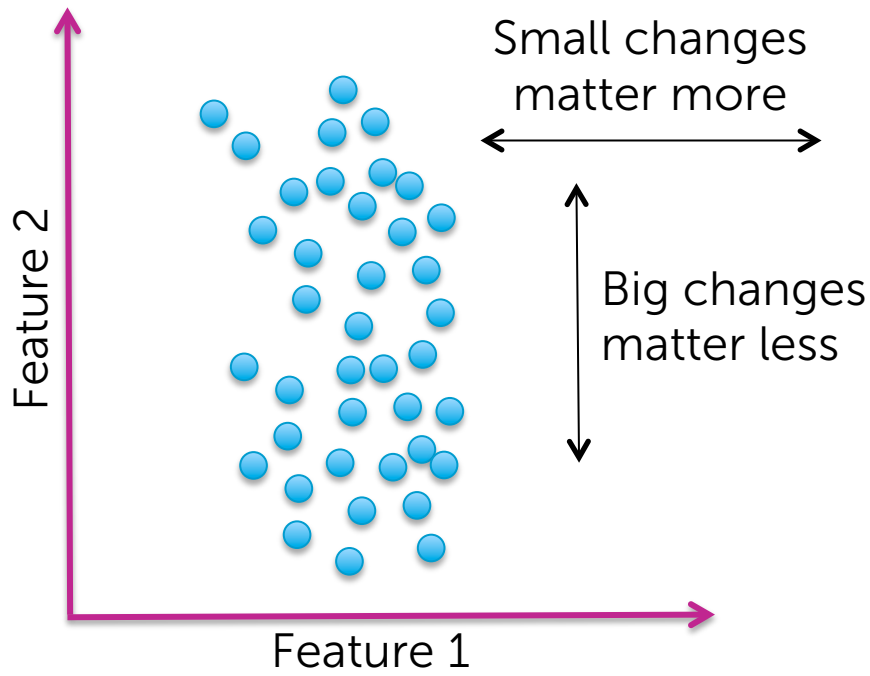
title
abstract
main body
conclusion

Weighting different features

Reasons:

– Some features are more relevant than others

– Some features vary more than others



Specify weights as a function of feature spread

For feature j:

$$\frac{1}{\max_i(\mathbf{x}_i[j]) - \min_i(\mathbf{x}_i[j])}$$

Scaled Euclidean distance

Formally, this is achieved via

$$\text{distance}(\mathbf{x}_i, \mathbf{x}_q) = \sqrt{a_1(\mathbf{x}_i[1]-\mathbf{x}_q[1])^2 + \ldots + a_d(\mathbf{x}_i[d]-\mathbf{x}_q[d])^2}$$

weight on each feature
(defining relative importance)

Effect of binary weights

$$\text{distance}(\mathbf{x}_i, \mathbf{x}_q) = \sqrt{a_1(\mathbf{x}_i[1]-\mathbf{x}_q[1])^2 + \ldots + a_d(\mathbf{x}_i[d]-\mathbf{x}_q[d])^2}$$

Setting weights as 0 or 1 is equivalent to **feature selection**

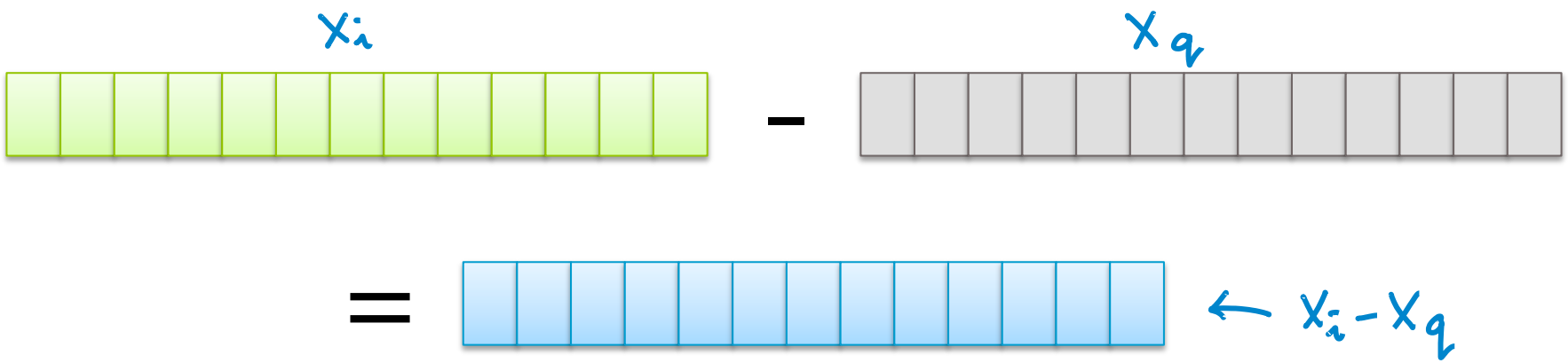
Feature engineering/ selection is **important**, but **hard**

(non-scaled) Euclidean distance

Defined in terms of inner product

$$\text{distance}(\mathbf{x}_i, \mathbf{x}_q) = \sqrt{(\mathbf{x}_i - \mathbf{x}_q)^T(\mathbf{x}_i - \mathbf{x}_q)}$$

$$= \sqrt{(\mathbf{x}_i[1] - \mathbf{x}_q[1])^2 + \ldots + (\mathbf{x}_i[d] - \mathbf{x}_q[d])^2}$$



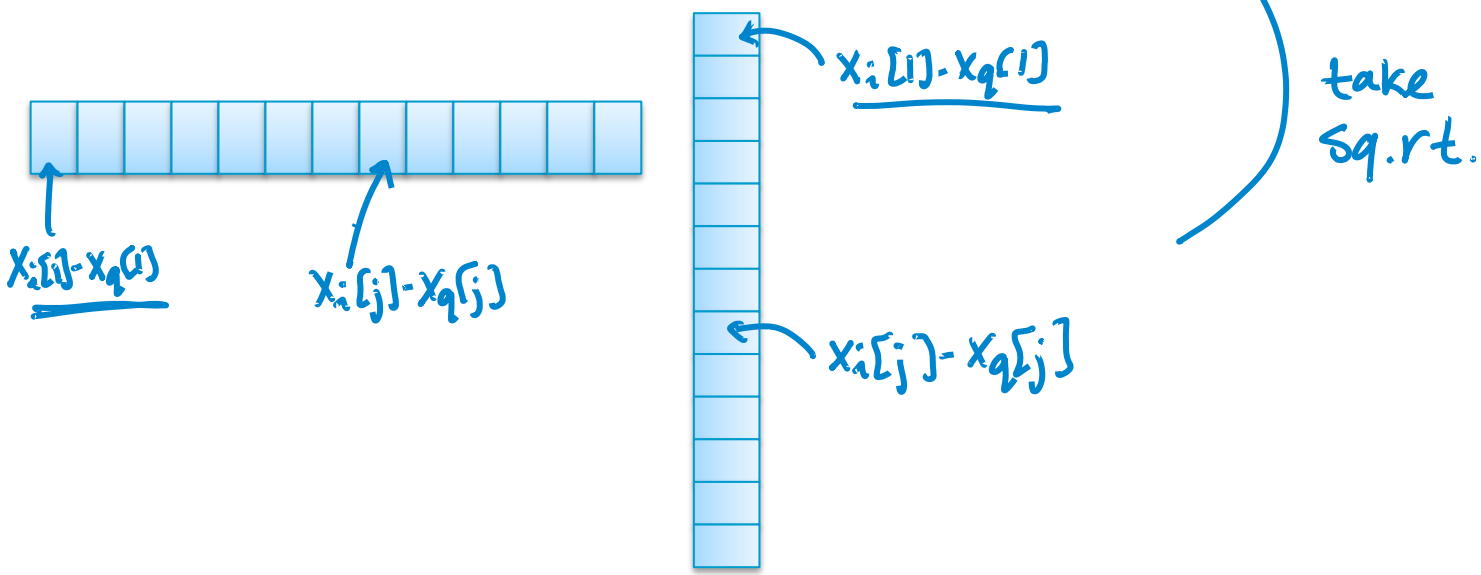
(non-scaled) Euclidean distance

Defined in terms of inner product

$$\text{distance}(\mathbf{x}_i, \mathbf{x}_q) = \sqrt{(\mathbf{x}_i - \mathbf{x}_q)^T(\mathbf{x}_i - \mathbf{x}_q)}$$

$$= \sqrt{(\mathbf{x}_i[1] - \mathbf{x}_q[1])^2 + \ldots + (\mathbf{x}_i[d] - \mathbf{x}_q[d])^2}$$

distance^2 =



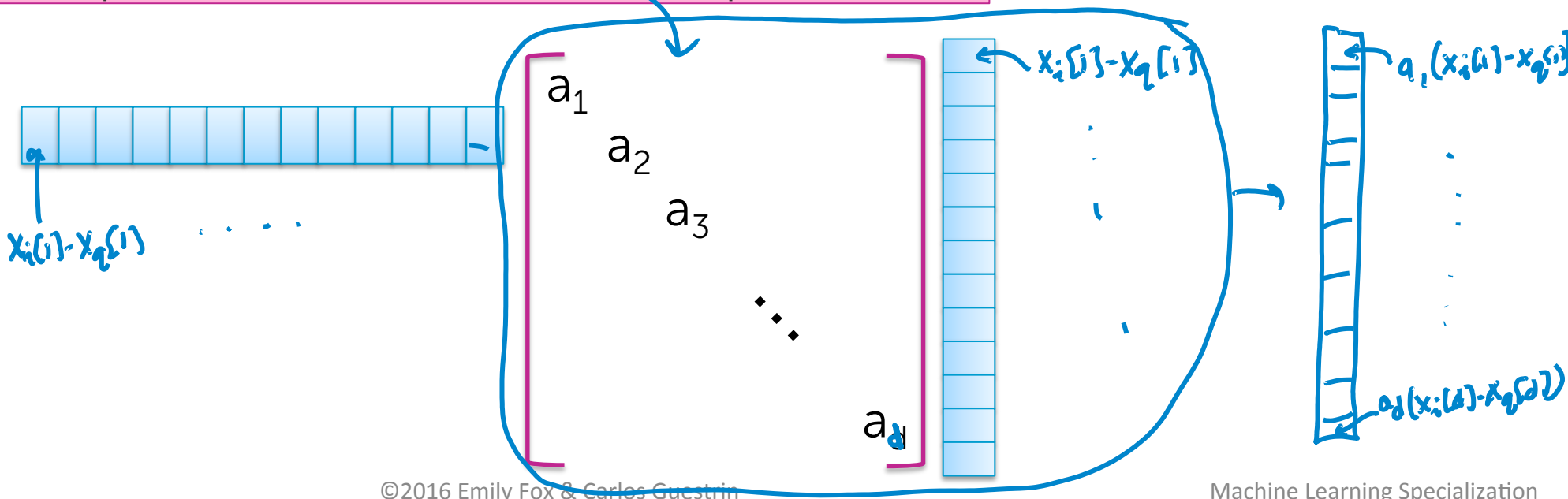
Scaled Euclidean distance

Defined in terms of inner product

$$\text{distance}(\mathbf{x}_i, \mathbf{x}_q) = \sqrt{(\mathbf{x}_i - \mathbf{x}_q)^T \mathbf{A} (\mathbf{x}_i - \mathbf{x}_q)}$$

$$= \sqrt{a_1(\mathbf{x}_i[1] - \mathbf{x}_q[1])^2 + \ldots + a_d(\mathbf{x}_i[d] - \mathbf{x}_q[d])^2}$$

$\text{distance}^2 =$



Another natural inner product measure

x_q

1	0	0	0	5	3	0	0	1	0	0	0	0

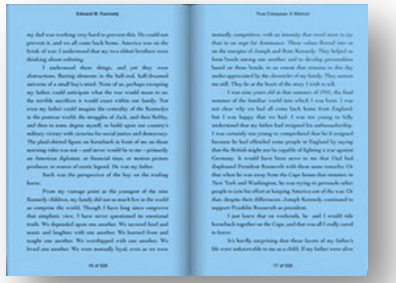
Similarity

$$= \mathbf{x}_i^T \mathbf{x}_q$$

$$= \sum_{j=1}^{d} \mathbf{x}_i[j]\, \mathbf{x}_q[j]$$

3	0	0	0	2	0	0	1	0	1	0	0	0

x_i



= 13

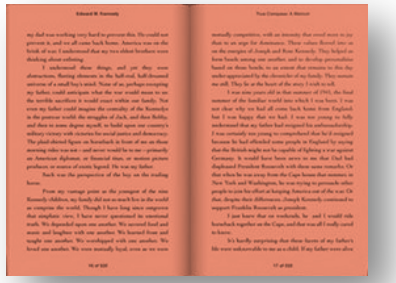
Another natural inner product measure

1	0	0	0	5	3	0	0	1	0	0	0	0

Similarity

= 0

0	0	1	0	0	0	9	0	0	6	0	4	0



Cosine similarity – normalize

Similarity = $\frac{\sum_{j=1}^{d} \mathbf{x}_i[j]\, \mathbf{x}_q[j]}{\sqrt{\sum_{j=1}^{d} (\mathbf{x}_i[j])^2}\sqrt{\sum_{j=1}^{d} (\mathbf{x}_q[j])^2}}$

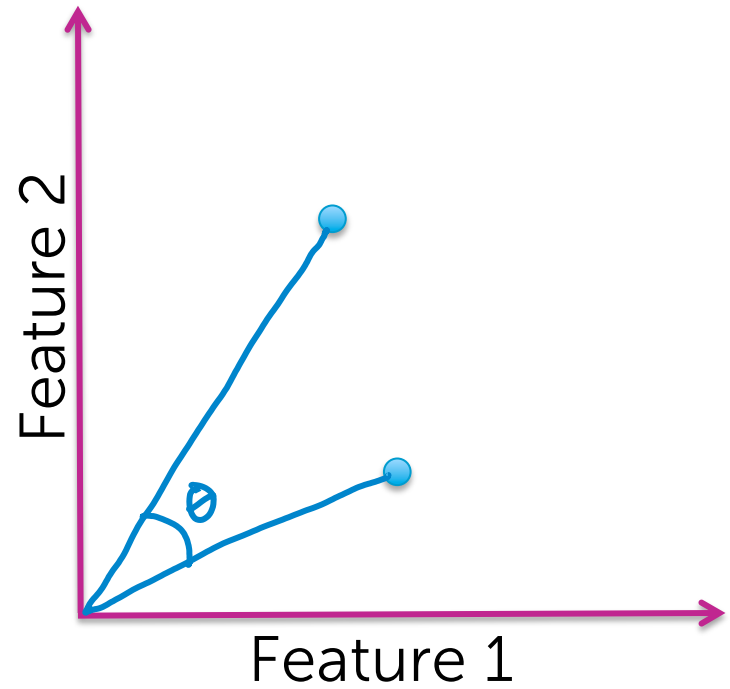
$a^T b = \|a\| \|b\| \cos(\theta)$

$\mathbf{x}_i^T \mathbf{x}_q = \cos(\theta)$

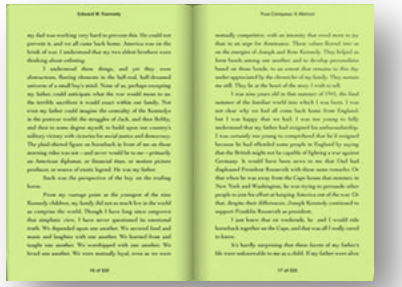
$\|\mathbf{x}_i\| \|\mathbf{x}_q\|$

$= \left(\frac{x_i}{\|x_i\|}\right)^T \left(\frac{x_q}{\|x_q\|}\right)$ first normalize

- Not a proper distance metric
- Efficient to compute for sparse vecs



Normalize



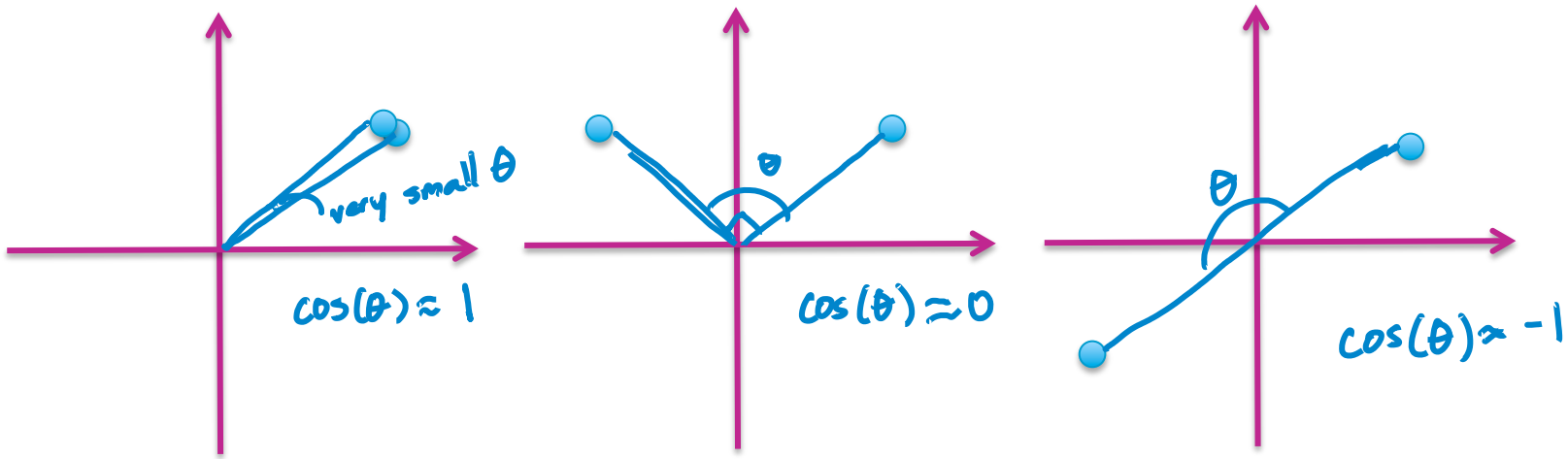
1	0	0	0	5	3	0	0	1	0	0	0	0

$\leftarrow x_i$

$$\sqrt{(1^2 + 5^2 + 3^2 + 1^2)} \leftarrow \|x_i\| = \sum_{j=1}^{d} x_i[j]^2$$

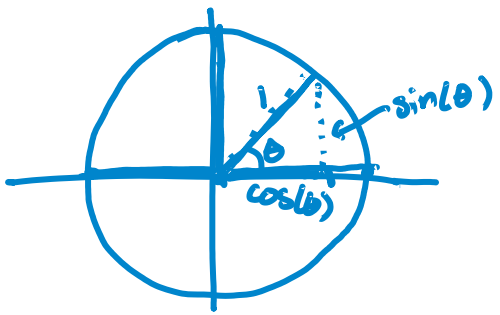
1/6	0	0	0	5/6	3/6	0	0	1/6	0	0	0	0

Cosine similarity



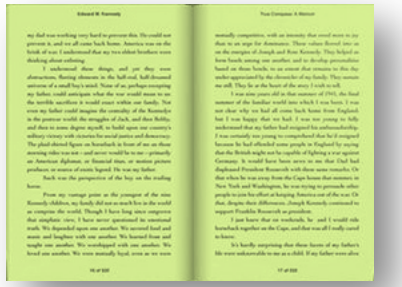
In general, $-1 <$ similarity < 1

For positive features (like tf-idf) $0 <$ similarity < 1 } our focus



Define **distance = 1-similarity**

To normalize or not?



1	0	0	0	5	3	0	0	1	0	0	0	0

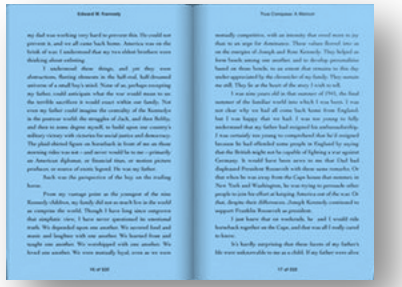
2	0	0	0	10	6	0	0	2	0	0	0	0

3	1	0	0	2	0	0	1	0	1	0	0	0

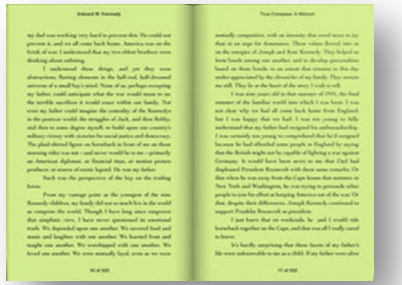
Similarity = 13

6	2	0	0	4	0	0	2	0	2	0	0	0

Similarity = 52



Normalize

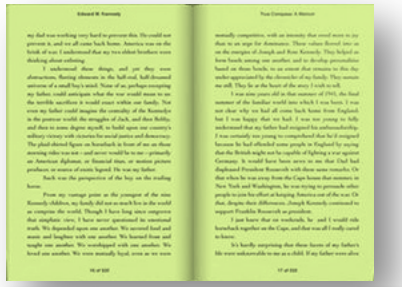


1	0	0	0	5	3	0	0	1	0	0	0	0

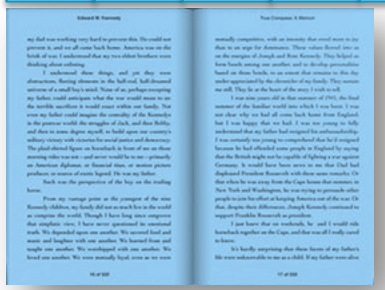
$$\sqrt{(1^2 + 5^2 + 3^2 + 1^2)}$$

1/6	0	0	0	5/6	3/6	0	0	1/6	0	0	0	0

In the normalized case

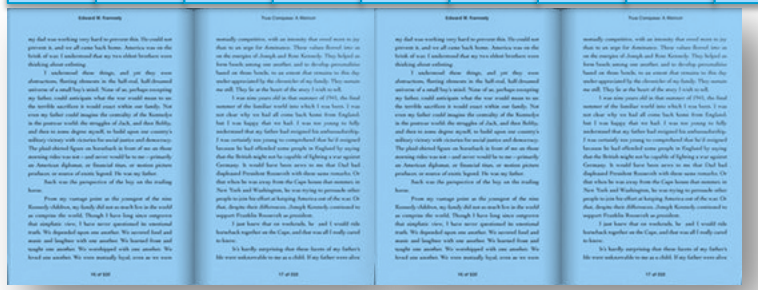


1/6	0	0	0	5/6	3/6	0	0	1/6	0	0	0	0
3/4	1/4	0	0	1/2	0	0	1/4	0	1/4	0	0	0



Similarity = 13/24

1/6	0	0	0	5/6	3/6	0	0	1/6	0	0	0	0
3/4	1/4	0	0	1/2	0	0	1/4	0	1/4	0	0	0



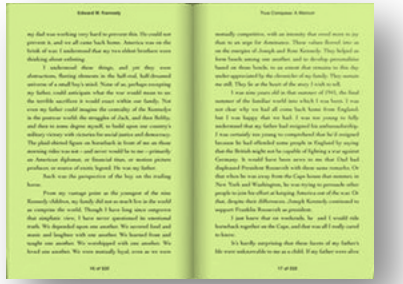
Similarity = 13/24

But not always desired…

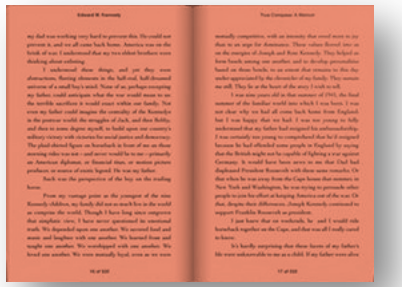
long document

short tweet

Normalizing can make dissimilar objects appear more similar



long document



long document

Common compromise:
Just cap maximum word counts

Other distance metrics

- Mahalanobis
- rank-based
- correlation-based
- Manhattan
- Jaccard
- Hamming
- …

Combining distance metrics

Example of document features:

1. Text of document
   - Distance metric: Cosine similarity
2. # of reads of doc
   - Distance metric: Euclidean distance

Add together with user-specified weights

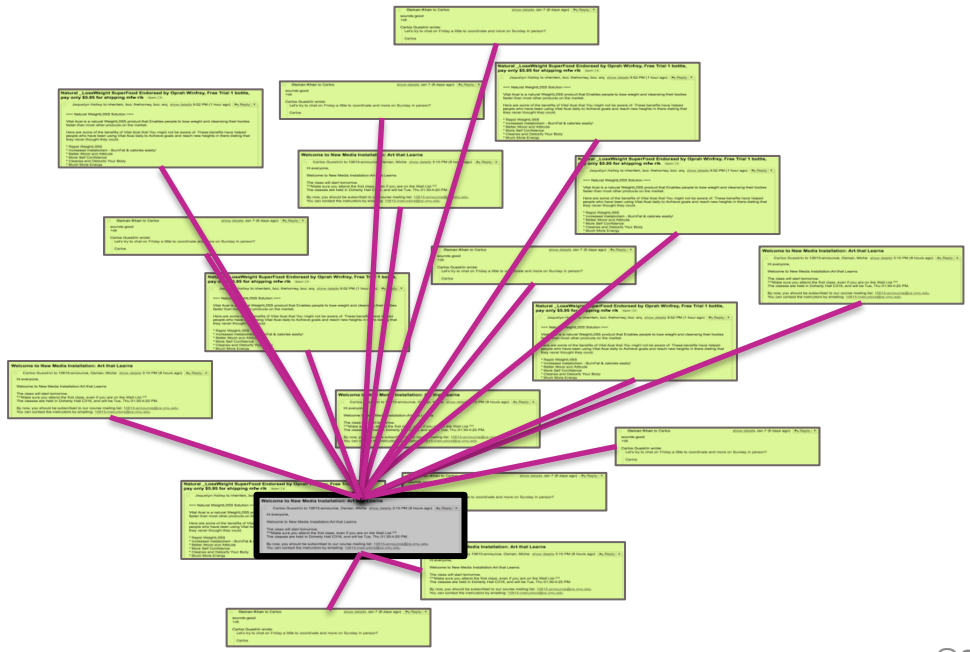
Scaling up k-NN search by storing data in a KD-tree

Complexity of search

Complexity of brute-force search

Given a query point, scan through each point

- O(*N*) distance computations per 1-NN query!
- O(*N*log*k*) per *k*-NN query!



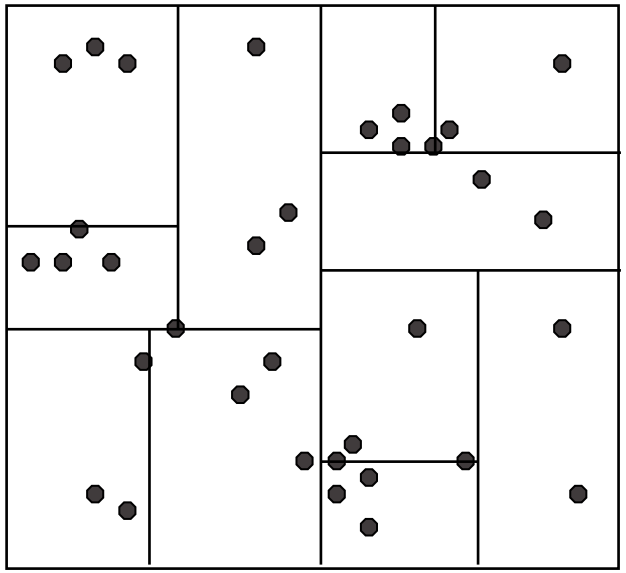
What if *N* is huge???
(and many queries)

KD-tree representation

KD-trees

Structured organization of documents

- Recursively partitions points into axis aligned boxes.

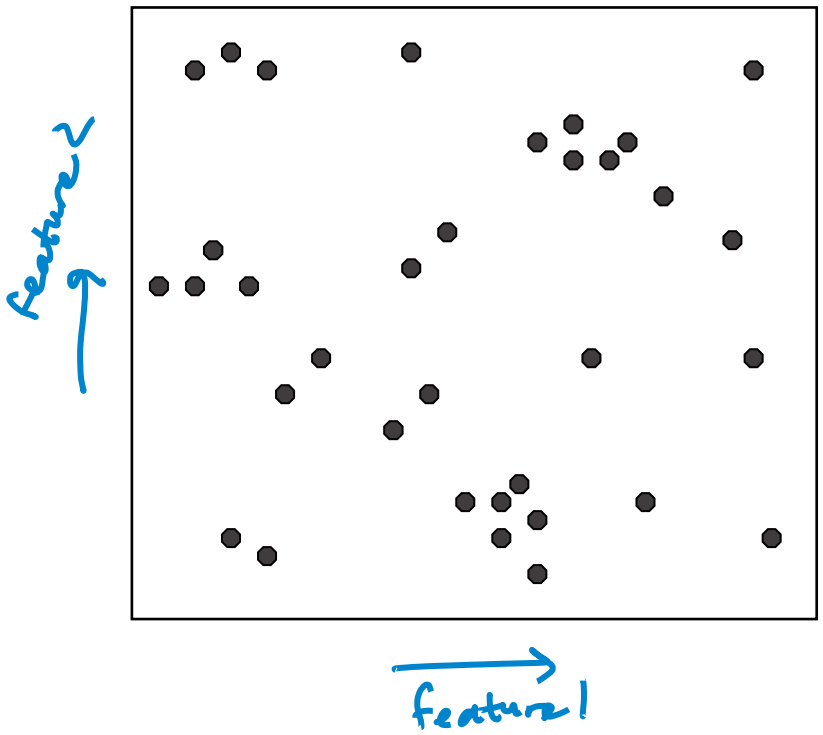


Enables more efficient pruning of search space

Works "well" in "low-medium" dimensions

- We'll get back to this…

KD-tree construction



Start with a list of d-dimensional points.

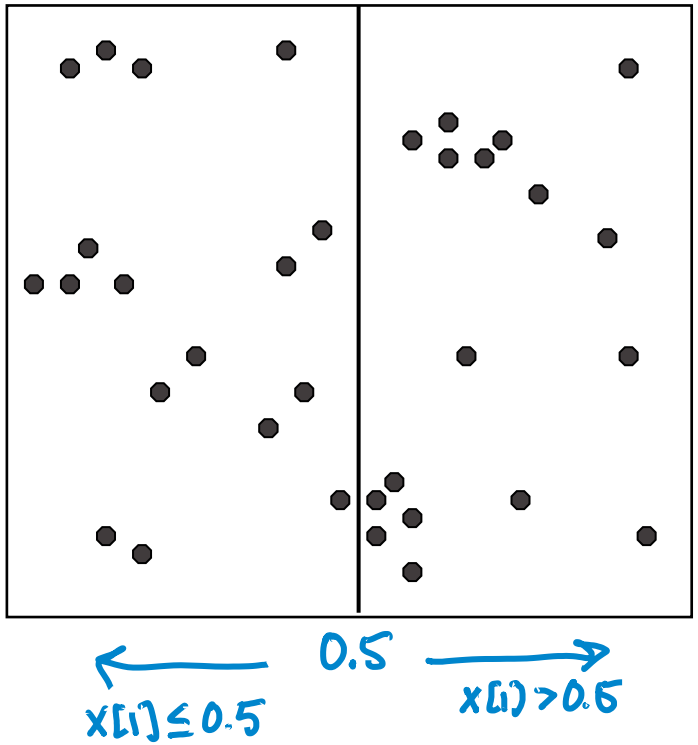
Pt	**x**[1]	**x**[2]
1	0.00	0.00
2	1.00	4.31
3	0.13	2.85
…	…	…

obs. indices

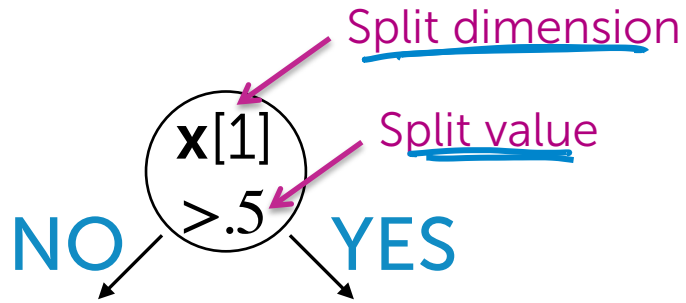
Feat. 1 (word 1)

Feat. 2 (word 2)

KD-tree construction



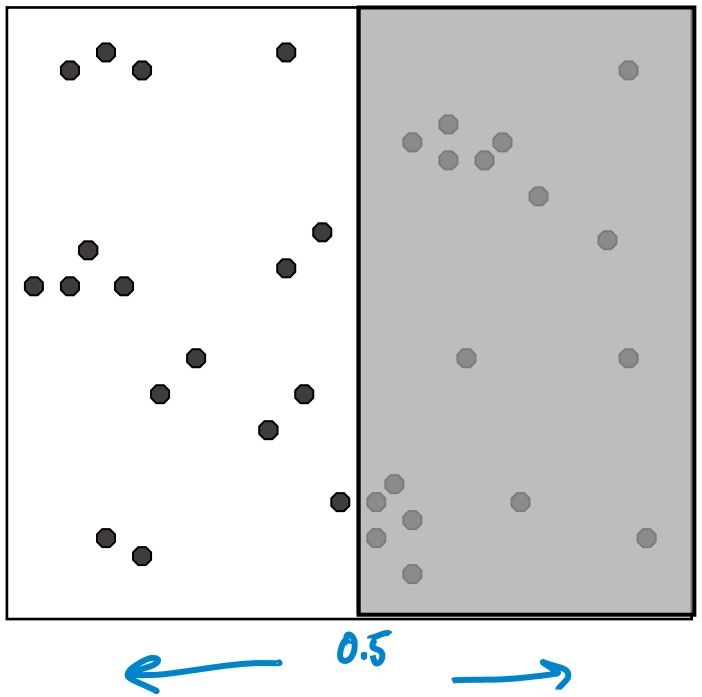
Split points into 2 groups



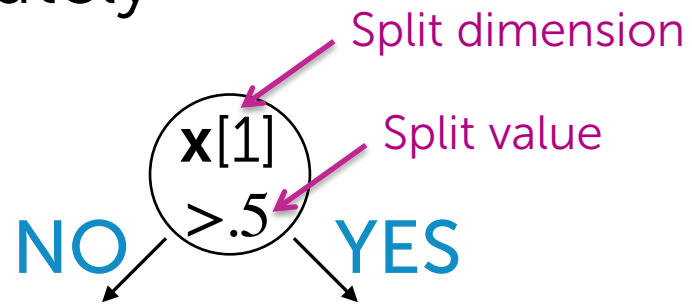
Pt	x[1]	x[2]
1	0.00	0.00
3	0.13	2.85
…	…	…

Pt	x[1]	x[2]
2	1.00	4.31
…	…	…

KD-tree construction



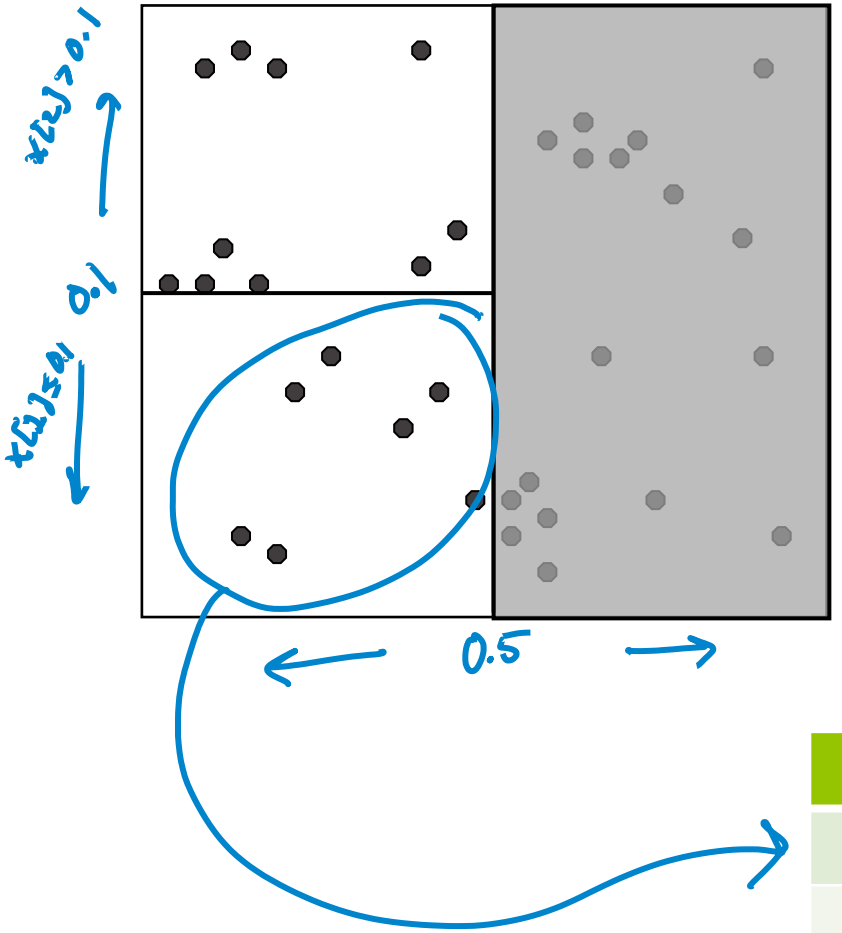
Recurse on each group separately



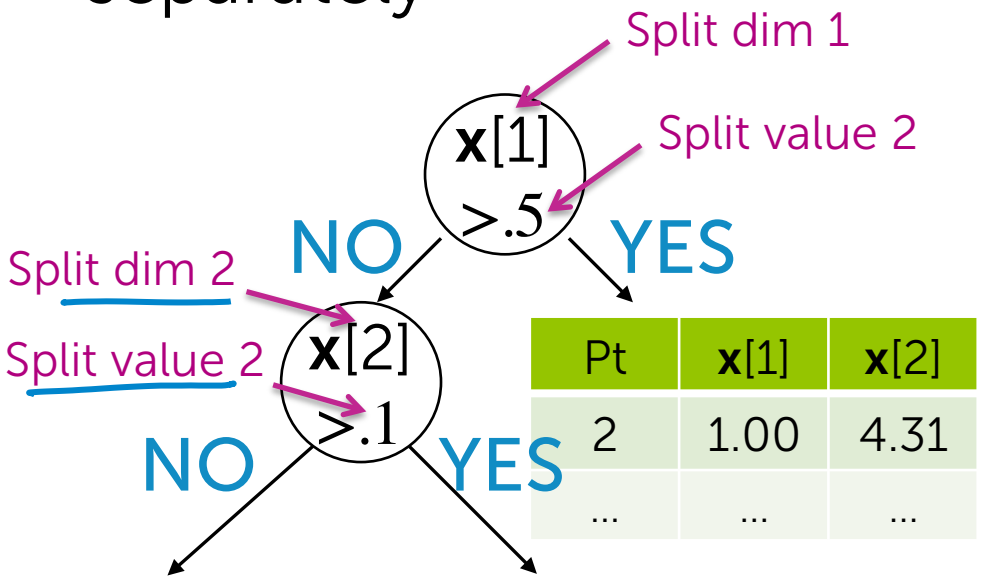
Pt	x[1]	x[2]
1	0.00	0.00
3	0.13	2.85
...	...	...

Pt	x[1]	x[2]
2	1.00	4.31
...	...	...

KD-tree construction



Recurse on each group separately

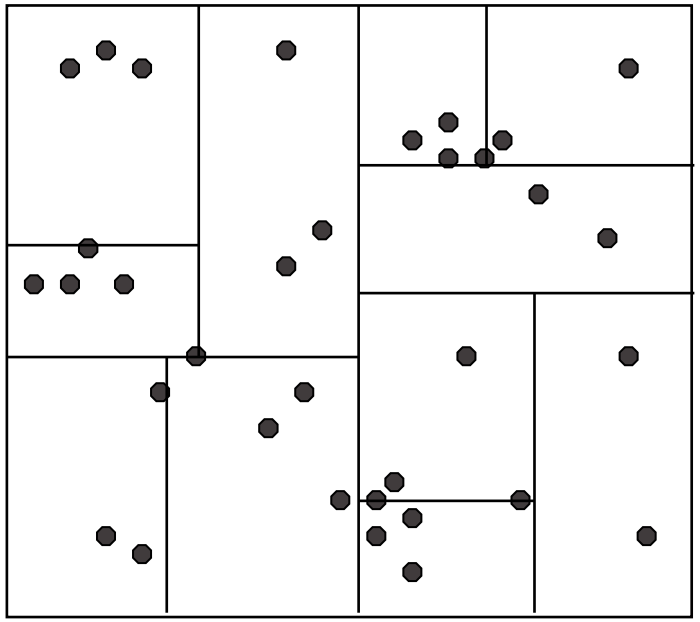


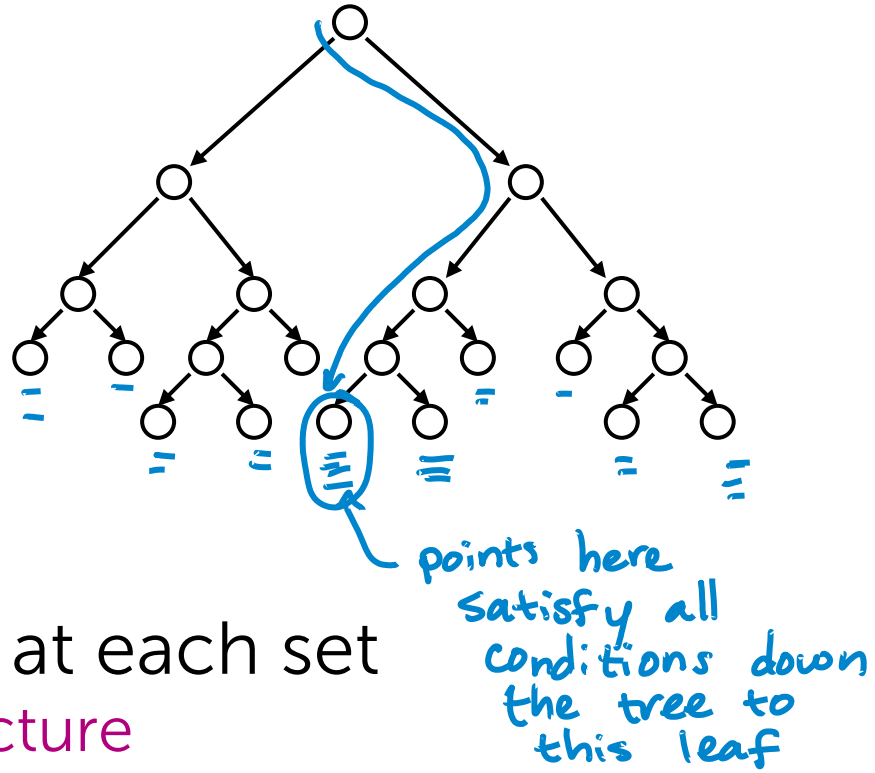
Pt	x[1]	x[2]
2	1.00	4.31
...	...	...

Pt	x[1]	x[2]
3	0.13	2.85
...	...	...

Pt	x[1]	x[2]
1	0.00	0.00
...	...	...

KD-tree construction



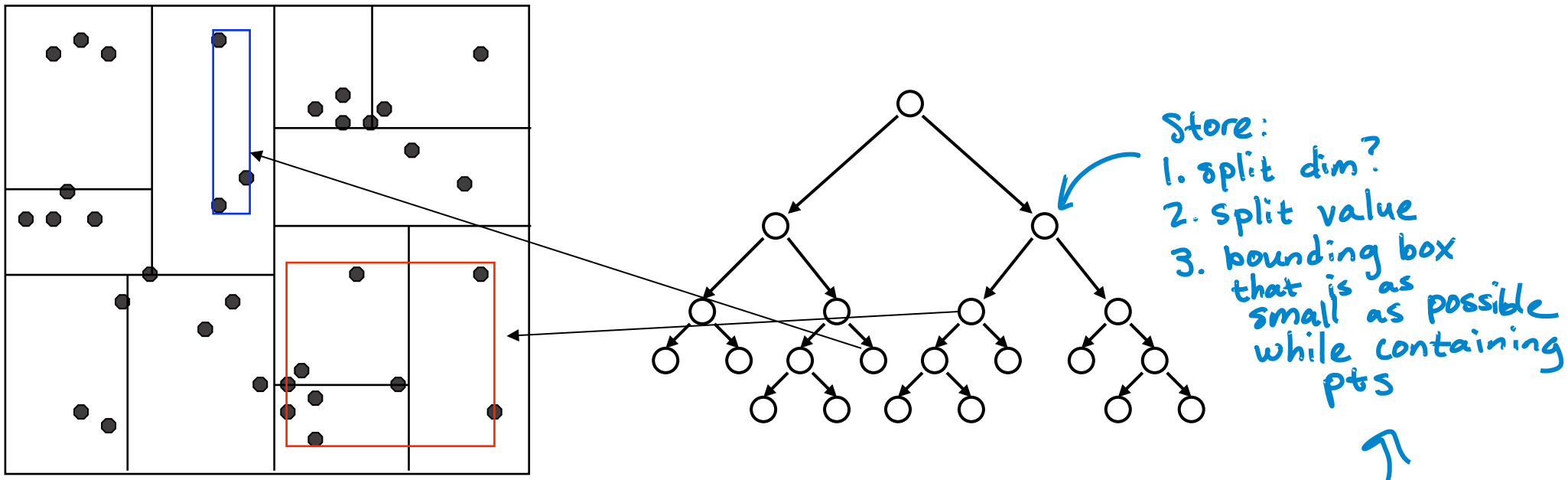


Continue splitting points at each set

- Creates a binary tree structure

Each leaf node contains a list of points

KD-tree construction



Keep one additional piece of info at each node:

#3– The (tight) bounds of points at or below node

KD-tree construction choices

Use heuristics to make splitting decisions:

- Which dimension do we split along?

  widest (or alternate)

- Which value do we split at?

  median (or center point of box, ignoring data in box)

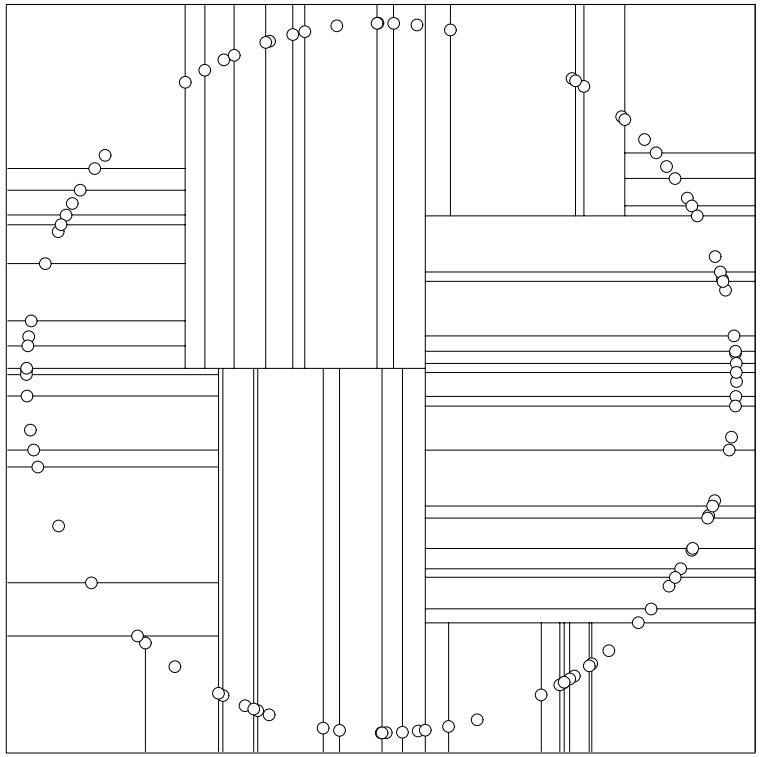
- When do we stop?

  fewer than m pts left

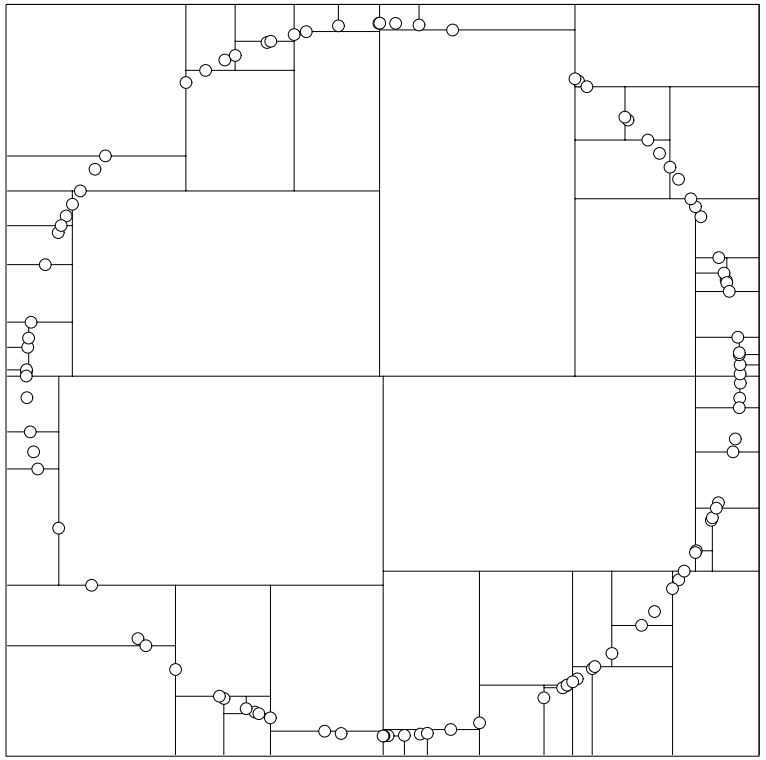
  or

  box hits minimum width

Many heuristics…



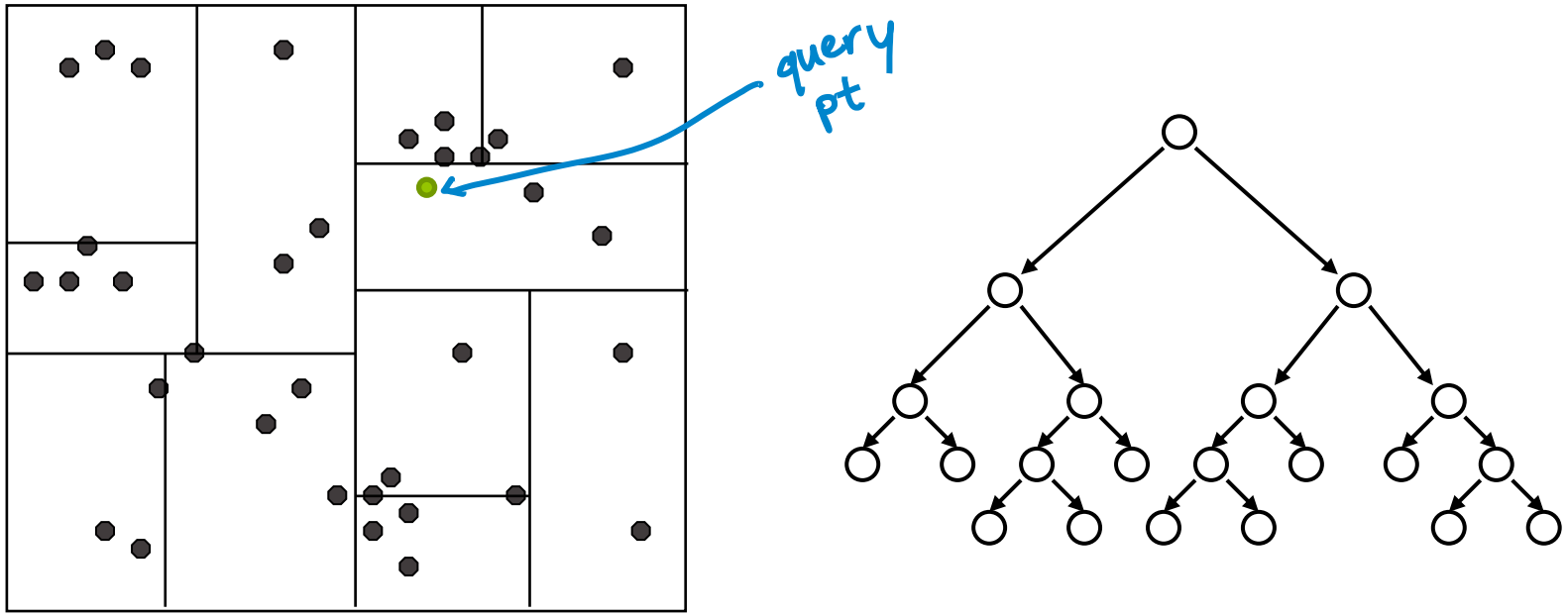
median heuristic



center-of-range heuristic

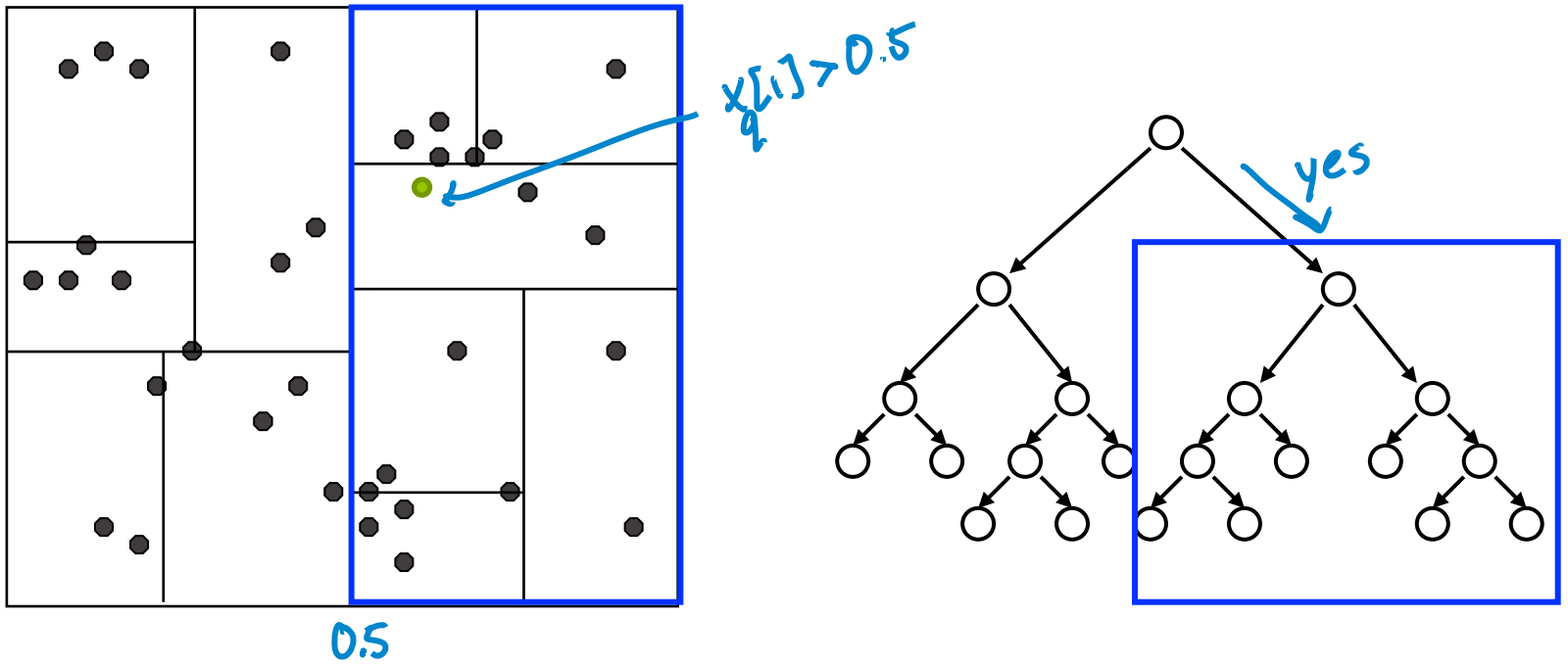
NN search with KD-trees

Nearest neighbor with KD-trees



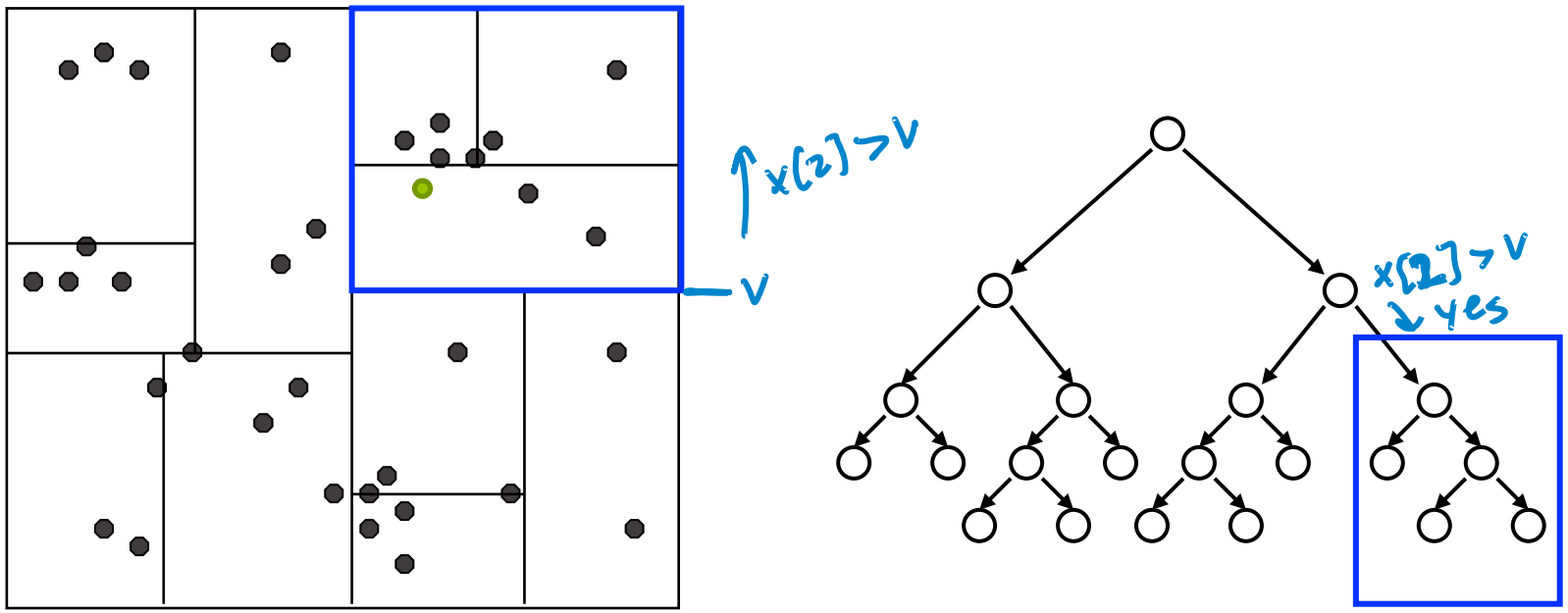
Traverse tree looking for nearest neighbor to query point

Nearest neighbor with KD-trees



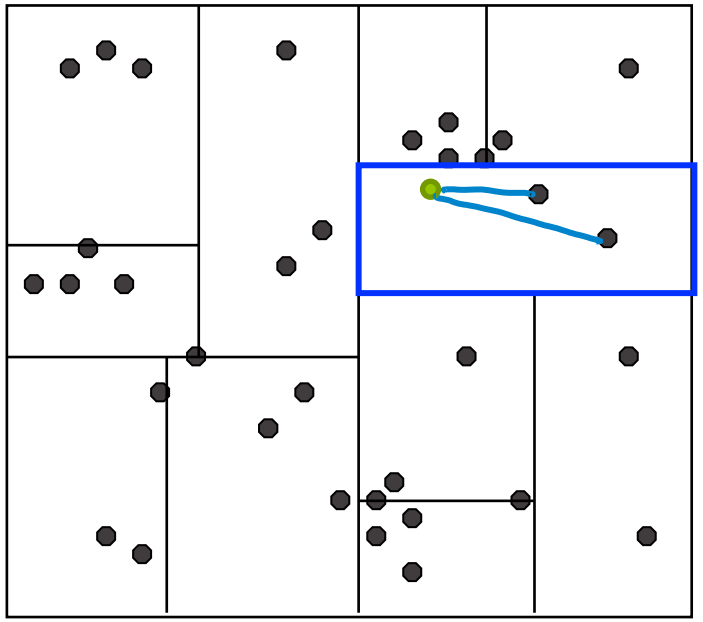
1. Start by exploring leaf node containing query point

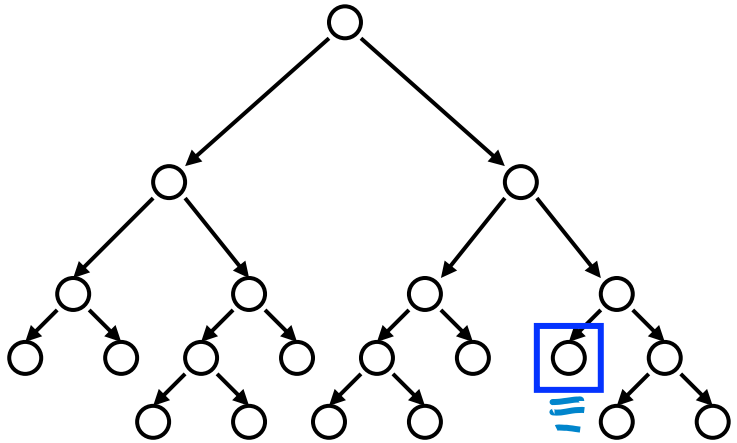
Nearest neighbor with KD-trees



1. Start by exploring leaf node containing query point

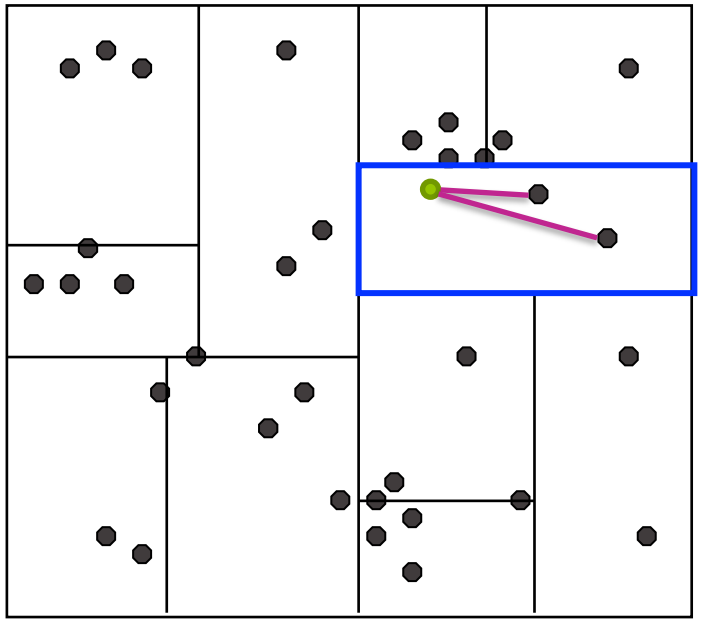
Nearest neighbor with KD-trees

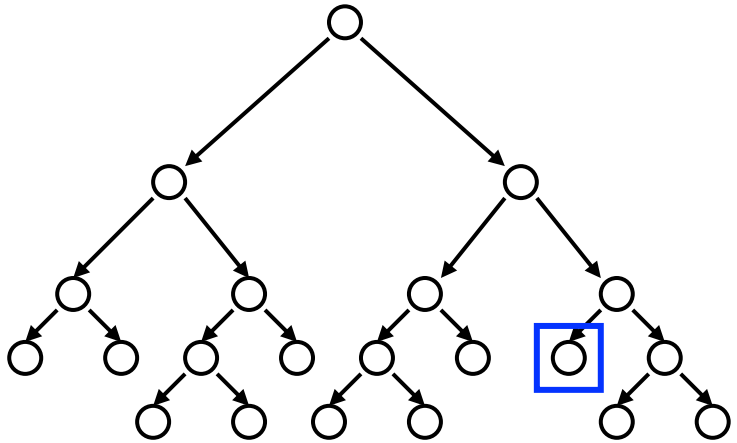




1. Start by exploring leaf node containing query point

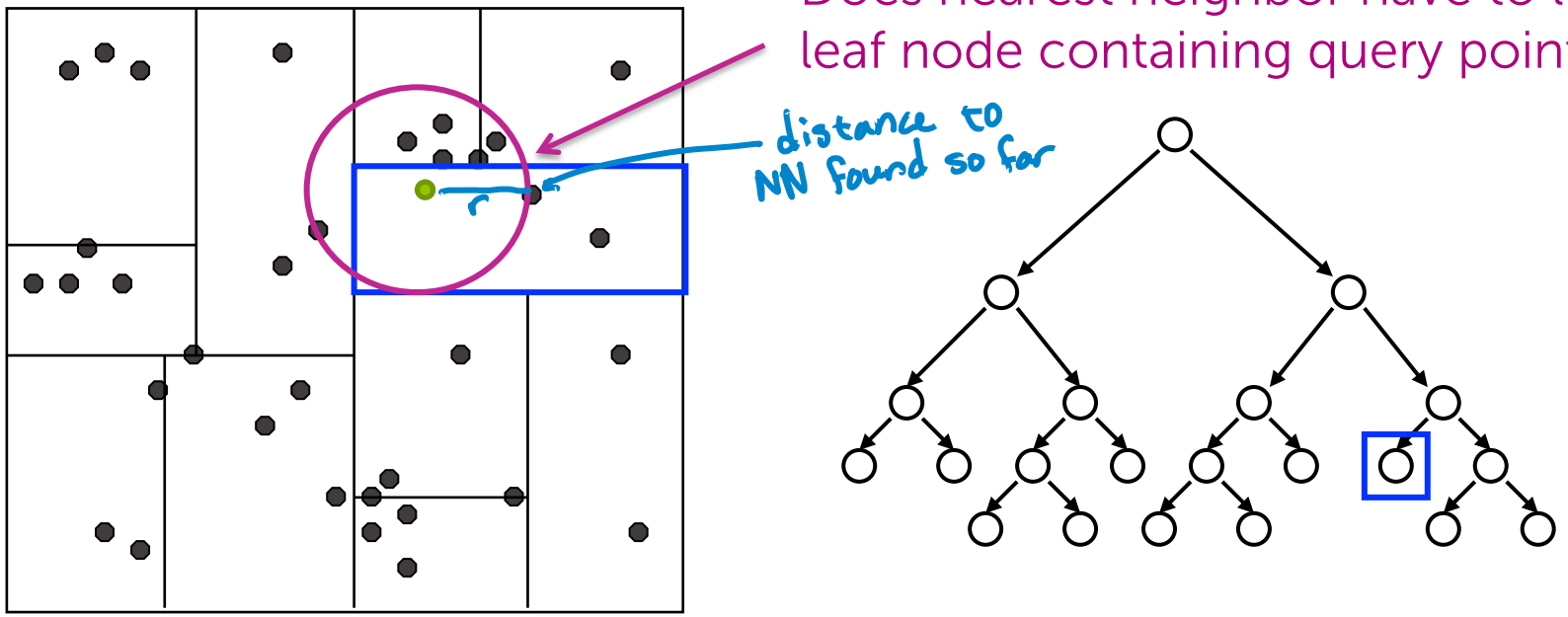
Nearest neighbor with KD-trees





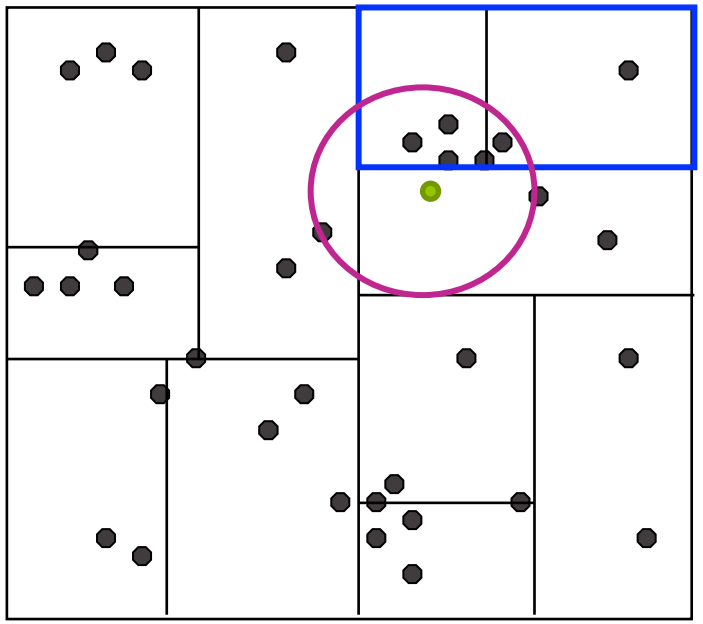
1. Start by exploring leaf node containing query point
2. Compute distance to each other point at leaf node

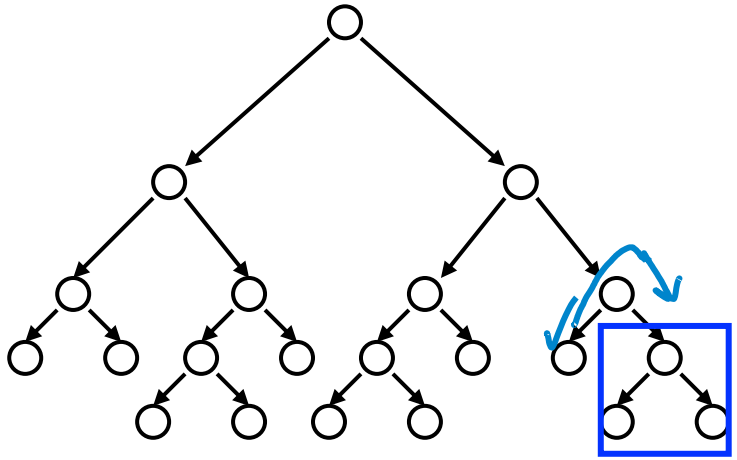
Nearest neighbor with KD-trees



1. Start by exploring leaf node containing query point
2. Compute distance to each other point at leaf node

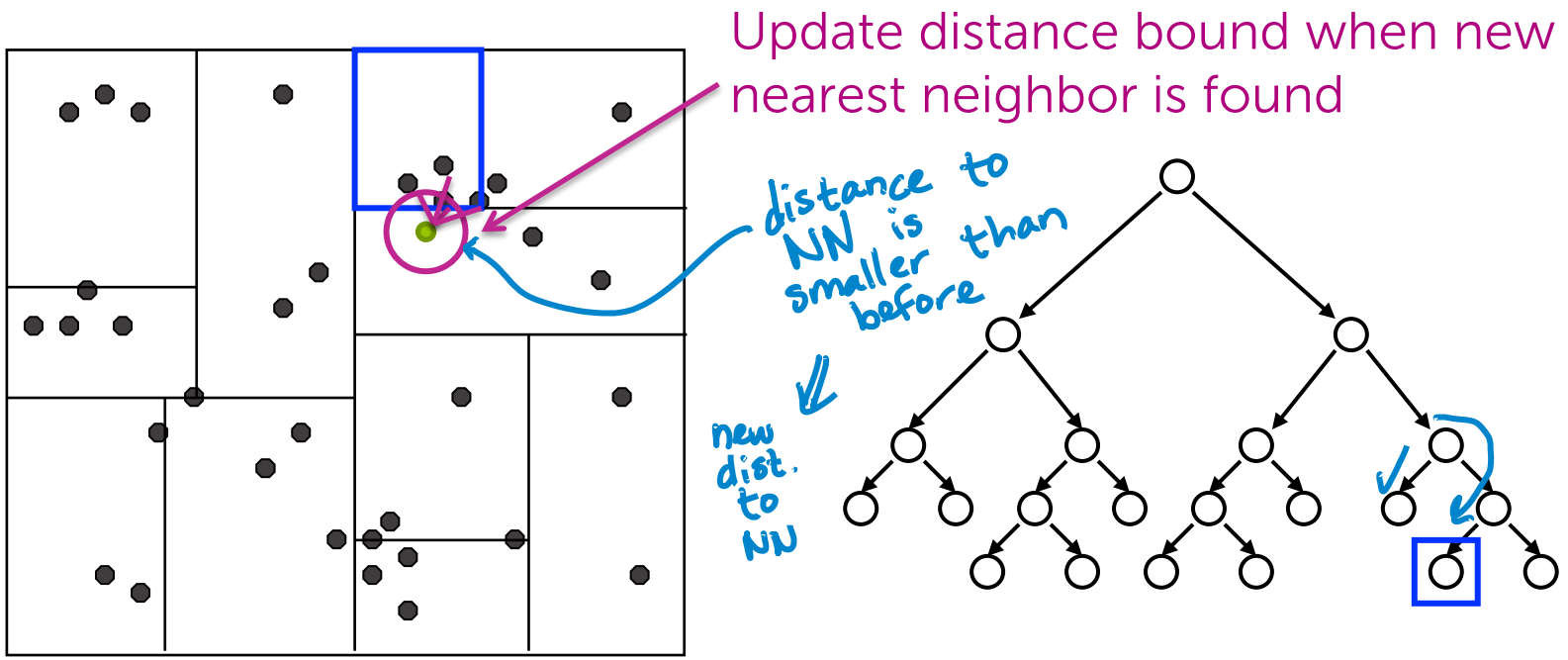
Nearest neighbor with KD-trees





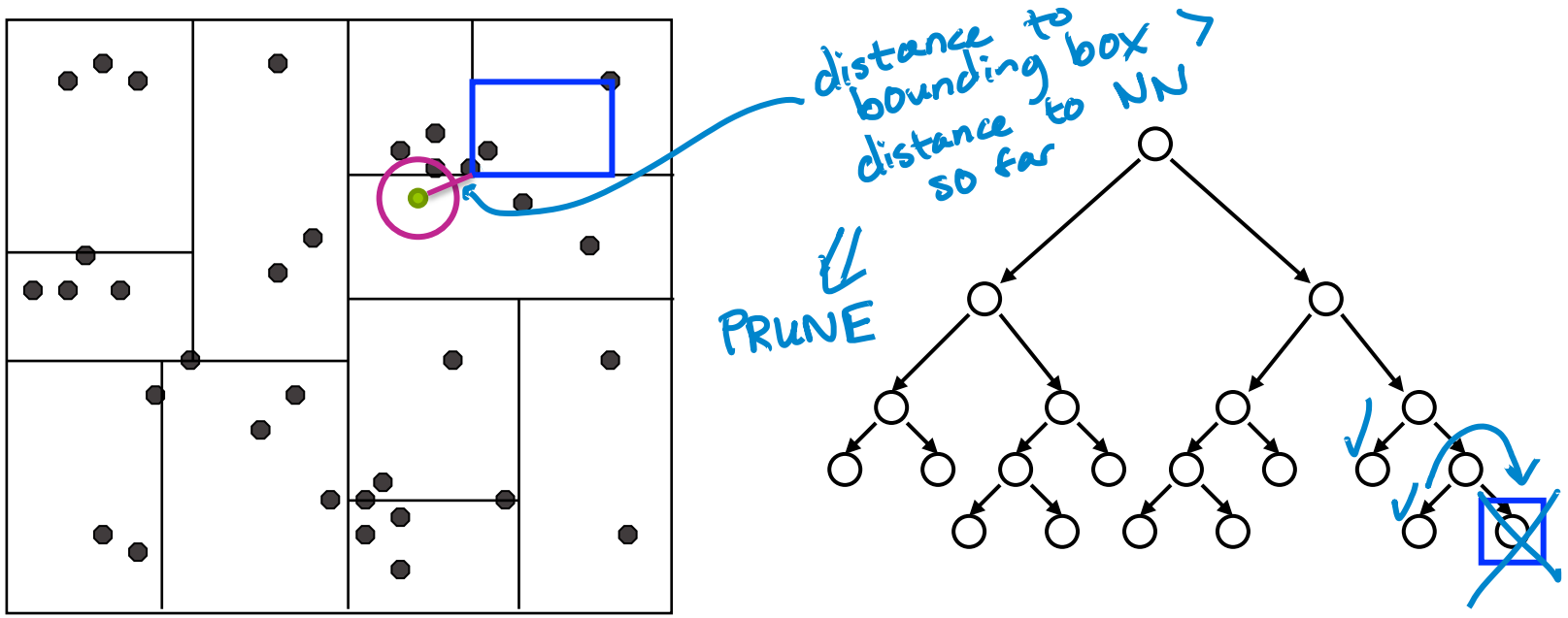
1. Start by exploring leaf node containing query point
2. Compute distance to each other point at leaf node
3. Backtrack and try other branch at each node visited

Nearest neighbor with KD-trees



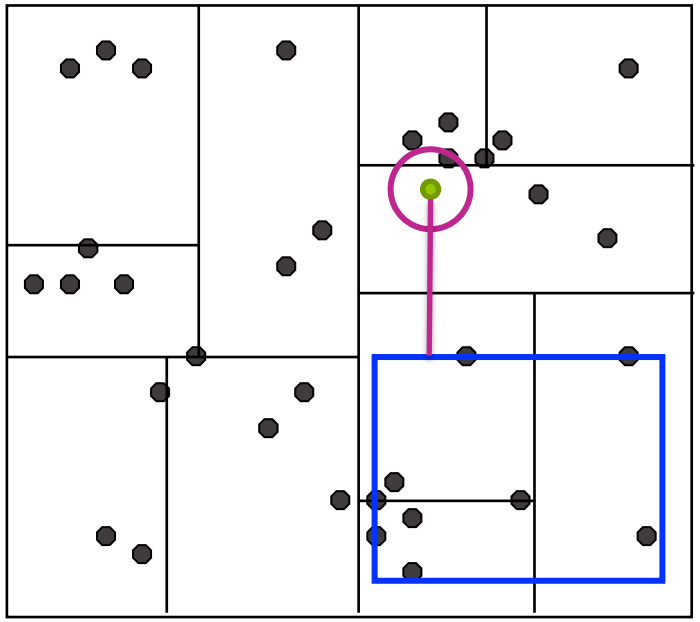
1. Start by exploring leaf node containing query point
2. Compute distance to each other point at leaf node
3. Backtrack and try other branch at each node visited

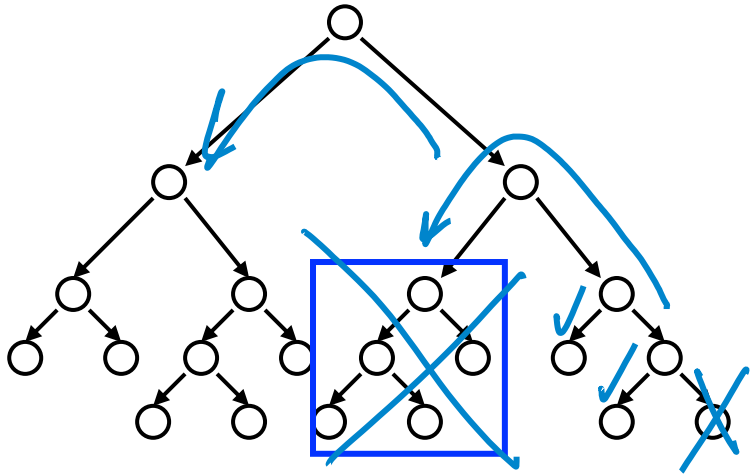
Nearest neighbor with KD-trees



Use distance bound and bounding box of each node to prune parts of tree that cannot include nearest neighbor

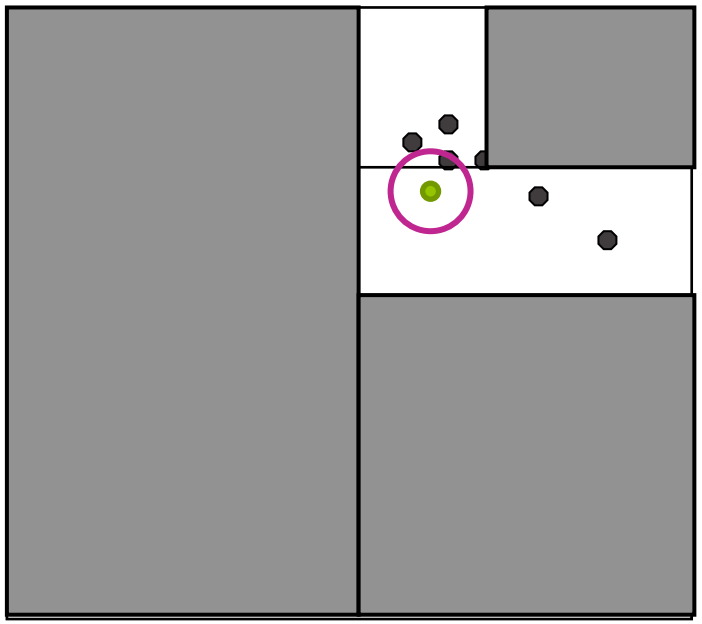
Nearest neighbor with KD-trees

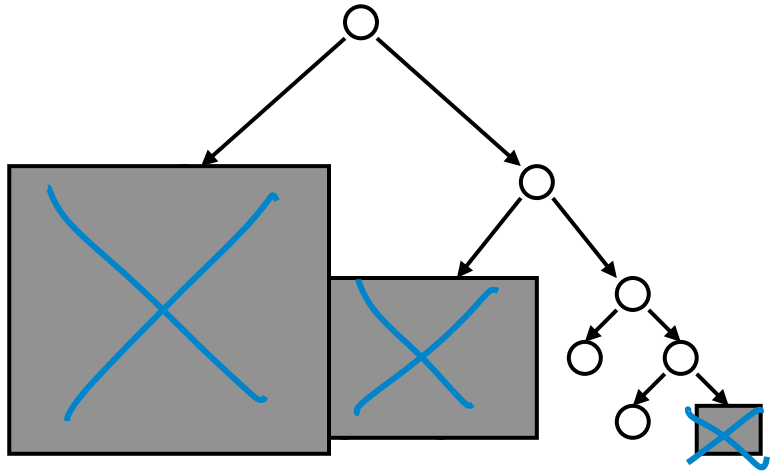




Use distance bound and bounding box of each node to prune parts of tree that cannot include nearest neighbor

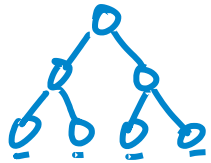
Nearest neighbor with KD-trees





Use distance bound and bounding box of each node to prune parts of tree that cannot include nearest neighbor

Complexity

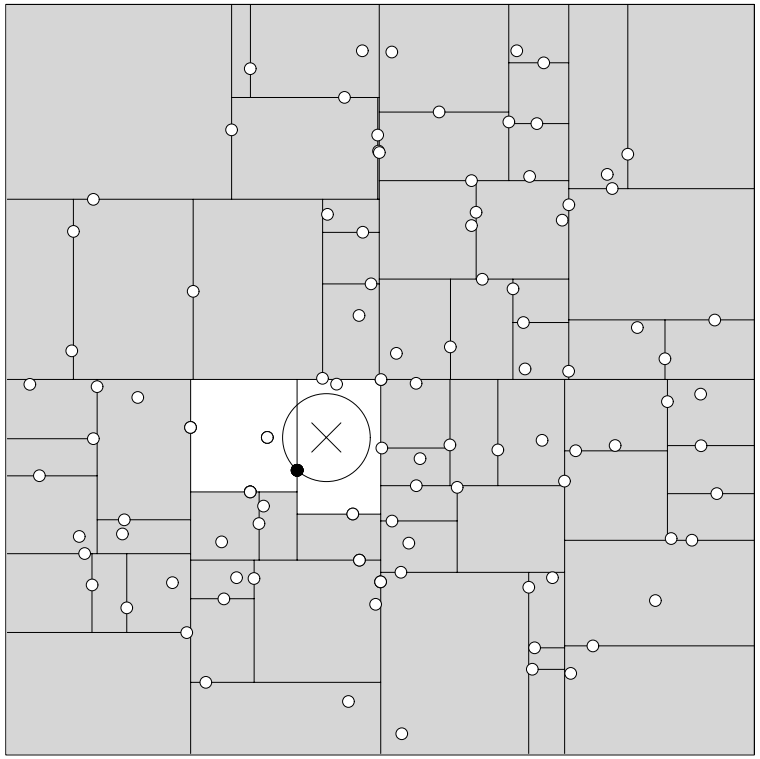


For (nearly) balanced, binary trees…

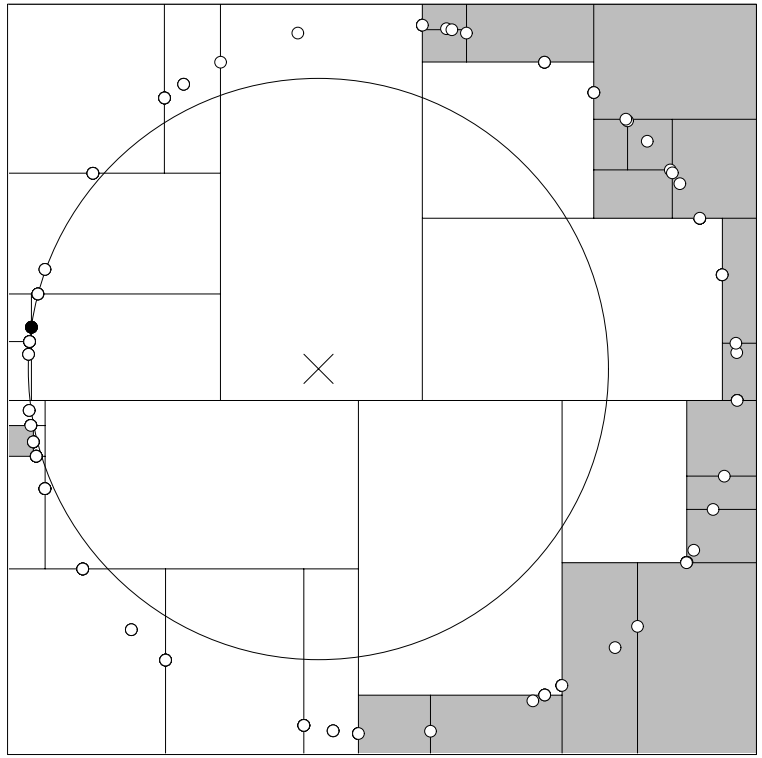
- Construction
  - Size: 2N-1 nodes if 1 datapt at each leaf → O(N)
  - Depth: O(log N)
  - Median + send points left right: O(N) at every level of the tree
  - Construction time: O(N log N)
- 1-NN query
  - Traverse down tree to starting point: O(log N)
  - Maximum backtrack and traverse: O(N) in worst case
  - Complexity range: O(log N) → O(N)

Under some assumptions on distribution of points, we get O(logN) but exponential in d

Complexity



pruned many
(closer to O(log N))



pruned few
(closer to O(N))

Complexity for N queries

- Ask for nearest neighbor to each doc

N queries

- Brute force 1-NN:

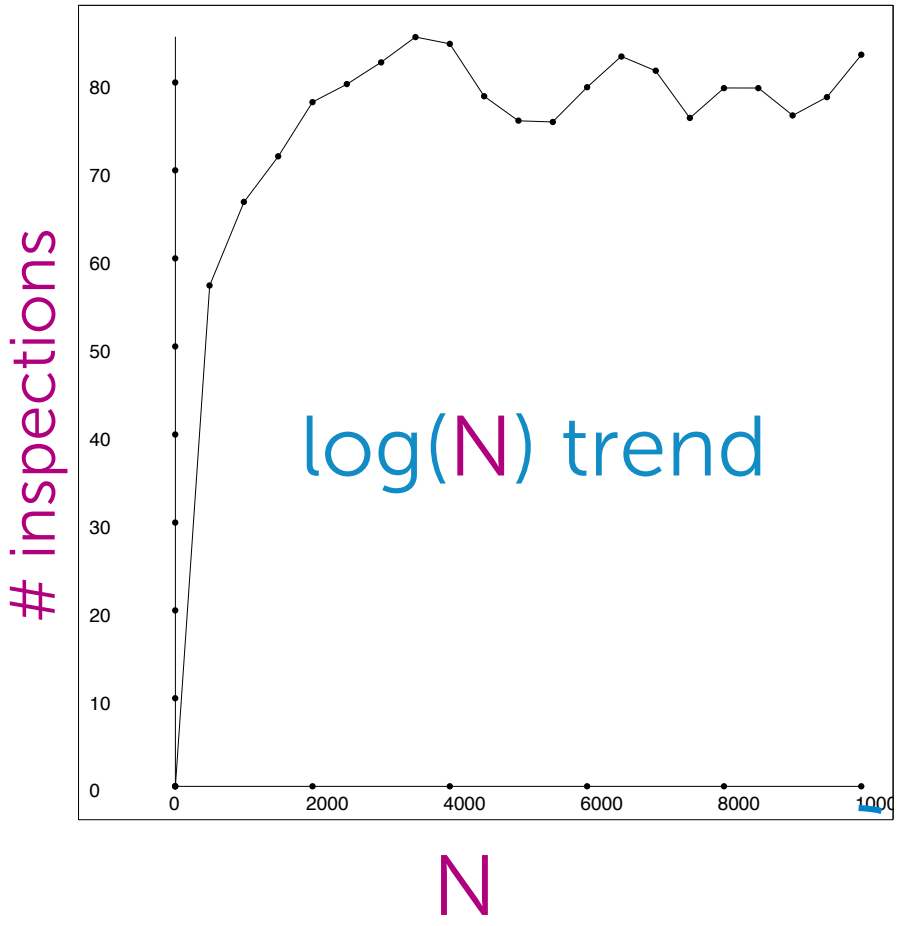
$O(N^2)$

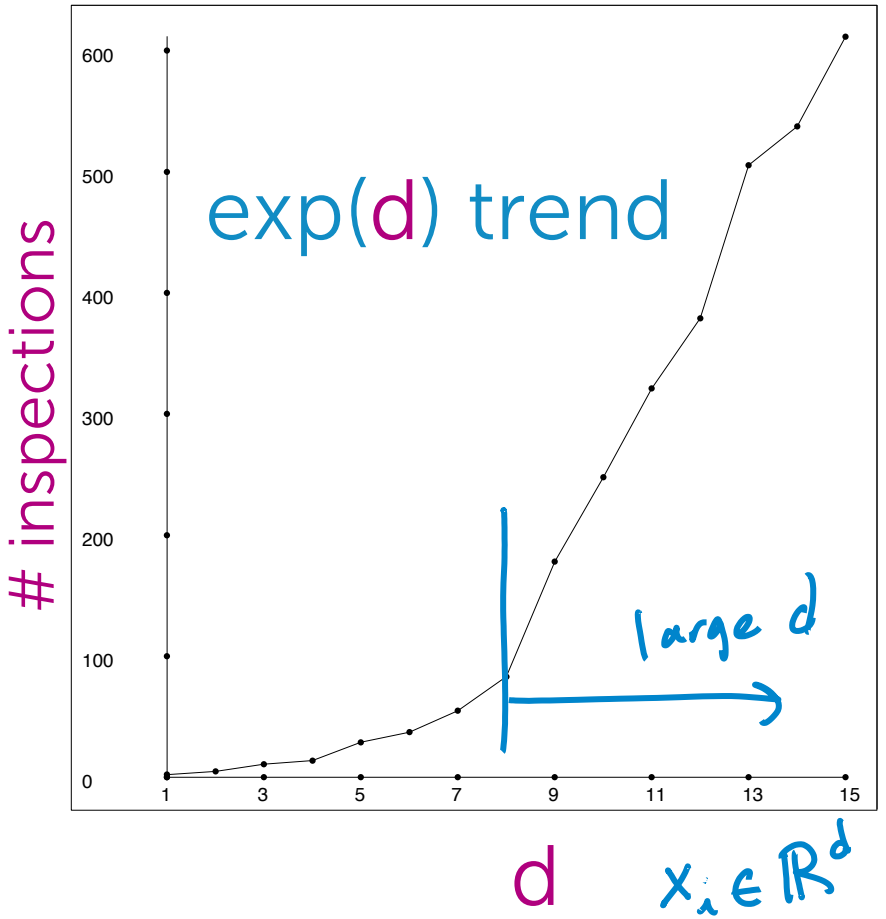
- kd-trees:

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

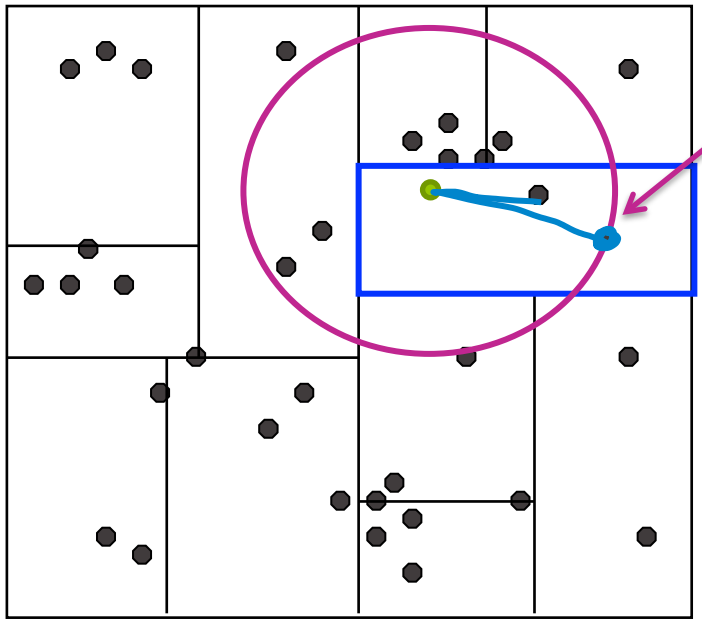
potentially very large savings for large N!

Inspections vs. N and d

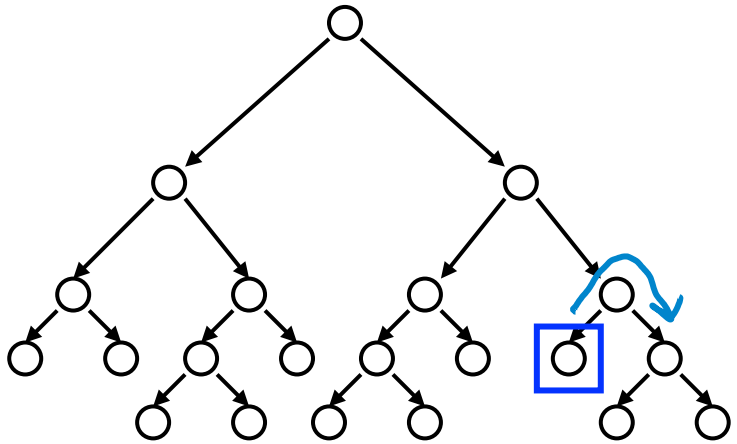




k-NN with KD-trees



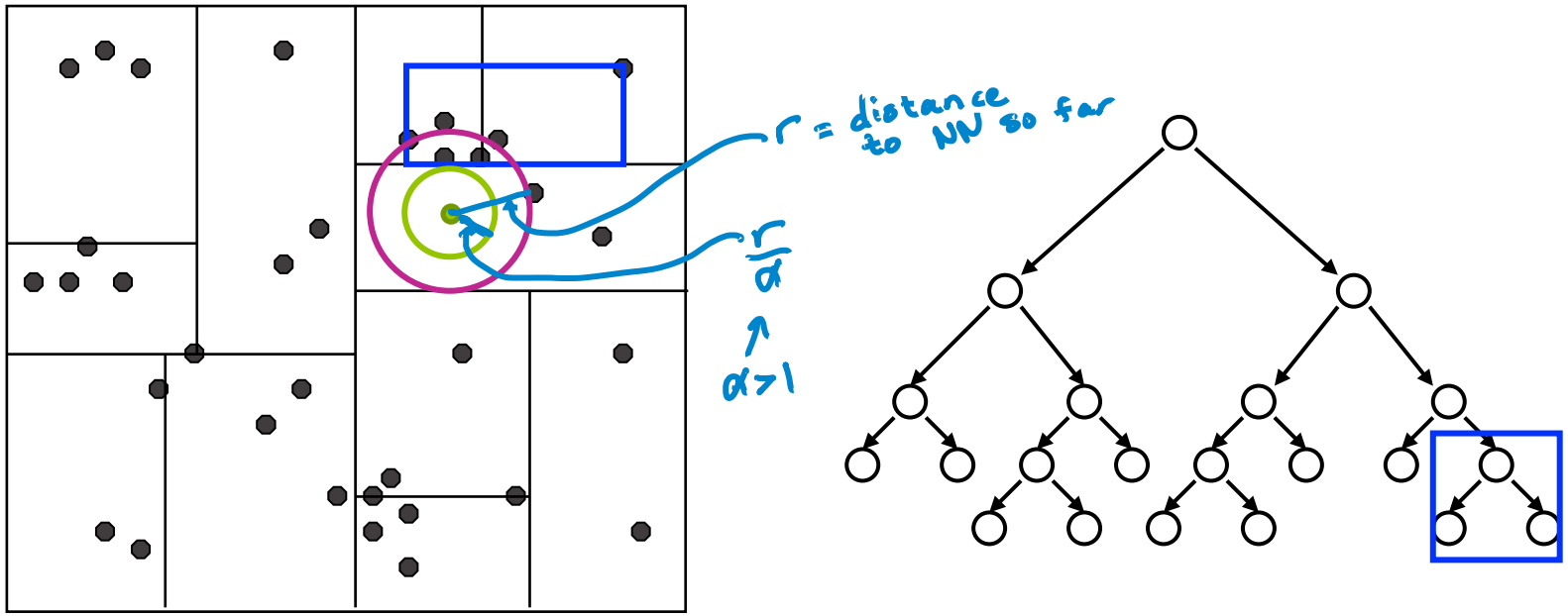
distance to 2nd nearest neighbor in 2-NN example

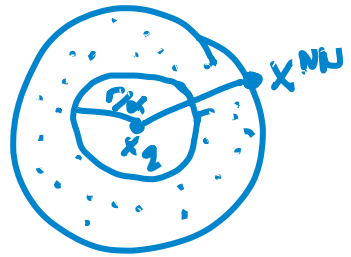


Exactly same algorithm, but maintain distance to furthest of current *k* nearest neighbors

Approximate k-NN search

Approximate k-NN with KD-trees





Before: Prune when distance to bounding box > r

Now: Prune when distance to bounding box > r/ α

Prunes more than allowed, but can guarantee that if we return a neighbor at distance r, then there is no neighbor closer than r/ α

Bound loose…In practice, often closer to optimal.

Saves lots of search time at little cost in quality of NN!

Closing remarks on KD-trees

Tons of variants of kd-trees

- 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 crucial to answer returned

For both, high-dim spaces are hard!

- Number of kd-tree searches can be exponential in dimension
  - Rule of thumb… $N >> 2^d$… Typically useless for large d.
- Distances sensitive to irrelevant features
  - Most dimensions are just noise → everything is far away
  - Need technique to learn which features are important to given task

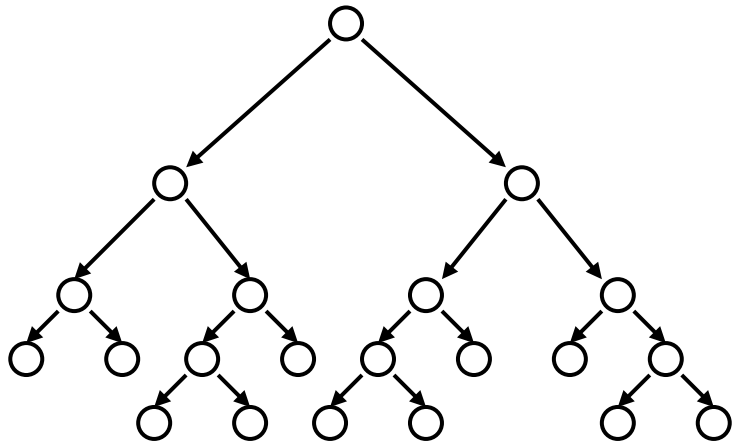
Acknowledgements

Thanks to Andrew Moore
(http://www.cs.cmu.edu/~awm/)
for the KD-trees slide outline

Locality sensitive hashing for approximate NN search

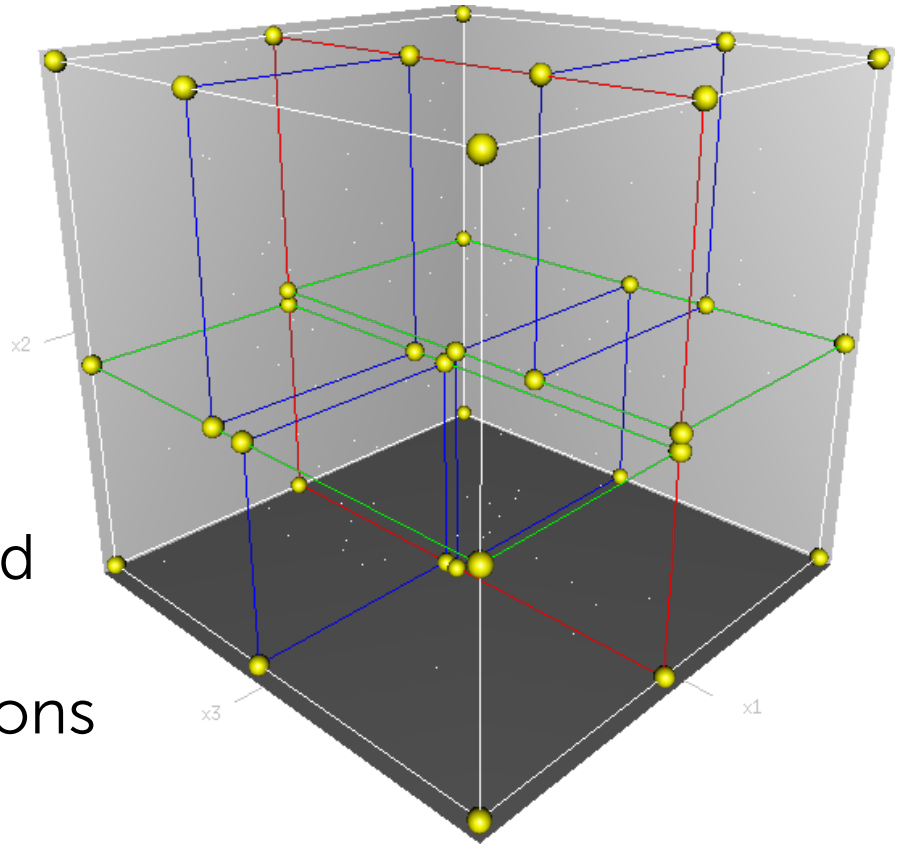
Motivating alternative approaches to approximate NN search

- KD-trees are cool, but…
  - Non-trivial to implement efficiently
  - Problems with high-dimensional data



KD-trees in high dimensions

- Unlikely to have any data points close to query point
- Once "nearby" point is found, the search radius is likely to intersect many hypercubes in at least one dim
- Not many nodes can be pruned
- Can show under some conditions that you visit at least 2^d nodes



Moving away from exact NN search

- Approximate neighbor finding…
  - Don't find exact neighbor, but that's okay for many applications

Out of millions of articles, do we need the closest article or just one that's pretty similar?

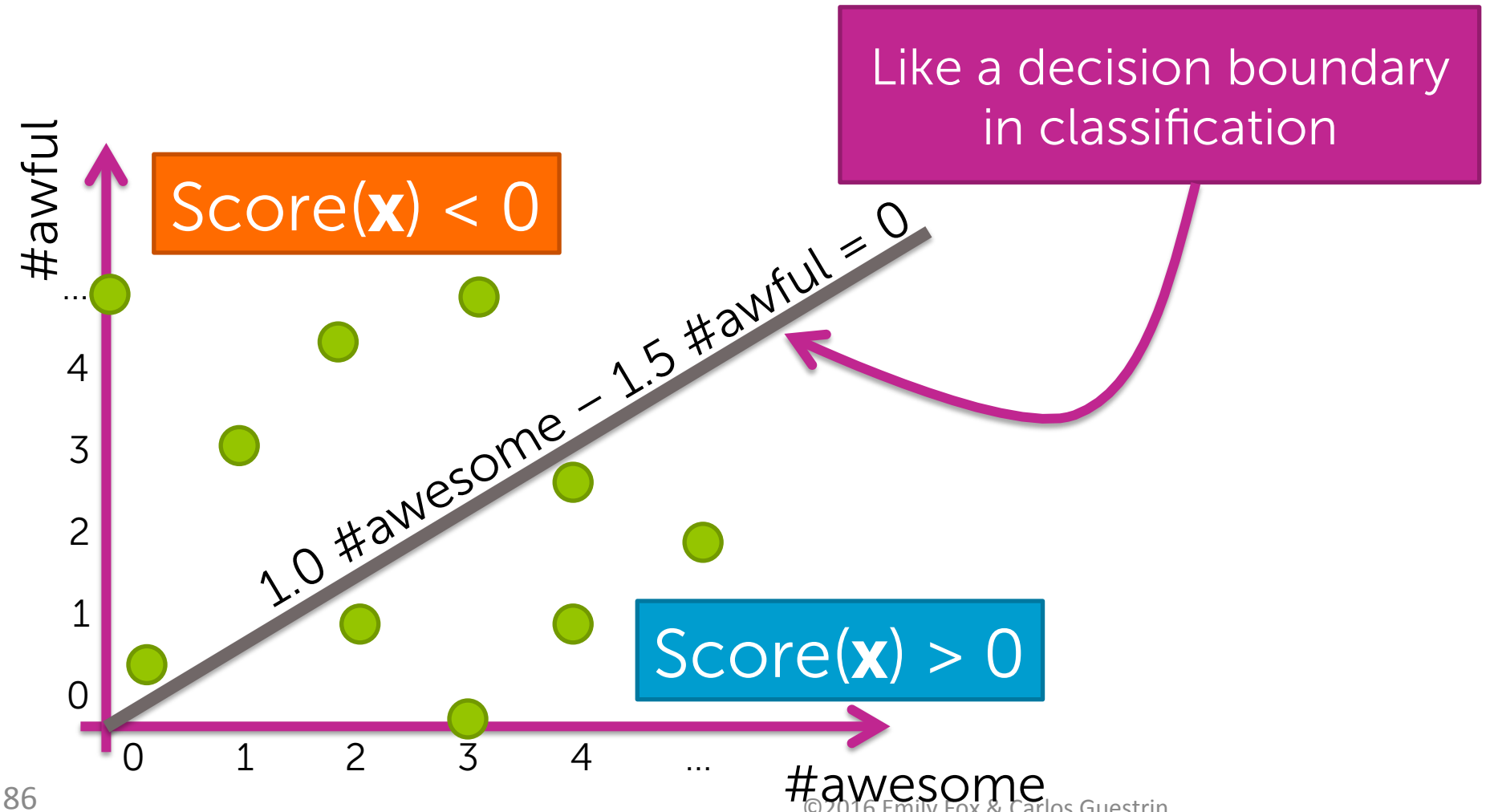
Do we even fully trust our measure of similarity???

- Focus on methods that provide good probabilistic guarantees on approximation

LSH as an alternative to KD-trees

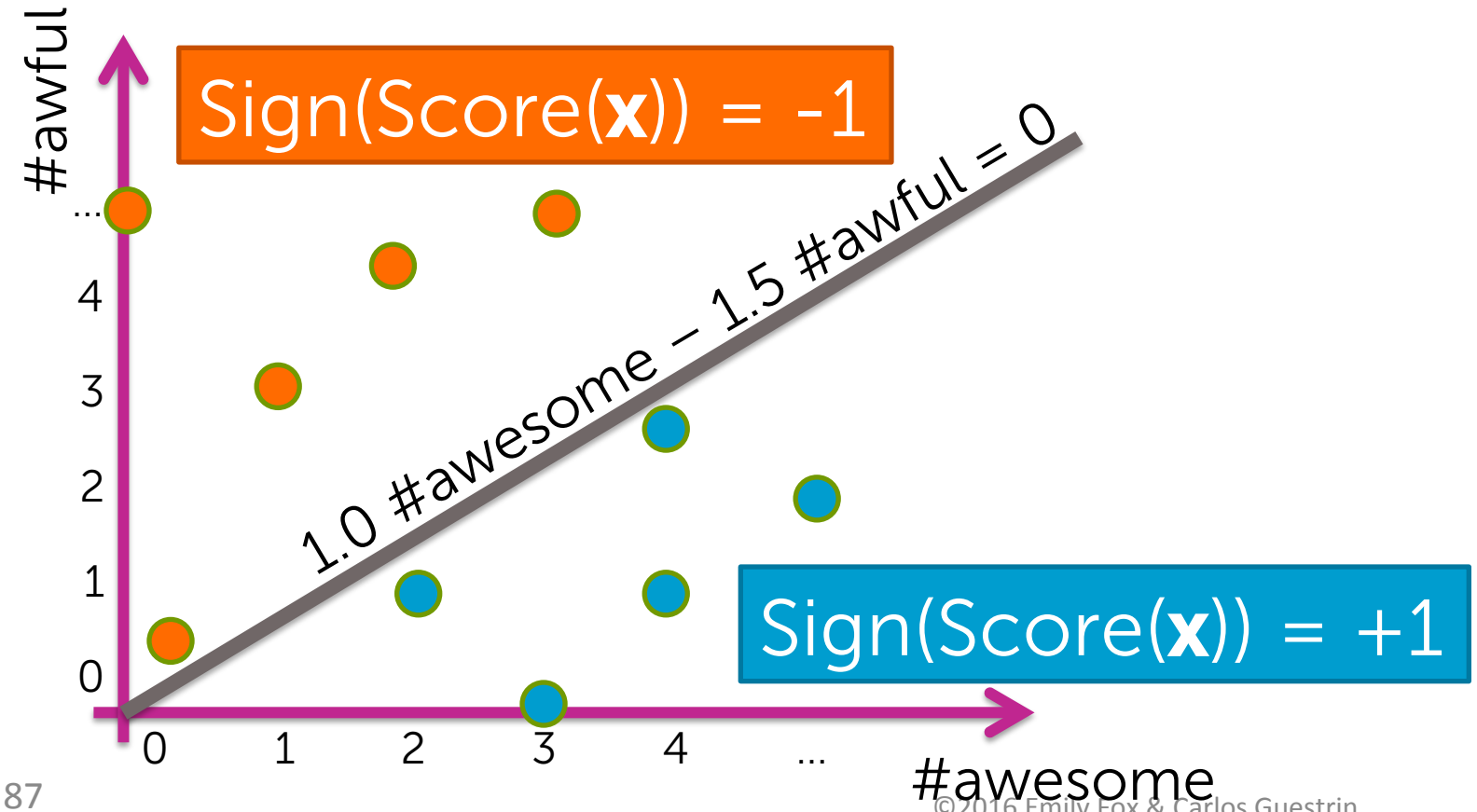
Simple "binning" of data into 2 bins

Score(**x**) = 1.0 #awesome – 1.5 #awful



Simple "binning" of data into 2 bins

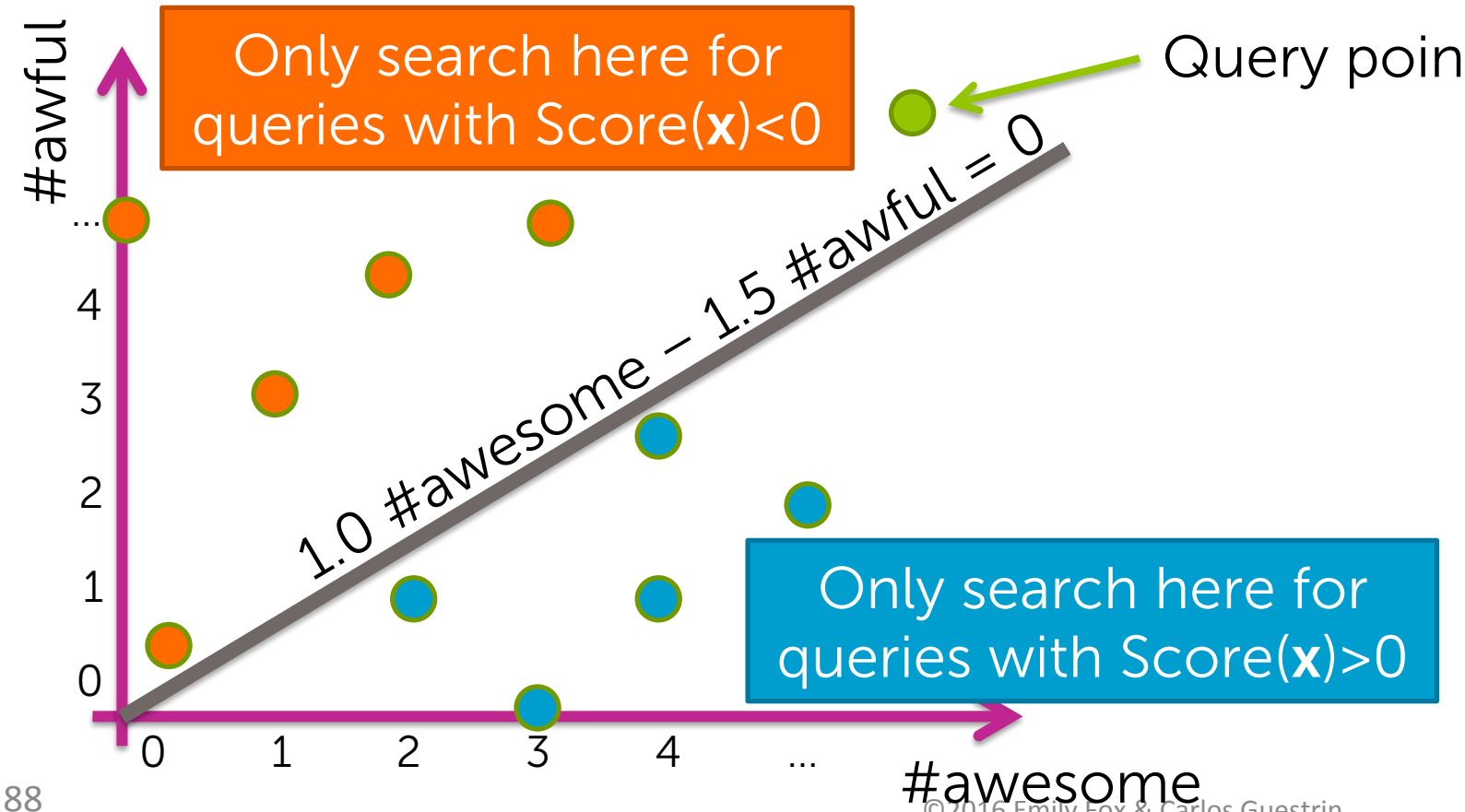
2D Data	Sign(Score)
$\mathbf{x}_1 = [0, 5]$	-1
$\mathbf{x}_2 = [1, 3]$	-1
$\mathbf{x}_3 = [3, 0]$	1
…	…



Using bins for NN search

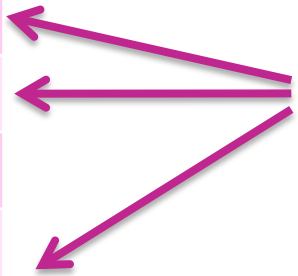
2D Data	Sign(Score)	Bin index
$\mathbf{x}_1$ = [0, 5]	-1	0
$\mathbf{x}_2$ = [1, 3]	-1	0
$\mathbf{x}_3$ = [3, 0]	1	1
…	…	…

candidate neighbors if Score(**x**)<0



Using score for NN search

2D Data	Sign(Score)	Bin index
$\mathbf{x}_1 = [0, 5]$	-1	0
$\mathbf{x}_2 = [1, 3]$	-1	0
$\mathbf{x}_3 = [3, 0]$	1	1
…	…	…



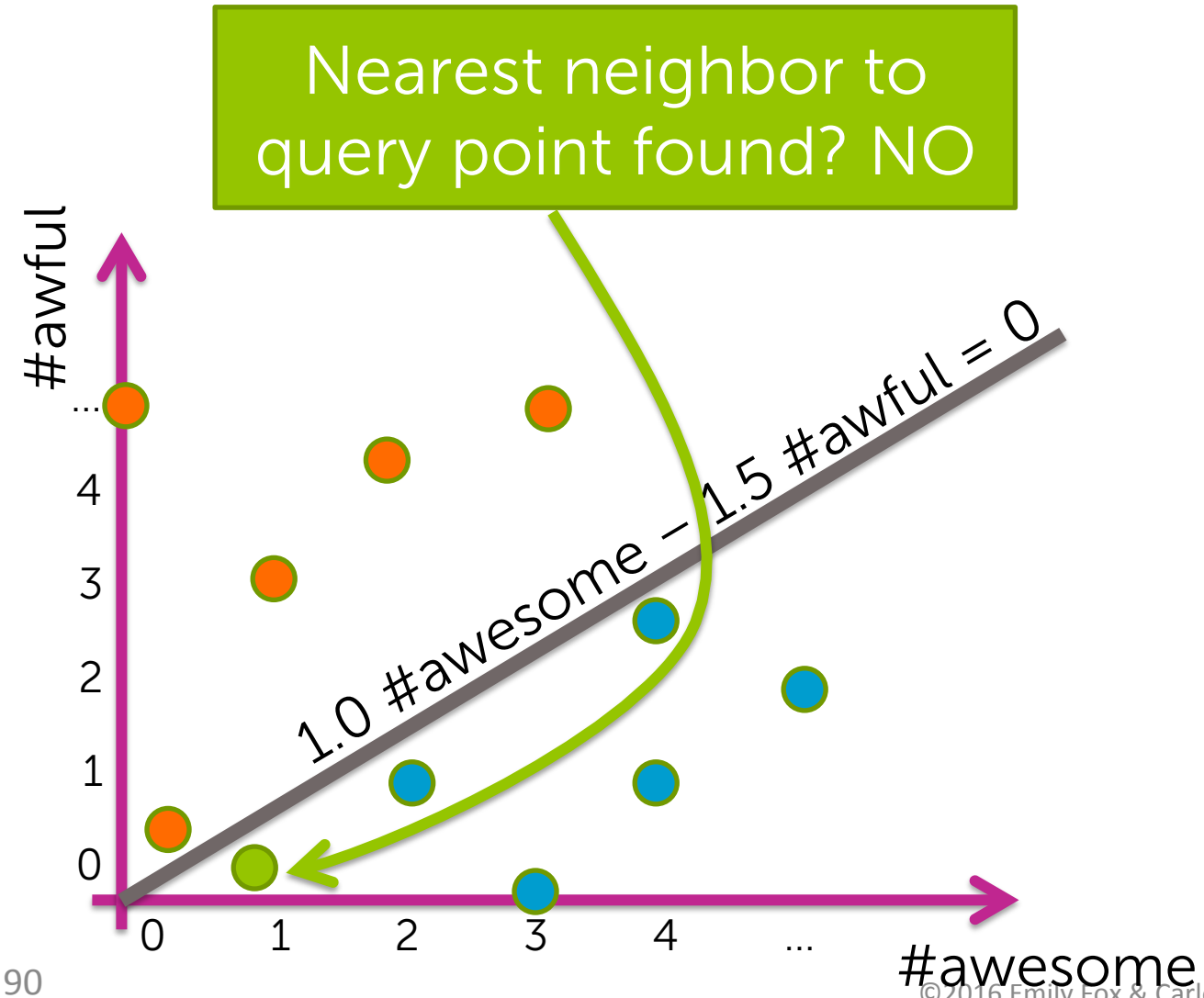
candidate neighbors if Score(x)<0

Bin	0	1
List containing indices of datapoints:	{1,2,4,7,…}	{3,5,6,8,…}

HASH TABLE

search for NN amongst this set

Provides approximate NN



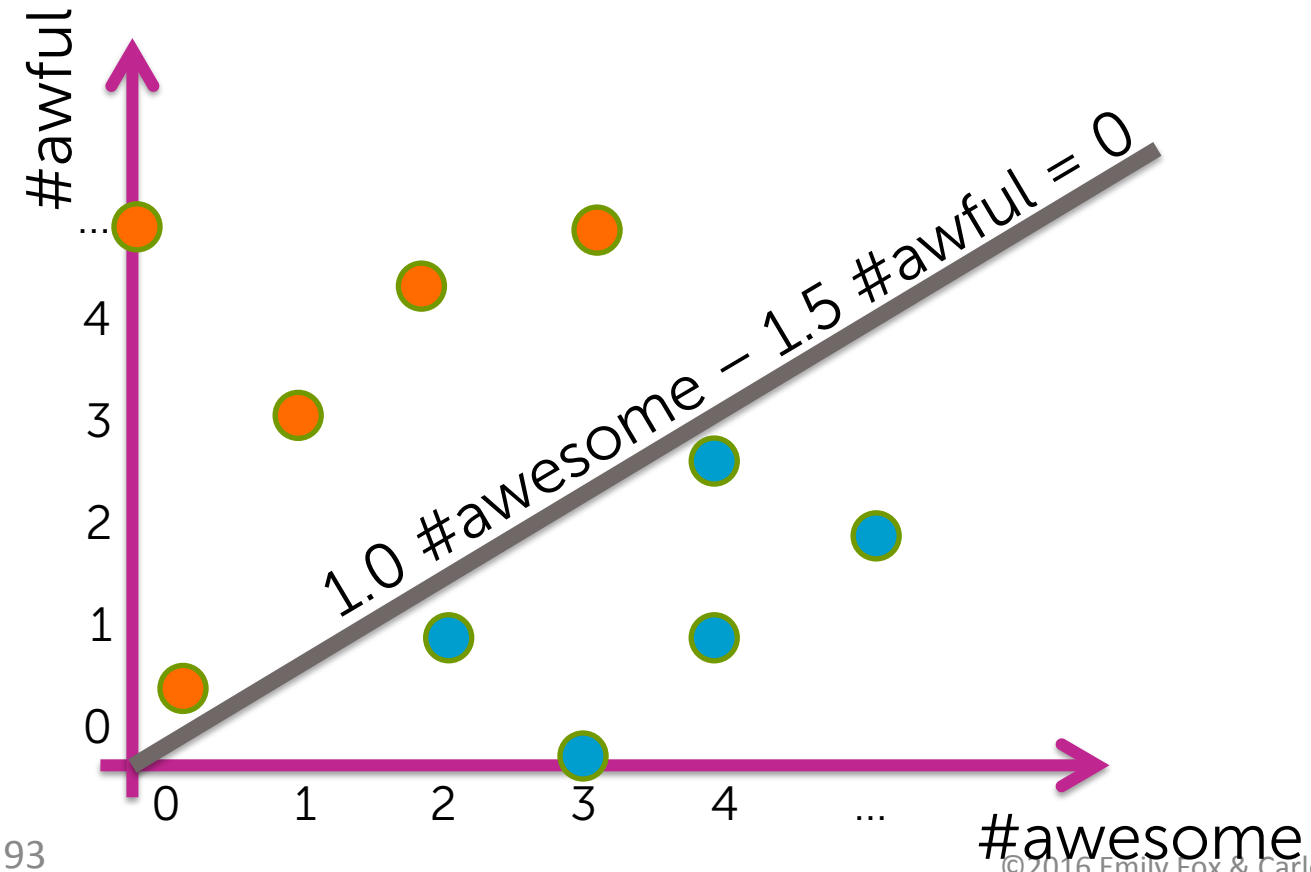
A practical implementation

Three potential issues with simple approach

1. Challenging to find good line
2. Poor quality solution:
   - Points close together get split into separate bins
3. Large computational cost:
   - Bins might contain many points, so still searching over large set for each NN query

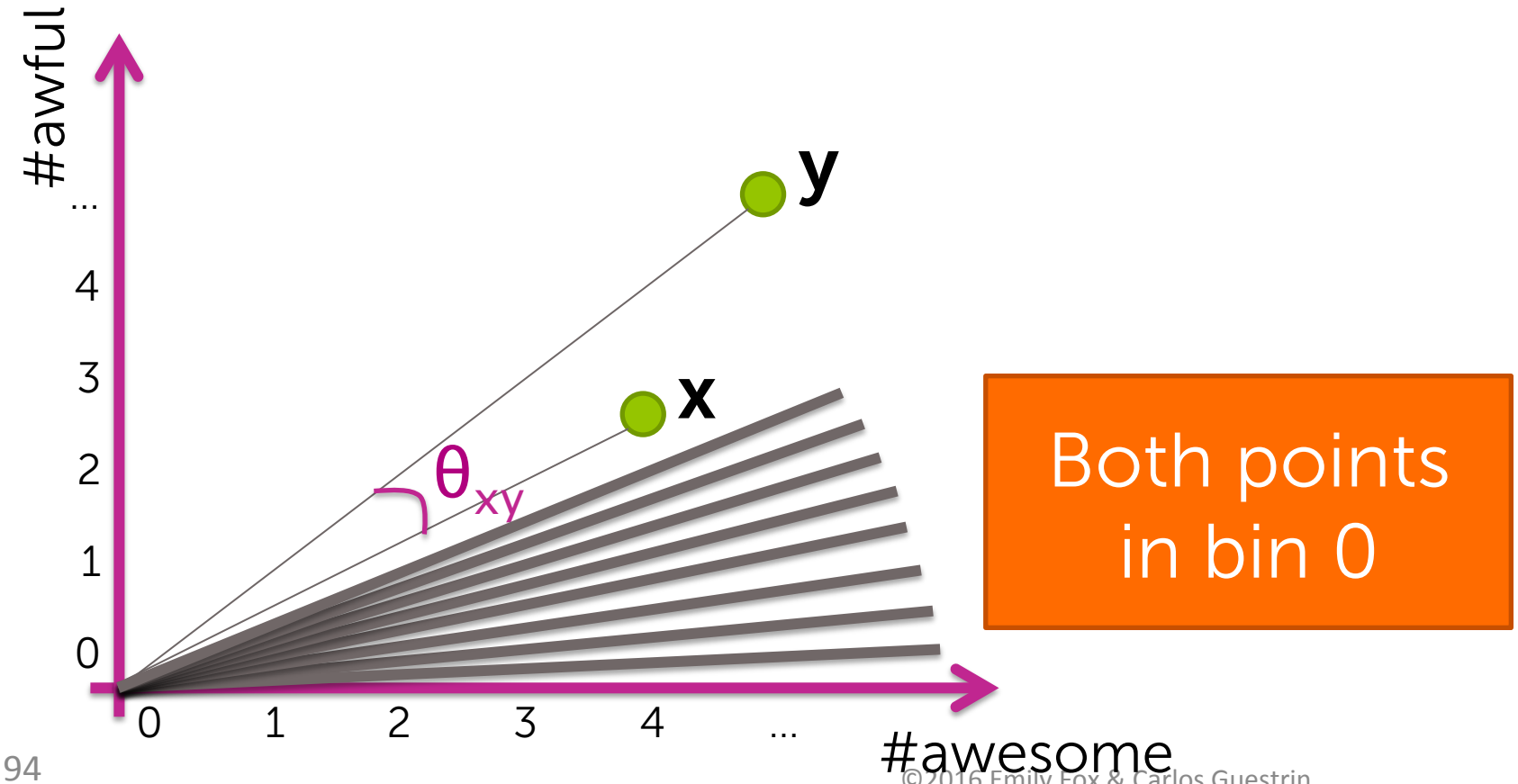
How to define the line?

Crazy idea:
Define line randomly!



How bad can a random line be?

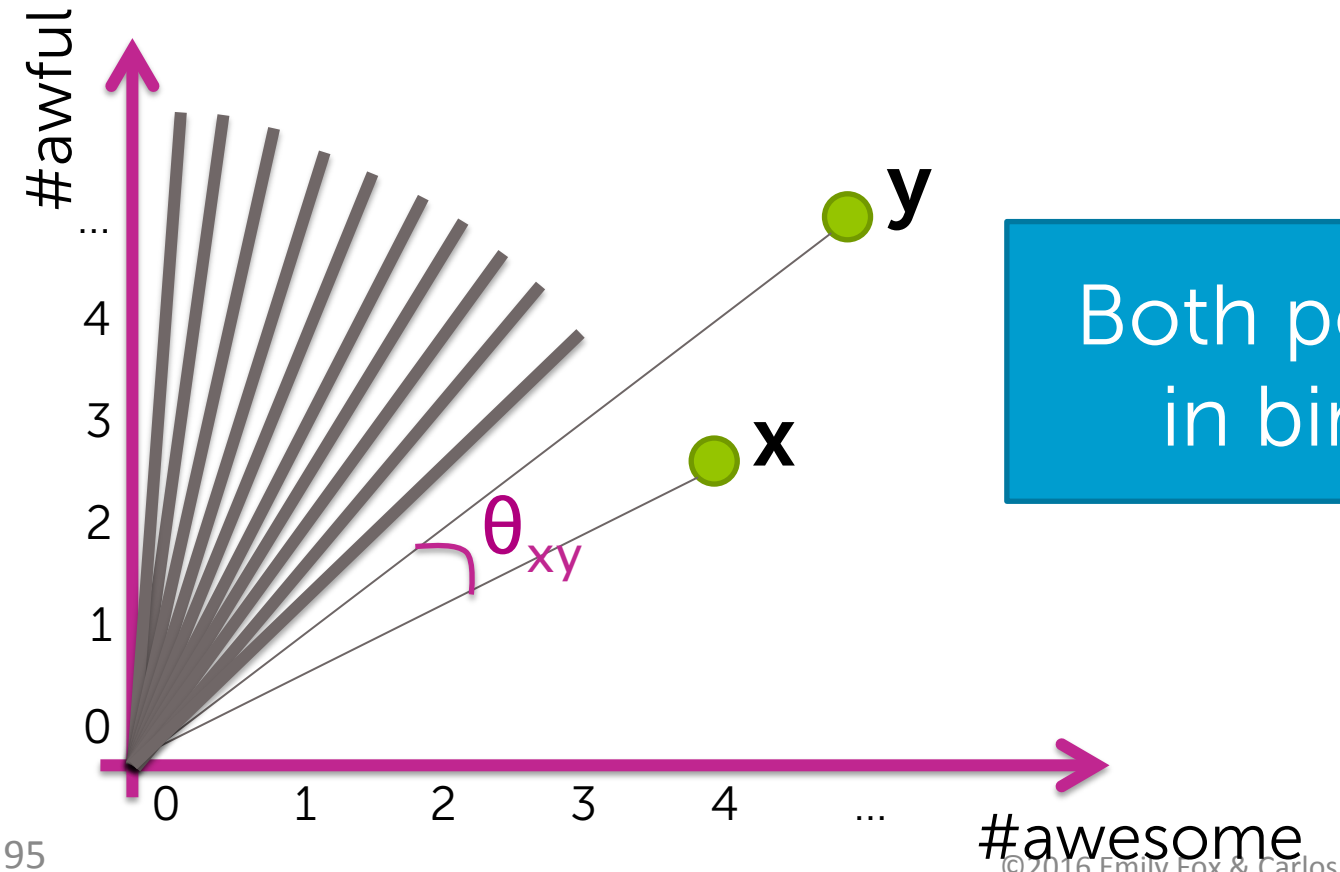
Goal: If **x**,**y** are close (according to cosine similarity), want binned values to be the same.



Both points in bin 0

How bad can a random line be?

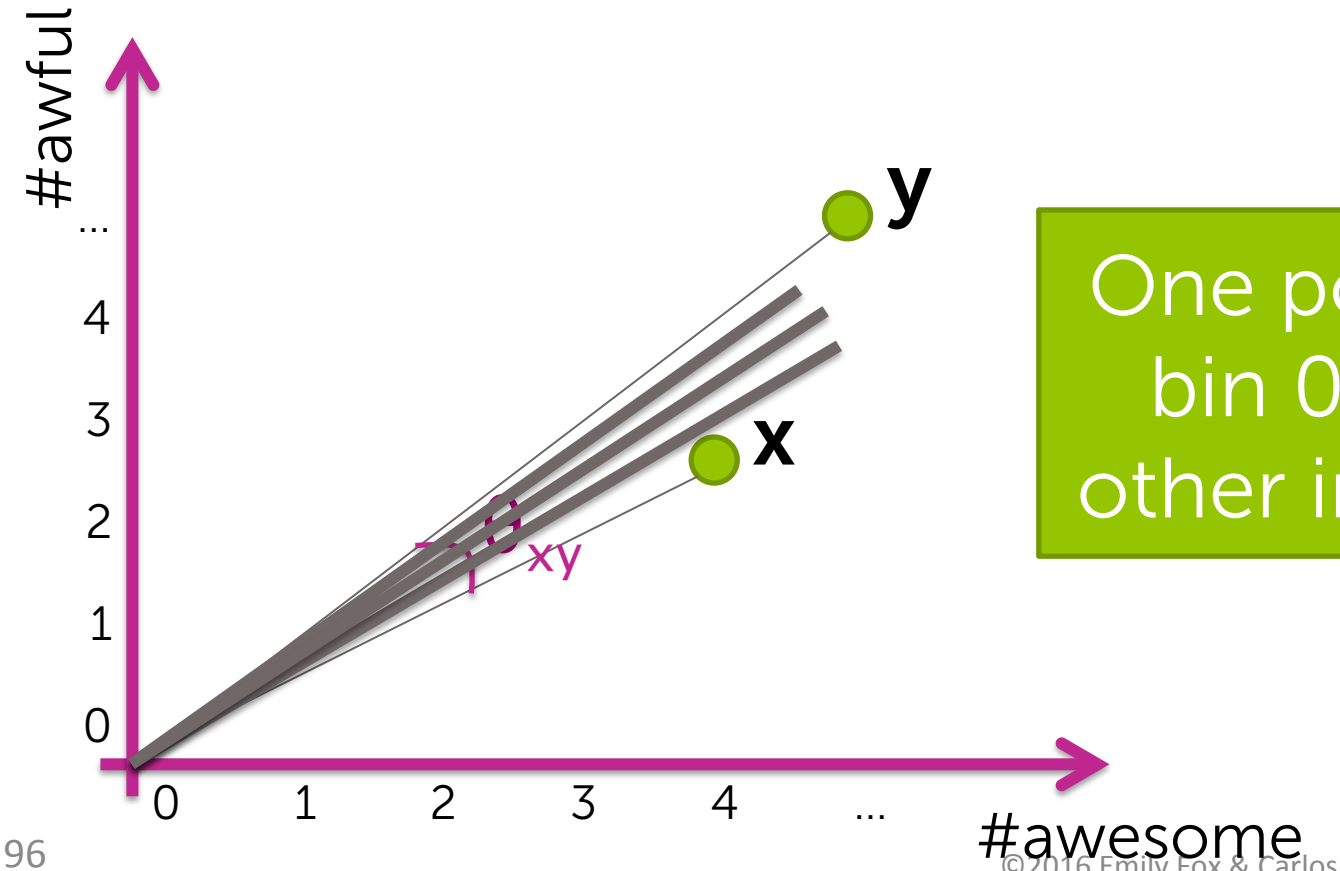
Goal: If **x**,**y** are close (according to cosine similarity), want binned values to be the same.



Both points in bin 1

How bad can a random line be?

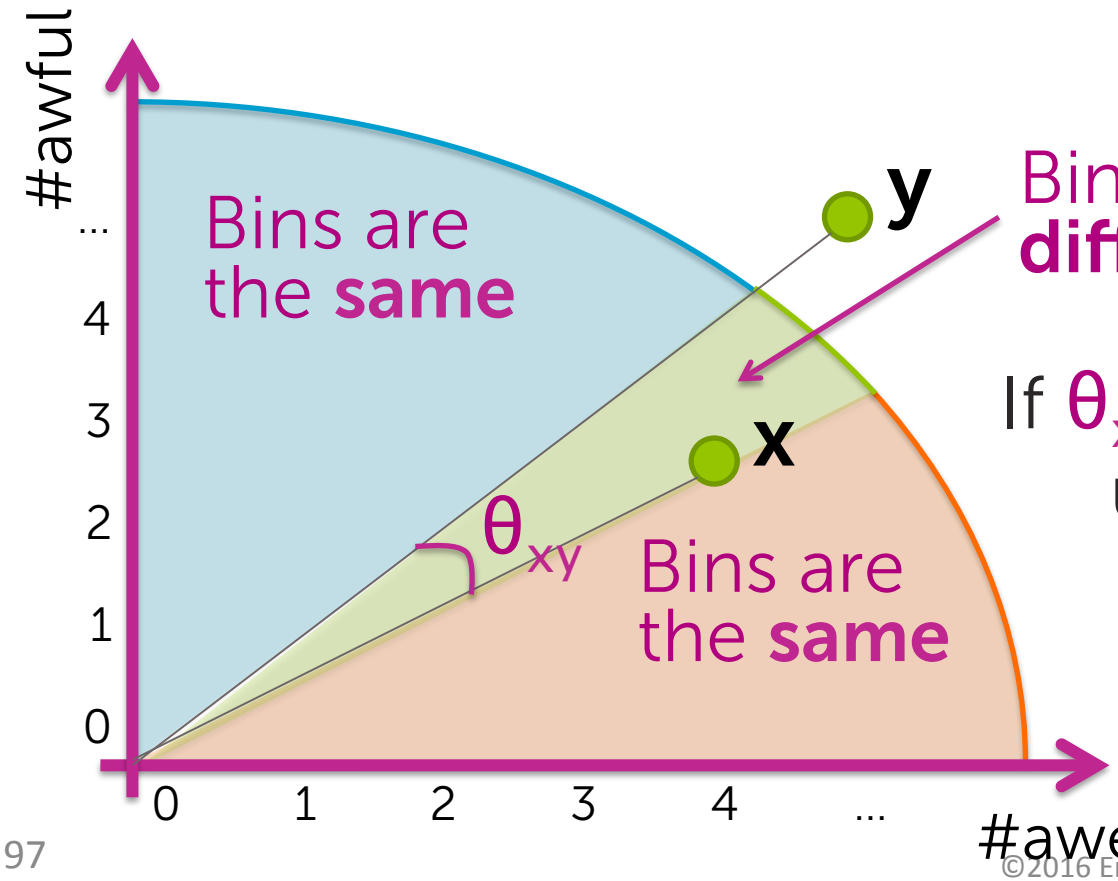
Goal: If **x**,**y** are close (according to cosine similarity), want binned values to be the same.



One point in bin 0 and other in bin 1

How bad can a random line be?

Goal: If **x**,**y** are close (according to cosine similarity), want binned values to be the same.



If θ_{xy} is small (**x**,**y** close), unlikely to be placed into separate bins

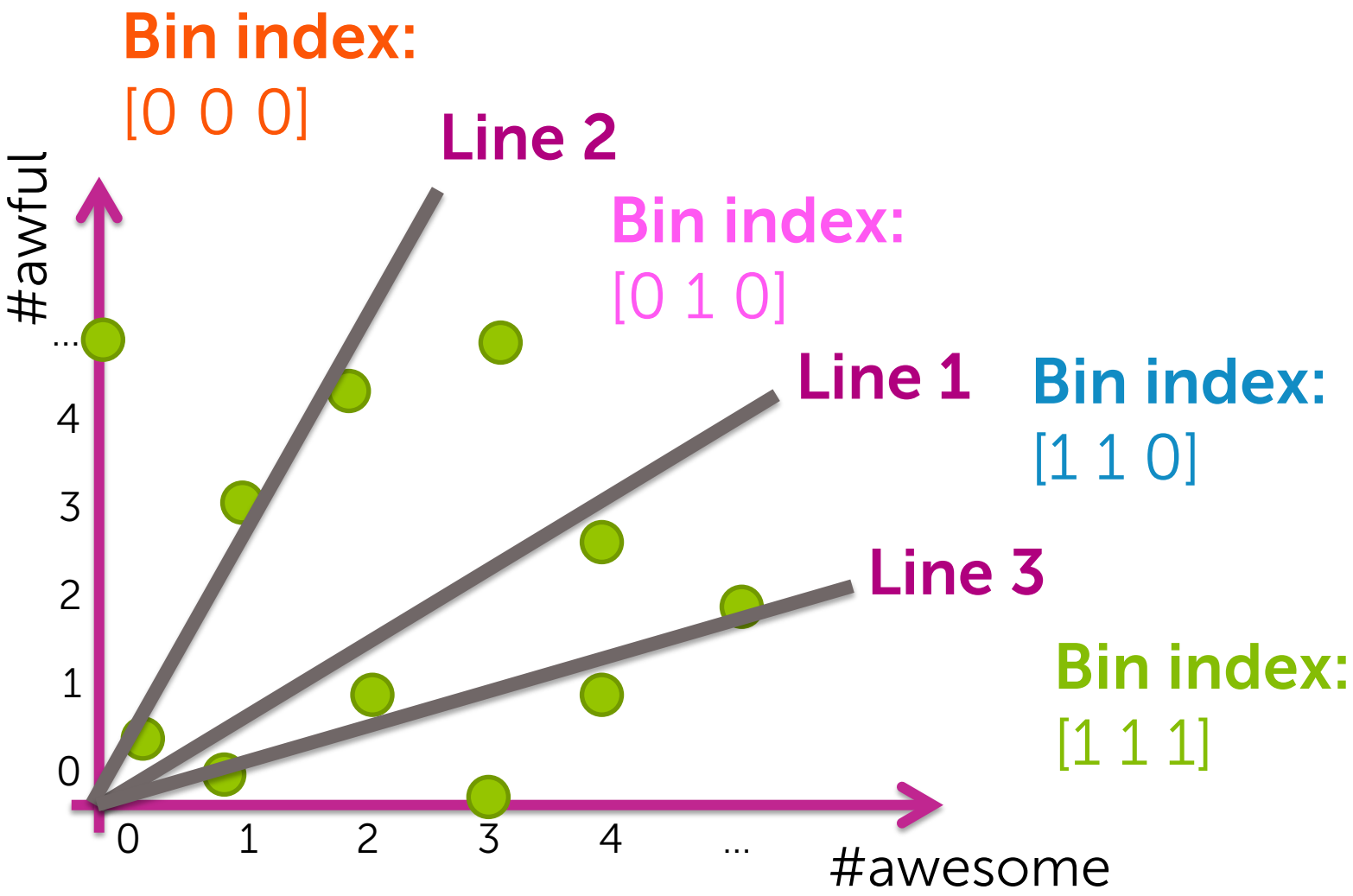
Three potential issues with simple approach

1. Challenging to find good line
2. Poor quality solution:
   - Points close together get split into separate bins
3. Large computational cost:
   - Bins might contain many points, so still searching over large set for each NN query

Bin	0	1
List containing indices of datapoints:	{1,2,4,7,...}	{3,5,6,8,...}

Improving efficiency:
Reducing # points examined per query

Reducing search cost through more bins



Using score for NN search

2D Data	Sign ($Score_1$)	Bin 1 index	Sign ($Score_2$)	Bin 2 index	Sign ($Score_3$)	Bin 3 index
$\mathbf{x}_1$ = [0, 5]	-1	0	-1	0	-1	0
$\mathbf{x}_2$ = [1, 3]	-1	0	-1	0	-1	0
$\mathbf{x}_3$ = [3, 0]	1	1	1	1	1	1
…	…	…	…	…	…	…

Bin	[0 0 0] = 0	[0 0 1] = 1	[0 1 0] = 2	[0 1 1] = 3	[1 0 0] = 4	[1 0 1] = 5	[1 1 0] = 6	[1 1 1] = 7
Data indices:	{1,2}	--	{4,8,11}	--	--	--	{7,9,10}	{3,5,6}

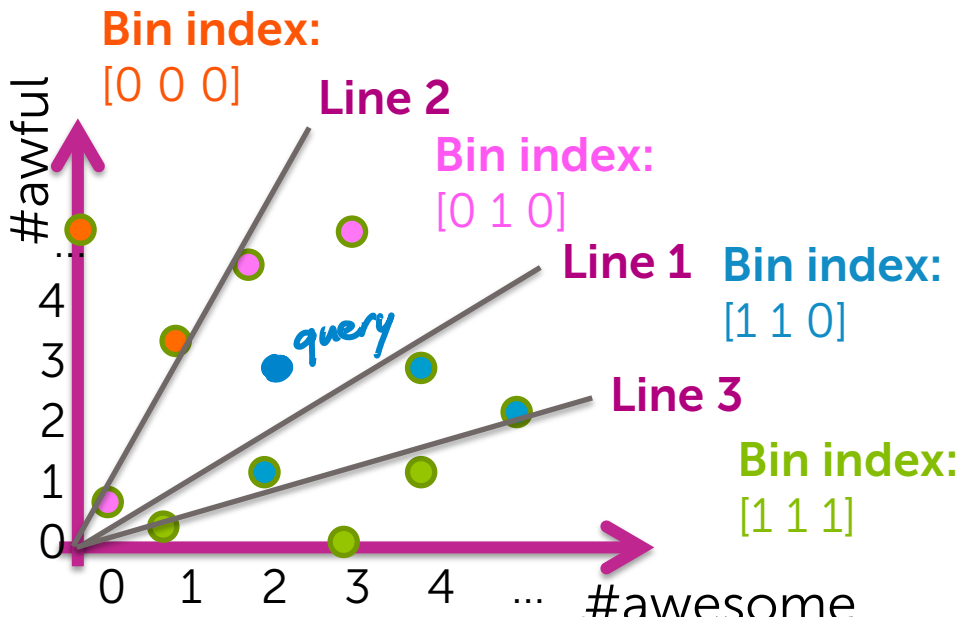
search for NN amongst this set

Improving search quality by searching neighboring bins

Bin	[0 0 0] = 0	[0 0 1] = 1	[0 1 0] = 2	[0 1 1] = 3	[1 0 0] = 4	[1 0 1] = 5	[1 1 0] = 6	[1 1 1] = 7
Data indices:	{1,2}	--	{4,8,11}	--	--	--	{7,9,10}	{3,5,6}

Query point here, but is NN?

Not necessarily

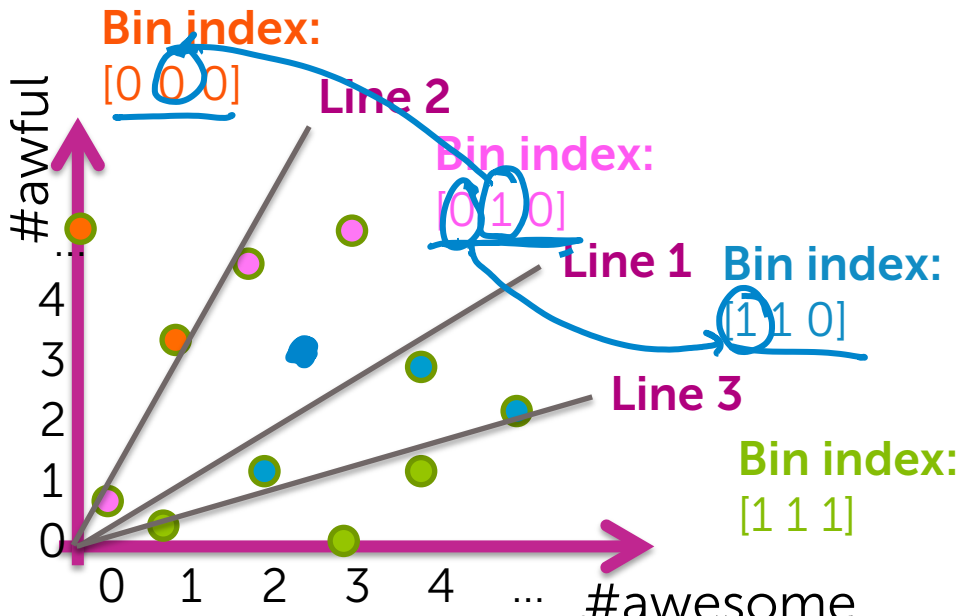


Even worse than before...Each line can split pts.
Sacrificing accuracy for speed

Improving search quality by searching neighboring bins

Bin	[0 0 0] = 0	[0 0 1] = 1	[0 1 0] = 2	[0 1 1] = 3	[1 0 0] = 4	[1 0 1] = 5	[1 1 0] = 6	[1 1 1] = 7
Data indices:	{1,2}	--	{4,8,11}	--	--	--	{7,9,10}	{3,5,6}

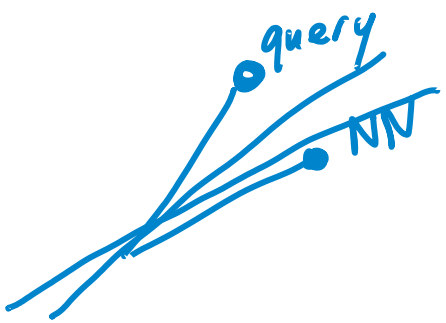
Next closest bins (flip 1 bit)

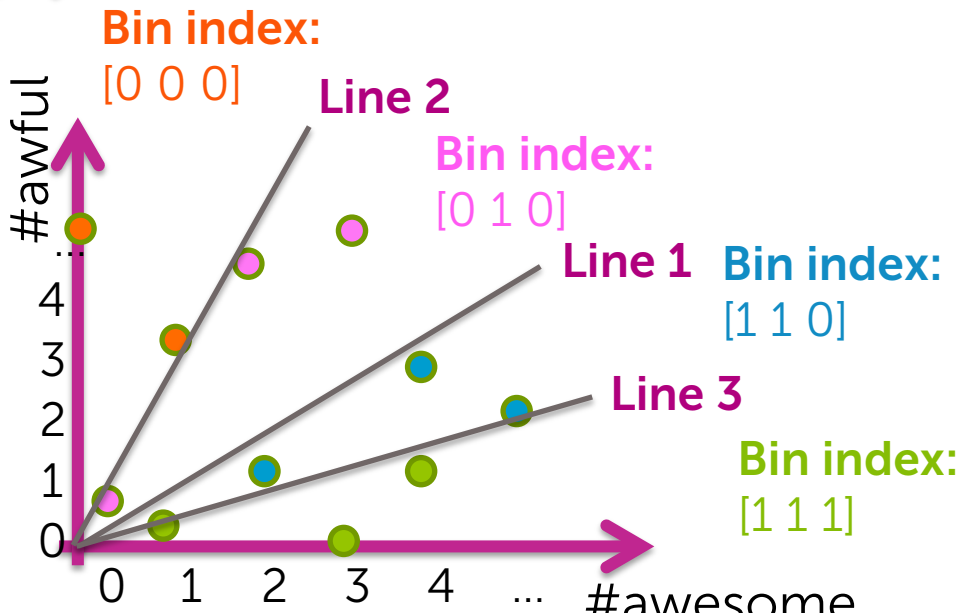


Improving search quality by searching neighboring bins

Bin	[0 0 0] = 0	[0 0 1] = 1	[0 1 0] = 2	[0 1 1] = 3	[1 0 0] = 4	[1 0 1] = 5	[1 1 0] = 6	[1 1 1] = 7
Data indices:	{1,2}	--	{4,8,11}	--	--	--	{7,9,10}	{3,5,6}

Further bin (flip 2 bits)





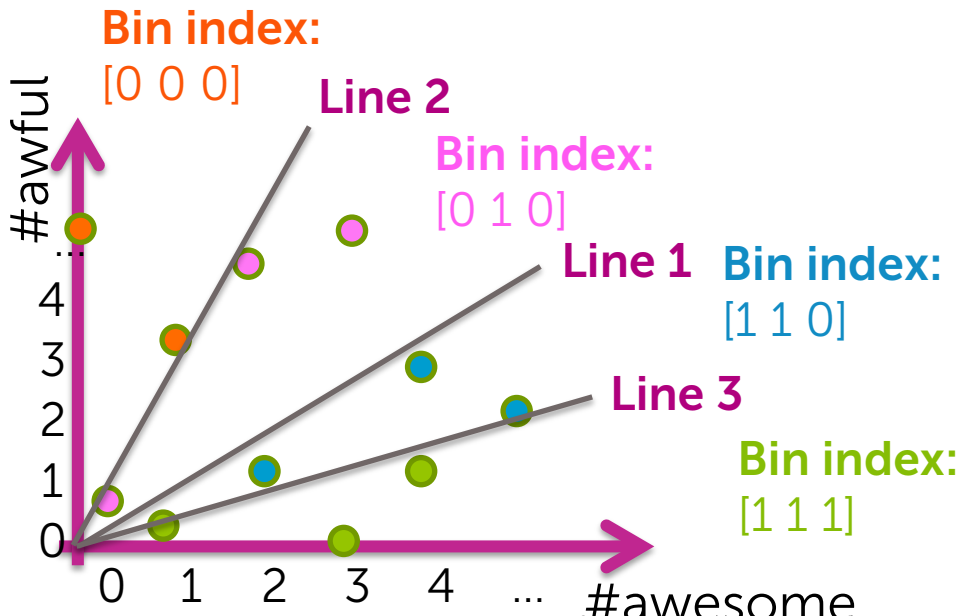
Improving search quality by searching neighboring bins

Bin	[0 0 0] = 0	[0 0 1] = 1	[0 1 0] = 2	[0 1 1] = 3	[1 0 0] = 4	[1 0 1] = 5	[1 1 0] = 6	[1 1 1] = 7
Data indices:	{1,2}	--	{4,8,11}	--	--	--	{7,9,10}	{3,5,6}

Quality of retrieved NN can only improve with searching more bins

Algorithm:

Continue searching until computational budget is reached or quality of NN good enough



LSH recap

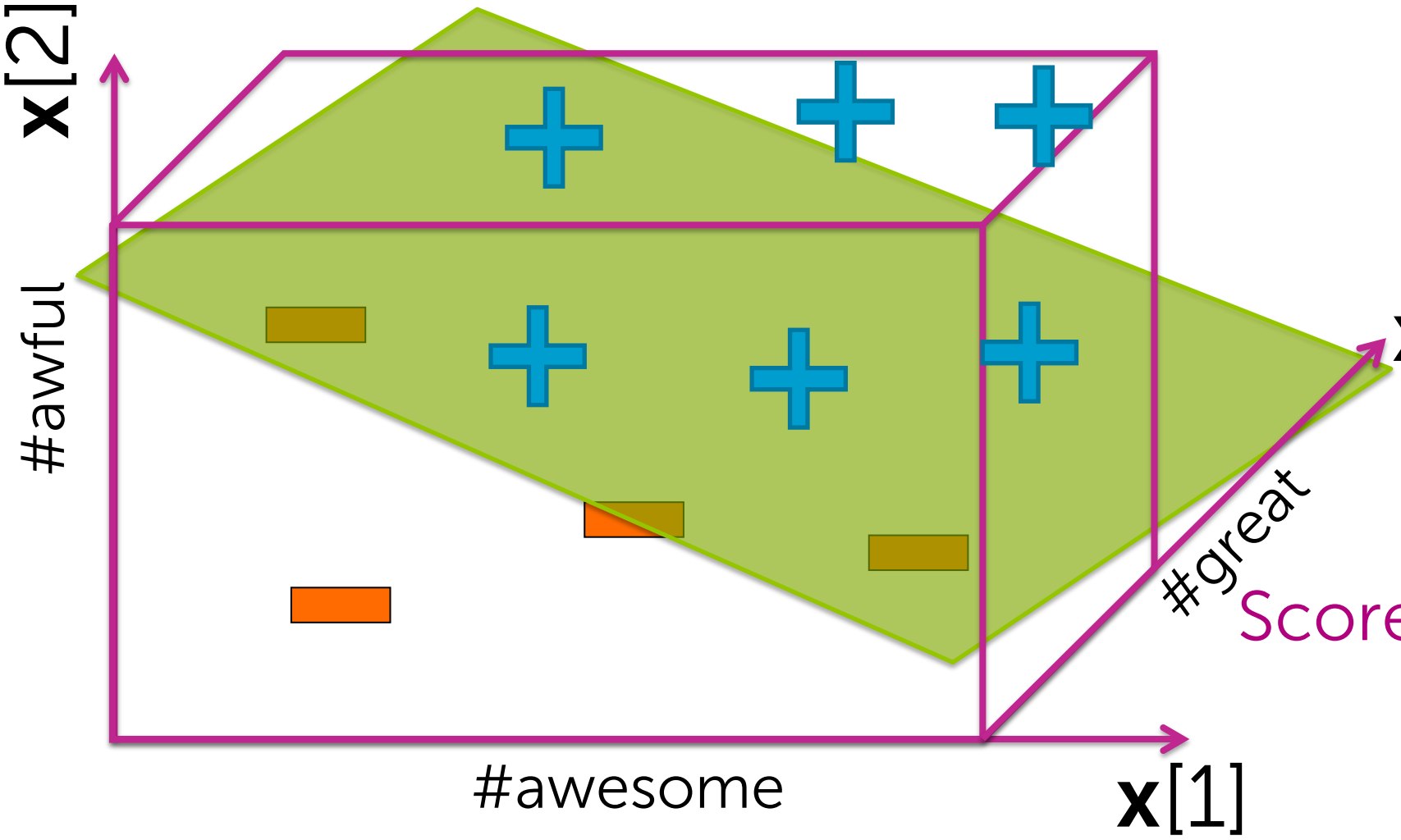
kd-tree competitor data structure

- Draw h random lines
- Compute "score" for each point under each line and translate to binary index
- Use h-bit binary vector per data point as bin index
- Create hash table

- For each query point **x**, search bin(**x**), then neighboring bins until time limit

Moving to higher dimensions d

Draw random *planes*



$$\text{Score}(\mathbf{x}) = v_1 \#\text{awesome} + v_2 \#\text{awful} + v_3 \#\text{great}$$

Cost of binning points in d-dim

$$\text{Score}(\mathbf{x}) = v_1^{(i)} \#\text{awesome} + v_2^{(i)} \#\text{awful} + v_3^{(i)} \#\text{great}$$

(i = i^{th} hyperplane)

Per data point, need d multiplies to determine bin index per plane

In high-dim, (and some applications) this is often a sparse mult.

One-time cost offset if many queries of fixed dataset

Using multiple tables for even greater efficiency in NN search

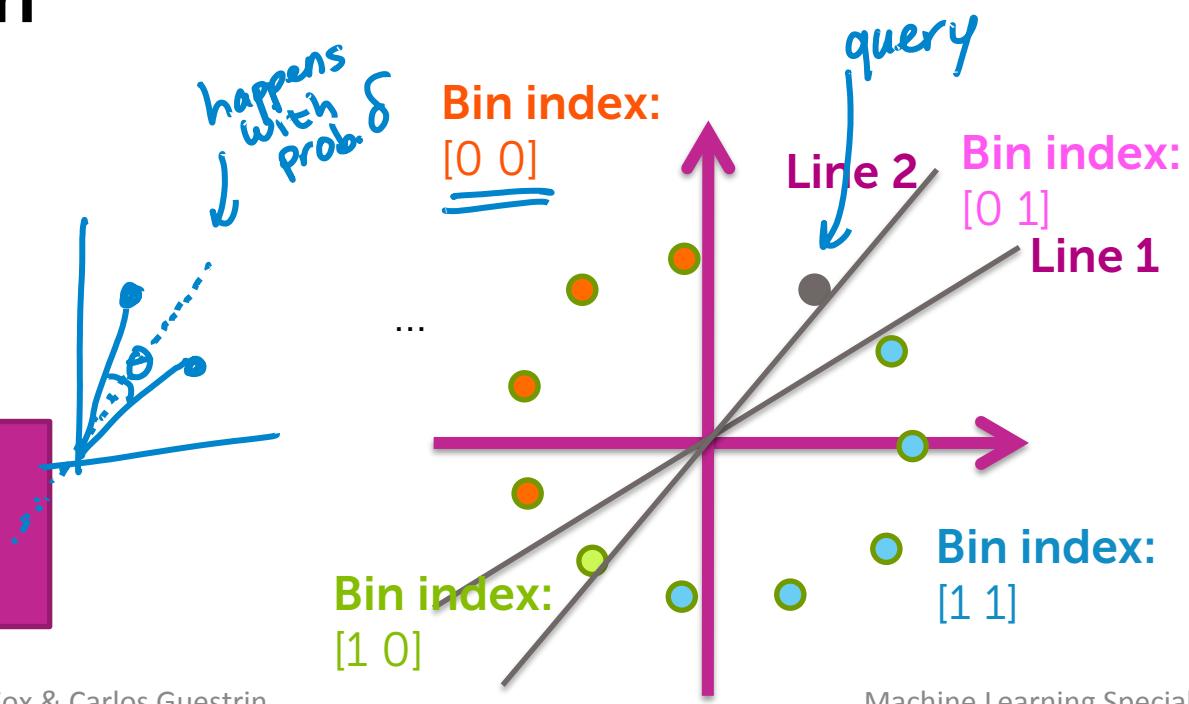
If I throw down 2 lines…

Bin	[0 0] = 0	[0 1] = 1	[1 0] = 2	[1 1] = 3
indices	L0	L1	L2	L3

For simplicity, assume we **search bins 1 bit off** from query

Let δ be the probability of a line falling between points θ apart

Search 3 bins and do not find NN with probability δ^2

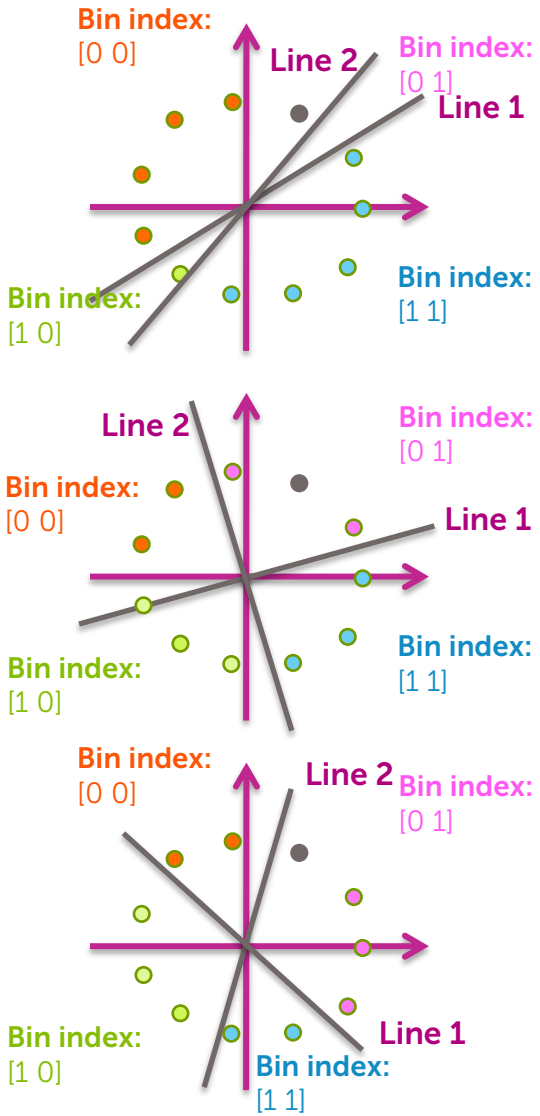


What if I repeat the 2-line binning?

Bin	[0 0] = 0	[0 1] = 1	[1 0] = 2	[1 1] = 3
indices	L0	L1	L2	L3

Bin	[0 0] = 0	[0 1] = 1	[1 0] = 2	[1 1] = 3
indices	L0	L1	L2	L3

Bin	[0 0] = 0	[0 1] = 1	[1 0] = 2	[1 1] = 3
indices	L0	L1	L2	L3

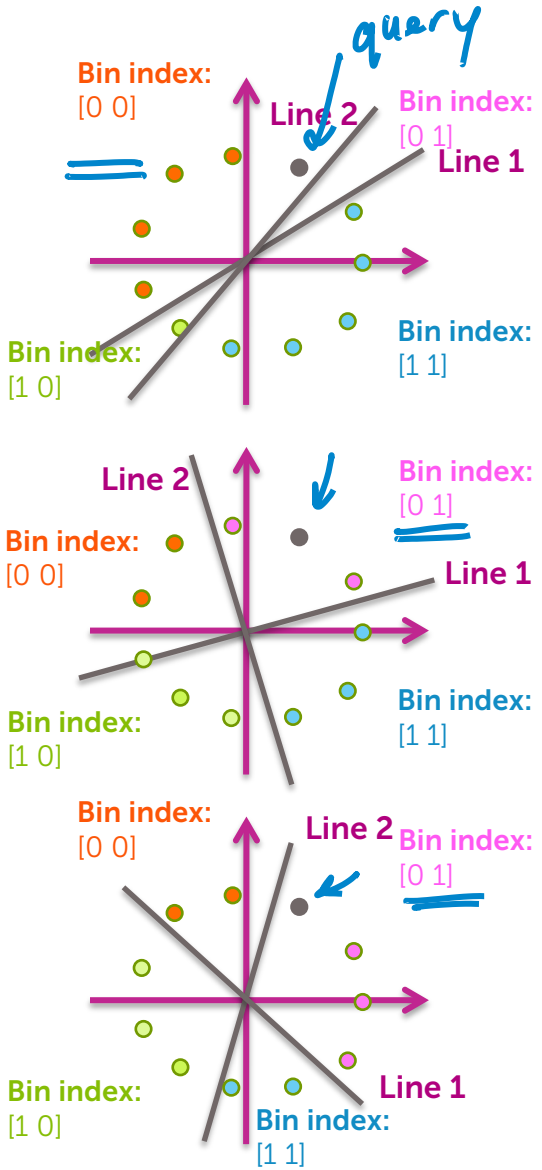


What if I repeat the 2-line binning?

Bin	[0 0] = 0	[0 1] = 1	[1 0] = 2	[1 1] = 3
indices	L0	L1	L2	L3

Bin	[0 0] = 0	[0 1] = 1	[1 0] = 2	[1 1] = 3
indices	L0	L1	L2	L3

Bin	[0 0] = 0	[0 1] = 1	[1 0] = 2	[1 1] = 3
indices	L0	L1	L2	L3



Now, search only query bin **per table**

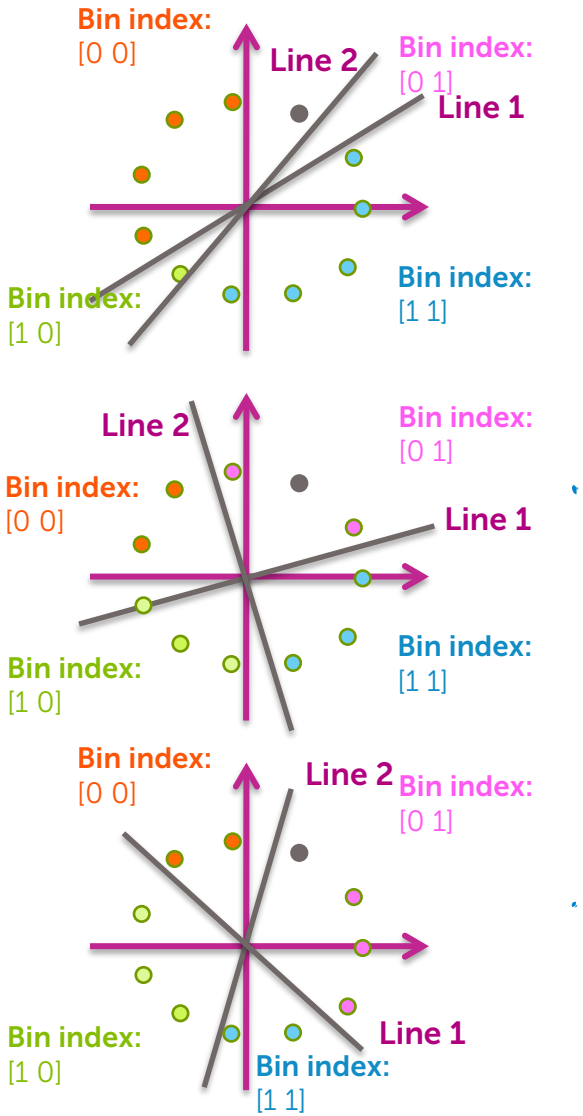
Still searching 3 bins, but what is chance of not finding NN?

What if I repeat the 2-line binning?

Bin	[0 0] = 0	[0 1] = 1	[1 0] = 2	[1 1] = 3
indices	L0	L1	L2	L3

Bin	[0 0] = 0	[0 1] = 1	[1 0] = 2	[1 1] = 3
indices	L0	L1	L2	L3

Bin	[0 0] = 0	[0 1] = 1	[1 0] = 2	[1 1] = 3
indices	L0	L1	L2	L3



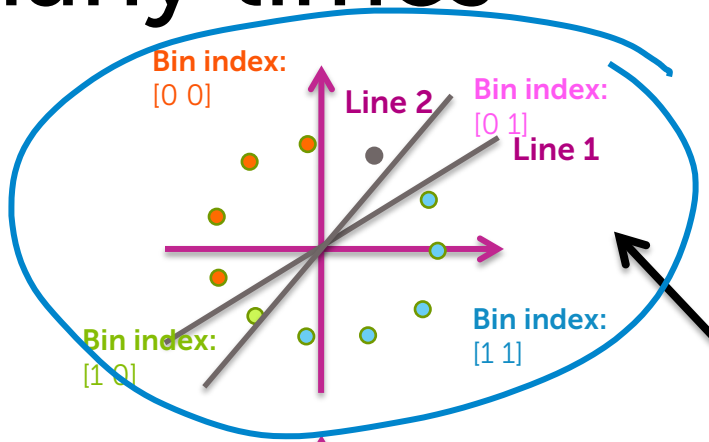
What is chance that query pt and NN are split in **all tables**?

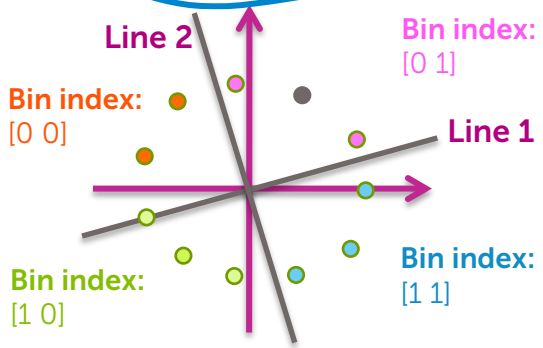
Probability of splitting neighboring points many times

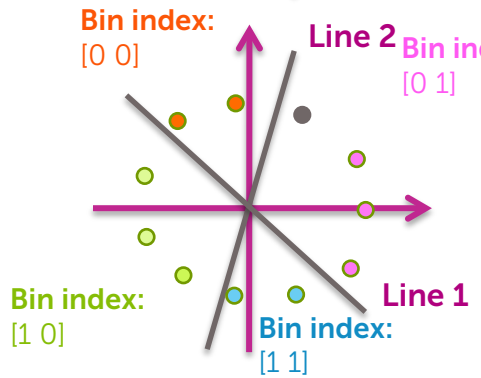
Bin	[0 0] = 0	[0 1] = 1	[1 0] = 2	[1 1] = 3
indices	L0	L1	L2	L3

Bin	[0 0] = 0	[0 1] = 1	[1 0] = 2	[1 1] = 3
indices	L0	L1	L2	L3

Bin	[0 0] = 0	[0 1] = 1	[1 0] = 2	[1 1] = 3
indices	L0	L1	L2	L3







Probability NN is in different bin:

Prob = 1-Pr(same bin)
= $1-(1-\delta)^2$
= $2\delta - \delta^2$

$1-\delta$ = prob. that 1 line does not split query + NN

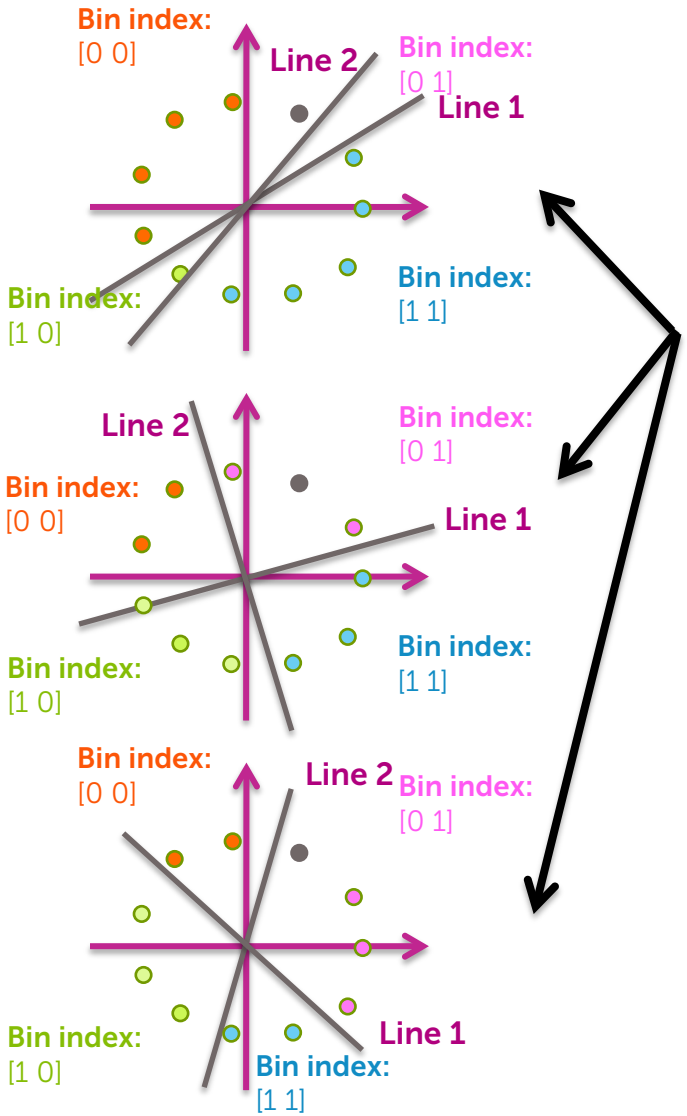
$(1-\delta)^2$ = prob. that 2 lines don't split pts

Probability of splitting neighboring points many times

Bin	[0 0] = 0	[0 1] = 1	[1 0] = 2	[1 1] = 3
indices	L0	L1	L2	L3

Bin	[0 0] = 0	[0 1] = 1	[1 0] = 2	[1 1] = 3
indices	L0	L1	L2	L3

Bin	[0 0] = 0	[0 1] = 1	[1 0] = 2	[1 1] = 3
indices	L0	L1	L2	L3

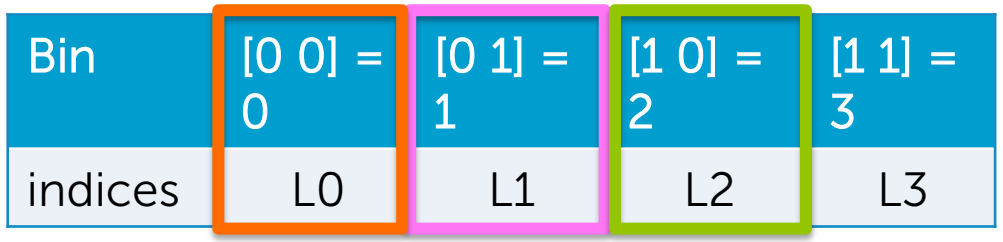


Probability NN is in different bin in **all 3 tables**:

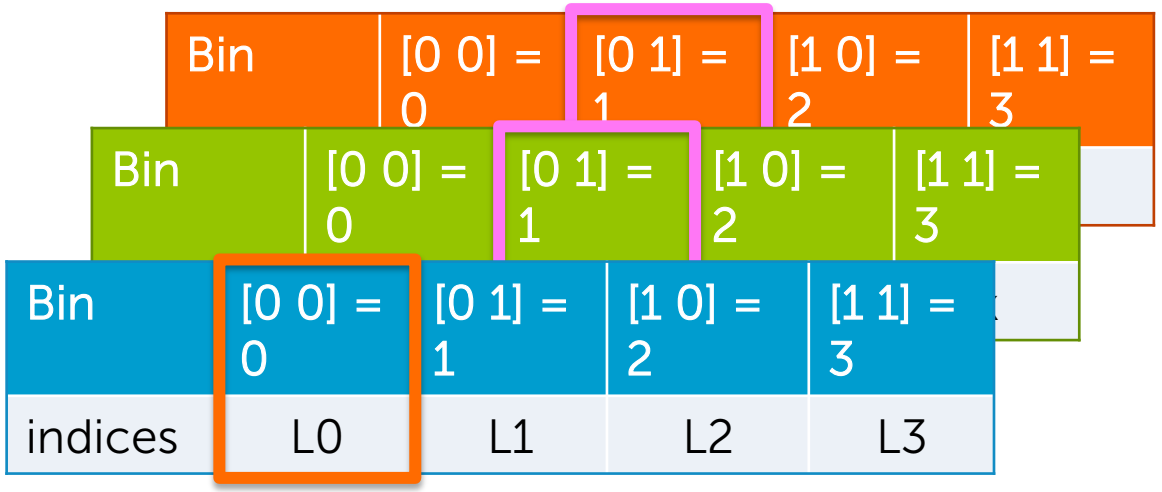
$$\text{Prob} = (2\delta - \delta^2)^3$$

# of hash tables

Comparing approaches for 2-bit tables

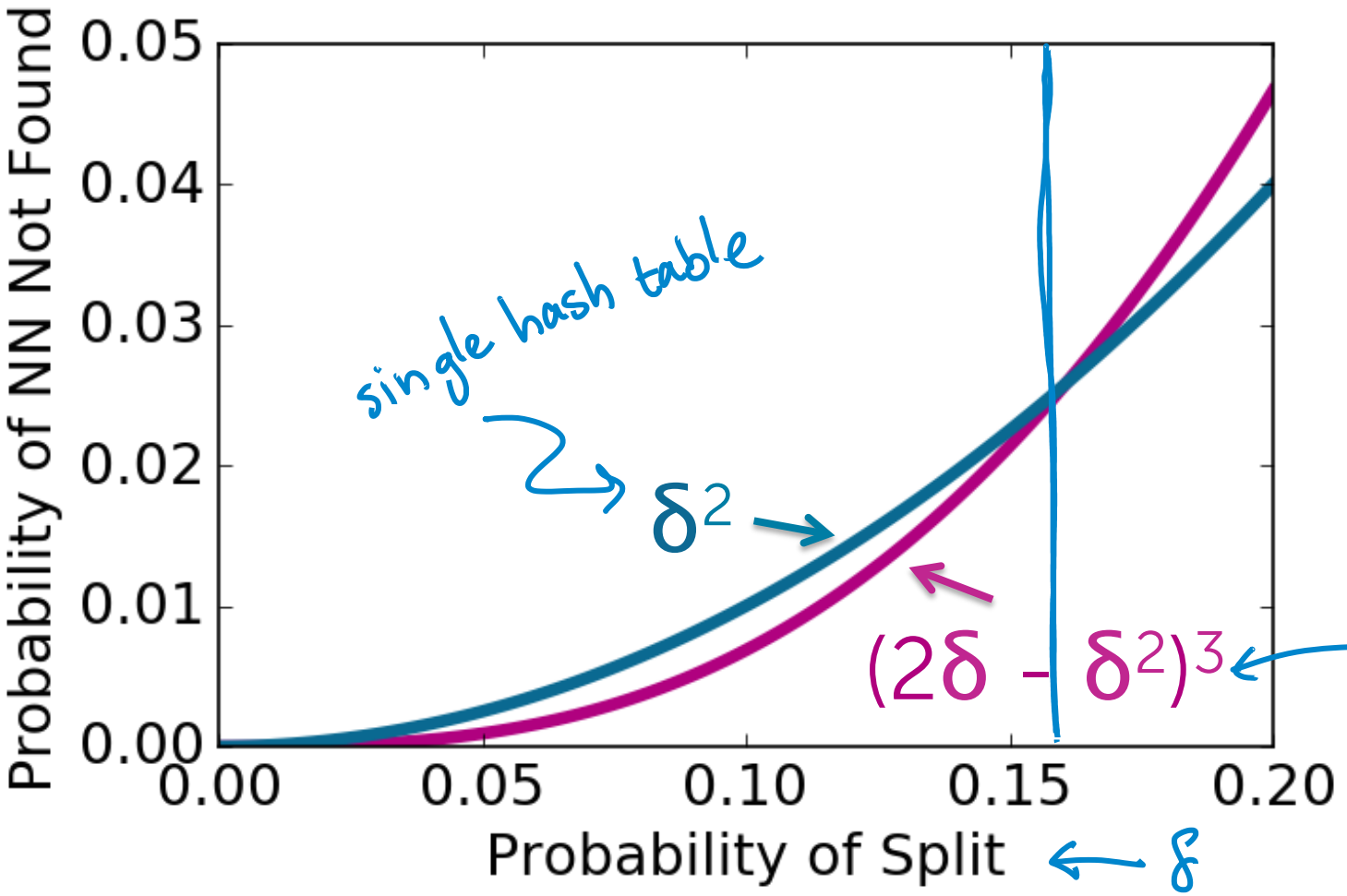


# bins searched	prob. of no NN
3	δ^2



3	$(2\delta - \delta^2)^3$

Comparing probabilities



If I throw down h lines…

Bin	[0 0 0] = 0	[0 0 1] = 1	[0 1 0] = 2	[0 1 1] = 3	[1 0 0] = 4	[1 0 1] = 5	[1 1 0] = 6	[1 1 1] = 7
indices	L0	L1	L2	L3	L4	L5	L6	L7

Still assume we **search bins 1 bit off** from query

Prob. of being > 1 bit away

= 1-Pr(same bin)-Pr(1 bin away)

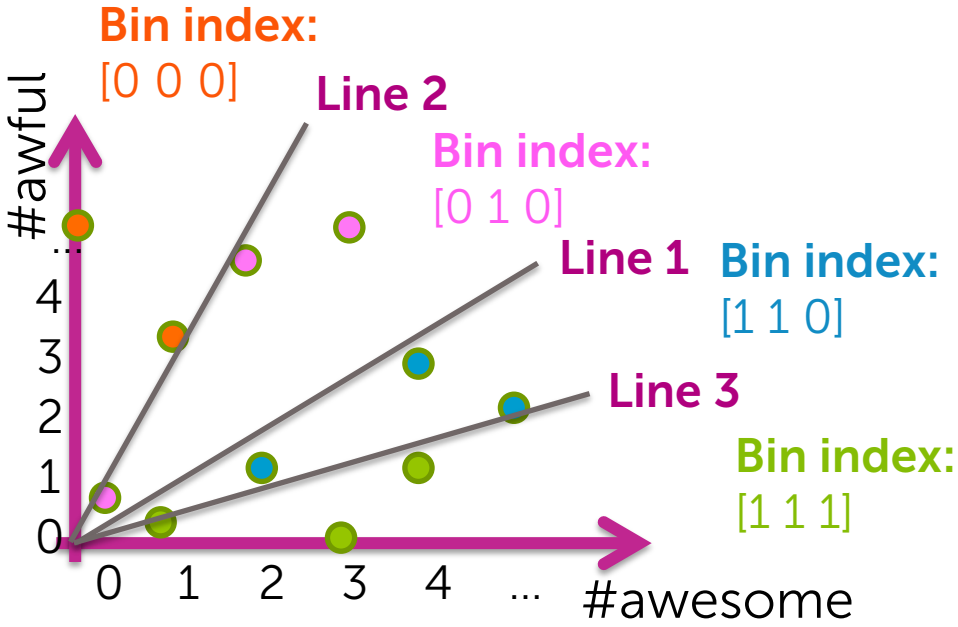
= 1-Pr(no split lines)-Pr(1 split line)

= $1-(1-\delta)^h - h\delta(1-\delta)^{h-1}$

Prob. of no split is $1-\delta$ + we have h lines ⇒ Prob. none of h lines split is $(1-\delta)^h$

Prob. of 1 line splitting

prob. of $h-1$ lines not splitting



If I throw down h lines…

Bin	[0 0 0] = 0	[0 0 1] = 1	[0 1 0] = 2	[0 1 1] = 3	[1 0 0] = 4	[1 0 1] = 5	[1 1 0] = 6	[1 1 1] = 7
indices	L0	L1	L2	L3	L4	L5	L6	L7

Still assume we **search bins 1 bit off** from query

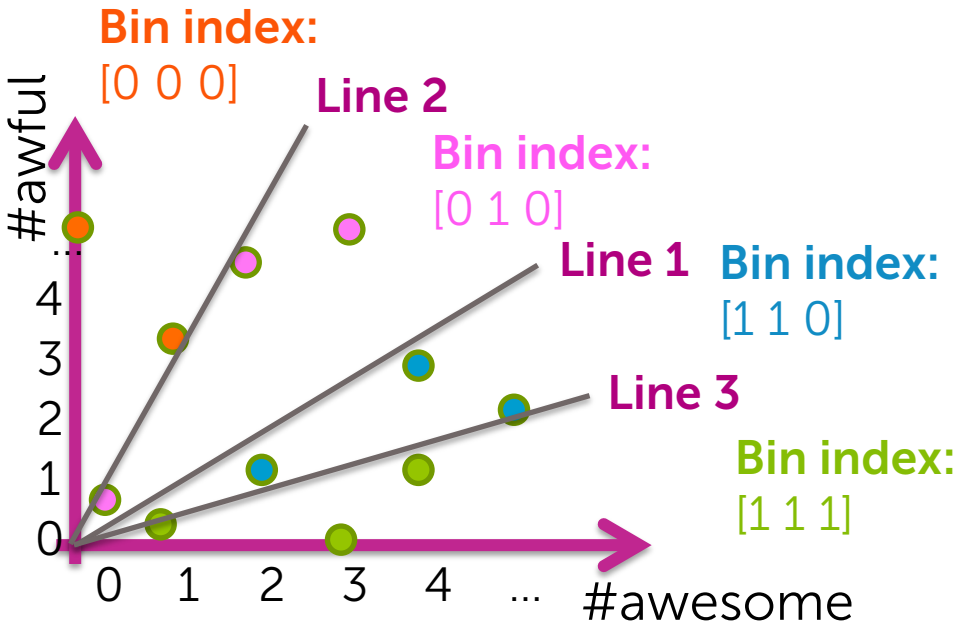
Prob. of being > 1 bit away

= 1-Pr(same bin)-Pr(1 bin away)

= 1-Pr(no split lines)-Pr(1 split line)

= $1-(1-\delta)^h-h\delta(1-\delta)^{h-1}$

Search h+1 bins and do not find NN with probability $1-(1-\delta)^h-h\delta(1-\delta)^{h-1}$



Probability of splitting neighboring points many times

Bin	[0 0 0] = 0	[0 0 1] = 1	[0 1 0] = 2	[0 1 1] = 3	[1 0 0] = 4	[1 0 1] = 5	[1 1 0] = 6	[1 1 1] = 7
indices	L0	L1	L2	L3	L4	L5	L6	L7

Bin	[0 0 0] = 0	[0 0 1] = 1	[0 1 0] = 2	[0 1 1] = 3	[1 0 0] = 4	[1 0 1] = 5	[1 1 0] = 6	[1 1 1] = 7
indices	L0	L1	L2	L3	L4	L5	L6	L7

Bin	[0 0 0] = 0	[0 0 1] = 1	[0 1 0] = 2	[0 1 1] = 3	[1 0 0] = 4	[1 0 1] = 5	[1 1 0] = 6	[1 1 1] = 7
indices	L0	L1	L2	L3	L4	L5	L6	L7

Bin	[0 0 0] = 0	[0 0 1] = 1	[0 1 0] = 2	[0 1 1] = 3	[1 0 0] = 4	[1 0 1] = 5	[1 1 0] = 6	[1 1 1] = 7
indices	L0	L1	L2	L3	L4	L5	L6	L7

Probability NN is in different bin in **all h+1 tables**

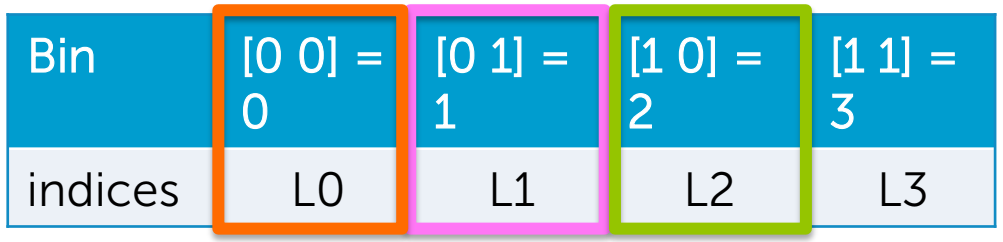
$$= (1-\Pr(\text{same bin}))^{h+1}$$
$$= (1-\Pr(\text{no split line}))^{h+1}$$
$$= (1-(1-\delta)^h)^{h+1}$$

holds for all hash tables

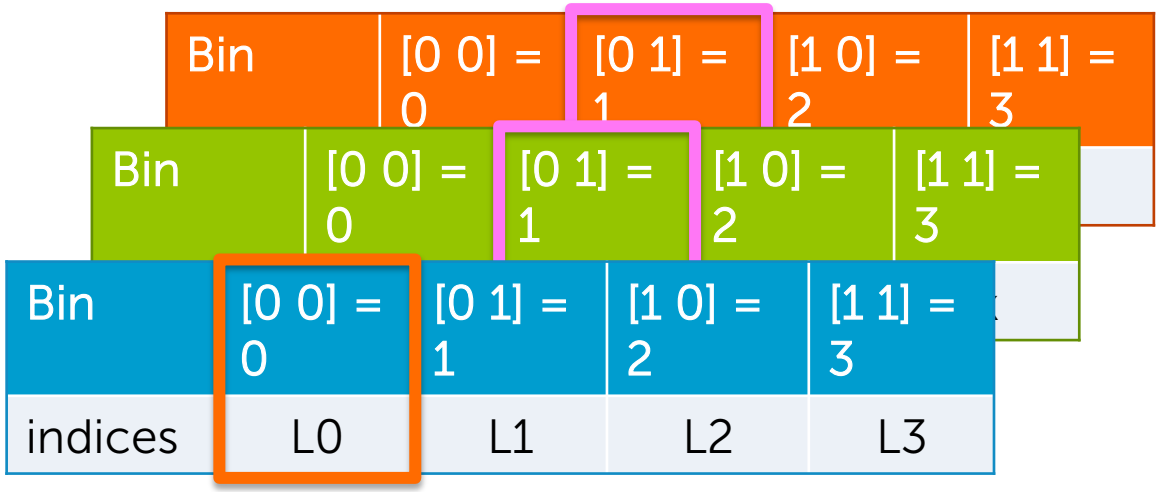
prob. of no split from any of h lines thrown down

Comparing approaches for h-bit tables

throw h down lines

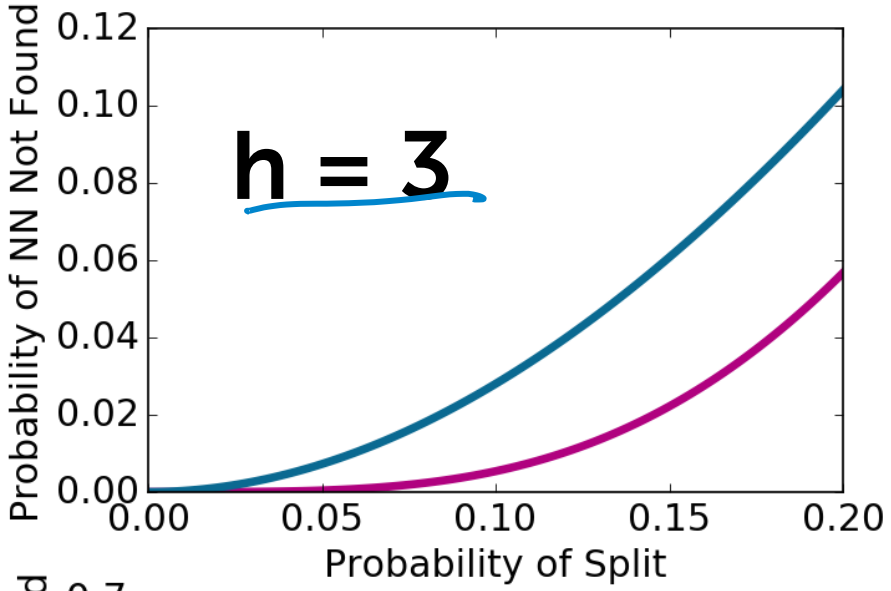


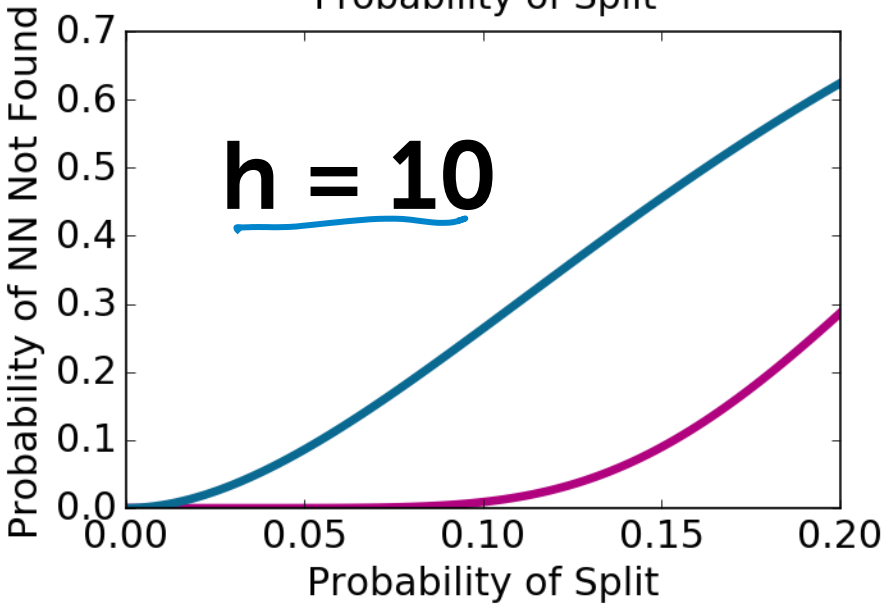
# bins searched	prob. of no NN
h+1	$1-(1-\delta)^h-h\delta(1-\delta)^{h-1}$



h+1	$(1-(1-\delta)^h)^{h+1}$

Comparing probabilities





one hash table

$1-(1-\delta)^h-h\delta(1-\delta)^{h-1}$

$(1-(1-\delta)^h)^{h+1}$

multiple hash table

Fix #bits and increase depth

Bin	[0 0 0] = 0	[0 0 1] = 1	[0 1 0] = 2	[0 1 1] = 3	[1 0 0] = 4	[1 0 1] = 5	[1 1 0] = 6	[1 1 1] = 7
indices	L0	L1	L2	L3	L4	L5	L6	L7

Bin	[0 0 0] = 0	[0 0 1] = 1	[0 1 0] = 2	[0 1 1] = 3	[1 0 0] = 4	[1 0 1] = 5	[1 1 0] = 6	[1 1 1] = 7
indices	L0	L1	L2	L3	L4	L5	L6	L7

Bin	[0 0 0] = 0	[0 0 1] = 1	[0 1 0] = 2	[0 1 1] = 3	[1 0 0] = 4	[1 0 1] = 5	[1 1 0] = 6	[1 1 1] = 7
indices	L0	L1	L2	L3	L4	L5	L6	L7

Bin	[0 0 0] = 0	[0 0 1] = 1	[0 1 0] = 2	[0 1 1] = 3	[1 0 0] = 4	[1 0 1] = 5	[1 1 0] = 6	[1 1 1] = 7
indices	L0	L1	L2	L3	L4	L5	L6	L7

Bin	[0 0 0] = 0	[0 0 1] = 1	[0 1 0] = 2	[0 1 1] = 3	[1 0 0] = 4	[1 0 1] = 5	[1 1 0] = 6	[1 1 1] = 7
indices	L0	L1	L2	L3	L4	L5	L6	L7

Probability NN is in different bin in **all tables** falls off exponentially fast

hash tables

Prob

$= (1\text{-Pr(same bin)})^m$

$= (1\text{-Pr(no split line)})^m$

$= (1-(1-\delta)^h)^m$

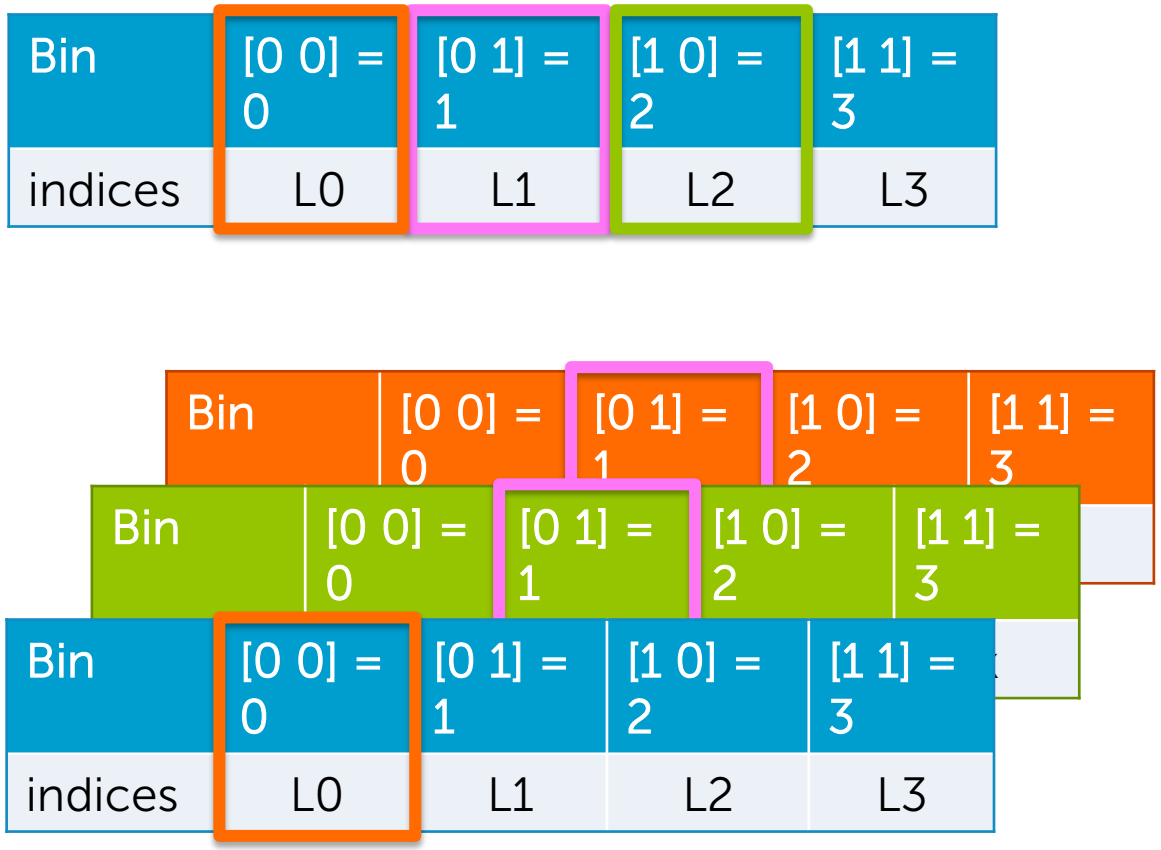
# of lines (bits)

Fix #bits and increase depth

Bin	[0 0 0] = 0	[0 0 1] = 1	[0 1 0] = 2	[0 1 1] = 3	[1 0 0] = 4	[1 0 1] = 5	[1 1 0] = 6	[1 1 1] = 7
indices	L0	L1	L2	L3	L4	L5	L6	L7

Typically higher probability of finding NN than searching m bins in 1 table

Summary of LSH approaches



Cost of binning points is **lower**, but likely need to search **more bins** per query

Cost of binning points is **higher**, but likely need to search **fewer bins** per query

Summary for retrieval using nearest neighbors, KD-trees, and locality sensitive hashing

What you can do now…

- Implement nearest neighbor search for retrieval tasks
- Contrast document representations (e.g., raw word counts, tf-idf,…)
  - Emphasize important words using tf-idf
- Contrast methods for measuring similarity between two documents
  - Euclidean vs. weighted Euclidean
  - Cosine similarity vs. similarity via unnormalized inner product
- Describe complexity of brute force search
- Implement KD-trees for nearest neighbor search
- Implement LSH for approximate nearest neighbor search
- Compare pros and cons of KD-trees and LSH, and decide which is more appropriate for given dataset