Randomized / Hashing Algorithms

Shannon Quinn

(with thanks to William Cohen of Carnegie Mellon University, and J. Leskovec, A. Rajaraman, and J. Ullman of Stanford University)

Outline

- Bloom filters
- Locality-sensitive hashing
- ~~Stochastic gradient descent~~ Already covered
- ~~Stochastic SVD~~ Next Wednesday's lecture

Hash Trick - Insights

- Save memory: don't store hash keys
- Allow collisions
  - even though it distorts your data some
- Let the learner (downstream) take up the slack

- Here's another famous trick that exploits these insights….

Bloom filters

- Interface to a Bloom filter
  - BloomFilter(int maxSize, double p);
  - void bf.add(String s); // insert s
  - bool bd.contains(String s);
    - // If s was added return true;
    - // else with probability at least *1-p* return false;
    - // else with probability at most *p* return true;

  - I.e., a noisy “set” where you can test membership (and that’s it)

One possible implementation

```
BloomFilter(int maxSize, double p) {
    set up an empty length-m array bits[];
}
void bf.add(String s) {
    bits[hash(s) % m] = 1;
}
bool bd.contains(String s) {
    return bits[hash(s) % m];
}
```

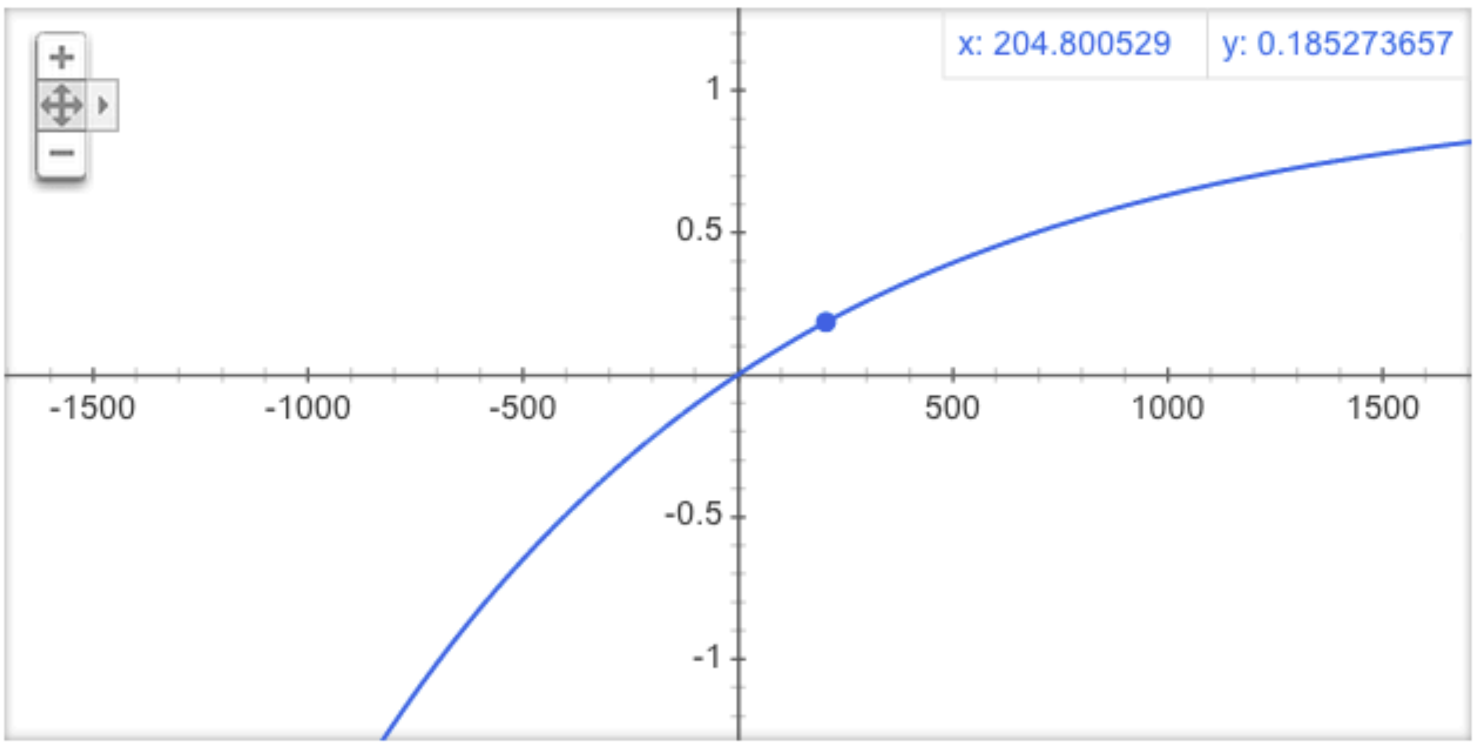
$$\Pr(fp \mid n \text{ prev inserts}) = 1 - \left(1 - \frac{1}{m}\right)^n$$

How well does this work?

$$\Pr(fp \mid x \text{ prev inserts}) = 1 - \left(1 - \frac{1}{m}\right)^x$$

Graph for 1-0.999^x

m=1,000, x~=200, y~=0.18

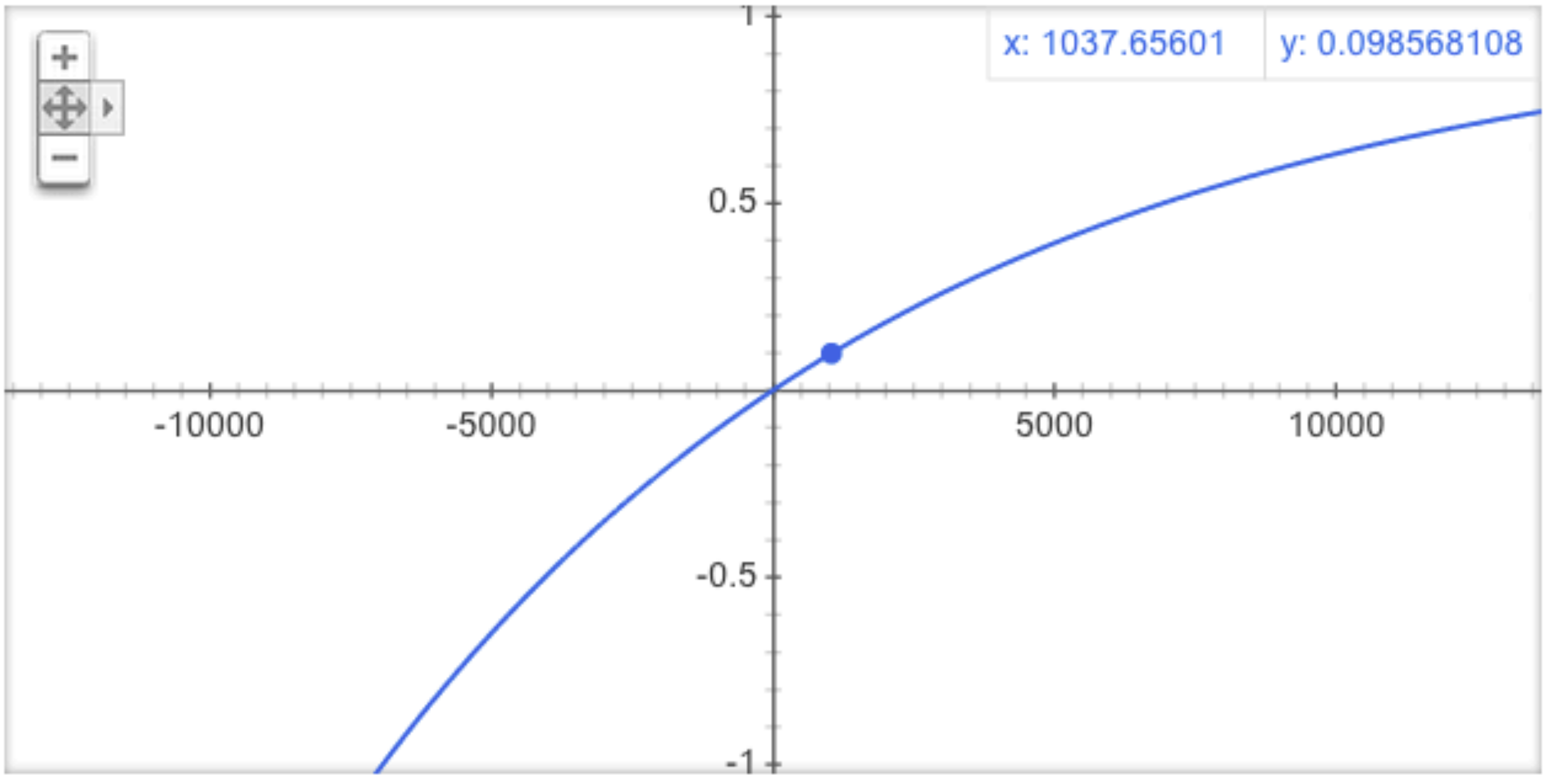


How well does this work?

$$\Pr(fp \mid x \text{ prev inserts}) = 1 - \left(1 - \frac{1}{m}\right)^x$$

Graph for 1-0.9999^x

m=10,000, x~=1,000, y~=0.10



A better??? implementation

```
BloomFilter(int maxSize, double p) {
    set up an empty length-m array bits[];
}
void bf.add(String s) {
    bits[hash1(s) % m] = 1;
    bits[hash2(s) % m] = 1;
}
bool bd.contains(String s) {
    return bits[hash1(s) % m] && bits[hash2(s) % m];
}
```

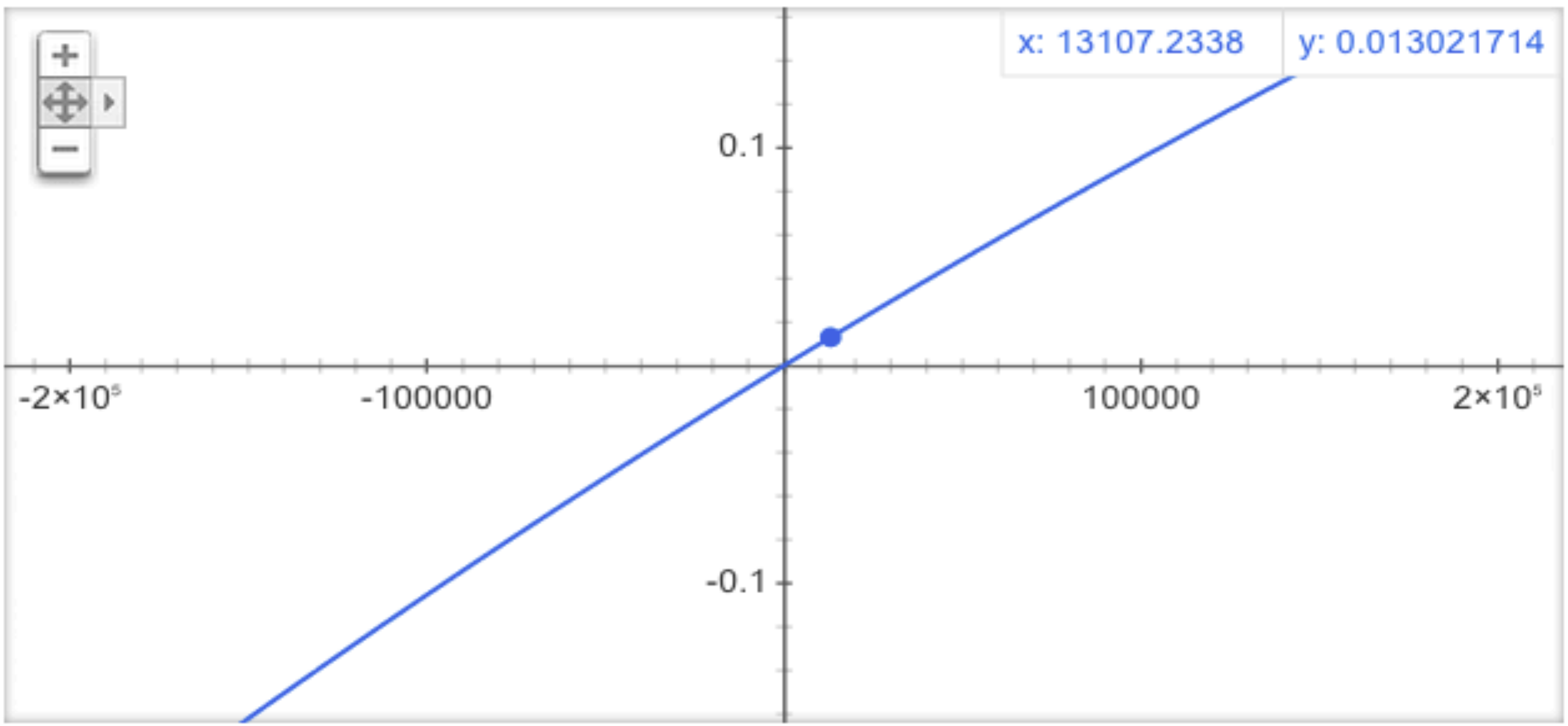
$$\Pr(fp \mid n \text{ prev inserts}) = 1 - \left(1 - \frac{1}{m}\right)^n \Rightarrow 1 - \left(1 - \frac{1}{m^2}\right)^n$$

How well does this work?

$$\Pr(fp \mid n \text{ prev inserts}) = 1 - \left[\left(1 - \frac{1}{m}\right)^2\right]^n$$

m=1,000, x~=13,000, y~=0.01

Graph for 1-(1-(1/1000)^2)^x



Bloom filters

- An example application
  - Finding items in "sharded" data
    - Easy if you know the sharding rule
    - Harder if you don't (like Google n-grams)
- Simple idea:
  - Build a BF of the contents of each shard
  - To look for *key,* load in the BF's one by one, and search only the shards that probably contain *key*
  - Analysis: you won't miss anything, you might look in some extra shards
  - You'll hit O(1) extra shards if you set p=1/#shards

Bloom filters

- An example application
  - discarding rare features from a classifier
  - seldom hurts much, can speed up experiments
- Scan through data once and check each w:
  - if bf1.contains(w):
    - if bf2.contains(w): bf3.add(w)
    - else bf2.add(w)
  - else bf1.add(w)
- Now:
  - bf2.contains(w) ⇔ w appears >= 2x
  - bf3.contains(w) ⇔ w appears >= 3x
- Then train, ignoring words not in bf3

Bloom filters

- Analysis (m bits, k hashers):
  - Assume hash(i,s) is a random function
  - Look at Pr(bit j is unset after n add's): $\left(1-\frac{1}{m}\right)^{kn}$
  - ... and Pr(collision):

$$p = \left(1-\left[1-\frac{1}{m}\right]^{kn}\right)^{k} \approx \left(1-e^{-kn/m}\right)^{k}$$

  - .... fix *m* and *n* and minimize *k:*

$$k = \frac{m}{n}\ln 2 \approx 0.7\frac{m}{n}$$

Bloom filters

- Analysis:
  - Plug optimal k=m/n*ln(2) back into Pr(collision):

$$p = \left(1 - \left[1 - \frac{1}{m}\right]^{kn}\right)^k \approx \left(1 - e^{-kn/m}\right)^k$$

  - Now we can fix any two of *p, n, m* and solve for the 3^{rd}:

$$p = \left(1 - e^{-(m/n \ln 2)n/m}\right)^{(m/n \ln 2)}$$

  - E.g., the value for *m* in terms of *n* and *p:*

$$m = -\frac{n \ln p}{(\ln 2)^2}.$$

Bloom filters: demo

- http://www.jasondavies.com/bloomfilter/

Locality Sensitive Hashing (LSH)

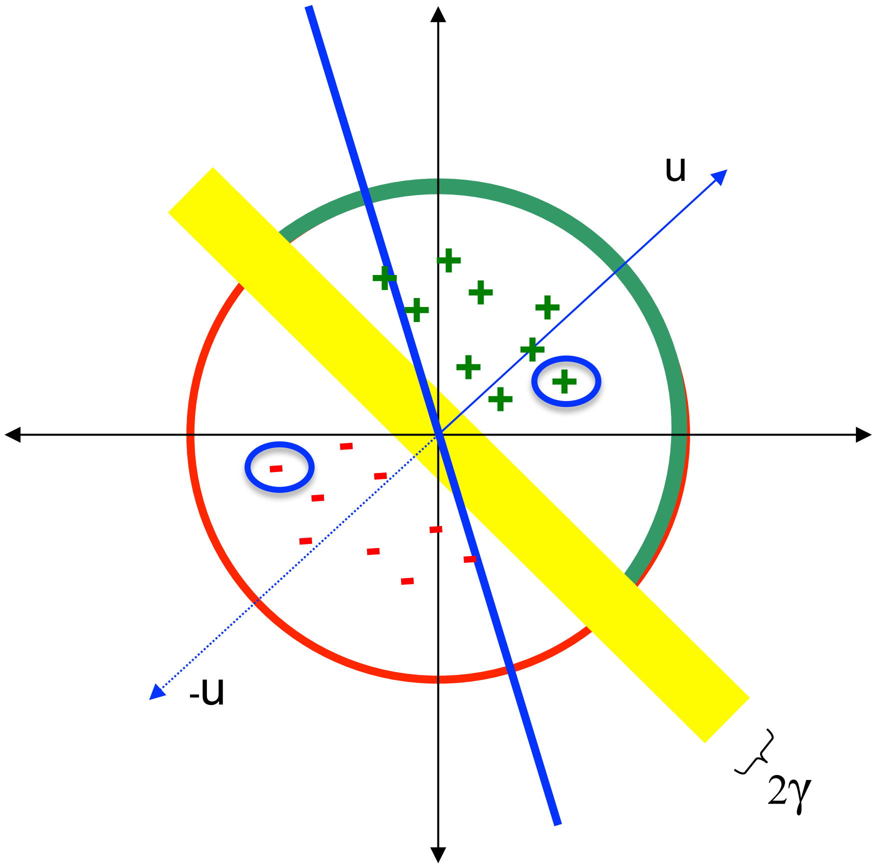
- Two main approaches
  - Random Projection
  - Minhashing

LSH: key ideas

- Goal:
  - map feature vector **x** to bit vector **bx**
  - ensure that **bx** preserves “similarity”

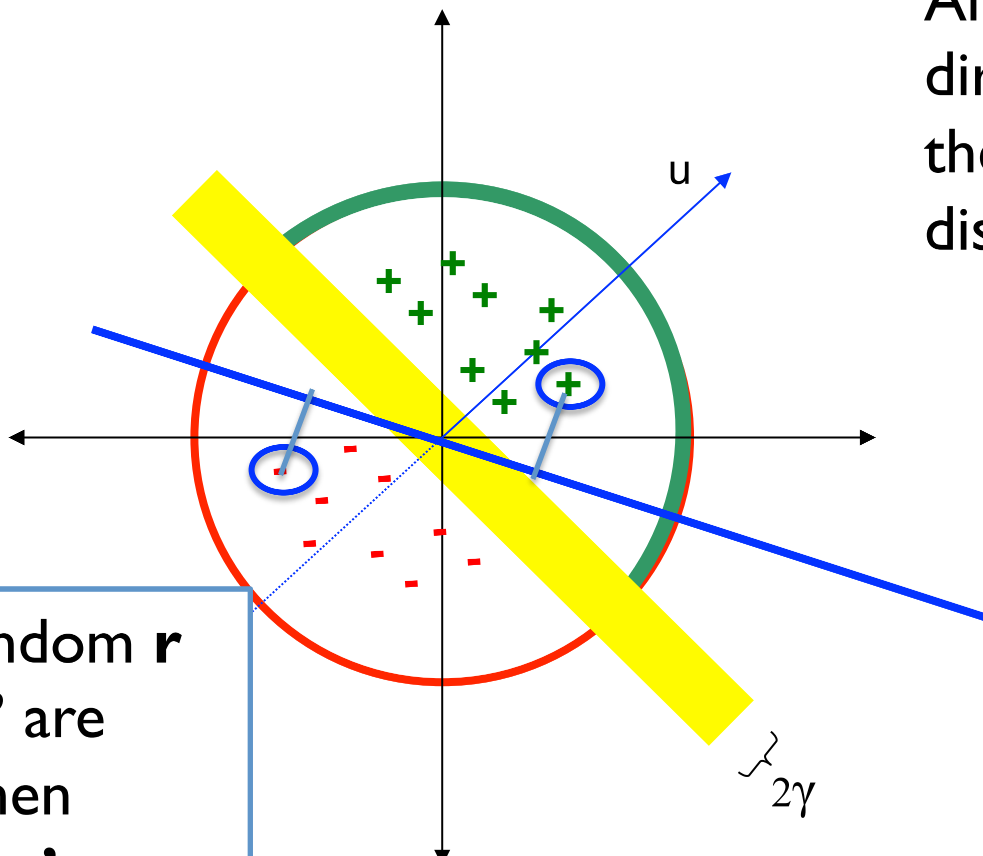
Random Projections

Random projections



To make those points “close” we need to project to a direction orthogonal to the line between them

Random projections



Any other direction will keep the distant points distant.

So if I pick a random **r** and **r.x** and **r.x'** are closer than γ then probably **x** and **x'** were close to start with.

LSH: key ideas

- Goal:
  - map feature vector **x** to bit vector **bx**
  - ensure that **bx** preserves “similarity”
- Basic idea: use random projections of **x**
  - Repeat many times:
    - Pick a random hyperplane **r**
    - Compute the inner product of **r** with **x**
    - Record if **x** is “close to” **r** (**r.x**>=0)
      - the next bit in **bx**
    - Theory says that is **x’** and **x** have small cosine distance then **bx** and **bx’** will have small Hamming distance

LSH: key ideas

- Naïve algorithm:
  - Initialization:
    - For i=1 to outputBits:
      - For each feature *f*:
        - » Draw r(f,i) ~ Normal(0,1)
  - Given an instance **x**
    - For i=1 to outputBits:

      LSH[i] =

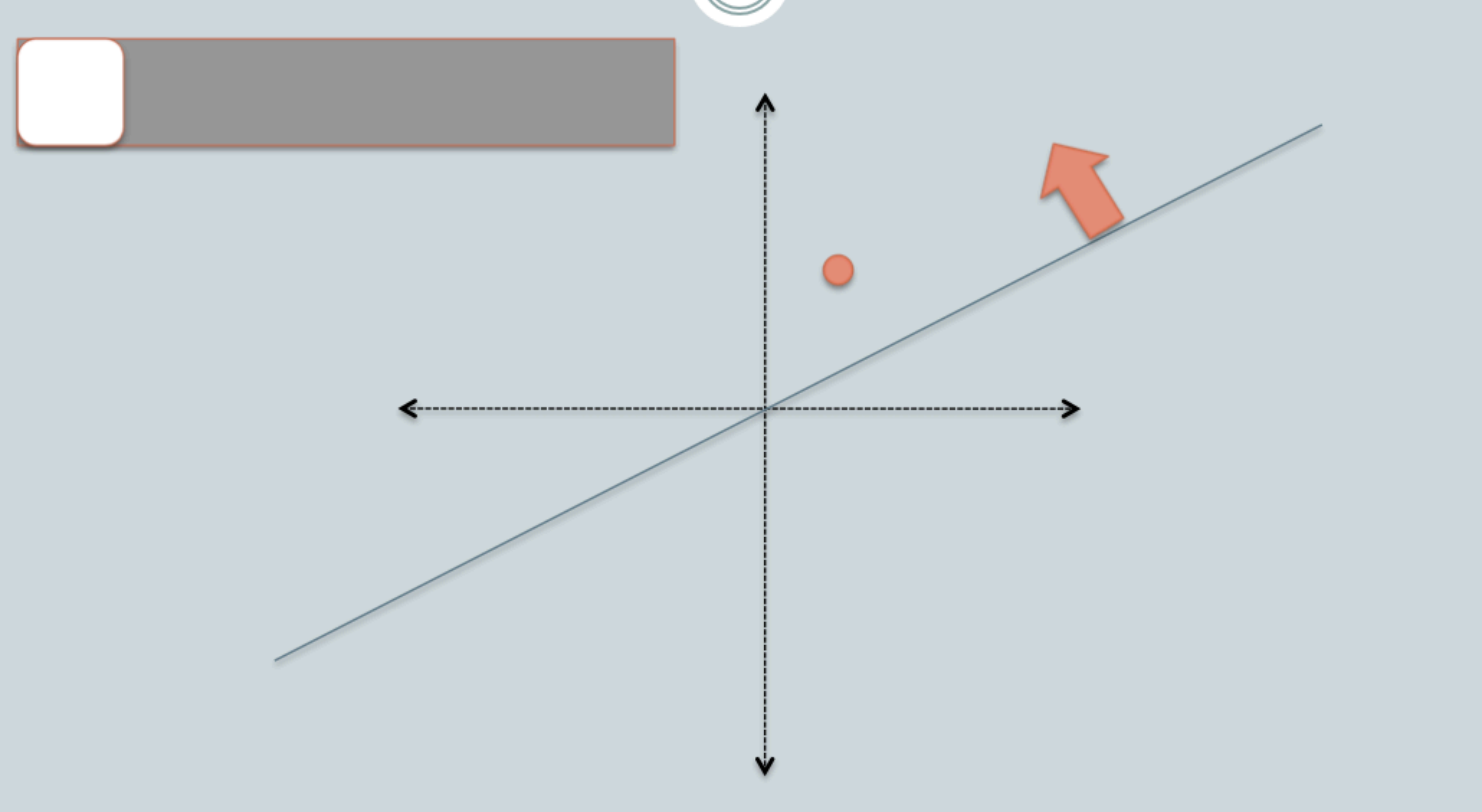
      sum(**x**[*f*]*r[i,*f*] for *f* with non-zero weight in **x**) > 0 ? 1 : 0
    - Return the bit-vector LSH
  - Problem:
    - the array of r's is very large

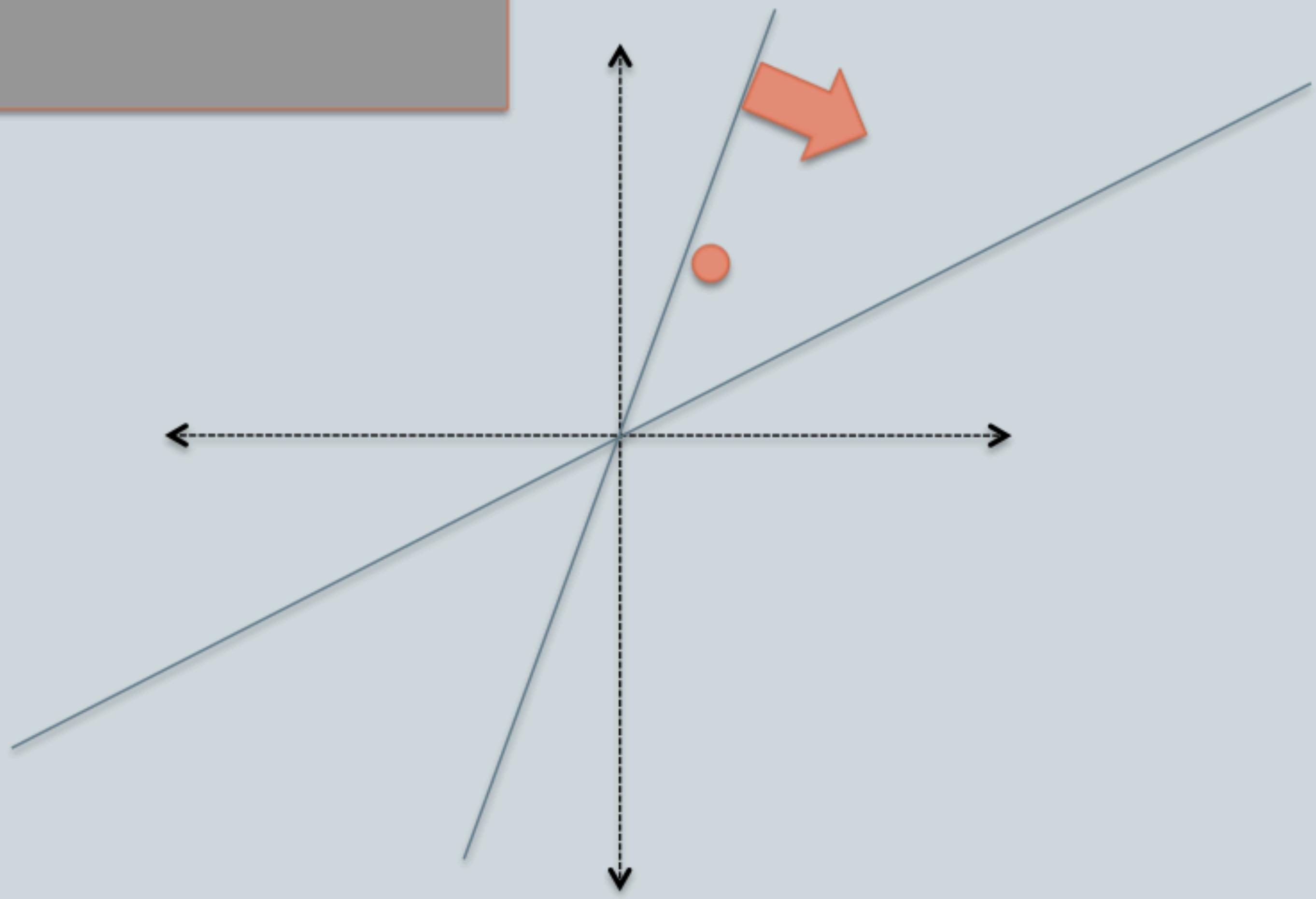
Online Generation of Locality Sensitive Hash Signatures

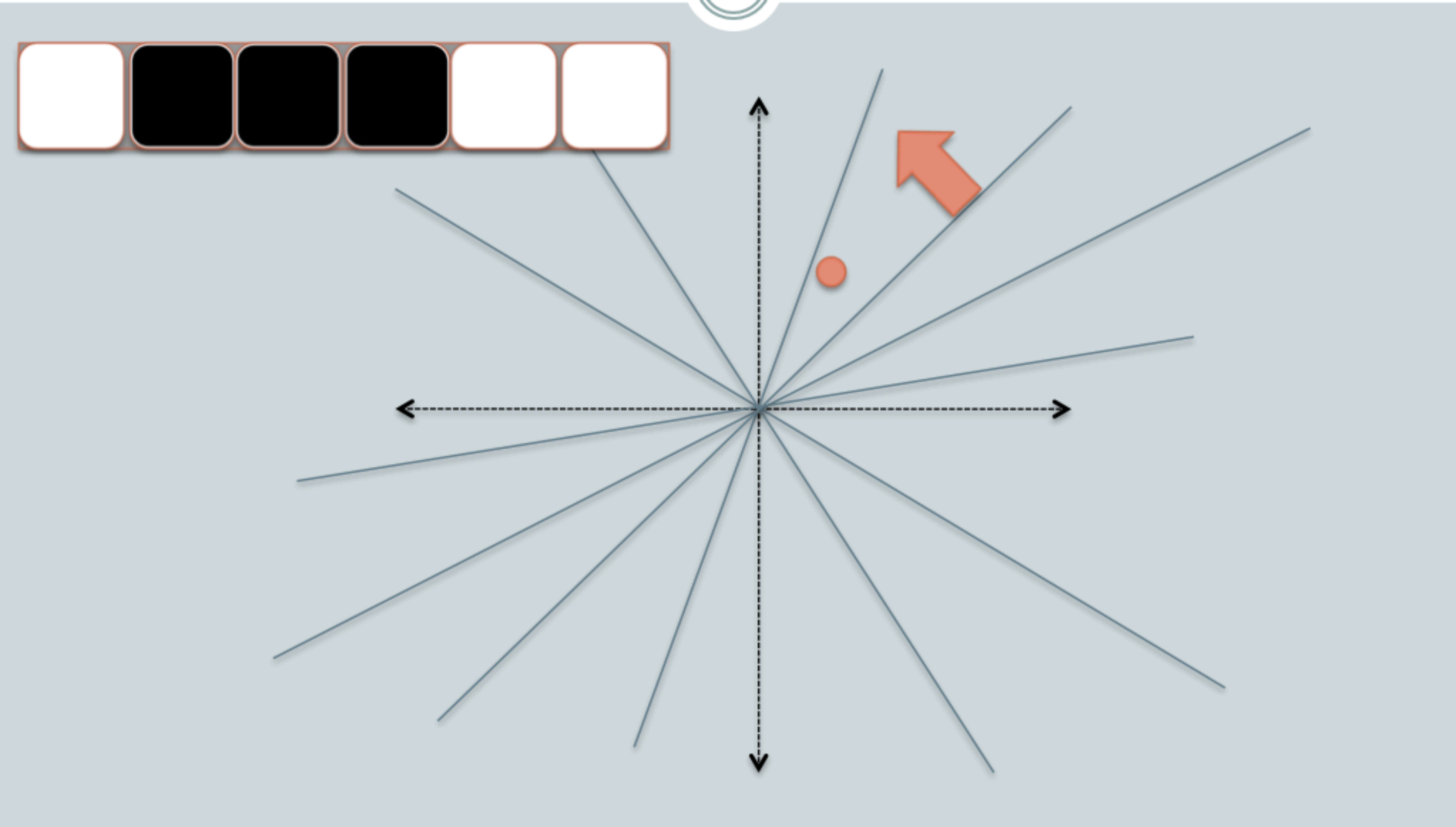
Benjamin Van Durme and Ashwin Lall

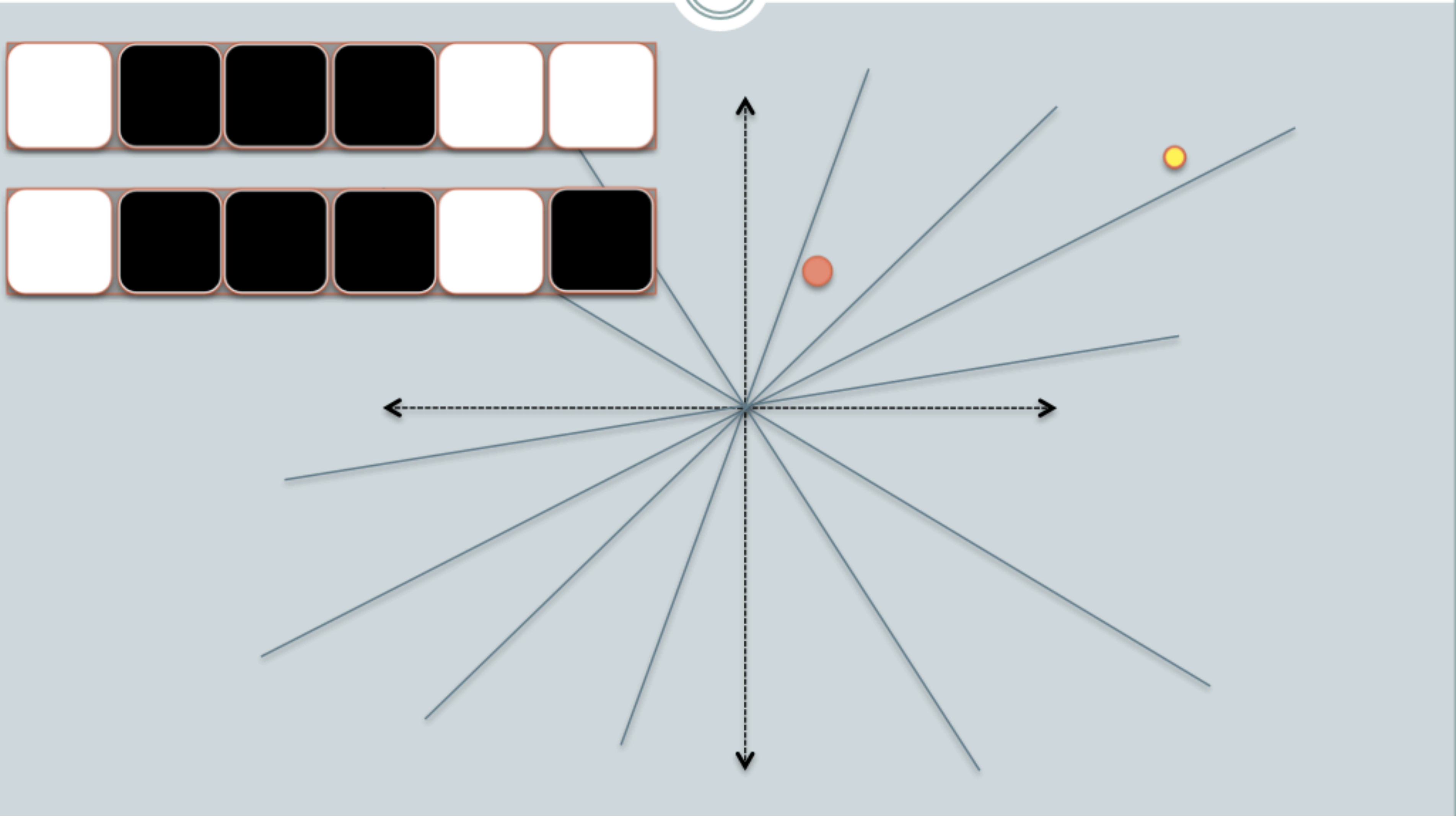
human language technology
center of excellence
JOHNS HOPKINS
UNIVERSITY

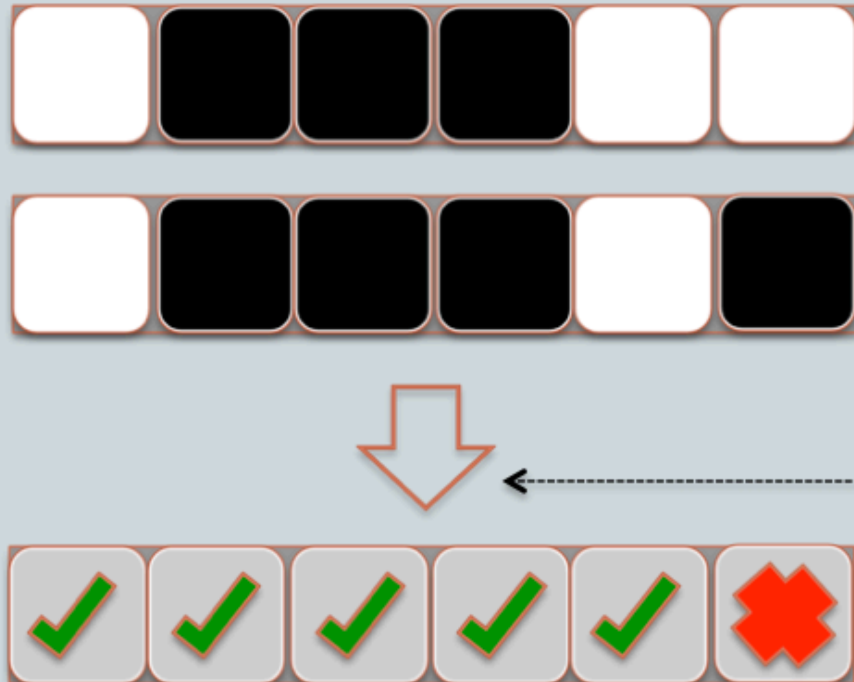
DENISON
UNIVERSITY

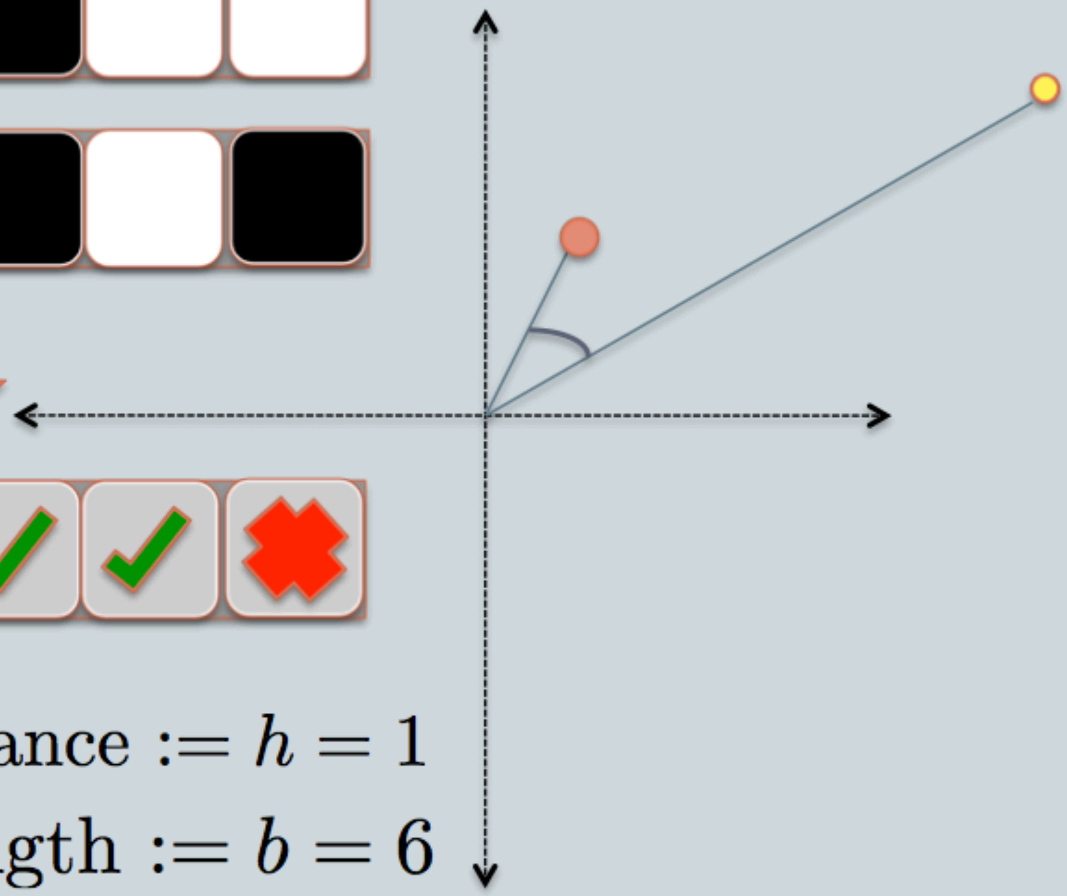










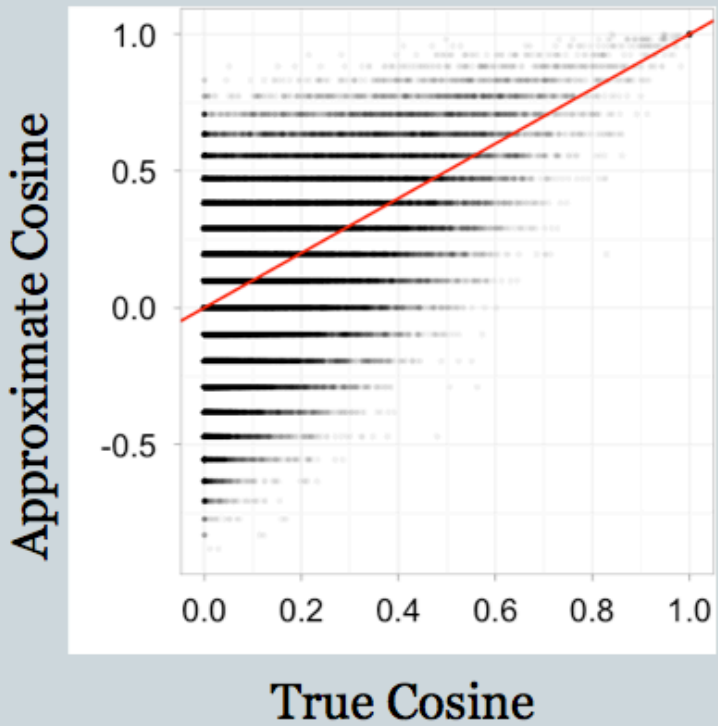


Hamming Distance $:= h = 1$

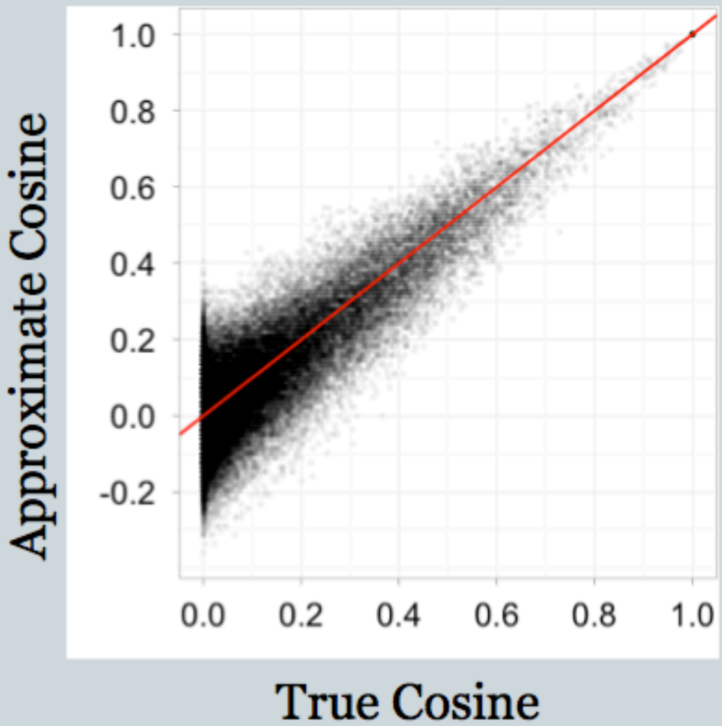
Signature Length $:= b = 6$

$$\cos(\theta) \approx \cos\left(\frac{h}{b}\pi\right) = \cos\left(\frac{1}{6}\pi\right)$$

32 bit signatures



256 bit signatures

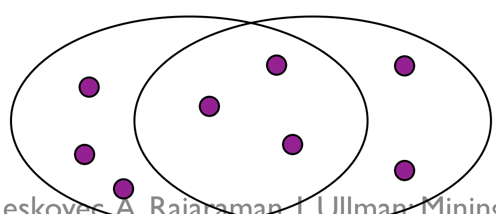


Cheap

Accurate

Distance Measures

- Goal: **Find near-neighbors in high-dim. space**
  - We formally define "near neighbors" as points that are a "small distance" apart
- For each application, we first need to define what "**distance**" means
- Today: **Jaccard distance/similarity**
  - The Jaccard similarity of two **sets** is the size of their intersection divided by the size of their union:
    $sim(C_1, C_2) = |C_1 \cap C_2| / |C_1 \cup C_2|$
  - Jaccard distance: $d(C_1, C_2) = 1 - |C_1 \cap C_2| / |C_1 \cup C_2|$



3 in intersection
8 in union
Jaccard similarity= 3/8
Jaccard distance = 5/8

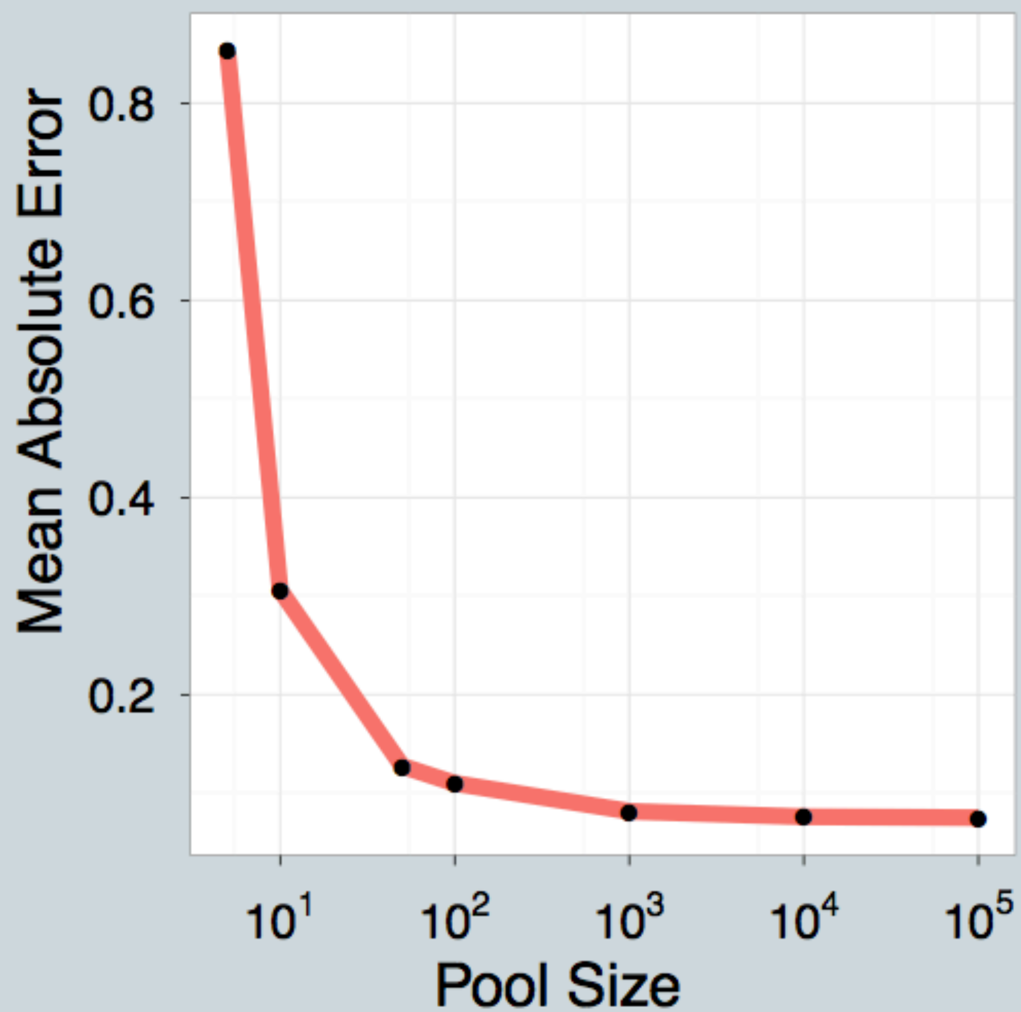
LSH: “pooling” (van Durme)

- Better algorithm:
  - Initialization:
    - Create a pool:
      - Pick a random seed *s*
      - For i=1 to poolSize:
        - » Draw pool[i] ~ Normal(0,1)
    - For i=1 to outputBits:
      - Devise a random hash function hash(i,*f*):
        - » E.g.: hash(i,f) = hashcode(f) XOR randomBitString[i]
  - Given an instance **x**
    - For i=1 to outputBits:

      LSH[i] = sum(

      x[f] * pool[hash(i,f) % poolSize] for *f* in **x**) > 0 ? 1 : 0
    - Return the bit-vector LSH

The Pooling Trick



LSH: key ideas: pooling

- Advantages:
  - with pooling, this is a compact re-encoding of the data
    - you don't need to store the **r**'s, just the pool
  - leads to very fast nearest neighbor method
    - just look at other items with **bx'=bx**
    - also very fast nearest-neighbor methods for Hamming distance
  - similarly, leads to very fast clustering
    - cluster = all things with same **bx** vector

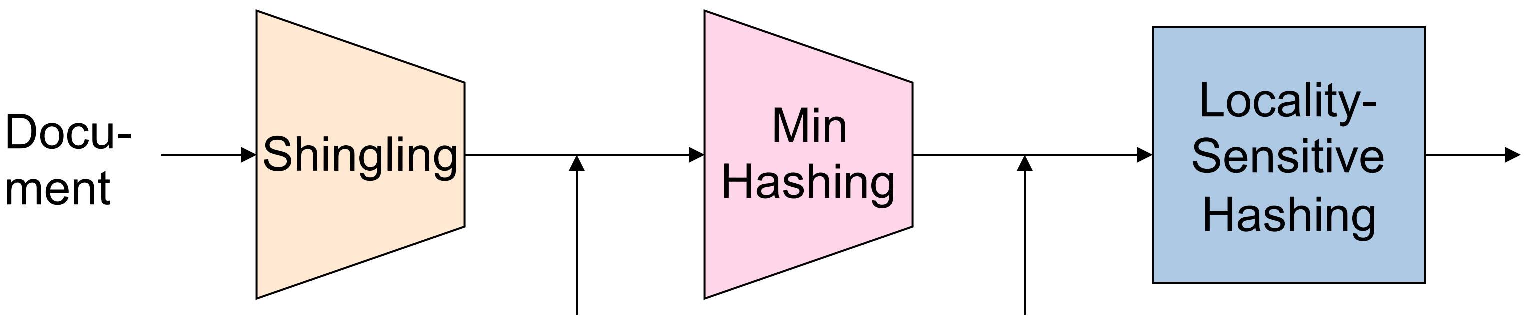
Finding Similar Documents with Minhashing

- **Goal:** Given a large number (in the millions or billions) of documents, find "near duplicate" pairs
- **Applications:**
  – Mirror websites, or approximate mirrors
    - Don't want to show both in search results
  – Similar news articles at many news sites
    - Cluster articles by "same story"
- **Problems:**
  – Many small pieces of one document can appear out of order in another
  – Too many documents to compare all pairs
  – Documents are so large or so many that they cannot fit in main memory

3 Essential Steps for Similar Docs

1. *Shingling:* Convert documents to sets
2. *Min-Hashing:* Convert large sets to short signatures, while preserving similarity
3. *Locality-Sensitive Hashing:* Focus on pairs of signatures likely to be from similar documents
   – Candidate pairs!

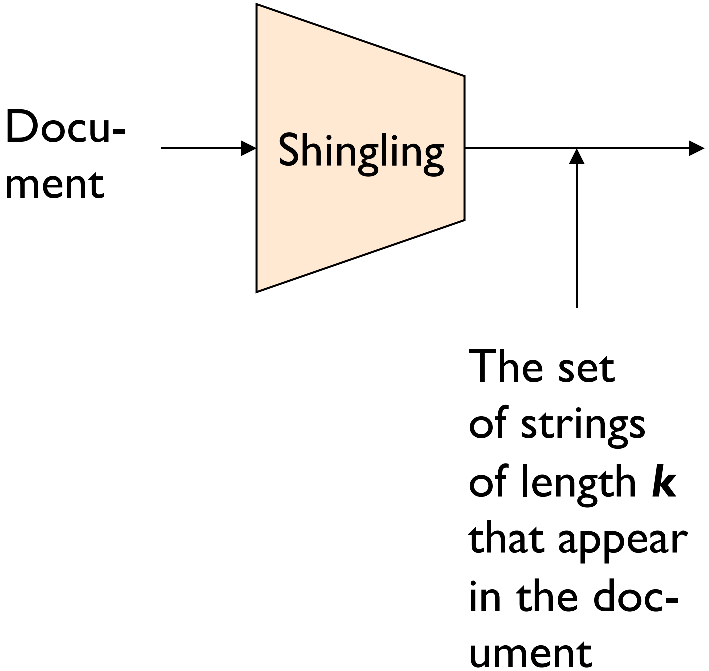
The Big Picture



The set of strings of length ***k*** that appear in the doc- ument

Signatures: short integer vectors that represent the sets, and reflect their similarity

Candidate pairs: those pairs of signatures that we need to test for similarity



Shingling

Step 1: ***Shingling:*** Convert documents to sets

Define: Shingles

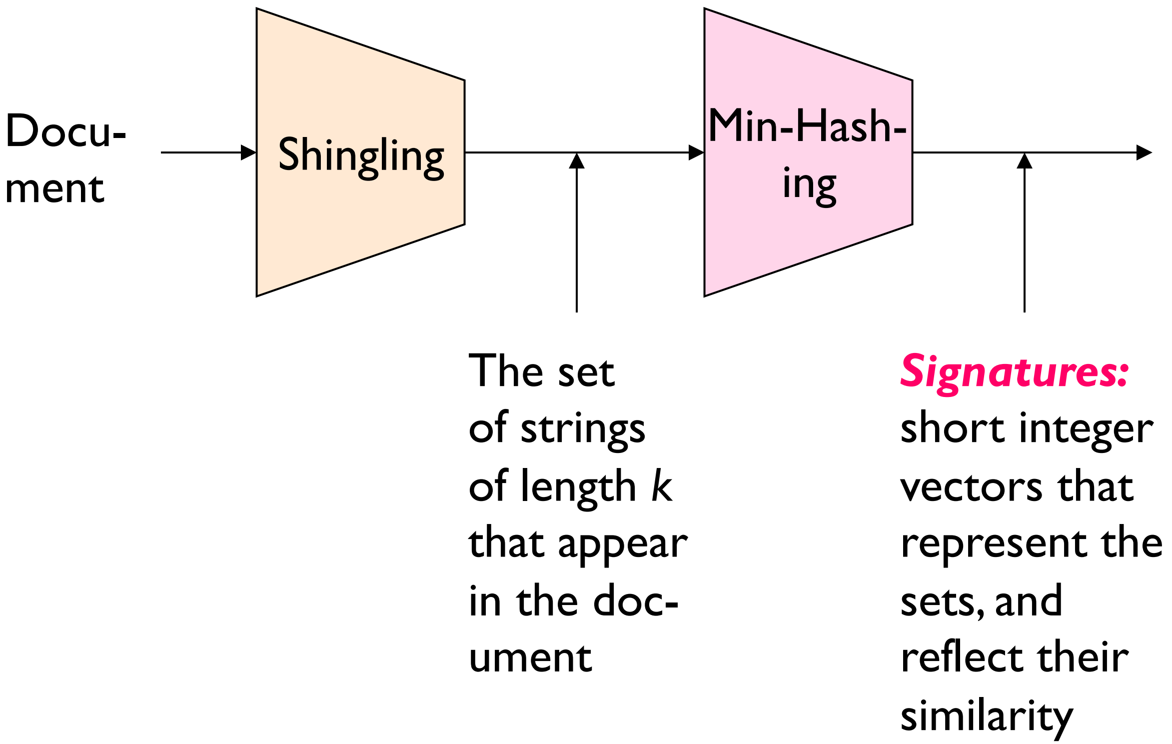
- A *k*-shingle (or *k*-gram) for a document is a sequence of *k* tokens that appears in the doc
  - Tokens can be characters, words or something else, depending on the application
  - Assume tokens = characters for examples

- Example: **k=2**; document $\mathbf{D_1}$ = abcab
  Set of 2-shingles: $\mathbf{S(D_1)}$ = {ab, bc, ca}
  - Option: Shingles as a bag (multiset), count ab twice: $\mathbf{S'(D_1)}$ = {ab, bc, ca, ab}

Working Assumption

- Documents that have lots of shingles in common have similar text, even if the text appears in different order

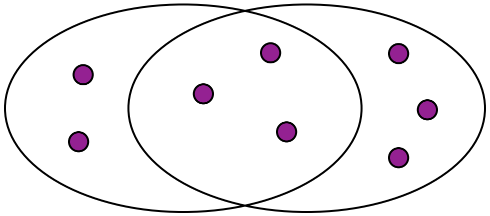
- Caveat: You must pick k large enough, or most documents will have most shingles
  - $k = 5$ is OK for short documents
  - $k = 10$ is better for long documents



MinHashing

Step 2: ***Minhashing:*** Convert **large sets** to **short signatures**, while **<u>preserving similarity</u>**

Encoding Sets as Bit Vectors



- Many similarity problems can be formalized as **finding subsets that have significant intersection**
- **Encode sets using 0/1 (bit, boolean) vectors**
  - One dimension per element in the universal set
- Interpret set intersection as bitwise **AND**, and set union as bitwise **OR**

- **Example:** $C_1 = 10111$; $C_2 = 10011$
  - Size of intersection = **3**; size of union = **4**,
  - **Jaccard similarity** (not distance) = **3/4**
  - **Distance: d(C_1,C_2) = 1 – (Jaccard similarity) = 1/4**

From Sets to Boolean Matrices

- **Rows** = elements (shingles)
- **Columns** = sets (documents)
  - 1 in row ***e*** and column ***s*** if and only if ***e*** is a member of ***s***
  - Column similarity is the Jaccard similarity of the corresponding sets (rows with value *1)*
  - Typical matrix is sparse!
- **Each document is a column:**
  - Example: $\text{sim}(C_1, C_2) = ?$
    - Size of intersection = 3; size of union = 6, Jaccard similarity (not distance) = 3/6
    - $d(C_1,C_2) = 1 - (\text{Jaccard similarity}) = 3/6$

Documents

Shingles

1	1	1	0
1	1	0	1
0	1	0	1
0	0	0	1
1	0	0	1
1	1	1	0
1	0	1	0

Min-Hashing

- **Goal:** **Find a hash function $h(\cdot)$ such that:**
  - if $sim(C_1,C_2)$ is high, then with high prob. $h(C_1) = h(C_2)$
  - if $sim(C_1,C_2)$ is low, then with high prob. $h(C_1) \neq h(C_2)$

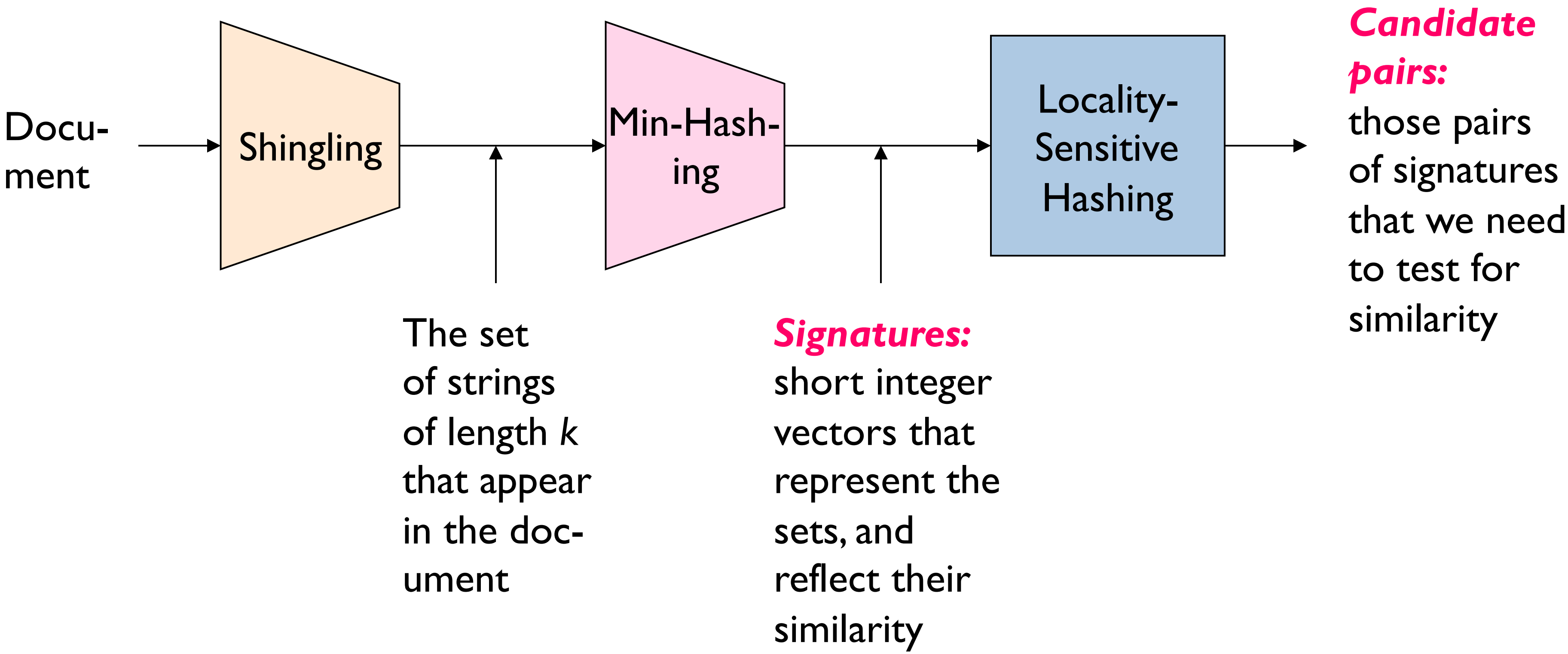
- **Clearly, the hash function depends on the similarity metric:**
  - Not all similarity metrics have a suitable hash function
- **There is a suitable hash function for the Jaccard similarity:** It is called **Min-Hashing**

Min-Hashing

- Imagine the rows of the boolean matrix permuted under **random permutation π**
- Define a **"hash" function $h_\pi(C)$** = the index of the **first** (in the permuted order π) row in which column C has value **1**:

$$h_\pi(C) = \min_\pi \pi(C)$$

- Use several (e.g., 100) independent hash functions (that is, permutations) to create a signature of a column



Locality Sensitive Hashing

Step 3: ***Locality-Sensitive Hashing:***
Focus on pairs of signatures likely to be from

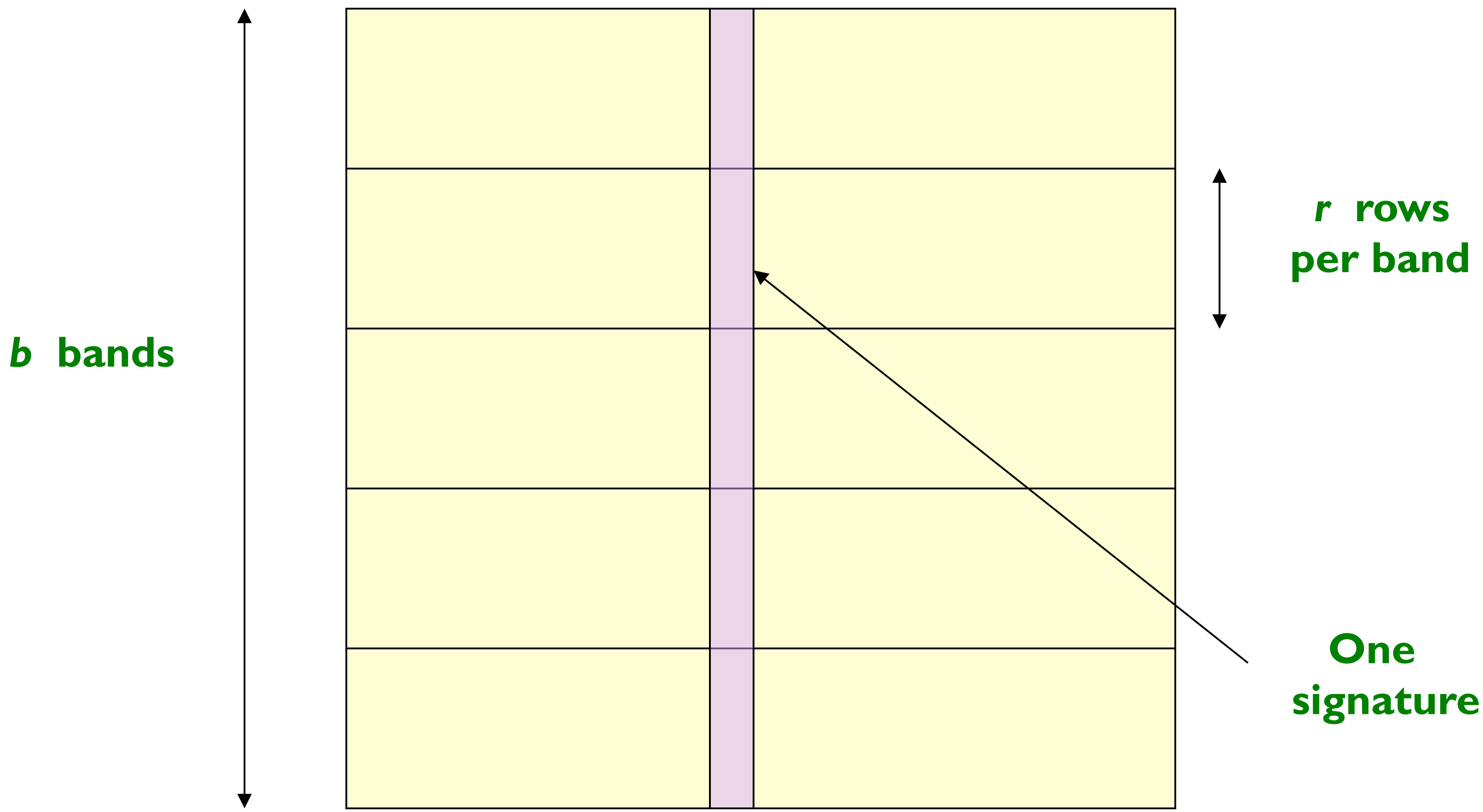
LSH: First Cut

2	1	4	1
1	2	1	2
2	1	2	1

- **Goal:** Find documents with Jaccard similarity at least ***s*** (for some similarity threshold, e.g., ***s***=0.8)
- **LSH – General idea:** Use a function ***f(x,y)*** that tells whether ***x*** and ***y*** is a ***candidate pair:*** a pair of elements whose similarity must be evaluated
- **For Min-Hash matrices:**
  - Hash columns of signature matrix ***M*** to many buckets
  - Each pair of documents that hashes into the same bucket is a candidate pair

Partition *M* into *b* Bands

2	1	4	1
1	2	1	2
2	1	2	1

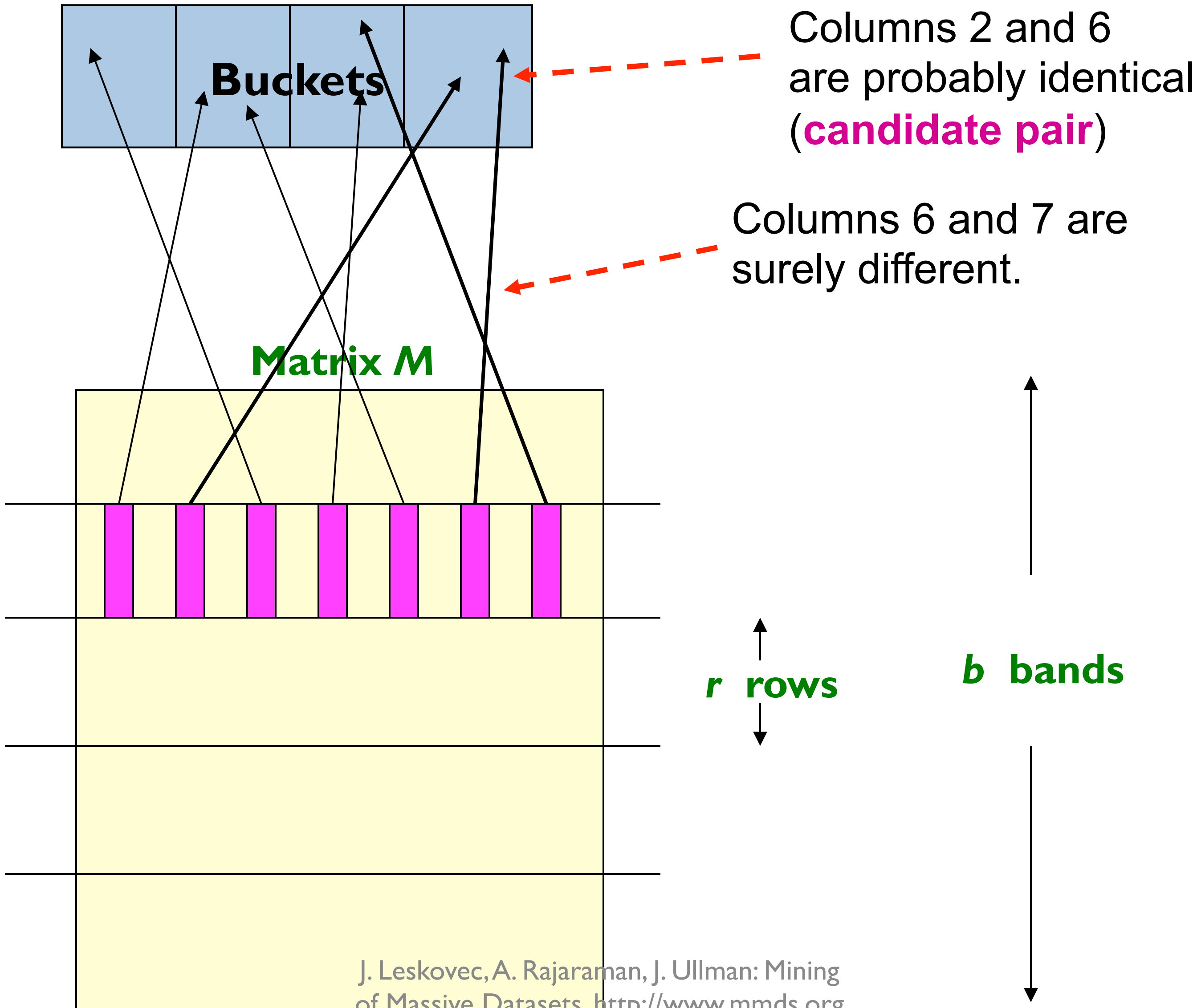


Signature matrix *M*

Partition M into Bands

- Divide matrix M into b bands of r rows
- For each band, hash its portion of each column to a hash table with k buckets
  - Make k as large as possible
- *Candidate* column pairs are those that hash to the same bucket for ≥ 1 band
- Tune b and r to catch most similar pairs, but few non-similar pairs

Hashing Bands



2	1	4	1
1	2	1	2
2	1	2	1

Example of Bands

Assume the following case:

- Suppose 100,000 columns of M (100k docs)
- Signatures of 100 integers (rows)
- Therefore, signatures take 40Mb
- Choose $b = 20$ bands of $r = 5$ integers/band

- **Goal:** Find pairs of documents that are at least $s = 0.8$ similar

C_1, C_2 are 80% Similar

2	1	4	1
1	2	1	2
2	1	2	1

- **Find pairs of ≥ *s*=0.8 similarity, set b=20, r=5**
- **Assume:** sim(C_1, C_2) = 0.8
  - Since sim(C_1, C_2) ≥ **s**, we want C_1, C_2 to be a **candidate pair**: We want them to hash to at **least 1 common bucket** (at least one band is identical)
- **Probability C_1, C_2 identical in one particular band:** $(0.8)^5 = 0.328$
- Probability C_1, C_2 are ***not*** similar in all of the 20 bands: $(1\text{-}0.328)^{20} = 0.00035$
  - i.e., about 1/3000th of the 80%-similar column pairs are **false negatives** (we miss them)
  - **We would find 99.965% pairs of truly similar documents**

C_1, C_2 are 30% Similar

2	1	4	1
1	2	1	2
2	1	2	1

- **Find pairs of ≥** s=0.8 similarity, set **b**=20, **r**=5
- **Assume:** sim(C_1, C_2) = 0.3
  - Since sim(C_1, C_2) < **s** we want C_1, C_2 to hash to **NO common buckets** (all bands should be different)
- **Probability C_1, C_2 identical in one particular band:** $(0.3)^5 = 0.00243$
- Probability C_1, C_2 identical in at least 1 of 20 bands: $1 - (1 - 0.00243)^{20} = 0.0474$
  - In other words, approximately 4.74% pairs of docs with similarity 0.3% end up becoming **candidate pairs**
    - They are **false positives** since we will have to examine them (they are candidate pairs) but then it will turn out their similarity is below threshold **s**

LSH Involves a Tradeoff

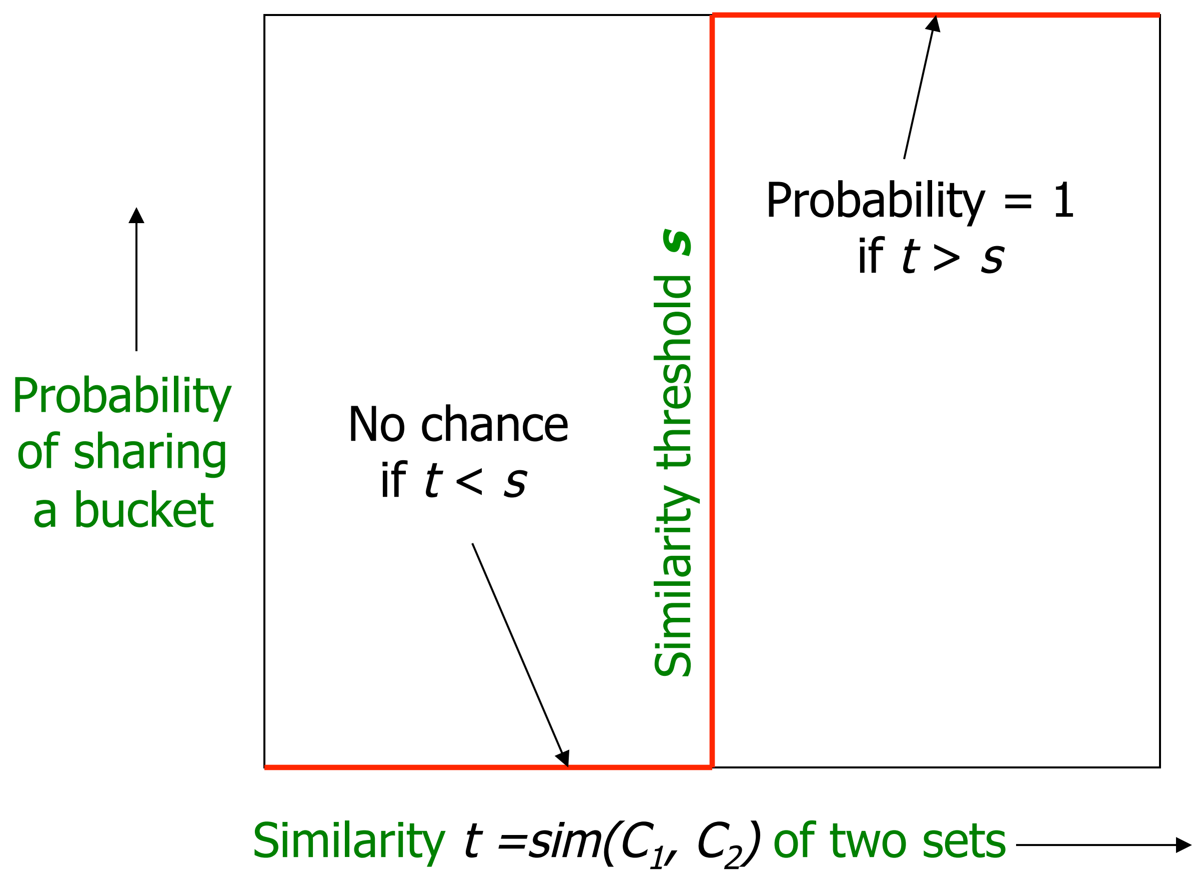
2	1	4	1
1	2	1	2
2	1	2	1

- Pick:
  - The number of Min-Hashes (rows of ***M***)
  - The number of bands ***b***, and
  - The number of rows ***r*** per band

  to balance false positives/negatives

- Example: If we had only 15 bands of 5 rows, the number of false positives would go down, but the number of false negatives would go up

Analysis of LSH – What We Want



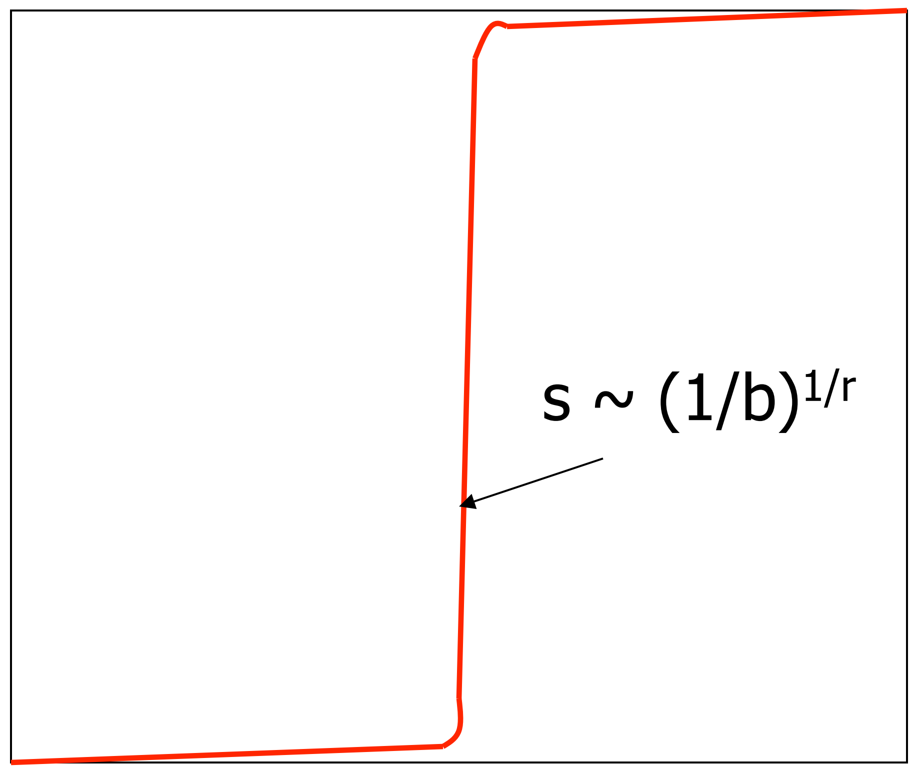
b bands, *r* rows/band

- Columns C_1 and C_2 have similarity t
- Pick any band (r rows)
  - Prob. that all rows in band equal = t^r
  - Prob. that some row in band unequal = $1 - t^r$
- Prob. that no band identical = $(1 - t^r)^b$
- Prob. that at least 1 band identical =

  $1 - (1 - t^r)^b$

What *b* Bands of *r* Rows Gives You

Probability of sharing a bucket



Similarity $t = sim(C_1, C_2)$ of two sets

At least one band identical

No bands identical

$$\mathbf{1 - (1 - t^{r})^{b}}$$

Some row of a band unequal

All rows of a band are equal

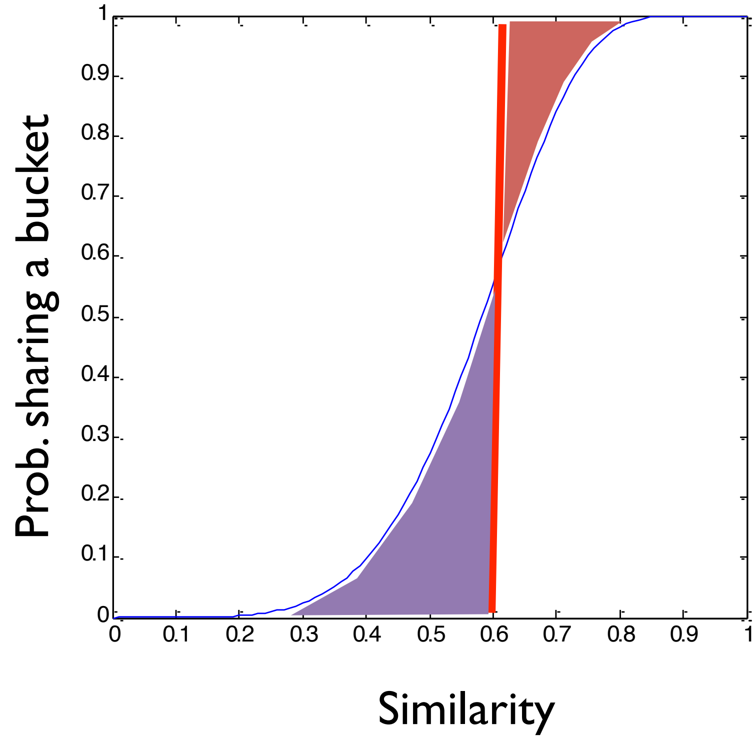
Example: $b = 20$; $r = 5$

- Similarity threshold s
- Prob. that at least 1 band is identical:

s	$1-(1-s^r)^b$
.2	.006
.3	.047
.4	.186
.5	.470
.6	.802
.7	.975
.8	.9996

Picking *r* and *b*: The S-curve

- Picking *r* and *b* to get the best S-curve
  - 50 hash-functions (r=5, b=10)



Red area: False Negative rate
Purple area: False Positive rate