

Nearest Neighbor Search

Sewoong Oh

CSE/STAT 416

University of Washington

Machine Learning

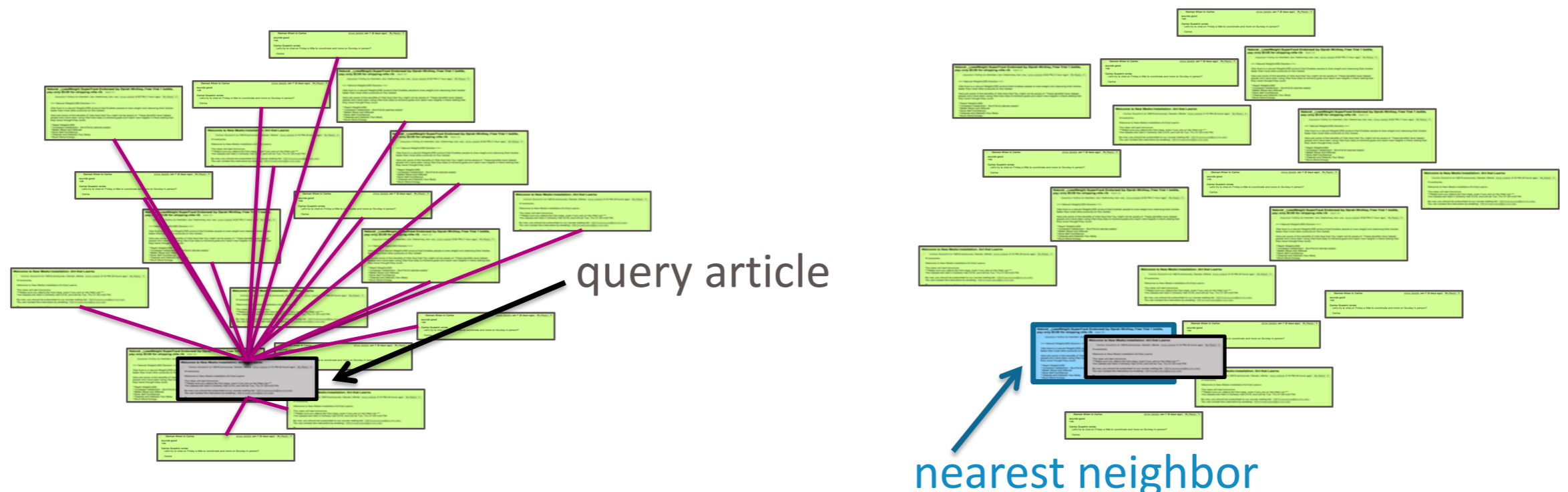
- Supervised Learning: predict y from x
 - Regression
 - Given training data $\{(x,y)\}$, with continuous y , learn $f(x) \sim y$
 - Classification
 - Given training data $\{(x,y)\}$, with categorical y , learn $f(x) \sim y$
- Unsupervised Learning: find pattern in x
 - Clustering
 - Given training data $\{x\}$, group similar data points together
 - Recommendation systems/Collaborative filtering
 - Given training data $\{x\}$ with missing values, fill in the missing values
- Idea: build a model of the data to solve these tasks

Embedding

- The raw data u is embedded into a feature vector x
- We build a data model for the feature vectors
- We un-embed when needed, to go back to raw u

Example: retrieving document

- Consider a scenario where you read a book, and want to find a book similar to it (Amazon has to solve such tasks)
- Challenges:
 - How should one measure similarity?
 - There are so many books (data points)
- **Nearest neighbor search** is used as a black-box tool for many tasks in regression, classification, clustering, and recommendation systems
 - We want to organize all books by similarity of the content such that it is easy to find nearest neighbors



Nearest Neighbor methods

1-Nearest Neighbor

- Input:
 - x_q : query example
 - x_1, \dots, x_N : Corpus of examples
- Output: Most similar example

- Find

$$x^{nn} = \arg \min_{i \in [N]} \text{distance}(x_q, x_i)$$

$$x^{nn} \in \arg \min_{i \in [N]} \text{distance}(x_q, x_i)$$

where $[N] = \{1, 2, \dots, N\}$ is the set of first N integers

- Proper definition of a **distance** function is critical
which is related to proper **representation** of the document

Run time

- How long does it take to find 1-NN (one nearest neighbor)?
 - N operations are necessary
- Pseudo code
 - Initialize Distance2NN=infinity
 - Initialize NN=0
 - For $i=1,2,\dots,N$
 - $D \leftarrow \text{distance}(x_q, x_i)$
 - If $d < \text{Distance2NN}$
 - $\text{NN} \leftarrow i$
 - $\text{Distance2NN} \leftarrow d$

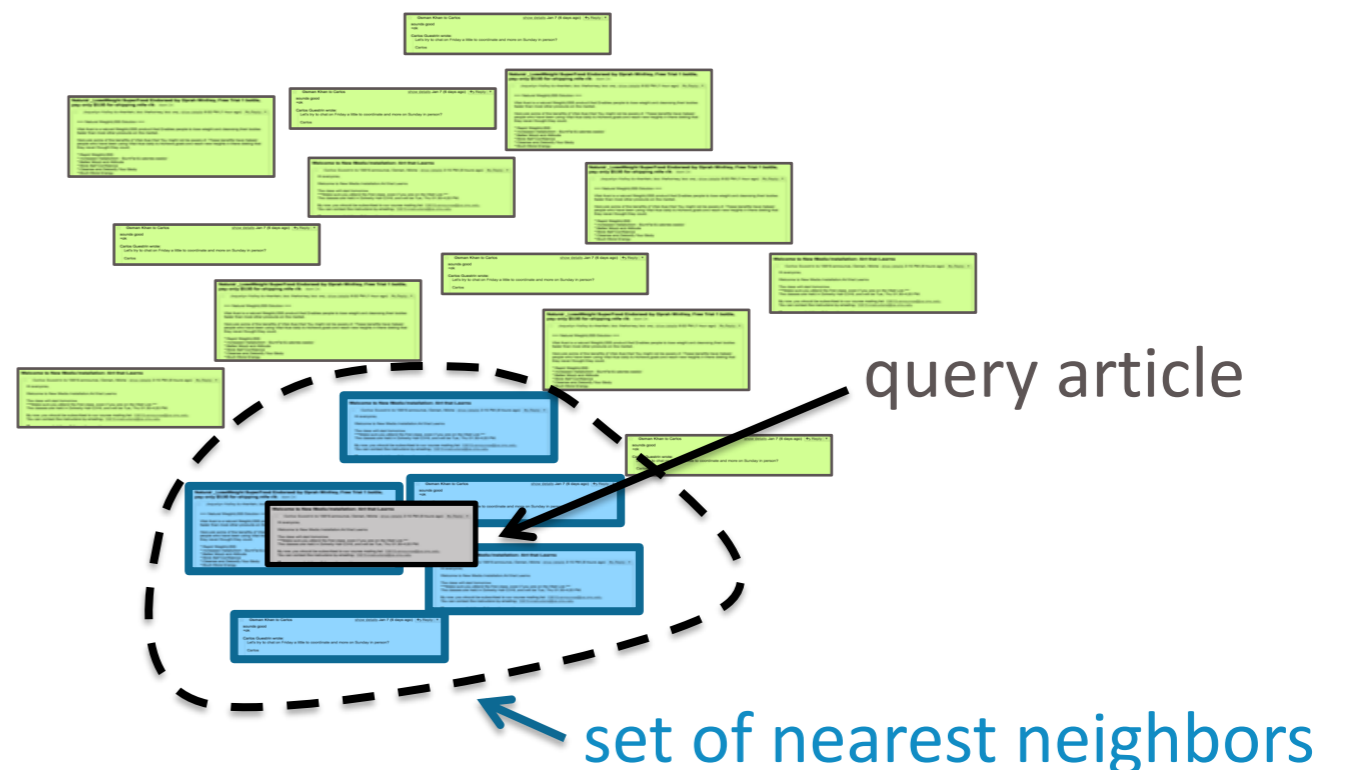
K-nearest neighbor

- Find a list of k similar examples

$$x^{nn} = \{x_1^{nn}, \dots, x_k^{nn}\} \text{ such that}$$

$$\max_{x_i \in x^{nn}} \text{distance}(x_q, x_i) \leq \min_{x_j \text{ not in } x^{nn}} \text{distance}(x_q, x_j)$$

- This takes kN operations
- Initialize
 - $\text{Dist2kNN} \leftarrow$
 - $\text{kNN} \leftarrow$
- If



Representation and distance (similarity)

Representing a Document as a vector

- **Bag-of-words (BoG)**
 - Ignore order of words
 - Count the number of occurrences
 - Size of the vector = number of words in a vocabulary
e.g. 171,476 for English, 370,000 for Chinese
- Main challenge
 - Common words like “the” or “have” dominate
Uncommon words
 - Oftentimes the uncommon words are more relevant to
figuring out which documents are similar

TF-IDF representation of documents

Emphasizes **important words**

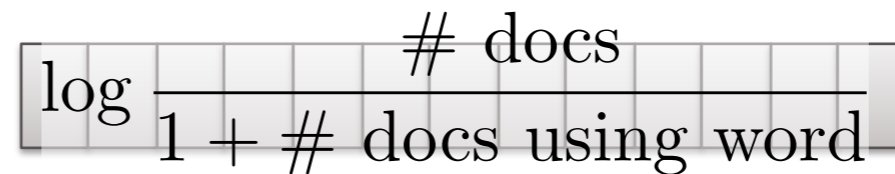
- Appears frequently in document (**common locally**)

Term frequency =



- Appears rarely in corpus (**rare globally**)

Inverse doc freq. =



tf * idf

- TF captures how common that word is in that document
- IDF measures how rare that word is in all corpus
 - For example,
 - “the” would have IDF~0
 - “Euclidean” would have IDF large
 - (use of) rare words distinguishes the document from others
- TF-IDF score of a document is the point-wise multiplication of TF and IDF

Distance

- 1-dimensional case (any distance is a monotonic function of this)

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

- multi-dimensional case

$$\text{distance}(x_q, x_i) = d(x_q[1], x_i[1]) + \dots + d(x_q[D], x_i[D])$$

- Euclidean distance (L2 distance)

$$\text{distance}(x_q, x_i) = \sqrt{(x_q[1] - x_i[1])^2 + \dots + (x_q[D] - x_i[D])^2}$$

- Manhattan distance (L1 distance)

$$\text{distance}(x_q, x_i) = |x_q[1] - x_i[1]| + \dots + |x_q[D] - x_i[D]|$$

- Hamming

$$\text{distance}(x_q, x_i) = \mathbb{I}(x_q[1] \neq x_i[1]) + \dots + \mathbb{I}(x_q[D] \neq x_i[D])$$

- Challenge in practice:

- What unit we use for each feature, changes the distance dramatically

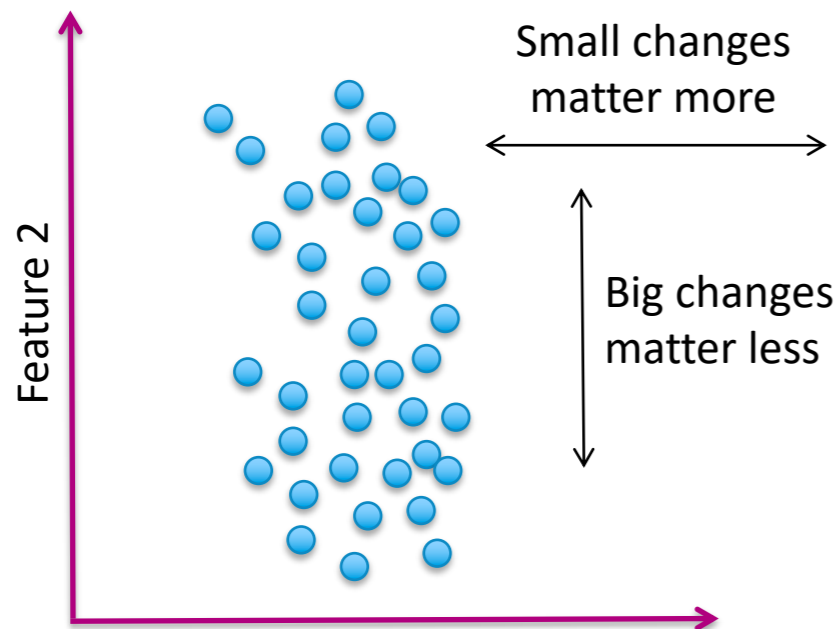
Weighted distances

- Some features vary more than others
 - Changing units should not change the distance



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

- One should weight each dimension differently



Specify weights as
a function of
feature spread


For feature j :
$$\frac{1}{\max_i(x_i[j]) - \min_i(x_i[j])} = a_j$$

- Weighted Euclidean distance
(this is coordinate-wise homogeneous)

$$\text{distance}(x_q, x_i) = \sqrt{a_1(x_q[1] - x_i[1])^2 + \dots + a_D(x_q[D] - x_i[D])^2}$$

Cosine similarity

- How strongly correlated are the two vectors?
 - Related to inner product, correlation,...




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


Similarity

$$= x_i^T x_q$$

$$= \sum_{j=1}^d x_i[j] x_q[j]$$

= 13






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

Similarity

= 0

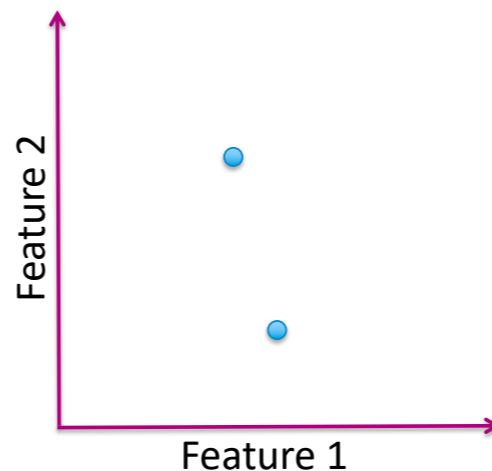
0	0	1	0	0	0	9	0	0	6	0	4	0
---	---	---	---	---	---	---	---	---	---	---	---	---



- Cosine similarity normalizes each vector

$$\text{similarity}(x_q, x_i) = \frac{x_q[1]x_i[1] + \dots + x_q[D]x_i[D]}{\sqrt{x_q[1]^2 + \dots + x_q[D]^2} \sqrt{x_i[1]^2 + \dots + x_i[D]^2}}$$

- Is this homogeneous?



Should we normalize or not?

- (unnormalized) similarity

$$\text{similarity}(x_q, x_i) = x_q[1]x_i[1] + \dots + x_q[D]x_i[D]$$

- Cosine similarity

$$\text{similarity}(x_q, x_i) = \frac{x_q[1]x_i[1] + \dots + x_q[D]x_i[D]}{\sqrt{x_q[1]^2 + \dots + x_q[D]^2} \sqrt{x_i[1]^2 + \dots + x_i[D]^2}}$$

- Normalization helps when documents of different size



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



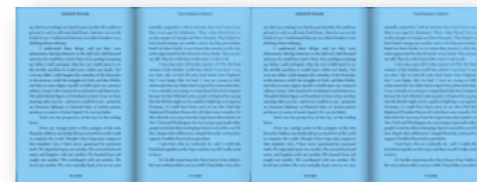
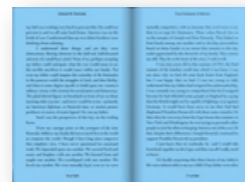
2 0 0 0 10 6 0 0 2 0 0 0 0

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



Cosine Similarity = 13/24

- Normalization undesired when comparing documents of different sizes



long document

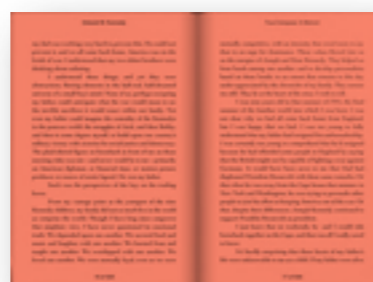


short tweet

Normalizing can
make dissimilar
objects appear more
similar



long document



long document

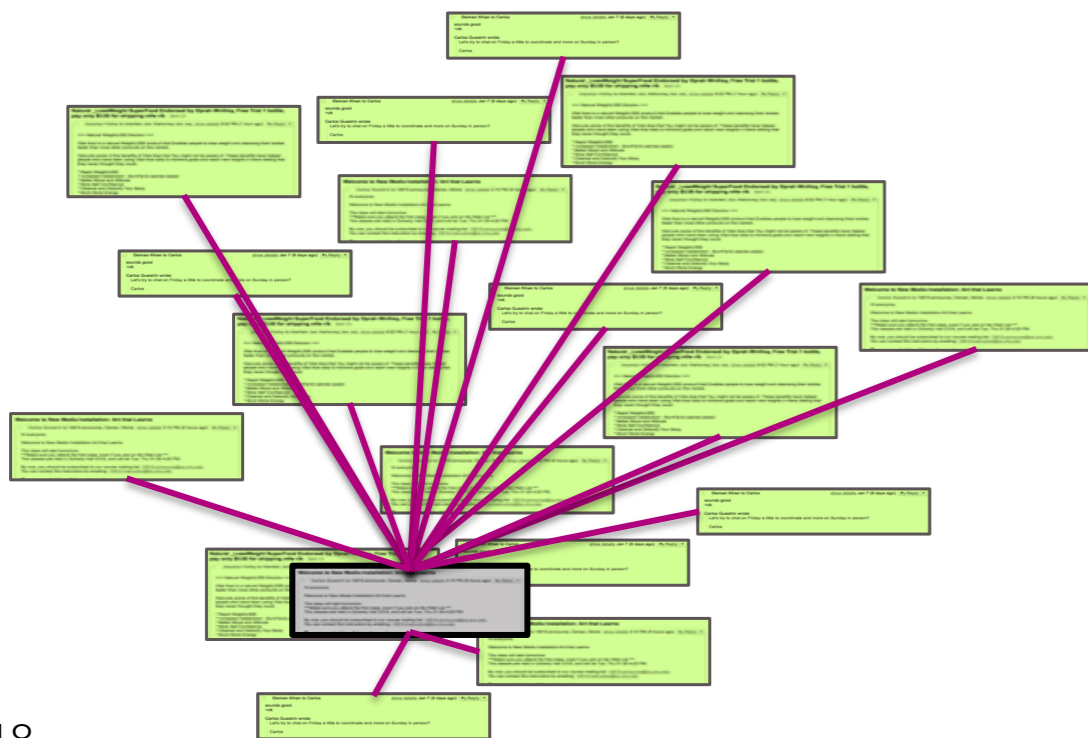
Common
compromise:
Just cap maximum
word counts

- In practice use multiple distance metrics

Locality sensitive hashing for approximate nearest-neighbor search

Finding exact nearest neighbor is computationally expensive

- Nearest neighbor search is one of the fundamental tasks in unsupervised learning, whose goal is to learn the patterns from the data
- Brute force search for nearest neighbors can be slow in the worst case
- Finding one-nearest-neighbor can take number of operations proportional to N , the number of examples
- Typical range of N can be larger than millions (number of books, number of webpages, number of images, etc.)

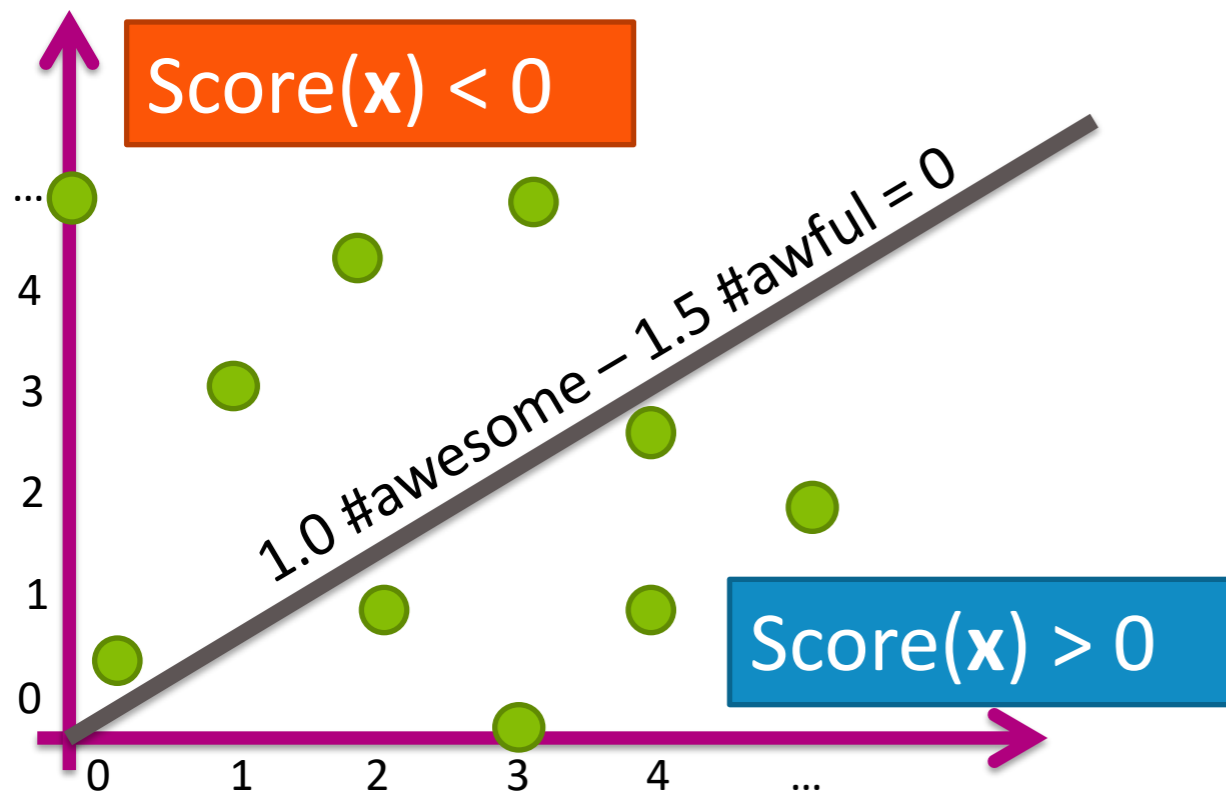


A breakthrough: locality sensitive hashing

- Main idea
 - Move away from exact nearest neighbor search
 - Find approximate nearest neighbors
 - Several applications are fine with “close enough neighbors”
 - The measure of similarity is not perfect
 - In practice, we are searching for a book recommendation from millions of books
- Approach
 - Design methods that have higher chance of finding neighbors that are closer
 - And provide probabilistic guarantees
 - We will skip the precise probabilistic guarantees, but learn the main techniques

Main component of locality sensitive hashing: binning

- Simple **binning** of data into 2 bins
- Consider examples in 2-dimensions $\mathbf{x}=(\mathbf{x}[1],\mathbf{x}[2])$
- In this example, the features are how many times the words **awesome** or **awful** occurred in a sentence
- Consider a linear score of the form $\text{score}(\mathbf{x}) = w_0 + w_1\mathbf{x}[1] + w_2\mathbf{x}[2]$



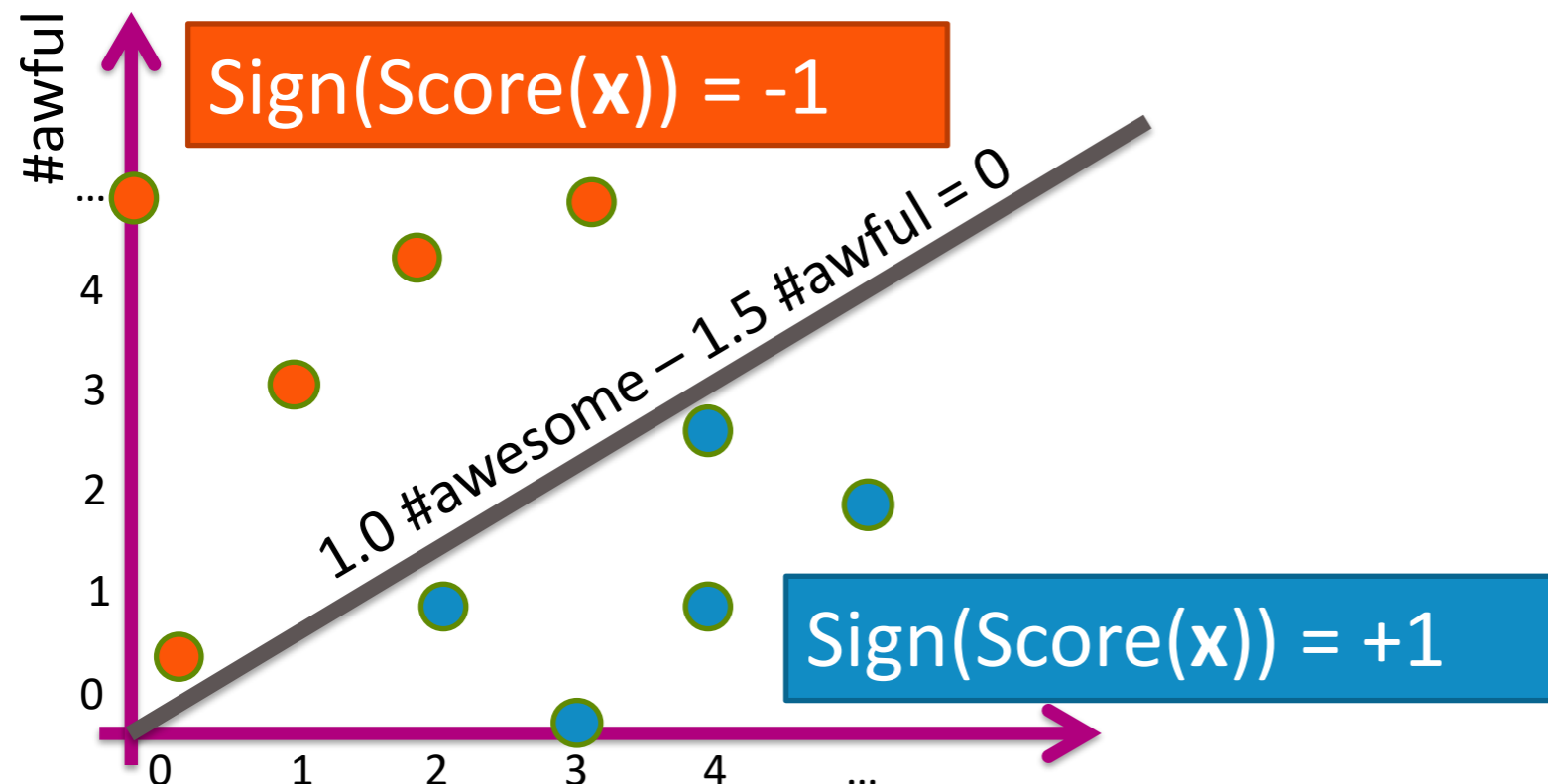
$$\text{Score}(\mathbf{x}) = 1.0 \# \text{awesome} - 1.5 \# \text{awful}$$

- It looks similar to classification
- But there is no training (the examples are unlabelled)
- The linear model above is manually chosen (for now)

Main component of locality sensitive hashing: binning

- Then take the sign of the score
- That divides the examples into two groups (or **bins**)

2D Data	Sign(Score)
$x_1 = [0, 5]$	-1
$x_2 = [1, 3]$	-1
$x_3 = [3, 0]$	1
...	...

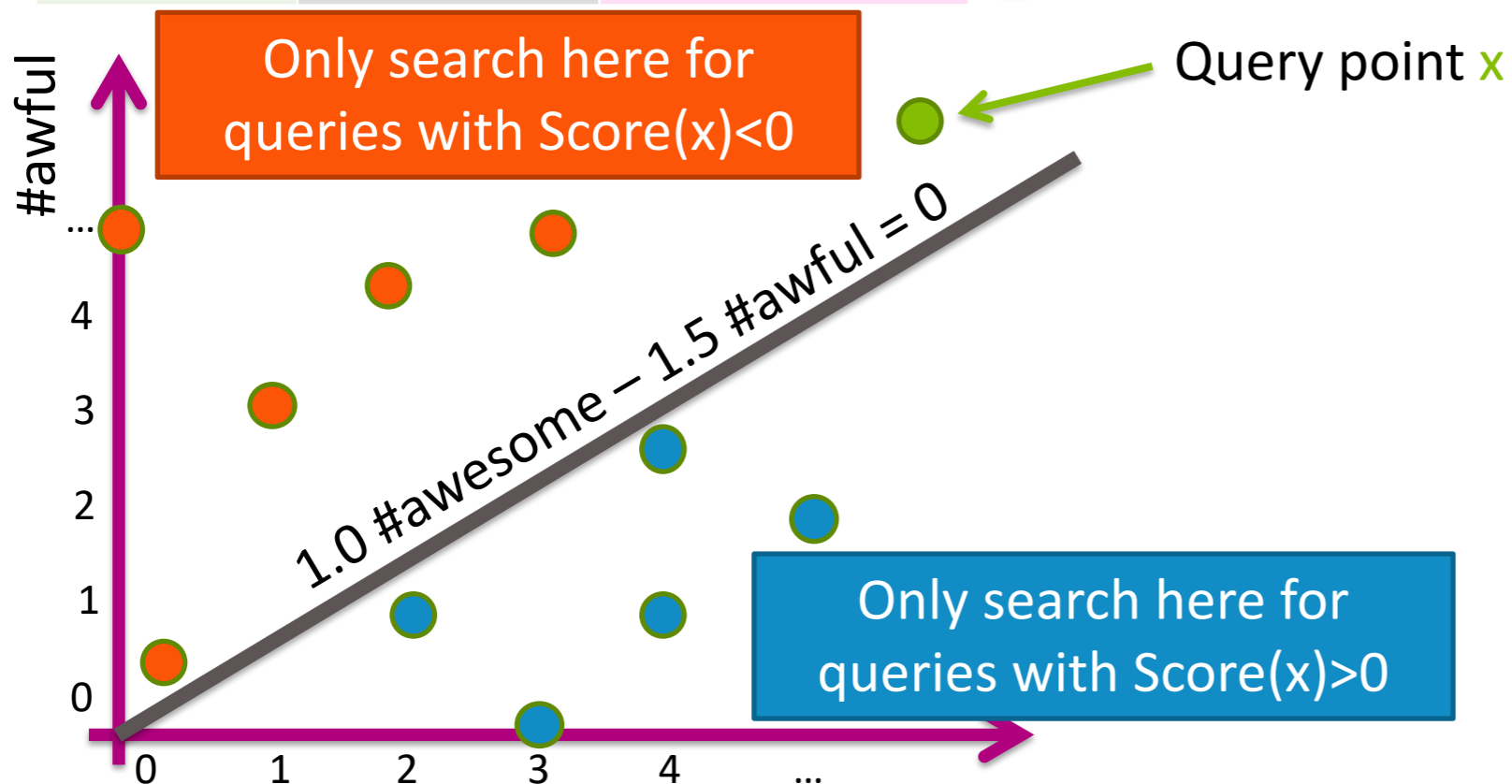


Main component of locality sensitive hashing: binning

- Then take the sign of the score
- That divides the examples into two groups (or **bins**)
- We will call one bin 0 and another 1 (this is referred to as bin index)

2D Data	Sign(Score)	Bin index
$x_1 = [0, 5]$	-1	0
$x_2 = [1, 3]$	-1	0
$x_3 = [3, 0]$	1	1
...

candidate neighbors if $\text{Score}(x) < 0$



- When asked to find a nearest neighbor of a query point, search for a nearest neighbor in the same **bin**
- This reduces the search space into $N/2$

Nearest neighbor search with 2 bins

- Create a table of all data points and the bin index each point belongs to

2D Data	Sign(Score)	Bin index
$x_1 = [0, 5]$	-1	0
$x_2 = [1, 3]$	-1	0
$x_3 = [3, 0]$	1	1
...

- Store it in a Hash table

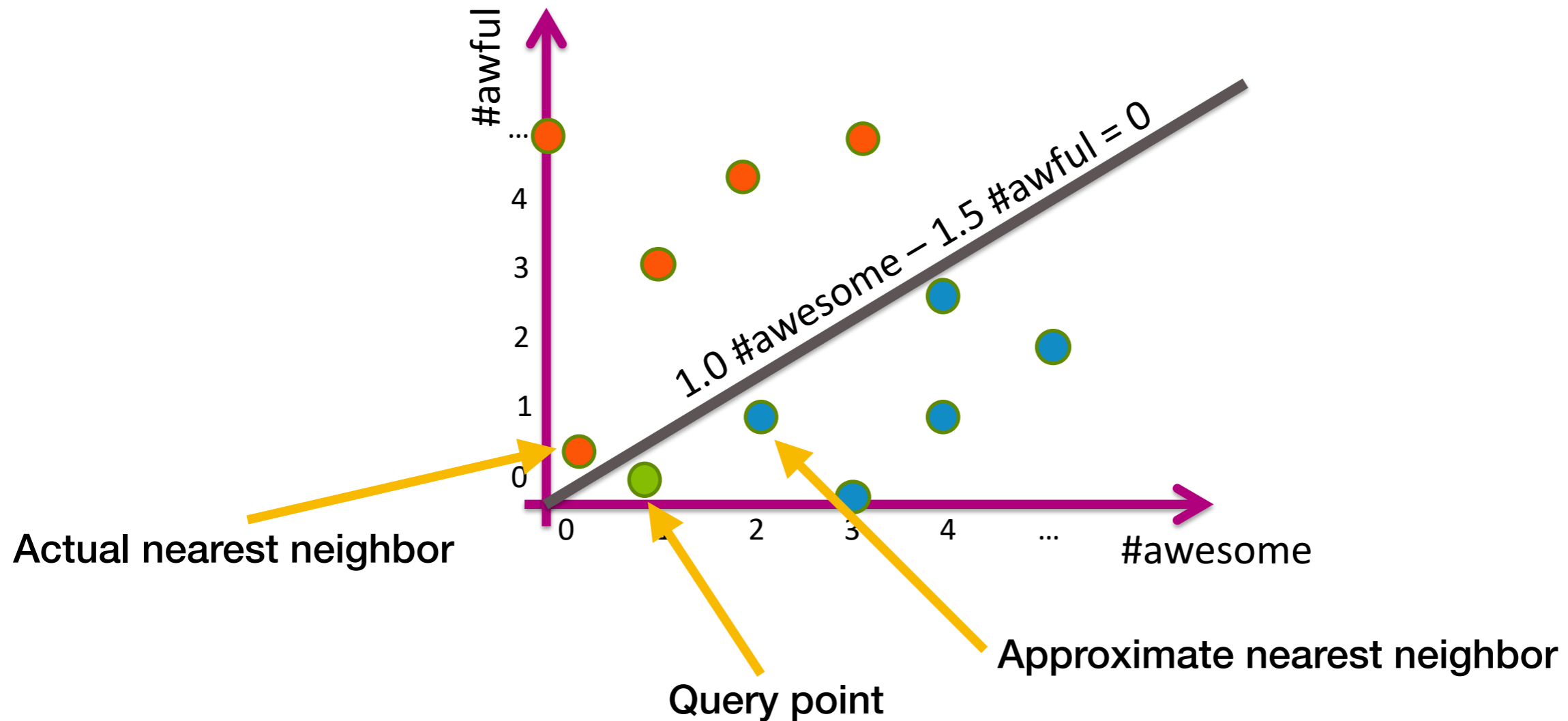
Bin	0	1
List containing indices of datapoints:	{1,2,4,7,...}	{3,5,6,8,...}

HASH
TABLE

- When searching for a nearest neighbor of a query point
 - Find bin index of the query point
 - Search only over those points in the same bin

Binning can only guarantee approximate nearest neighbor

- Actual nearest neighbor might not be found with 2-bin hashing



Three issues with simple 2 bin hashing

1. Challenging to find a good line

2. Poor quality solution:

–Points close together get split into separate bins

3. Large computational cost:

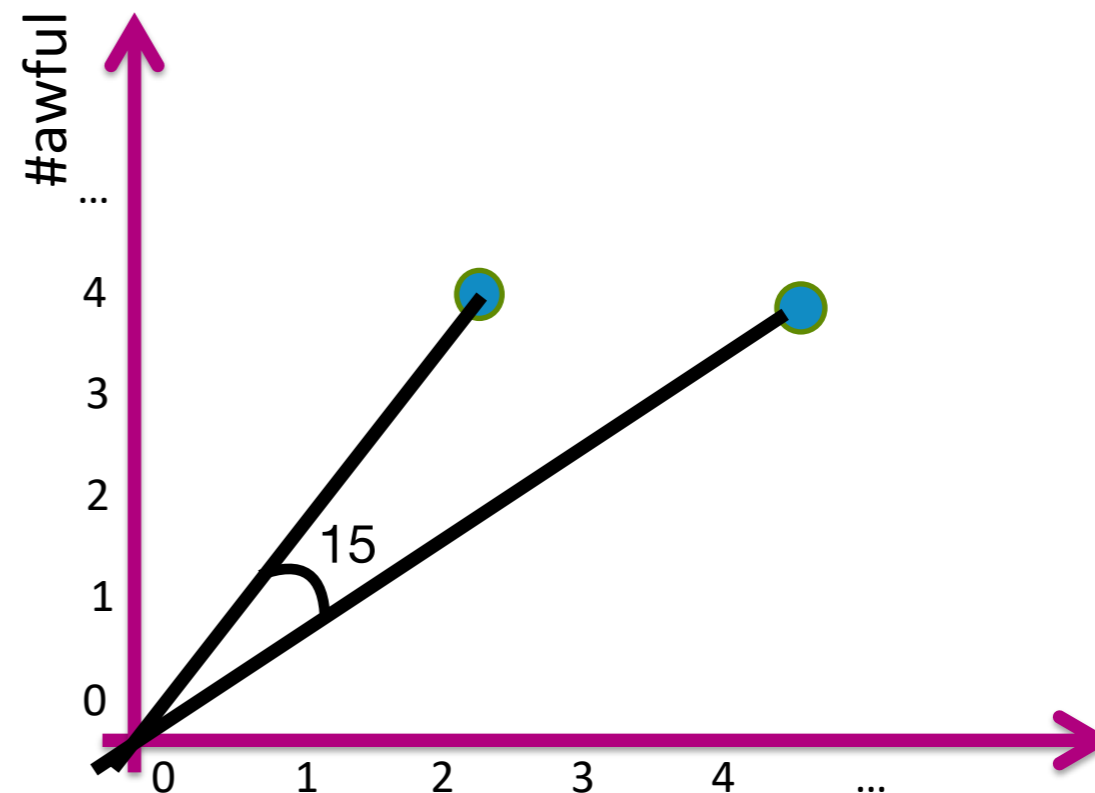
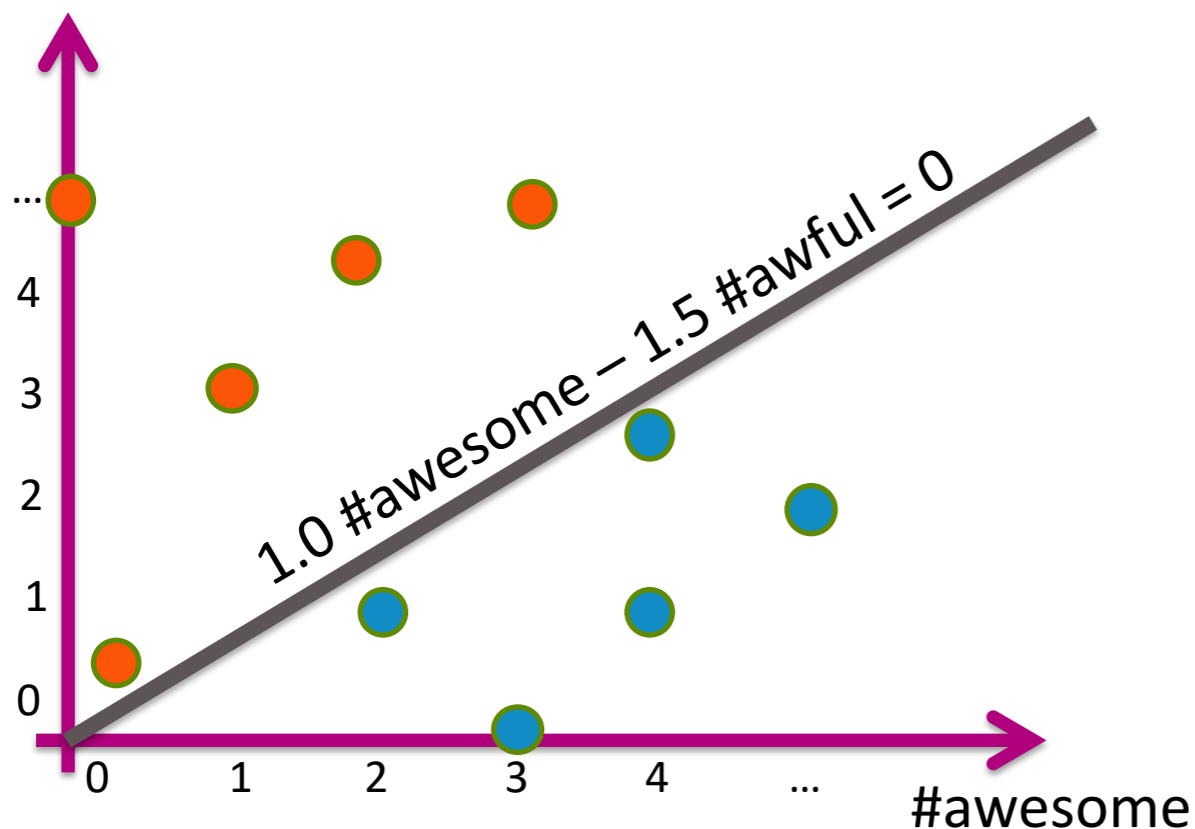
–Bins might contain many points, so still searching over large set for each Nearest Neighbor query

How do you find a good line?

- Crazy idea: choose a line randomly

- before, we were manually choosing a line that we think would work well

- now, we use a randomly chosen line of slope between 0 and 90 degrees
- What is the probability that the line separates the two points?
- Does the probability increase for two points that are closer (in angle)?
- Which similarity metric corresponds to angles?



How bad is a random line?

- Three possible outcomes, for this example with two points

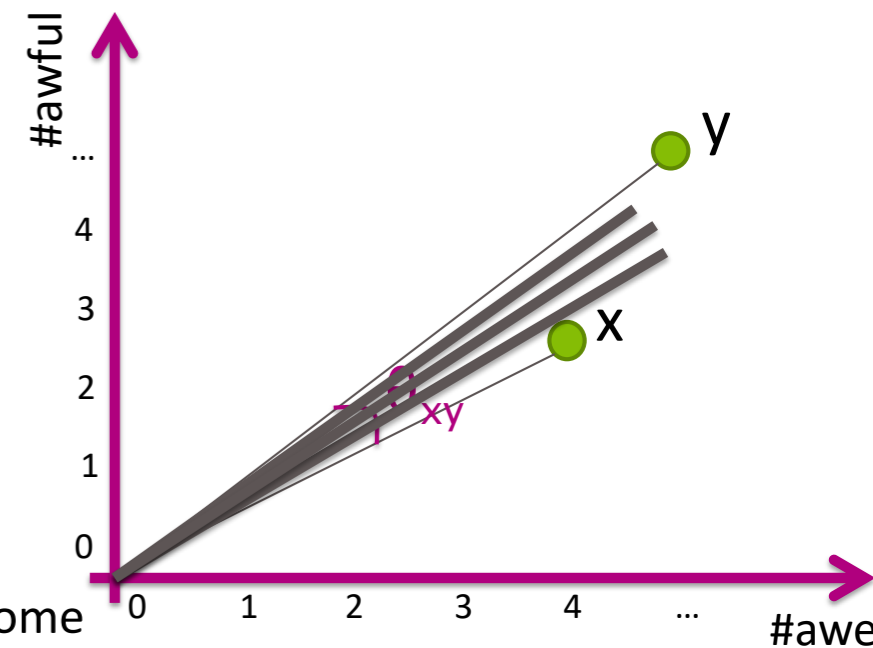
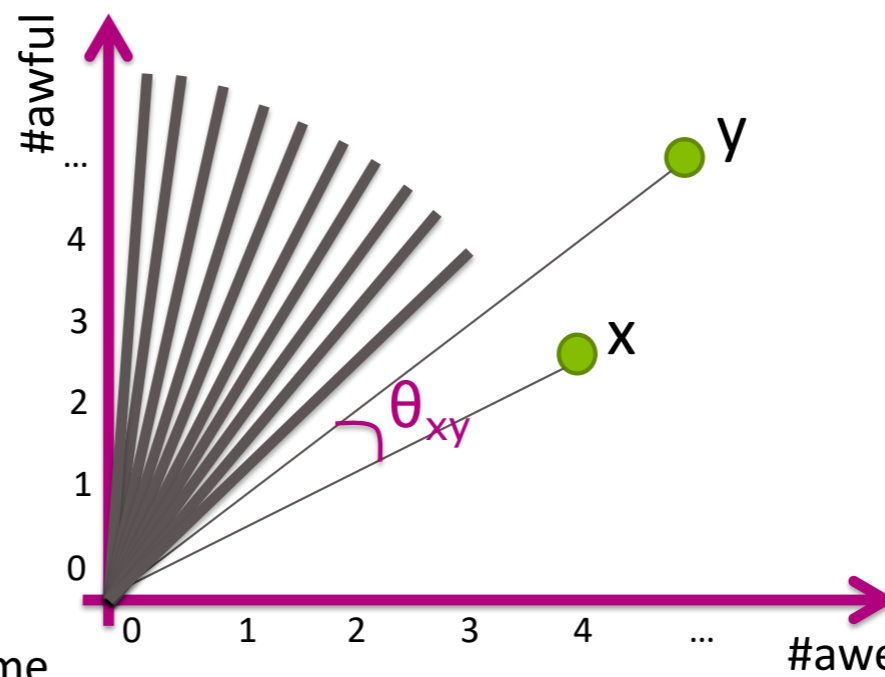
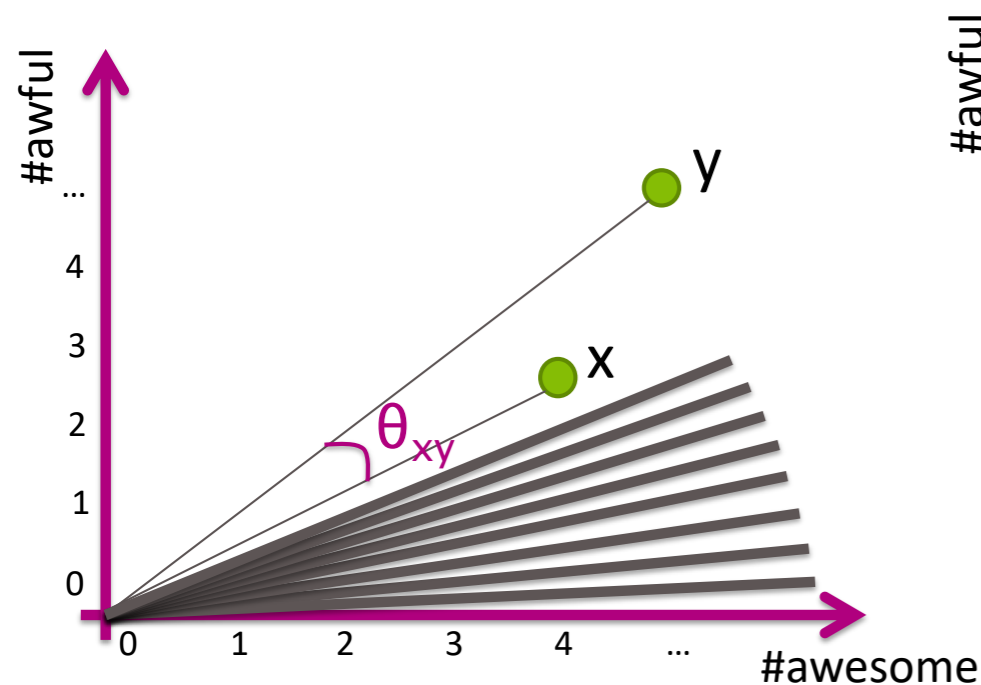
- Both in bin 0

- Both in bin 1

- Two points in separate bins

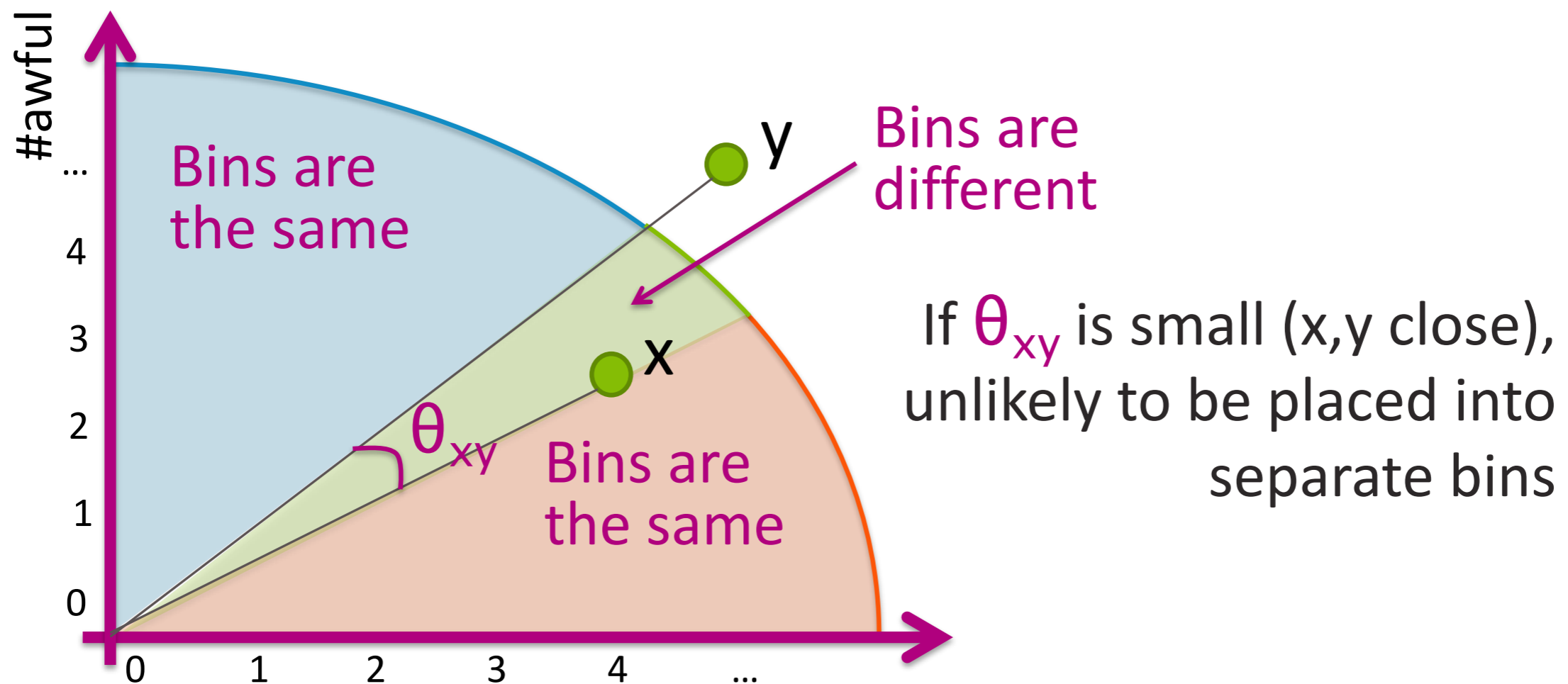
- This happens with probability $\frac{\theta}{\pi/2}$

- Cosine similarity between these two points is $\cos(\theta)$



How bad is a random line?

- The areas represent the probability of the events: both in bin 1, two in different bins, or both in bin 0
- Random lines tend to put similar points (as measured by cosine similarity) in the same bin



- Random lines are pretty good, but using 1 line and 2 bins is problematic

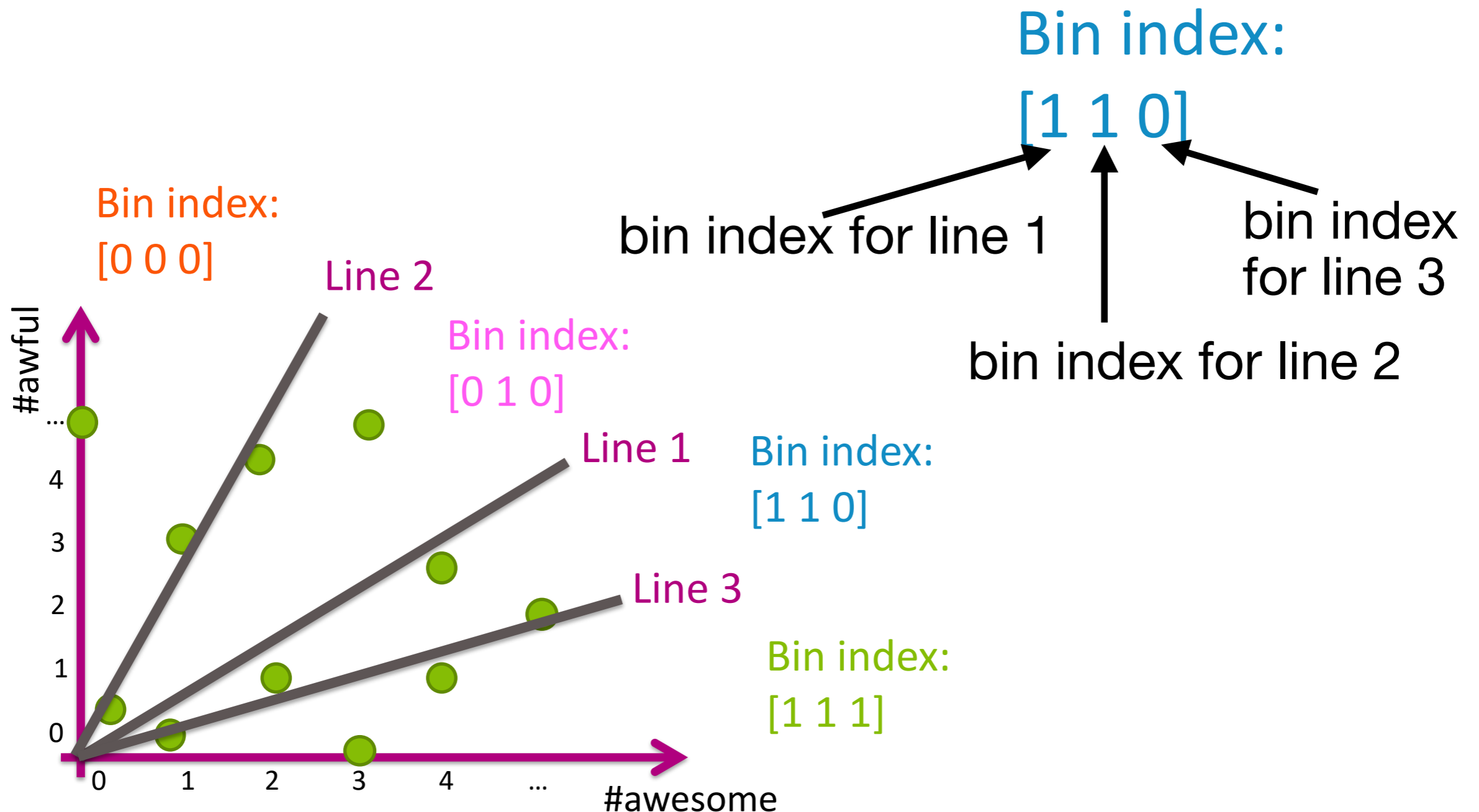
Bin	0	1
List containing indices of datapoints:	{1,2,4,7,...}	{3,5,6,8,...}

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 Nearest Neighbor query

How can we reduce the number of points we need to query to find one-nearest-neighbor of a query point?

Reducing search cost with more bins

- Each line partitions the points into two bins, with its own bin indices 0 or 1
- k bins give k bin indices for each data point



Creating a Hash table

- Use linear scores to find bin index of each point

2D Data	Sign (Score ₁)	Bin 1 index	Sign (Score ₂)	Bin 2 index	Sign (Score ₃)	Bin 3 index
$x_1 = [0, 5]$	-1	0	-1	0	-1	0
$x_2 = [1, 3]$	-1	0	-1	0	-1	0
$x_3 = [3, 0]$	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}

- When finding a nearest neighbor of a query point
 - Find the bin indices of that point
 - Search over only the points in the same bin

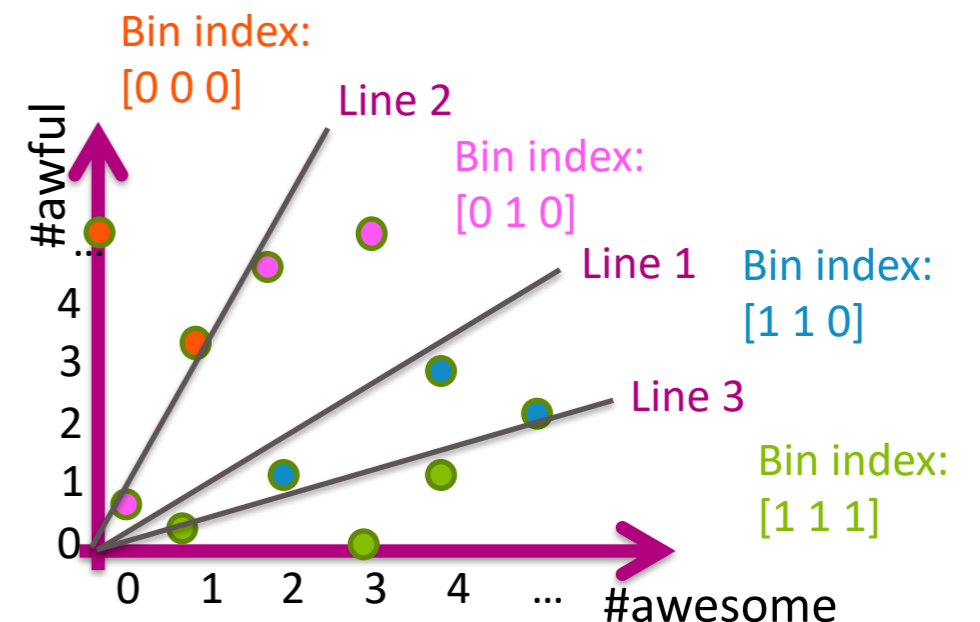
Improving search quality by Searching neighboring bins

- For example, consider the case when you query a point in bin (0,1,0)

Hash table

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}

- By using multiple lines, we have reduced the search space, but the chance of the nearest neighbor being in the same bin is reduced -as there are many lines cutting
- We can include points in a near-by bins, to gracefully trade-off accuracy vs. computation
- But what is a close-by bin?



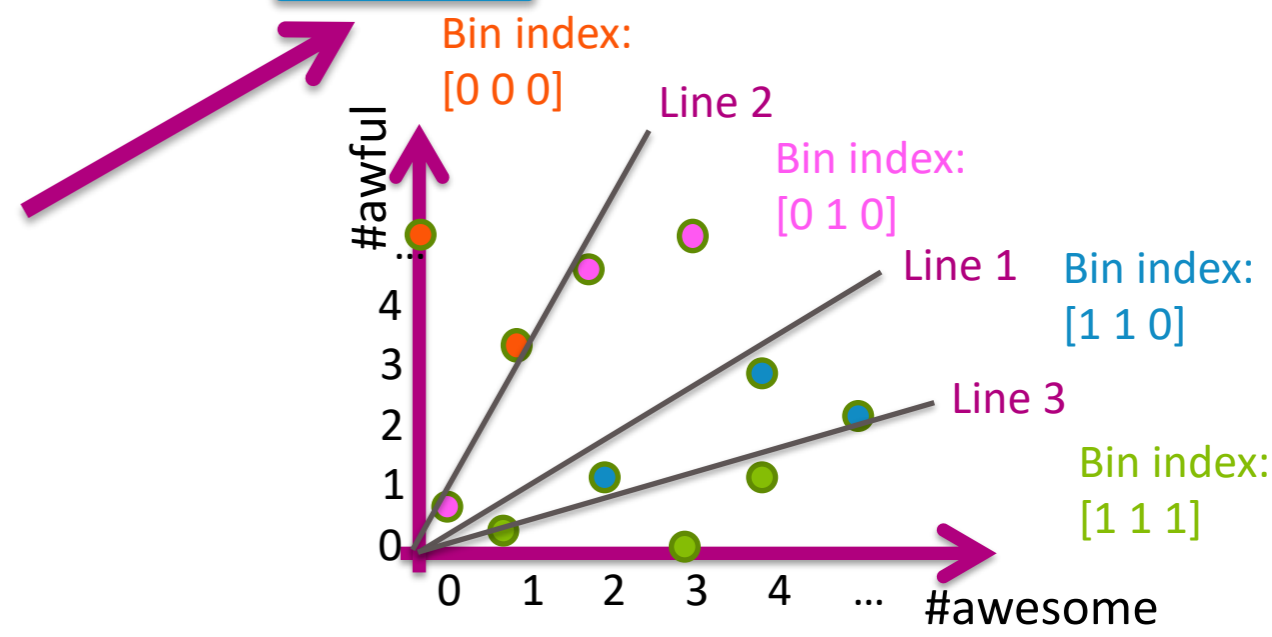
Hamming distance

- Hamming distance between binary vectors
- $x = [0\ 0\ 1\ 0\ 1]$
- $y = [1\ 0\ 1\ 1\ 1]$
- The hamming distance is defined as
$$d_H(x,y) = \# \text{ of coordinates that disagree}$$
$$= 2$$
- The same definition holds for general vectors, not necessarily binary
- $x = [0.1\ 3.1\ 0.5\ 1]$
- $y = [0.\ 3.1\ 1.0\ 0]$
- $d_H(x,y) = 3$

- The closest bin, is the one with smallest flips of 1's (this is called Hamming distance between the bin indices)
- Each flip means that you crossed a (randomly chosen) line
- Hence, small number of flips corresponds to closeness

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)



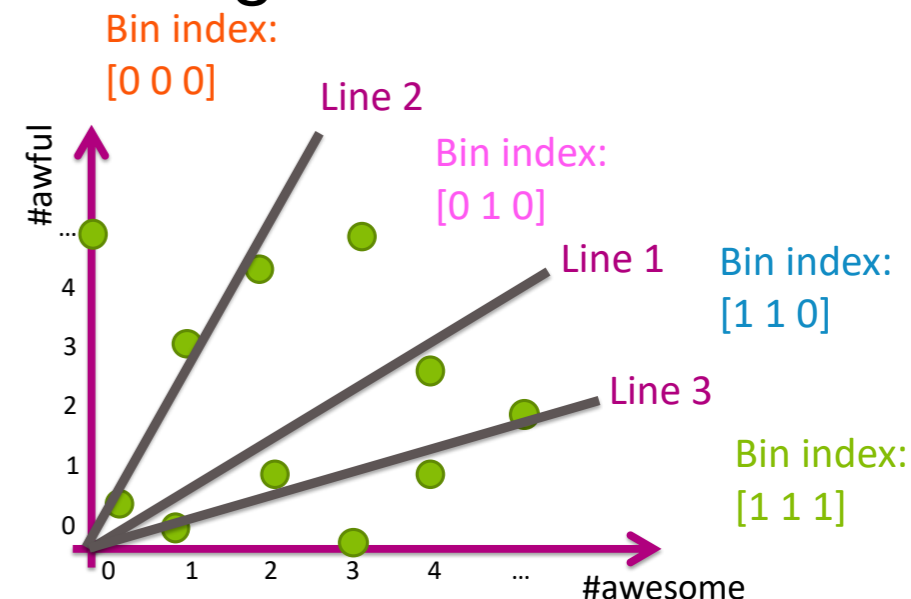
- In practice, continue searching in near-by bins until computational budget runs out

Locality Sensitive Hashing

- This strategy is referred to as “Locality Sensitive Hashing”

- Locality Sensitive Hashing

- Draw h random lines
- Compute score for each point e.r.t. each line and translate it into a binary indices
- Represent each data point using h dimensional binary vector
- Create Hash table



2D Data	Sign (Score ₁)	Bin 1 index	Sign (Score ₂)	Bin 2 index	Sign (Score ₃)	Bin 3 index
$x_1 = [0, 5]$	-1	0	-1	0	-1	0
$x_2 = [1, 3]$	-1	0	-1	0	-1	0
$x_3 = [3, 0]$	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}

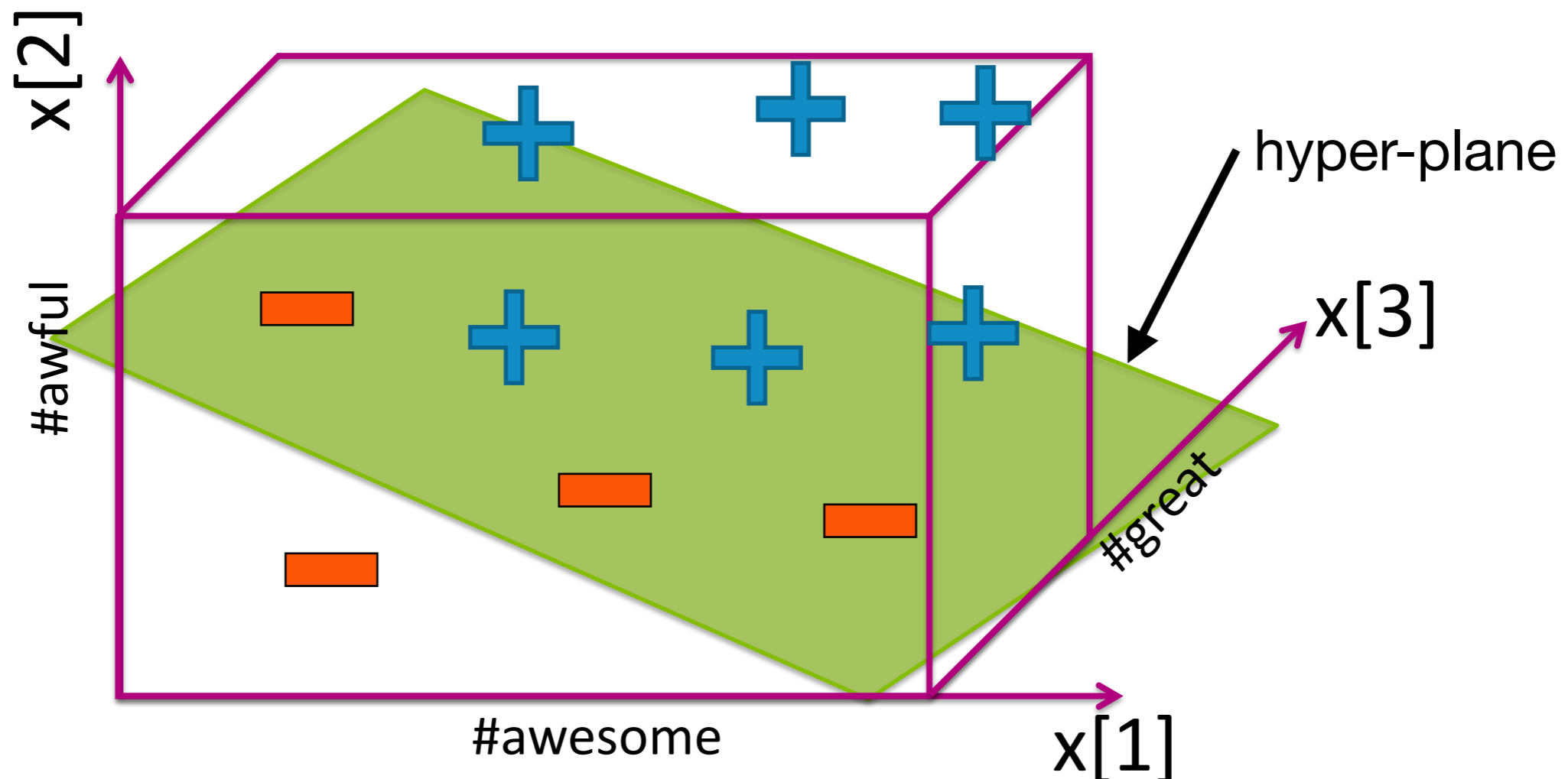
- Querying in a locality sensitive hashing
 - For a query point x , search for the nearest neighbor among all points in the bin with a binary index vector $bin(x)$, then continue searching close-by bins until budget runs out

Moving to high dimensional data

Partition by hyper-planes

- In dimensions higher than 2, randomly draw hyper-planes and compute the score
- This partitions the space into two

$$\text{Score}(\mathbf{x}) = w_1 \# \text{awesome} + w_2 \# \text{awful} + w_3 \# \text{great}$$



Computational cost of binning in d-dimensions

$$\text{Score}(x) = w_1 \#awesome + w_2 \#awful + w_3 \#great$$

- Each data point requires d multiplications to compute the score (and compute the bin indices)
- There is a one-time cost, that can be pre-computed before any query comes in
- Often times query complexity is more important
- How should the number of planes h scale with the dimension d ?
- What is the **goal** of locality sensitive hashing?
 - Find a representation of a point, such that
 - It is lower dimensional than d , i.e. $h < d$
 - While preserving the distance between two points
- $h = \log(d)$ is sufficient to achieve this goal