INTRODUCING RETRIEVAL ALGORITHMS

Scott O'Hara
Metrowest Developers Machine Learning Group
05/12/2021

REFERENCES

The material for this talk is primarily drawn from the notes, slides and lectures of the courses and book below:

Machine Learning Foundations: A Case Study Approach

University of Washington, Prof. Emily Fox & Prof. Carlos Guestrin

https://www.coursera.org/learn/ml-foundations

CSE/STAT 416 - Intro to Machine Learning Spring 2018

University of Washington, Prof. Emily Fox, Spring 2018

https://courses.cs.washington.edu/courses/cse416/18sp/index.html

University of Washington, Sewoong Oh, Spring 2019

https://courses.cs.washington.edu/courses/cse416/19sp/index.html

University of Washington, Vinitra Swamy, Summer 2020

https://courses.cs.washington.edu/courses/cse416/20su/index.html

Last Time:

Recommender Systems

Personalization is transforming our experience of the world

100 Hours a Minute

What do I care about?

Information overload

↓

Browsing is "history"

– Need new ways to discover content

Personalization: Connects *users & items*

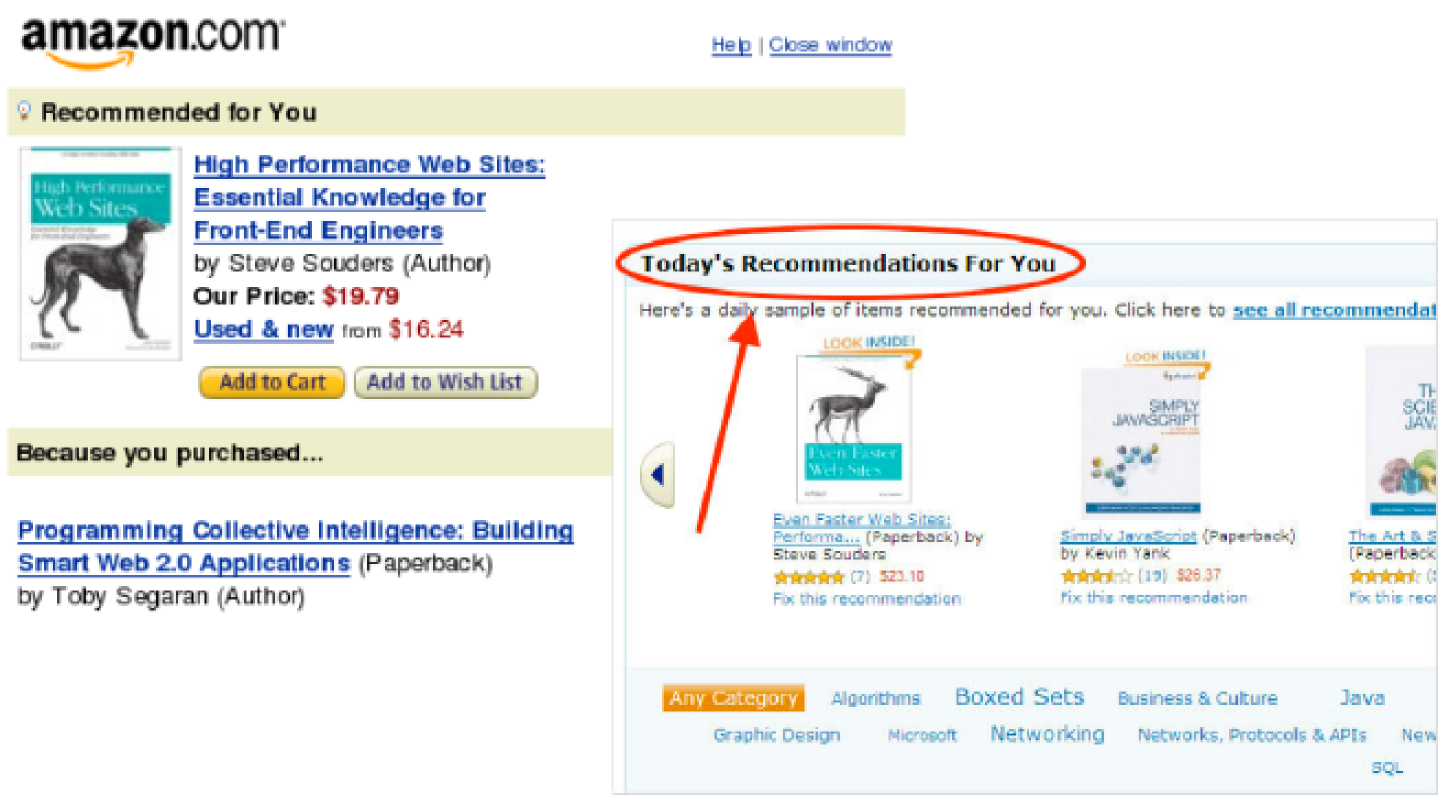
viewers ↗ *users*

videos ↖ *items*

Movie recommendations

Connect users with movies they may want to watch

Product recommendations


amazon.com

Help | Close window

Recommended for You

High Performance Web Sites: Essential Knowledge for Front-End Engineers
by Steve Souders (Author)
Our Price: $19.79
Used & new from $16.24

Add to Cart | Add to Wish List

Because you purchased...

Programming Collective Intelligence: Building Smart Web 2.0 Applications (Paperback)
by Toby Segaran (Author)

Today's Recommendations For You

Here's a daily sample of items recommended for you. Click here to see all recommendat

Even Faster Web Sites: Performa... (Paperback) by Steve Souders
(7) $23.10
Fix this recommendation

Simply JavaScript (Paperback) by Kevin Yank
(19) $26.37
Fix this recommendation

The Art & S
(Paperback
Fix this rec

Any Category | Algorithms | Boxed Sets | Business & Culture | Java | Graphic Design | Microsoft | Networking | Networks, Protocols & APIs | New | SQL

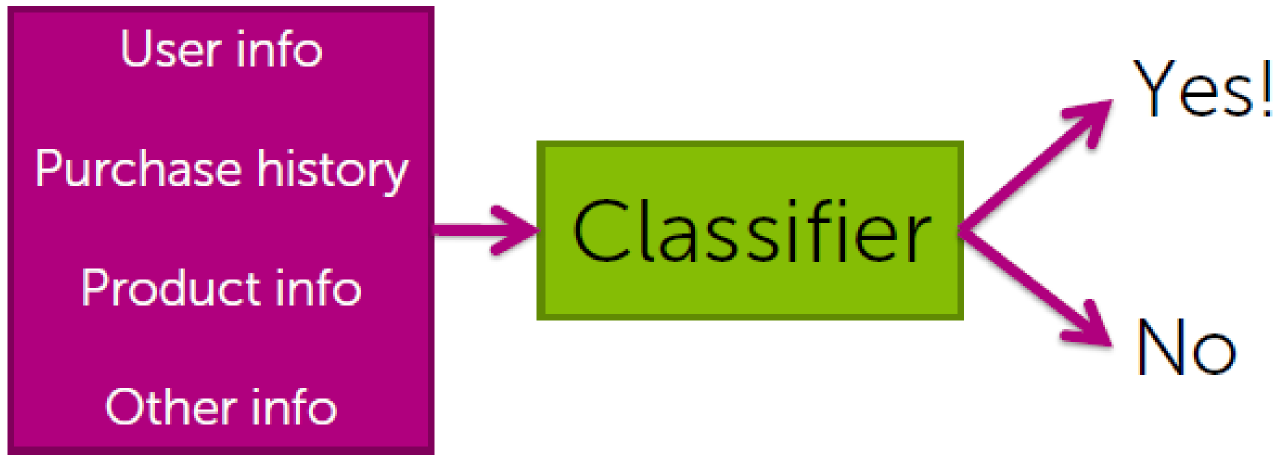
Recommendations combine global & session interests

Approaches to Building a Recommender System

- Solution 0: Popularity
- Solution 1: Content-based Filtering (Classification Model)
- Solution 2: Collaborative Filtering

  (People who bought this also bought …)
- Solution 3: Discovering Hidden Structure by Matrix Factorization
- Bringing it all together: Featurized Matrix Factorization

What's the probability I'll buy this product?



Pros:

- Personalized:
  Considers user info & purchase history
- Features can capture context:
  Time of the day, what I just saw,…
- Even handles limited user history: Age of user, …

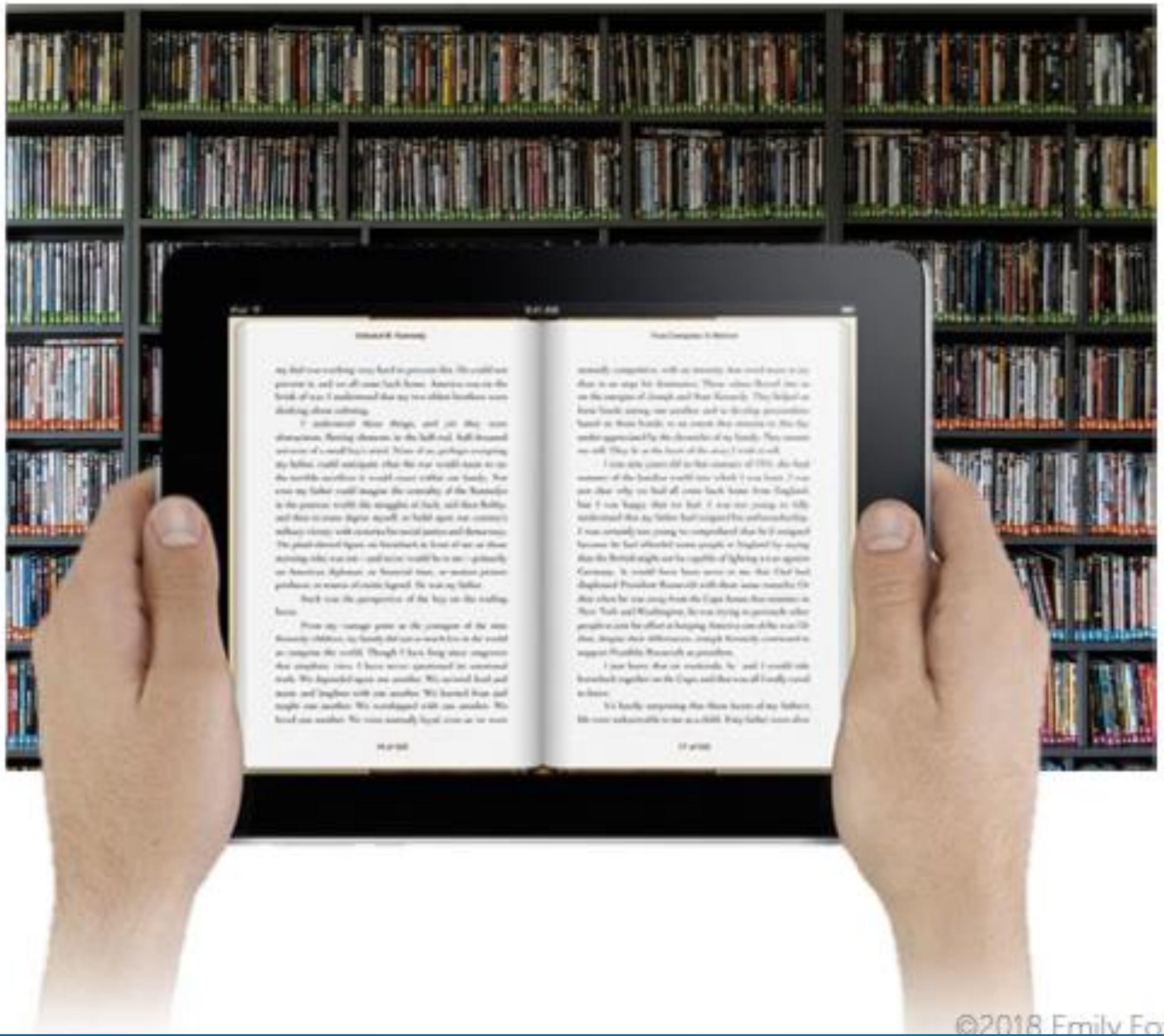
Top K versus diverse outputs

- Top K recommendations may be very redundant
  - *People who liked Rocky 1 also enjoyed Rocky 2, Rocky 3, Rocky 4, Rocky 5,…*

- Diverse recommendations
  - Users are multi-facetted & want to hedge our bets
  - Rocky 2, It's Always Sunny in Philadelphia, Gandhi

But sometimes, the most similar items are exactly what you are looking for…

Document retrieval



Document retrieval

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

Challenges

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

Three Retrieval Algorithms

- KNN – K Nearest Neighbors

  The basic retrieval idea. Easy to understand, but inefficient.

- KD-Trees – K-dimensional Trees

  More efficient than KNN, but subject to curse of dimensionality.

- LSH – Locality Sensitive Hashing

  Hash items so that that similar items are likely to end up in the same bins. Much more efficient than KD-Trees and curse of dimensionality is much less a problem.

Presentation Order

- KNN – K Nearest Neighbors

  The basic retrieval idea. Easy to understand, but inefficient.

- LSH – Locality Sensitive Hashing

  Hash items so that that similar items are likely to end up in the same bins. Much more efficient than KD-Trees and curse of dimensionality is much less a problem.

- KD-Trees – K-dimensional Trees

  More efficient than KNN, but subject to curse of dimensionality.

Important Points

While the formalization of these algorithms are fairly tedious, the intuition is fairly simple. Find the 1 or k nearest neighbors to a given document and return those as the answer.

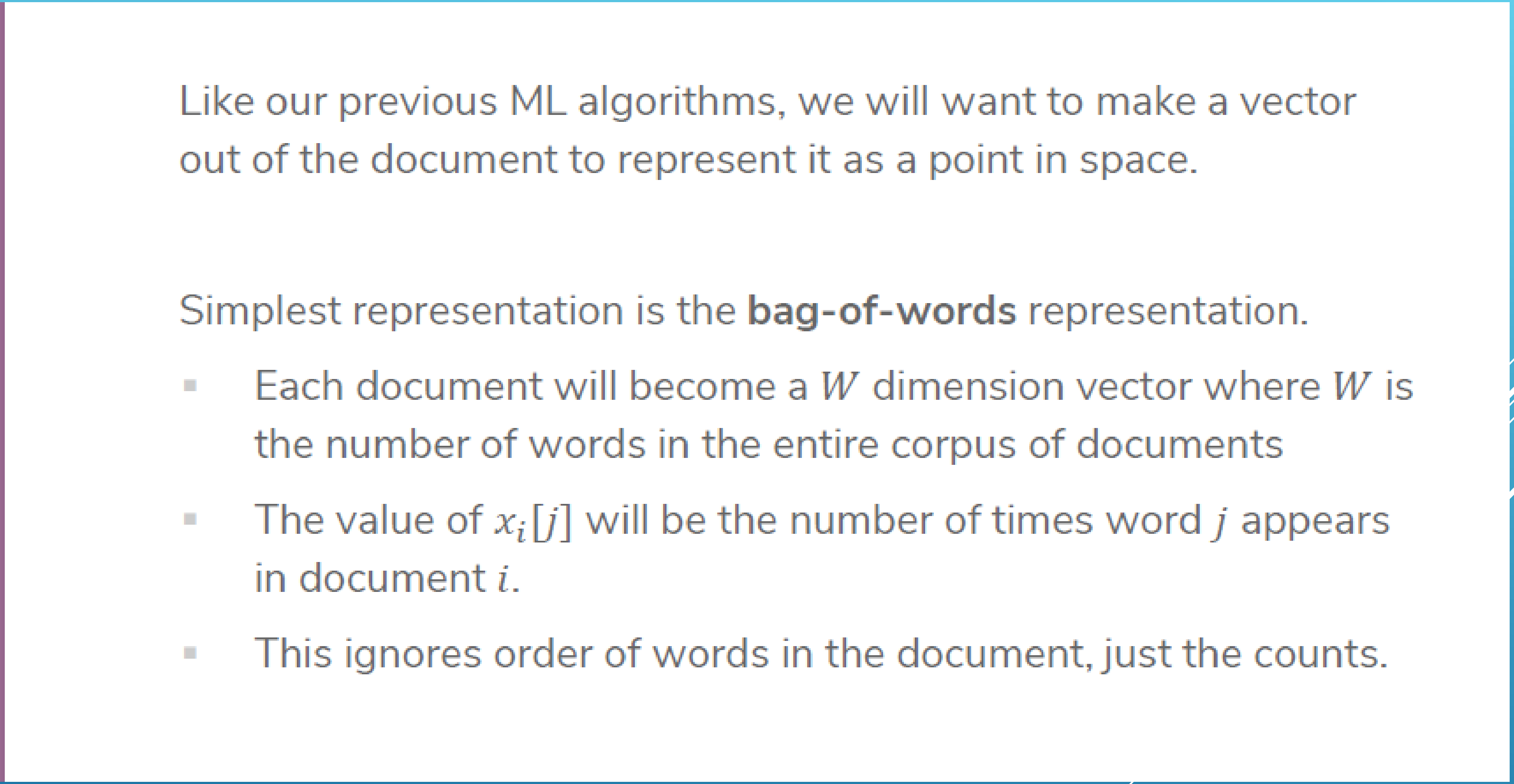
This intuition relies on answering two important questions

- How do we represent the documents x_i?
- How do we measure the distance $distance(x_q, x_i)$?

Two Important Questions

- Document Representation
  - Bag of words
  - TF-IDF
- Similarity/Distance Measures
  - Euclidean distance
  - Similarity represented by vector dot products
  - Cosine similarity
  - Many more …

Document Representation

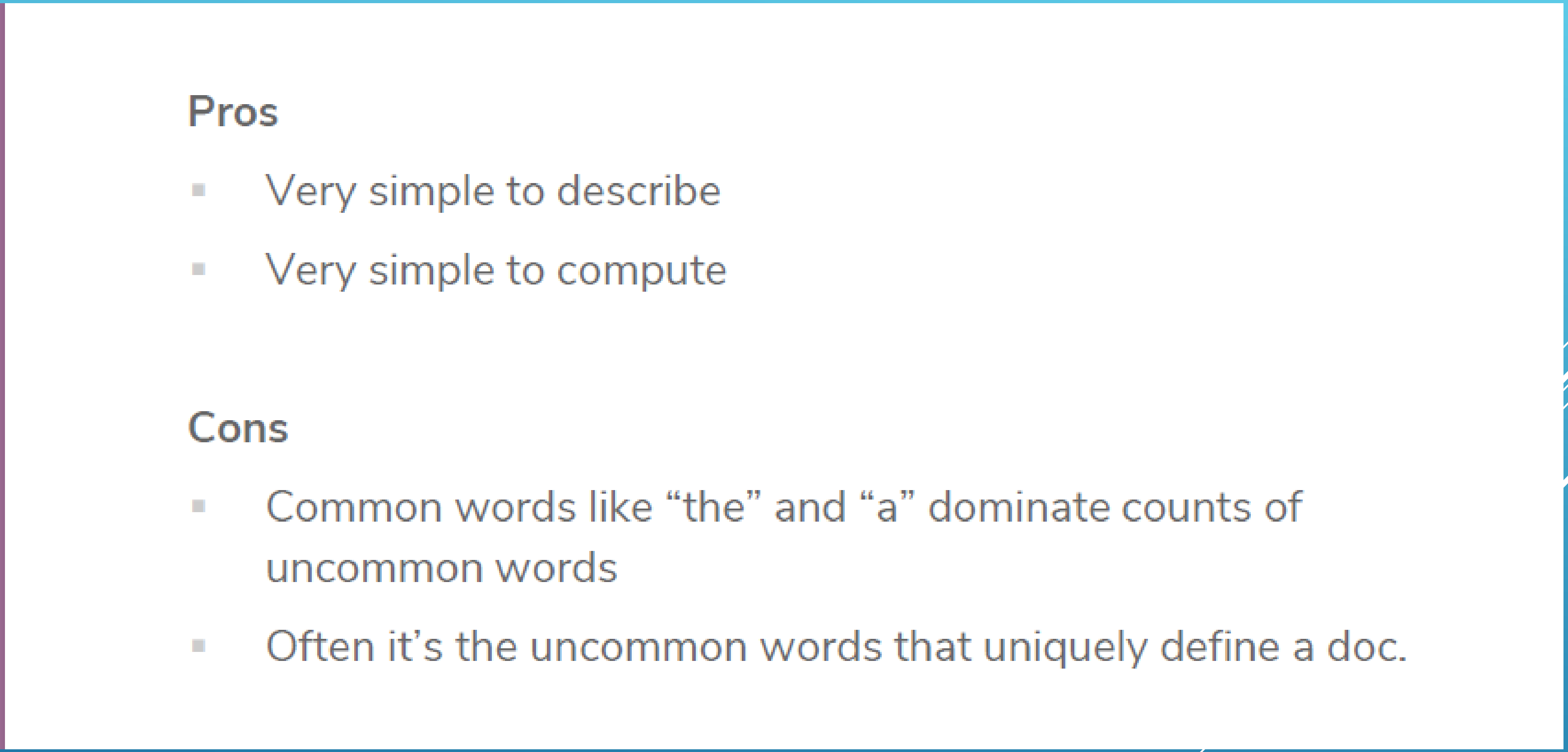


Like our previous ML algorithms, we will want to make a vector out of the document to represent it as a point in space.

Simplest representation is the **bag-of-words** representation.

- Each document will become a W dimension vector where W is the number of words in the entire corpus of documents
- The value of $x_i[j]$ will be the number of times word j appears in document i.
- This ignores order of words in the document, just the counts.

Bag of Words



Pros

- Very simple to describe
- Very simple to compute

Cons

- Common words like “the” and “a” dominate counts of uncommon words
- Often it’s the uncommon words that uniquely define a doc.

TF-IDF

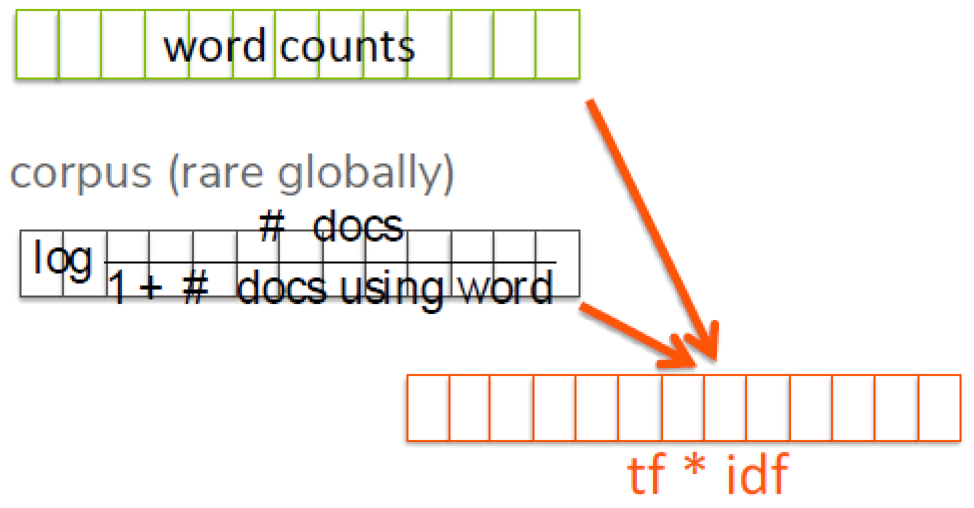
Goal: Emphasize important words

- Appear frequently in the document (common locally)

Term frequency =

- Appears rarely in the corpus (rare globally)

Inverse doc freq. =



Do a pair-wise multiplication to compute the TF-IDF for each word

- Words that appear in every document will have a small IDF making the TF-IDF small!

Distance

Now we will define what similarity/distance means

Want to define how “close” two vectors are. A smaller value for distance means they are closer, a large value for distance means they are farther away.

The simplest way to define distance between vectors is the **Euclidean distance**

$$distance(x_i, x_q) = \left|\left|x_i - x_q\right|\right|_2$$

$$= \sqrt{\sum_{j=1}^{D} (x_i[j] - x_q[j])^2}$$

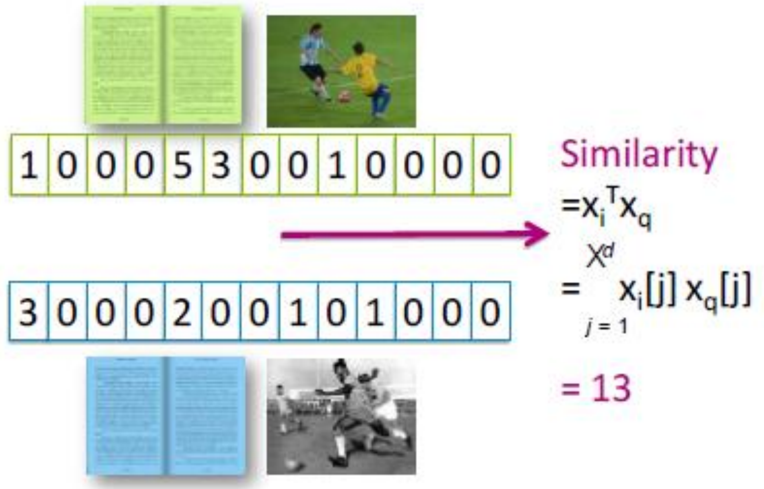
Similarity

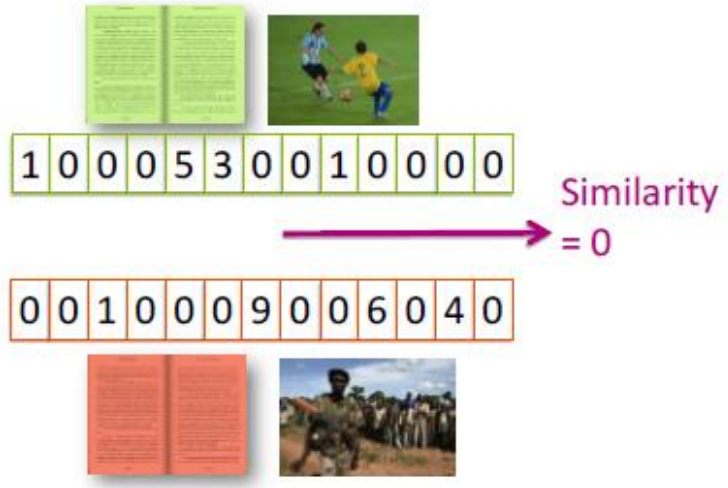
Another natural similarity measure would use

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

Notice this is a measure of similarity, not distance

- This means a bigger number is better





Cosine Similarity

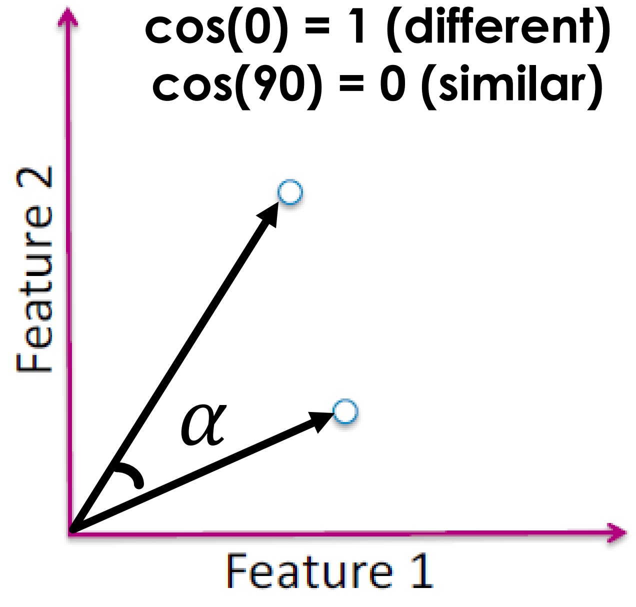
Should we normalize the vectors before finding the similarity?

$$similarity = \frac{x_i^T x_q}{\left|\left|x_i\right|\right|_2 \left|\left|x_q\right|\right|_2} = \cos(\theta)$$

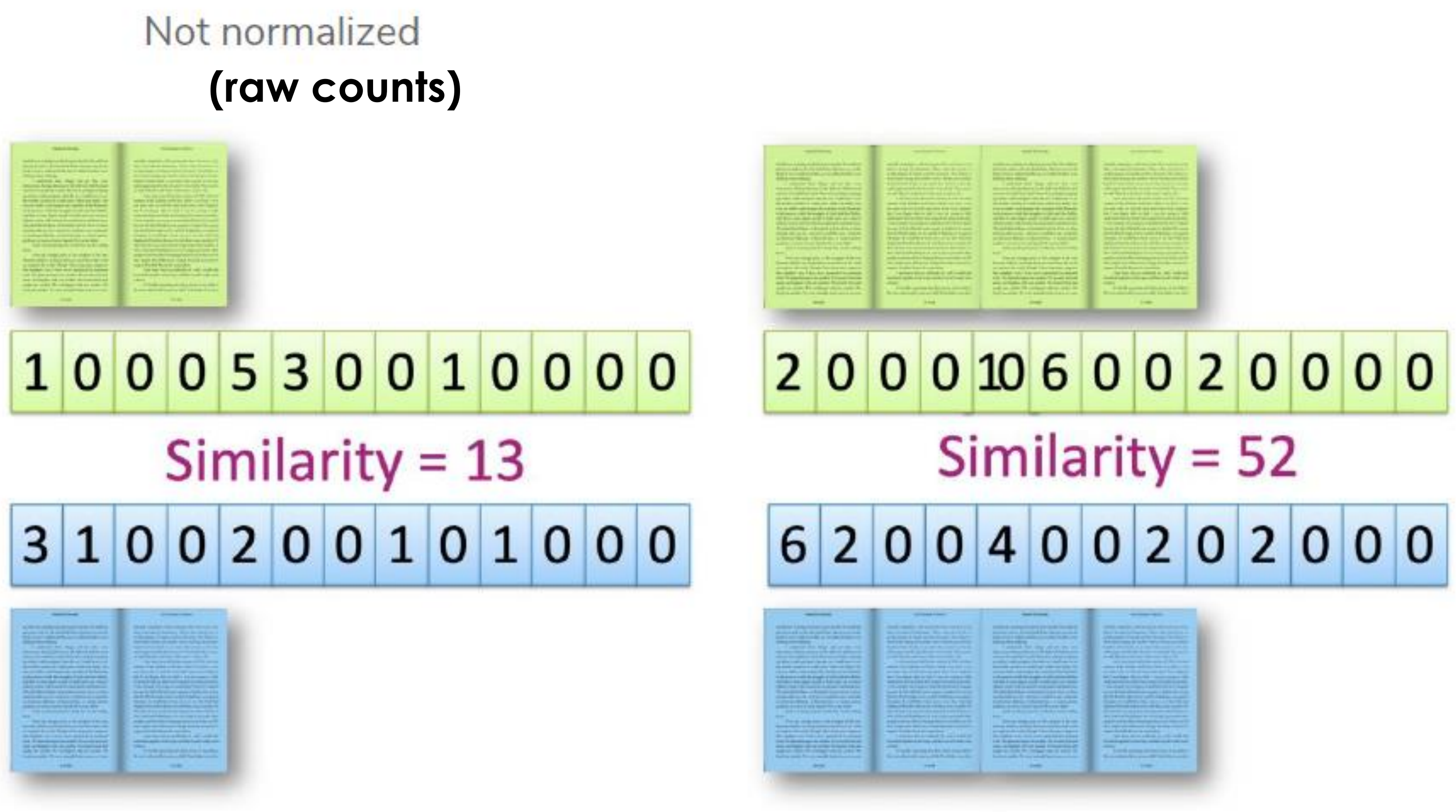
cos(0) = 1 (different)
cos(90) = 0 (similar)

Note:

- Not a true distance metric
- Efficient for sparse vectors!



To Normalize or Not To Normalize?



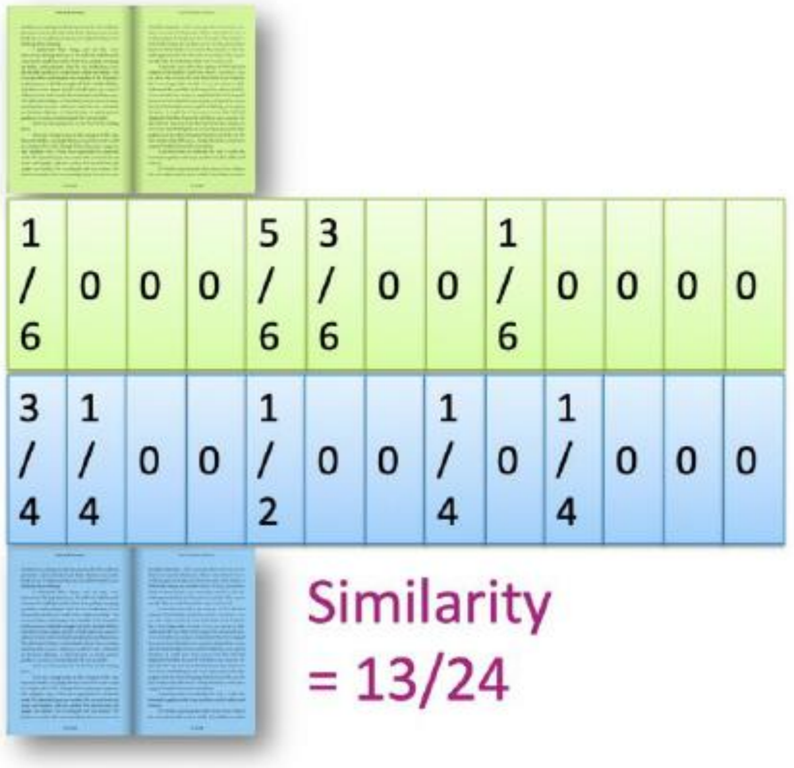
1	0	0	0	5	3	0	0	1	0	0	0	0
3	1	0	0	2	0	0	1	0	1	0	0	0

2	0	0	0	10	6	0	0	2	0	0	0	0
6	2	0	0	4	0	0	2	0	2	0	0	0

To Normalize or Not To Normalize?

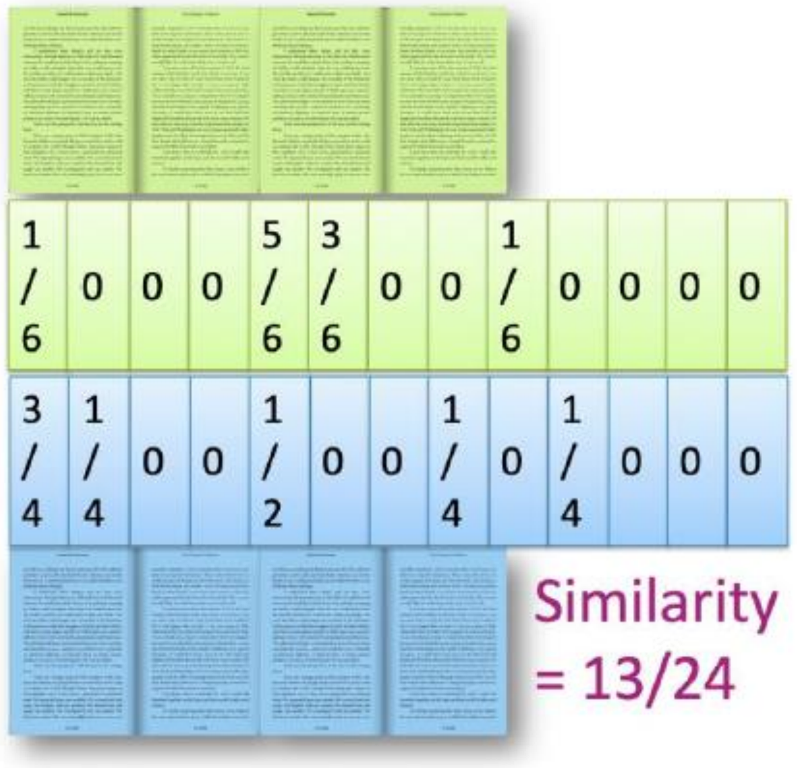
Normalized

(using TF-IDF)



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

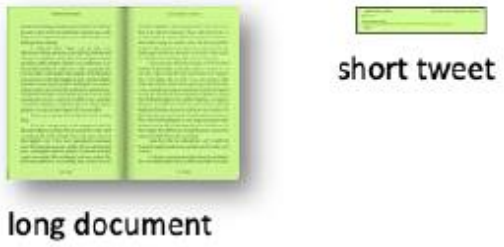


1/6	0	0	0	5/6	3/6	0	0	1/6	0	0	0	0
3/4	1/4	0	0	1/2	0	0	1/4	0	1/4	0	0	0

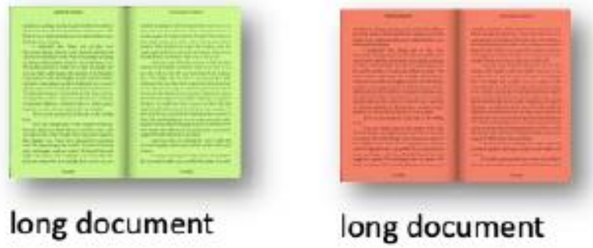
Similarity = 13/24

To Normalize or Not To Normalize?

Normalization is not desired when comparing documents of different sizes since it ignores length.



Normalizing can make dissimilar objects appear more similar



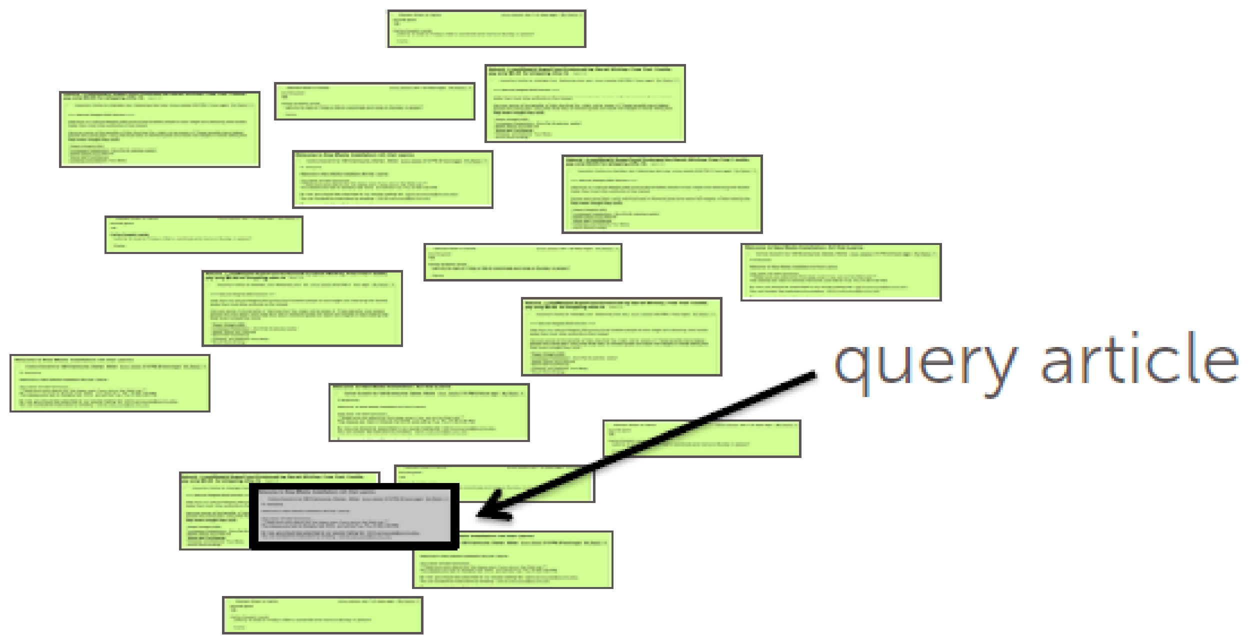
Common compromise: Just cap maximum word counts

In practice, can use multiple distance metrics and combine them using some defined weights

Retrieval as K-nearest neighbor search

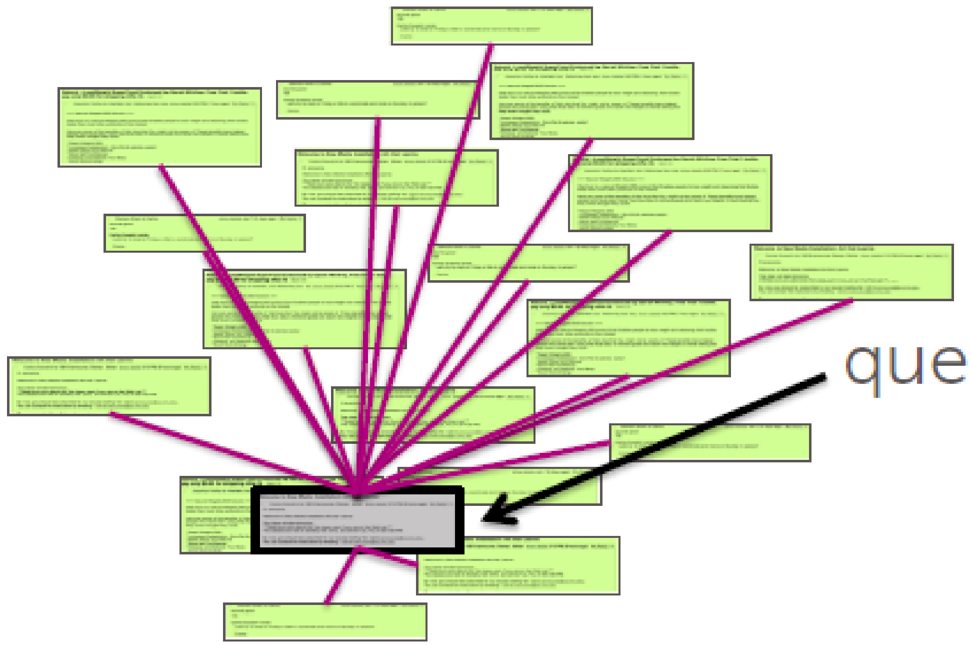
1-NN search for retrieval

Space of all articles,
organized by similarity of text



Compute distances to all docs

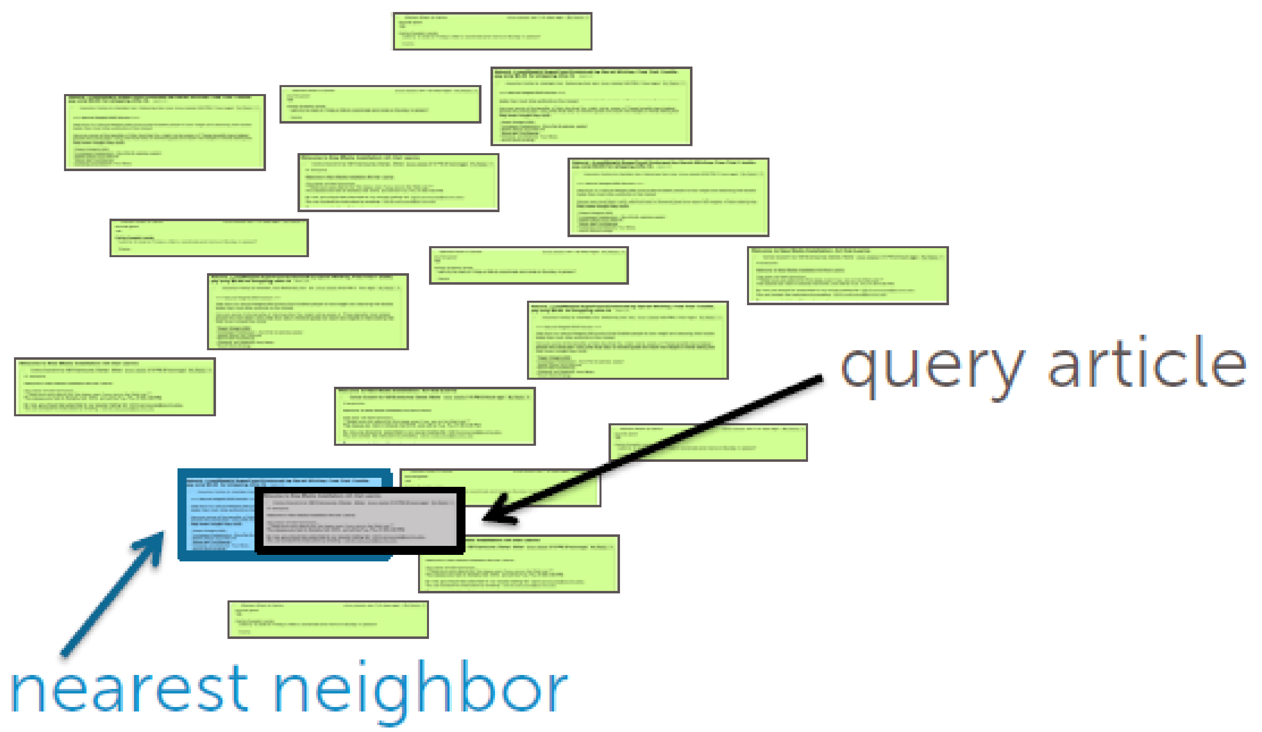
Space of all articles,
organized by similarity of text



query article

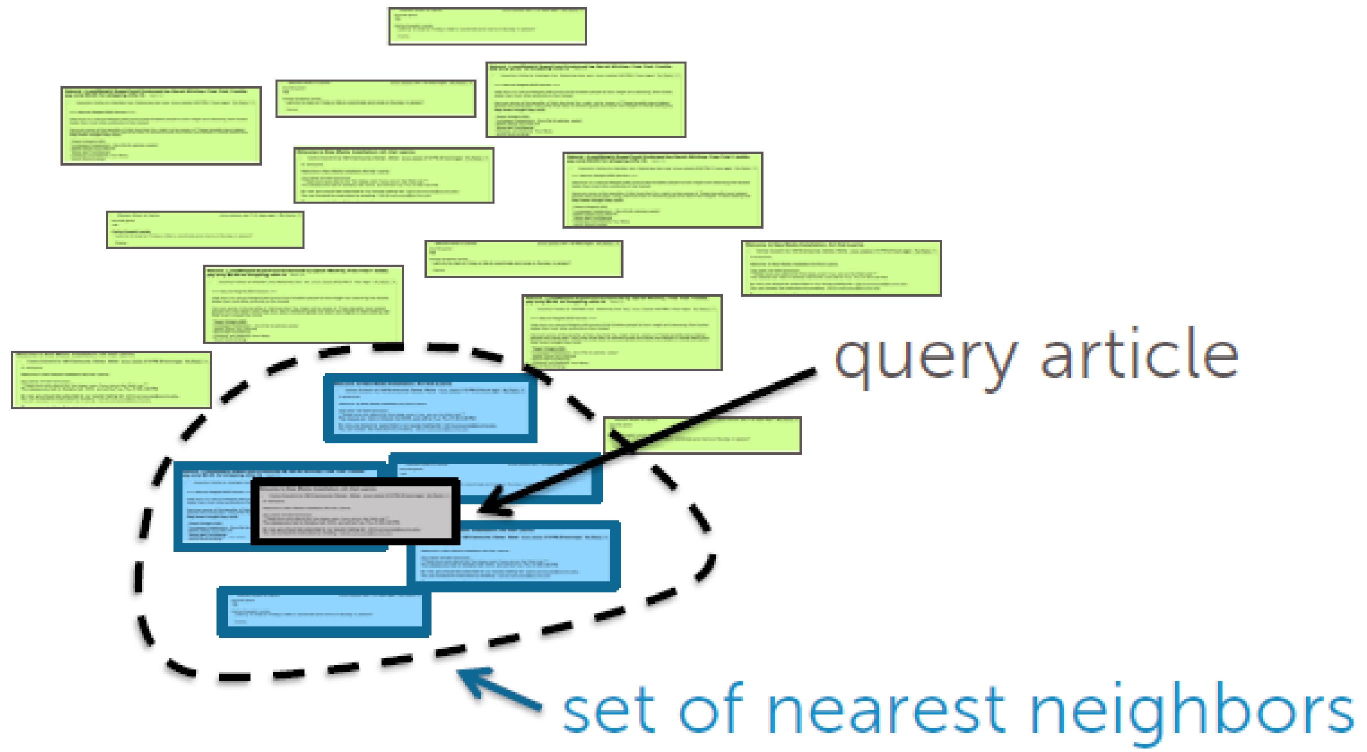
Retrieve "nearest neighbor"

Space of all articles,
organized by similarity of text



Or set of nearest neighbors

Space of all articles,
organized by similarity of text



1-Nearest Neighbor

Input

- x_q: Query example (e.g. my book)
- $x_1, \ldots, x_n$: Corpus of documents (e.g. Amazon books)

Output

- The document in corpus that is most similar to x_q

$$x^{NN} = \underset{x_i \in [x_1, \ldots, x_n]}{\arg\min}\ distance(x_q, x_i)$$

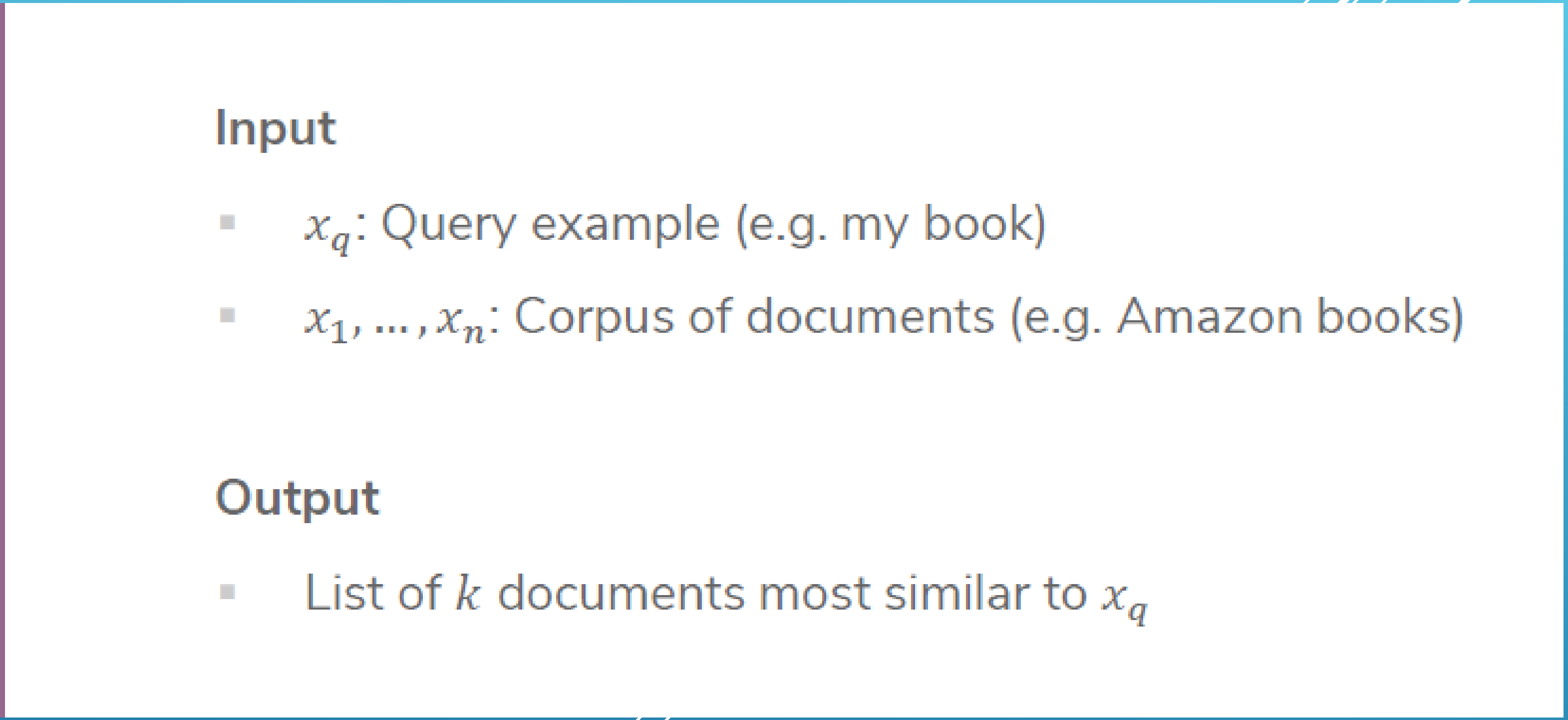
It's very critical to properly define how we represent each document x_i and the similarity metric $distance$! Different definitions will lead to very different results.

1-Nearest Neighbor

How long does it take to find the 1-NN? About n operations

```
**Input:** $x_q$
$x^{NN} = \emptyset$
$nn\_dist = \infty$
$for\ x_i \in [x_1, \dots, x_n]$:
    $dist = distance(x_q, x_i)$
    $if\ dist < nn\_dist$:
        $x^{NN} = x_i$
        $nn\_dist = dist$
**Output:** $x^{NN}$
```

k-Nearest Neighbors



Input

- x_q: Query example (e.g. my book)
- $x_1, \ldots, x_n$: Corpus of documents (e.g. Amazon books)

Output

- List of k documents most similar to x_q

k-Nearest Neighbors

Same idea as 1-NN algorithm, but maintain list of k-NN

```
Input: $x_q$
$X^{k-NN} = [x_1, \ldots, x_k]$
$nn\_dists = [dist(x_1, x_q), dist(x_2, x_q), \ldots, dist(x_k, x_q)]$
$for\ x_i \in [x_{k+1}, \ldots, x_n]$:
    $dist = distance(x_q, x_i)$
    $if\ dist < \max(nn\_dists)$ :
        $remove\ largest\ dist\ from\ X^{k-NN}\ and\ nn\_dists$
        $add\ x_i\ to\ X^{k-NN} and\ distance(x_q, x_i)\ to\ nn\_dists$
Output: $X^{k-NN}$
```

k-Nearest Neighbors

Can be used in many circumstances!

Retrieval

Return X^{k-NN}

Regression

$$\hat{y}_i = \frac{1}{k}\sum_{j=1}^{k} x^{NN_j}$$

Presentation last saved: Just now

Classification

$$\hat{y}_i = majority\_class(X^{k-NN})$$

Retrieval with LSH (Locality Sensitive Hashing)

Nearest Neighbor Efficiency

Nearest neighbor methods are good because they require no training time (just store the data, compute NNs when predicting).

How slow can that be? Very slow if there is a lot of data!

- $\mathcal{O}(n)$ if there are n data points.
- If n is in the hundreds of billions, this will take a while…

There is not an obvious way of speeding this up unfortunately.

Big Idea: Sacrifice accuracy for speed. We will look for an approximate nearest neighbor to return results faster

Approximate Nearest Neighbor

Don’t find the exact NN, find one that is “close enough”.

Many applications are okay with approximate answers

- The measure of similarity is not perfect
- Clients probably can’t tell the difference between the most similar book and a book that’s pretty similar.

We will use **locality sensitive hashing** to answer this approximate nearest neighbor problem.

High level approach

- Design an algorithm that yields a close neighbor with high probability
- These algorithms usually come with a “guarantee” of what probability they will succeed, won’t discuss that in detail but is important when making a new approximation algorithm.

Locality Sensitive Hashing (LSH)

Locality Sensitive Hashing is an algorithm that answers the approximate nearest neighbor problem.

Big Idea

- Break the data into smaller bins based on how close they are to each other
- When you want to find a nearest neighbor, choose an appropriate bin and do an exact nearest neighbor search for the points in that bin.

More bins → Fewer points per bin → Faster search

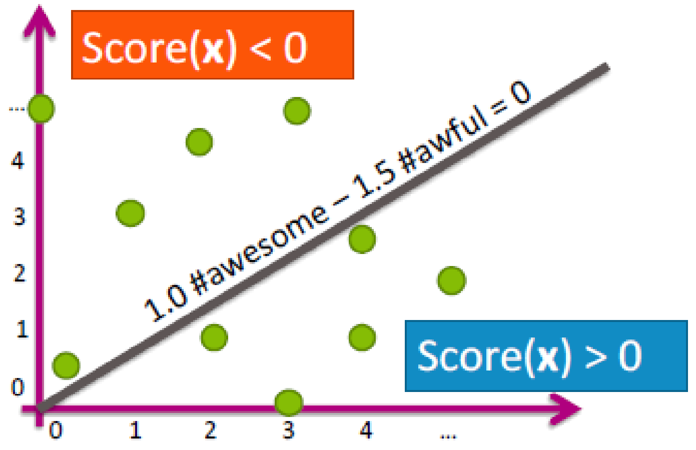
More bins → More likely to make errors if we aren't careful

Binning

How do we make the bins?

What if we pick some line that separates the data and then put them into bins based on the $Score(x)$ for that line?

Looks like classification, but we don't have labelled data here. Will explain shortly how to find this line.

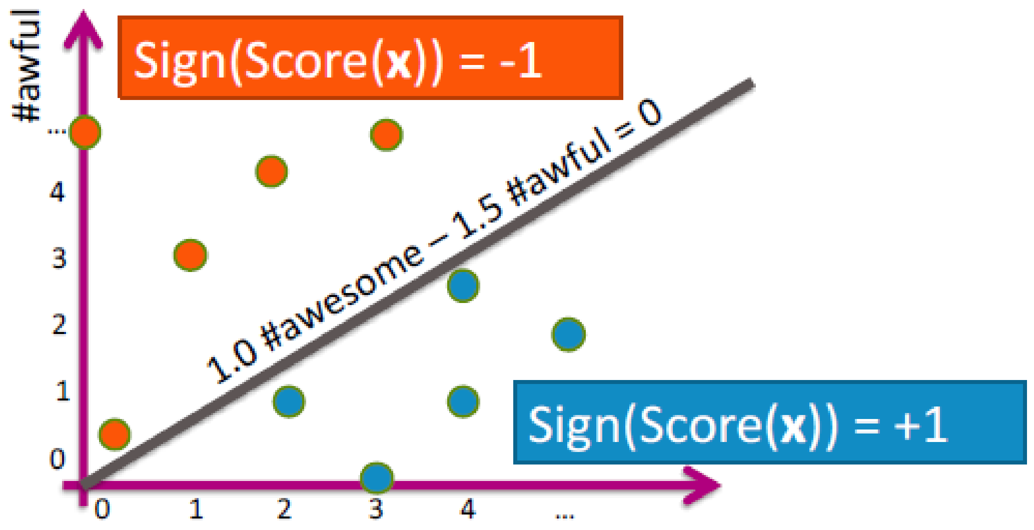


Binning

Put the data in bins based on the sign of the score (2 bins total)

Call negative score points bin 0, and the other bin 1 (**bin index**)

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

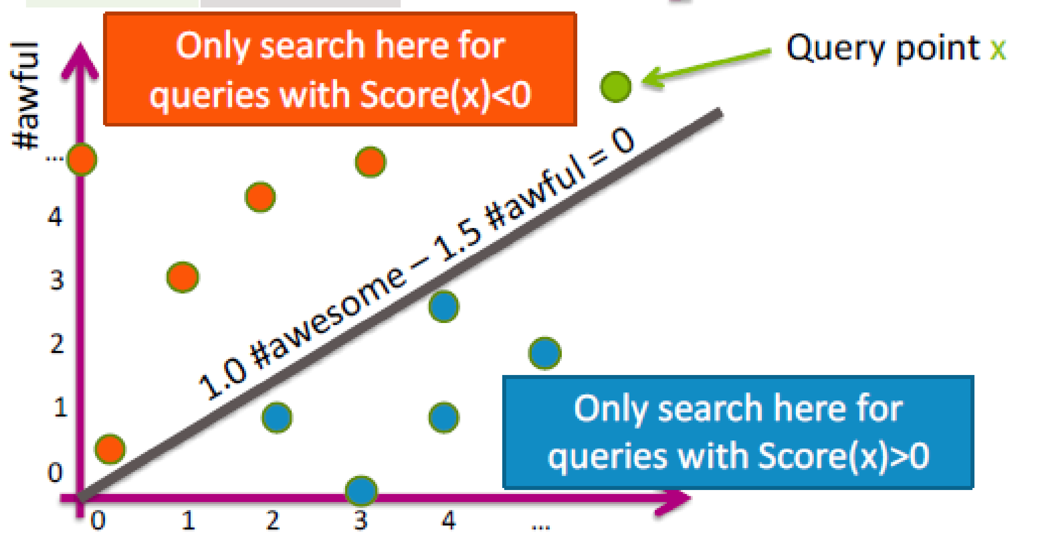


Binning

Put the data in bins based on the sign of the score (2 bins total)

Call negative score points bin 0, and the other bin 1 (**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 Score(x)<0
Only search here for queries with Score(x)<0
Query point x
#awful
1.0 #awesome – 1.5 #awful = 0
Only search here for queries with Score(x)>0
0
1
2
3
4
...

When asked to find neighbor for query point, only search through points in the same bin!

This reduces the search time to $\frac{n}{2}$ if we choose the line right.

LSH with 2 bins

Create a table of all data points and calculate their bin index based on some chosen line

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 for fast lookup

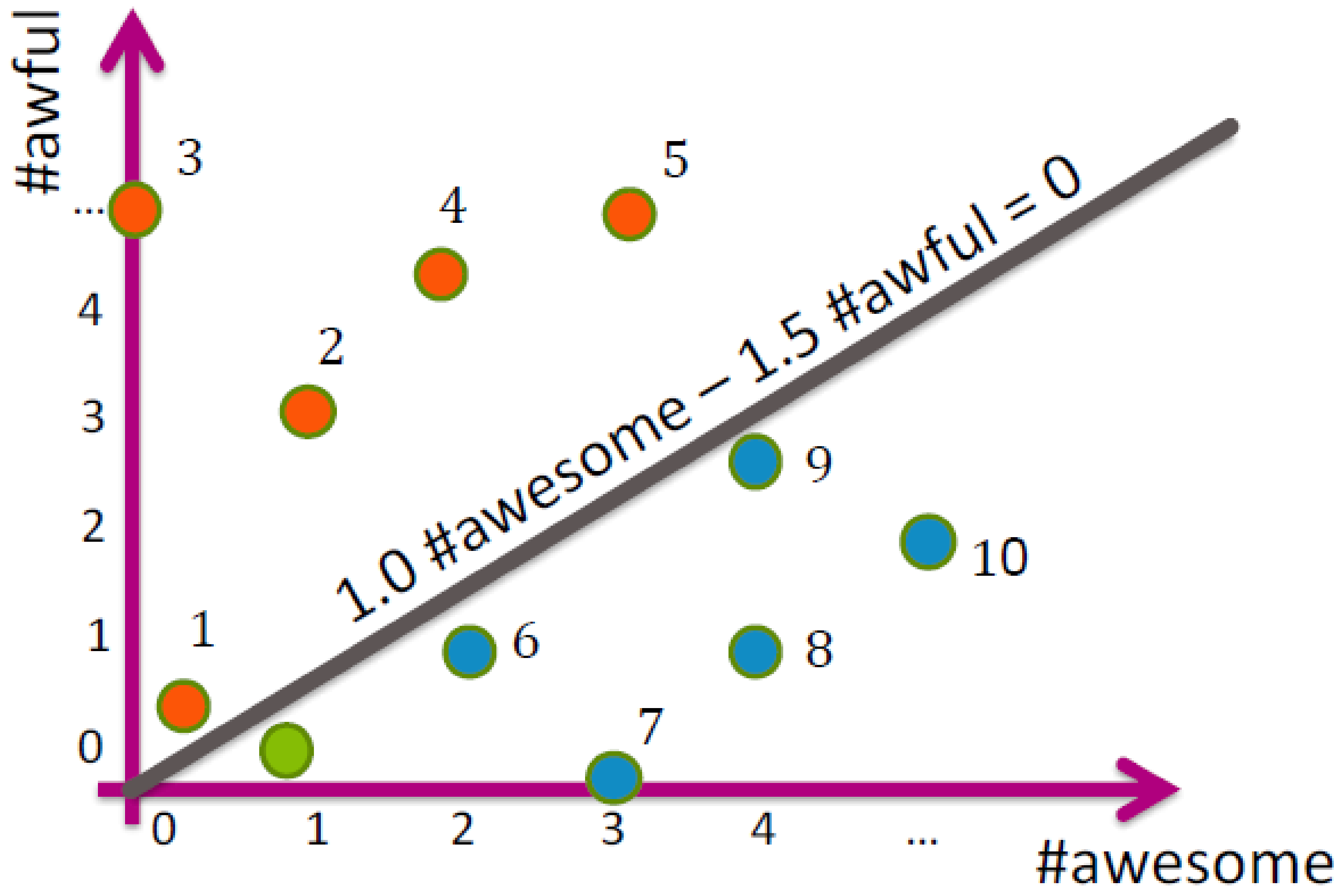
Bin	0	1
List containing indices of datapoints:	{1,2,4,7,...}	{3,5,6,8,...}

HASH TABLE

When searching for a point x_q:

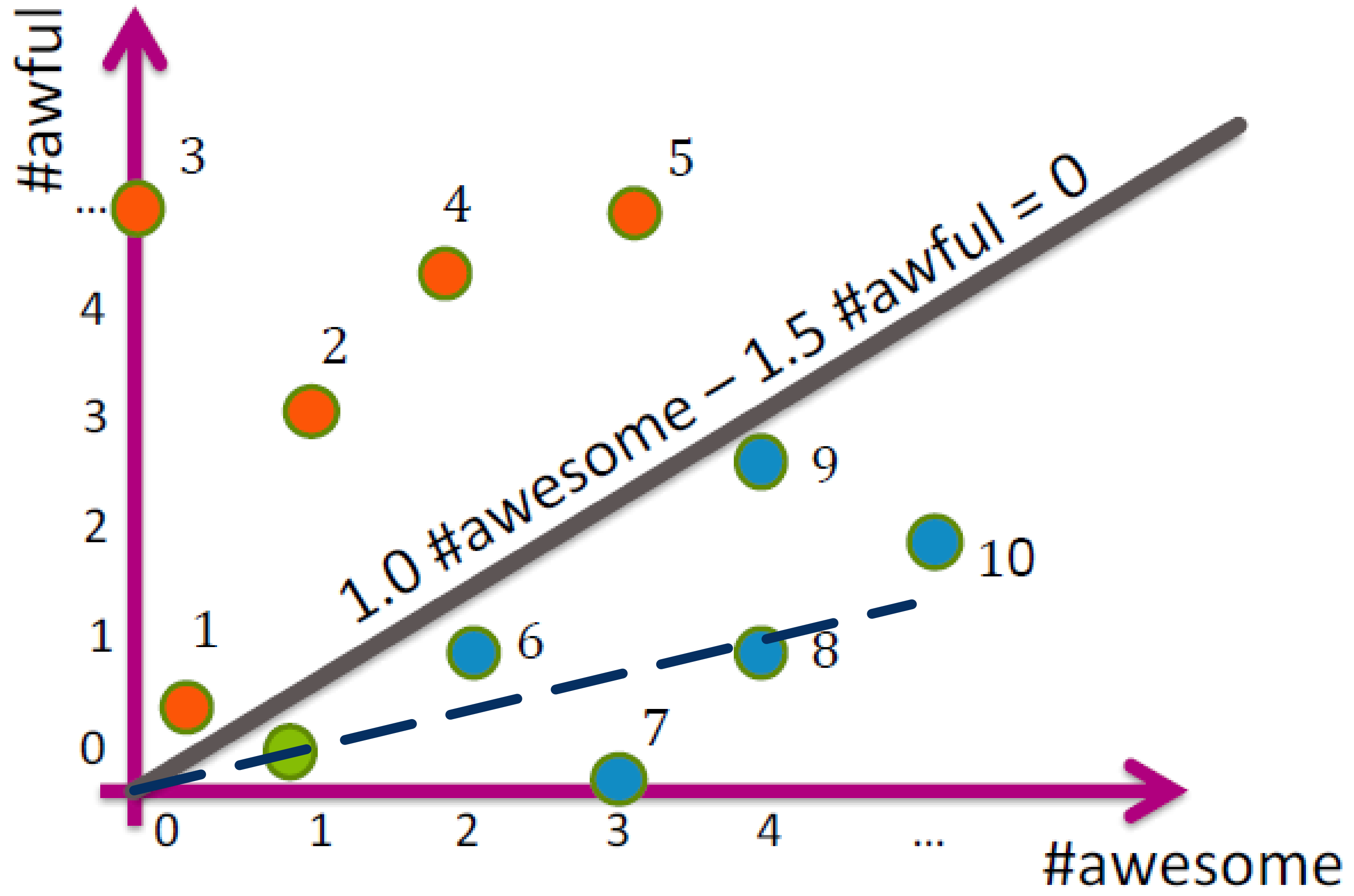
- Find its bin index based on that line
- Search over the points in that bin

If we used LSH with this line, what would be the result returned for searching for the nearest neighbor of the green query point?



?

If we used LSH with this line, what would be the result returned for searching for the nearest neighbor of the green query point?



8

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,…}

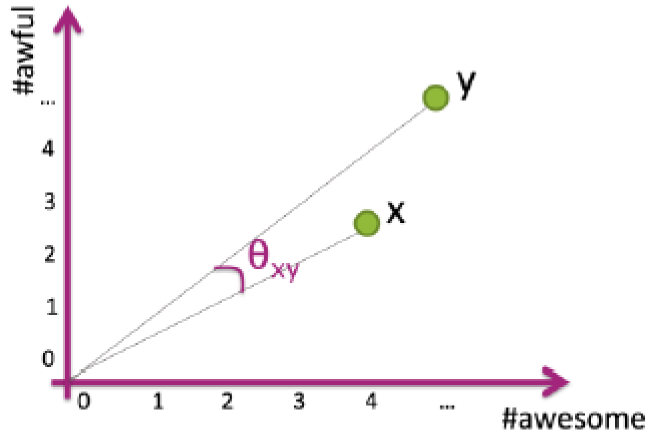
1. How to choose line?

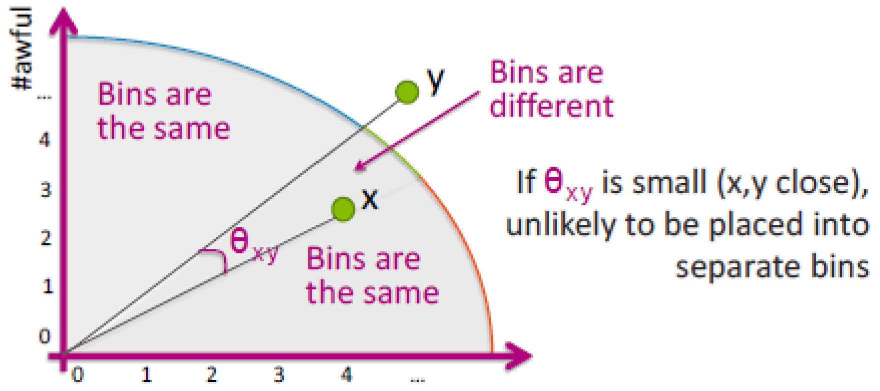
Crazy Idea: Choose the line randomly!

- Choose a slope randomly between 0 and 90 degrees

How bad can randomly picking it be?

- If two points have a small cosine distance, it is unlikely that we will split them into different bins!

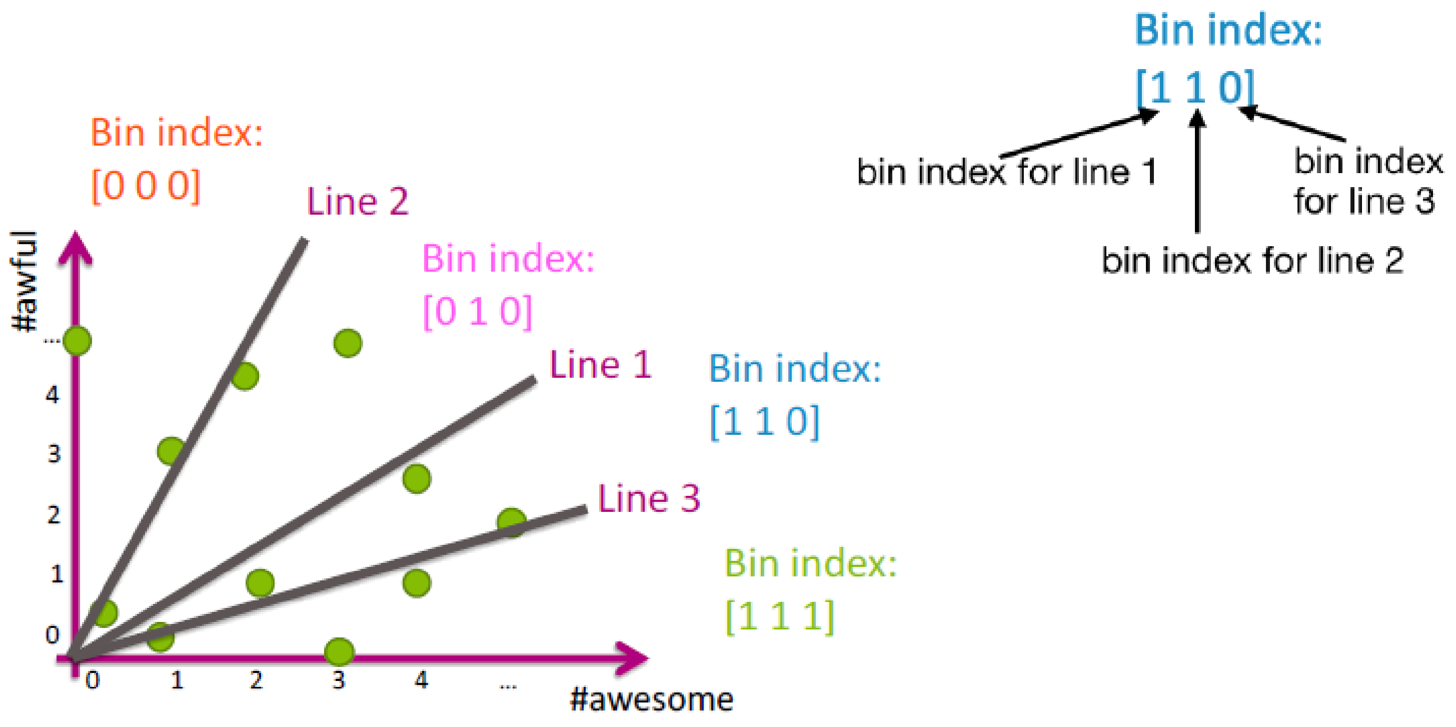




More Bins

Can reduce search cost by adding more lines, increasing the number of bins.

For example, if we use 3 lines, we can make more bins!



LSH with Many Bins

Create a table of all data points and calculate their bin index based on some chosen lines. Store points in hash table indexed by all bin indexes

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}

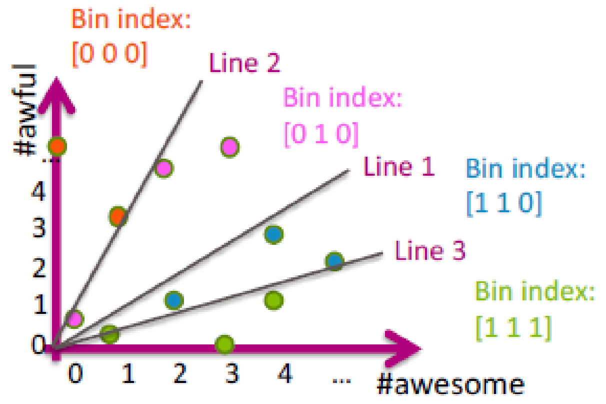
When searching for a point x_q:

- Find its bin index based on the lines
- Only search over the points in that bin

LSH Example

Imagine my query point was (2, 2)

This has bin index [0 1 0]



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 bins, we have reduced the search time!

However, it's more likely that we separate points from their true nearest neighbors since we do more splits ☹

- Often with approximate methods, there is a tradeoff between speed and accuracy.

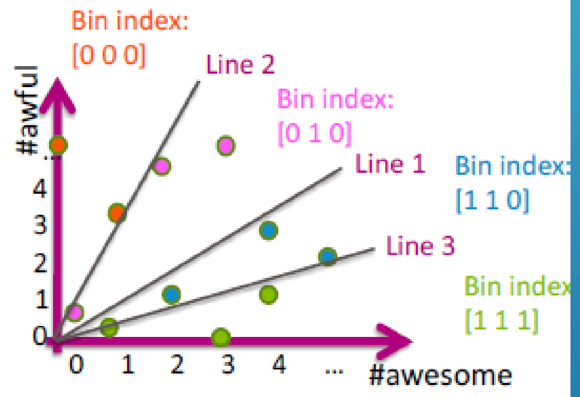
Improve Quality

The nice thing about LSH is we can actually tune this tradeoff by looking at nearby bins. If we spend longer searching, we are likely to find a better answer.

What does "nearby" mean for 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)



In practice, set some time "budget" and keep searching nearby bins until budget runs out

Reference: CSE/STAT 416 Intro to Machine Learning, Summer 2020, University of Washington, Instructor: Vinitra Swamy

Locality Sensitive Hashing (LSH)

Pre-Processing Algorithm

- Draw h lines randomly
- For each data point, compute $Score(x_i)$ for each line
- Translate the scores into binary indices
- Use binary indices as a key to store the point in a hash table

Querying LSH

- For query point x_q compute $Score(x_q)$ for each of the h lines
- Translate scores into binary indices. Lookup all data points that have the same key.
- Do exact nearest neighbor search just on this bin.
- If there is more time in the computation budget, go look at nearby bins until this budget runs out.

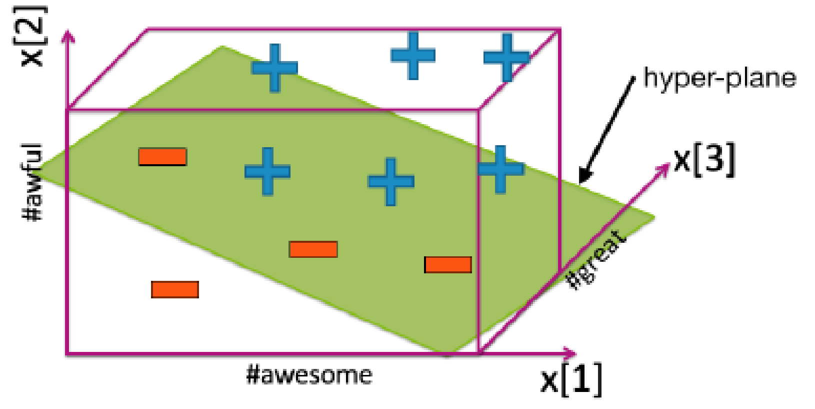
Higher Dimensions

Pick random hyper-plane to separate points for points in higher dimensions.

Unclear how to pick h for LSH and you can't do cross-validation here (why?)

- Generally people use $h \approx \log(d)$

Score(x) = w_1 #awesome + w_2 #awful + w_3 #great



Locality Sensitive Hashing (LSH)

Pre-Processing Algorithm

- Draw h lines randomly
- For each data point, compute $Score(x_i)$ for each line
- Translate the scores into binary indices
- Use binary indices as a key to store the point in a hash table

Querying LSH

- For query point x_q compute $Score(x_q)$ for each of the h lines
- Translate scores into binary indices. Lookup all data points that have the same key.
- Do exact nearest neighbor search just on this bin.
- If there is more time in the computation budget, go look at nearby bins until this budget runs out.

Reference: CSE/STAT 416 Intro to Machine Learning, Summer 2020, University of Washington, Instructor: Vinitra Swamy

Retrieval with KD-Trees (K-dimensional Trees)

INTRODUCING RETRIEVAL ALGORITHMS

Scott O’Hara
Metrowest Developers Machine Learning Group
05/12/2021

Reference: CSE/STAT 416 Intro to Machine Learning, Spring 2019, University of Washington, Instructor: Sewoong Oh

Reference: CSE/STAT 416 Intro to Machine Learning, Summer 2020, University of Washington, Instructor: Vinitra Swamy

Reference: CSE/STAT 416 Intro to Machine Learning, Summer 2020
University of Washington, Instructor: Vinitra Swamy