ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Data Mining
Learning from Large Data Sets

Lecture 2 – Approximate Retrieval

263-5200-00L

Andreas Krause

Announcement

- Homework 1 out by tomorrow
- Self-assessment test posted online

- Recitations
  - *Tue 13-14, CAB G61*. Last names starting with A-L
  - *Fri 14-15, NO C6.* Last names starting with M-Z

Data Mining Goals

- **Approximate retrieval**
  - Given a query, find "most similar" items in a large data set
  - *Applications*: GoogleGoggles, Shazam, …
- **Supervised learning** (Classification, Regression)
  - Learn a concept (function mapping queries to labels)
  - *Applications*: Spam filtering, predicting relevance…
- **Unsupervised learning** (Clustering, dimension reduction)
  - Identify clusters, regularities, …
  - *Applications*: Explorative analysis, Anomaly detection, …
- **Recommender Systems / interactive learning**
  - Learning from limited feedback / through experimentation
  - *Applications*: Online advertising, learning rankings, …

Today:

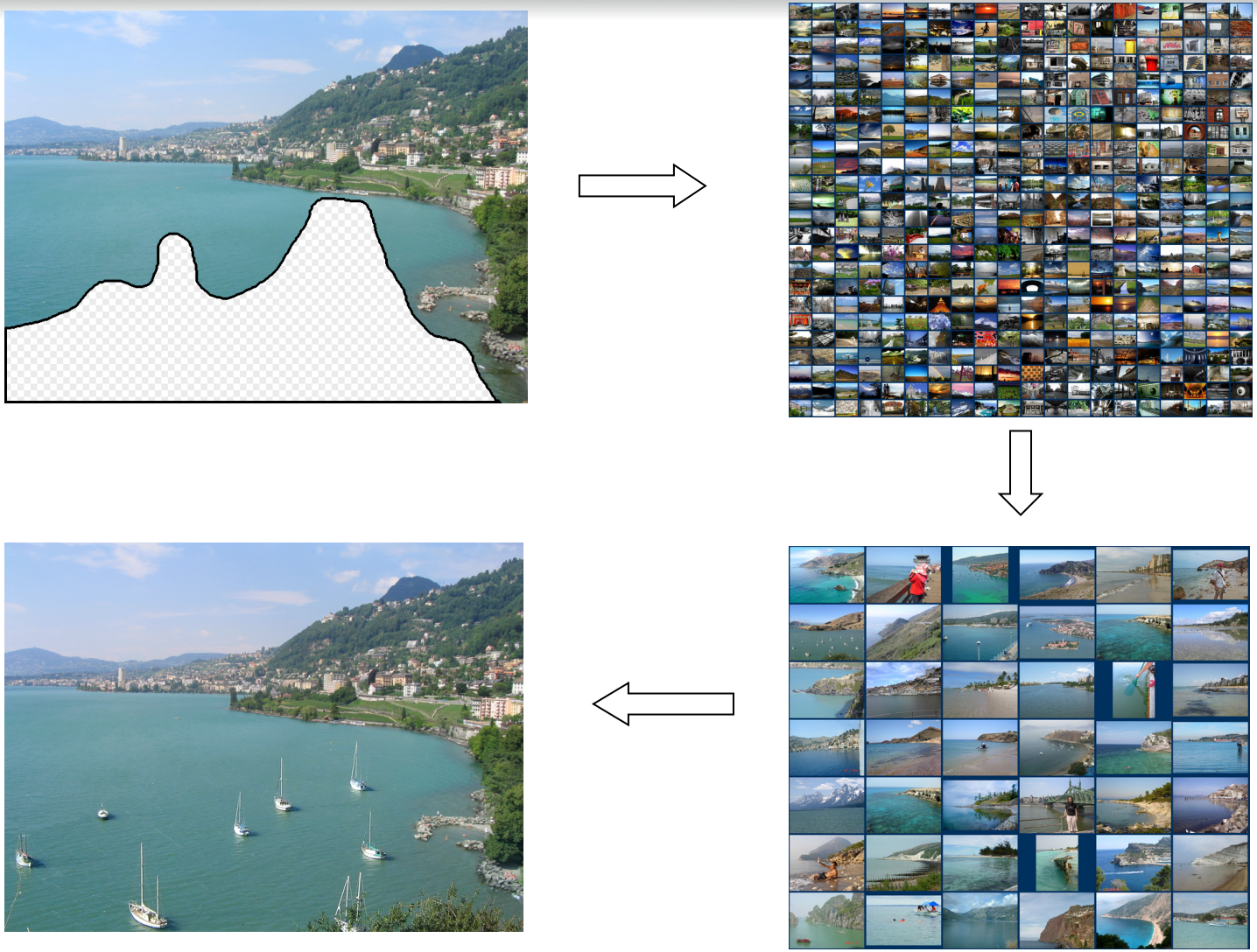
How can one quickly find nearest neighbors in high dimensions?

Application: Multimedia retrieval

Google Goggles

shazam
midomi

Application: Image completion

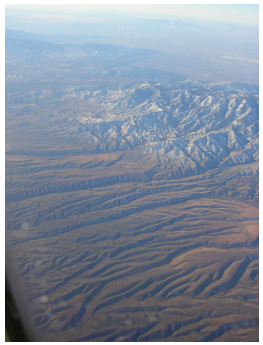


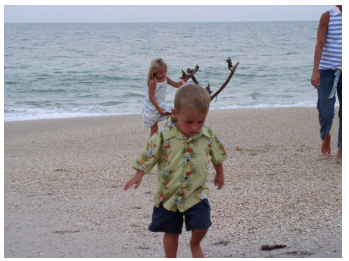
[Hays and Efros, SIGGRAPH 2007]

Near(est)-neighbor search

Closer

Far





How can we find the nearest neighbor to a query?
How can we find all near duplicates?

Properties of distance fn's

A function $d : S \times S \to \mathbb{R}$

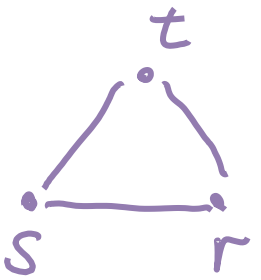
is called a distance function if it satisfies

$$\forall s, t \in S : d(s,t) \geq 0$$

$$\forall s : d(s,s) = 0$$

$$\forall s, t : d(s,t) = d(t,s)$$ Symmetry

$$\forall s, t, r : d(s,t) + d(t,r) \geq d(s,r)$$ Triangle inequality



Representing objects as vectors

[.3 .01 .1 2.3 0 0 1.1 …]

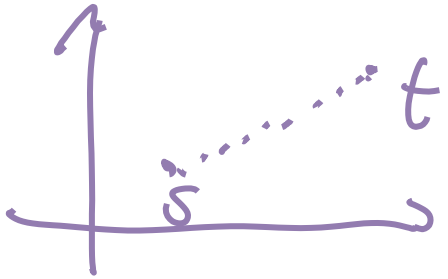
The quick brown fox jumps over the lazy dog …

[0 1 0 0 0 3 2 0 1 0 0 0]

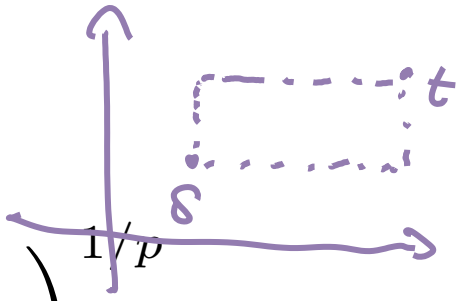
- Often, represent objects as vectors
  - Bag of words for documents; tf-idf; …
  - Feature vectors for images (SIFT, GIST, PHOG, etc.)
  - …
- Allows to use the same distances / same algorithms for different object types

Examples: Distance of vectors in R^D

- ℓ_2 (Euclidean) distance $\sqrt{\sum_{i=1}^{D}(x_i - x'_i)^2}$



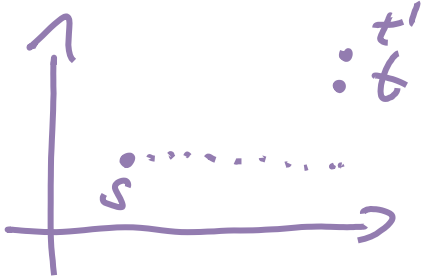
- ℓ_1 (Manhattan) distance $\sum_{i=1}^{D}|x_i - x'_i|$



- ℓ_p distances: $d_p(x, x') = \left(\sum_{i=1}^{D}|x_i - x'_i|^p\right)$

$\|x\|_p = d_p(0, x)$

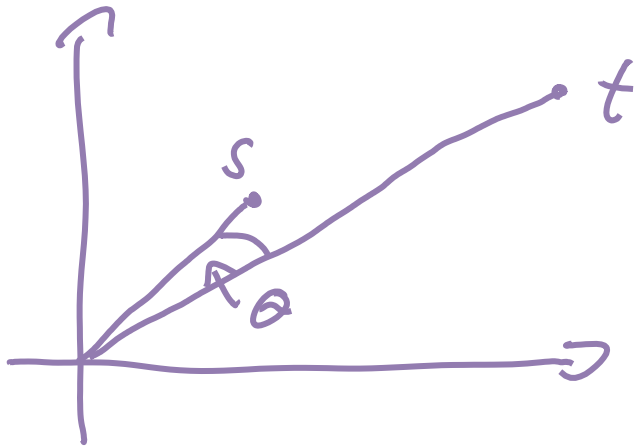
- ℓ_∞ distance: $d_\infty(x, x') = \max_i |x_i - x'_i|$



Example: Cosine distance

- Cosine distance

$$d(x, x') = \arccos \frac{x^T x'}{||x||_2 ||x'||_2}$$



More complex example: Edit distance

Edit distance: How many inserts and deletes are necessary to transform one string to another?

Example:

- d("The quick brown fox","The quikc brwn fox") = 3
- d("GATTACA","ATACAT")

- Allows various extensions (mutations; reversal; …)
- Can compute in polynomial time, but expensive for large texts

➔ We will focus on vector representation

Approximate retrieval

- Input: data set S, distance function d

- Problem 1: Nearest neighbor
  - Given query q in S, find nearest s* in S

$$s^* \in \operatorname*{argmin}_{s \in S} d(q, s)$$

- Problem 2: Near-duplicate detection
  - Find all s,s' in S with distance at most ε

$$C = \{ (s, s') \in S \times S : d(s, s') \leq \varepsilon \}$$

Finding nearest neighbors in low-D

- Naive approach:
  - Compute distance of query to all examples
  - Retrieve closest one
  - doesn‘t scale for massive problems
- In low dimensions, finding near(est) neighbors is relatively simple
- Specialized data structures
  - k-d-Trees
  - R-Trees
  - …
- Not focus of this course

Many real-world problems are high-dimensional

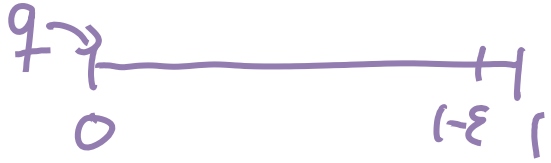
- Text on the web
  - Billions of documents, millions of terms
  - In *bag of words* representation, each term is a dimension..
- Scene completion, image classification, …
  - Large # of image features
- Scientific data
  - Imaging / other high-dimensional measurements
- Product recommendations and advertising
  - Millions of customers, millions of products
  - Traces of behavior (websites visited, searches, …)

Curse of dimensionality

- Suppose we would like to find neighbors B of maximum distance at most 1- ε in $[0,1]^D$
- Suppose we have N data points sampled uniformly at random from $[0,1]^D$

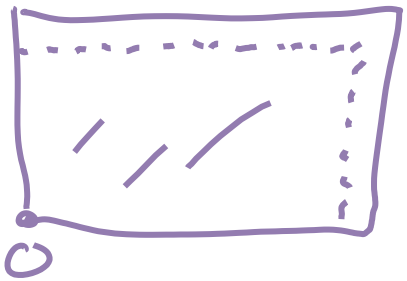
$d_\infty(x,x) = \max_i |x_i - y_i|$

D = 1



$E[B] = (1-\varepsilon)\,N$

D = 2



$E[B] = (1-\varepsilon)^2\,N$

D > 2

$E[B] = (1-\varepsilon)^D\,N \to 0$

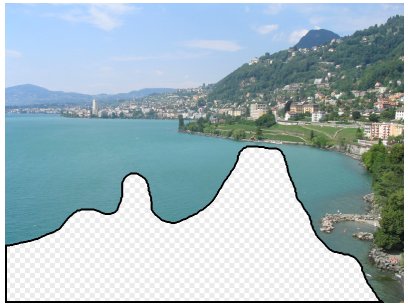
Curse of dimensionality

- **Theorem [Beyer et al. '99]** Fix ε>0 and N. If data is "truly" high dimensional, then, under fairly weak assumptions on the distribution of the data

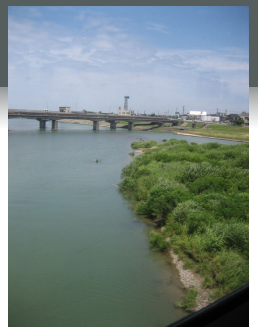
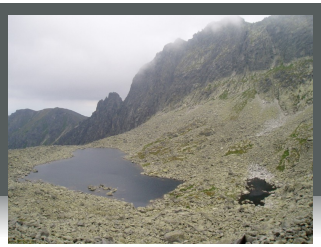
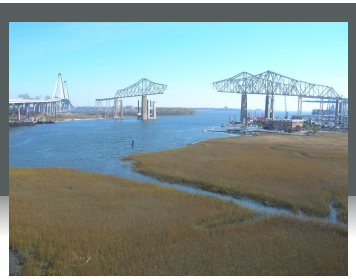
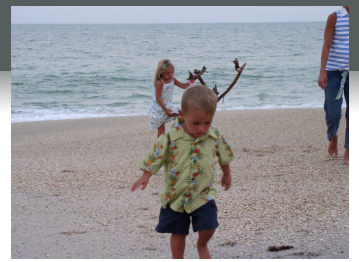
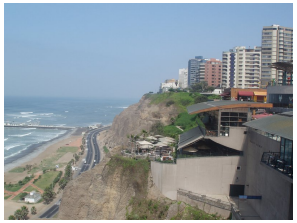
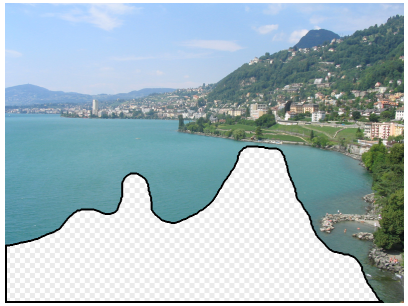
$$\lim_{D \to \infty} P[d_{\max}(N, D) \leq (1 + \varepsilon) d_{\min}(N, D)] = 1$$

- Fortunately, often data is not "truly" high-dimensional..

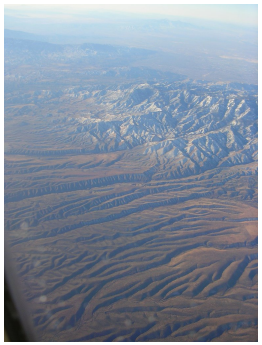
The Blessing of Large Data



Hays and Efros, SIGGRAPH 2007

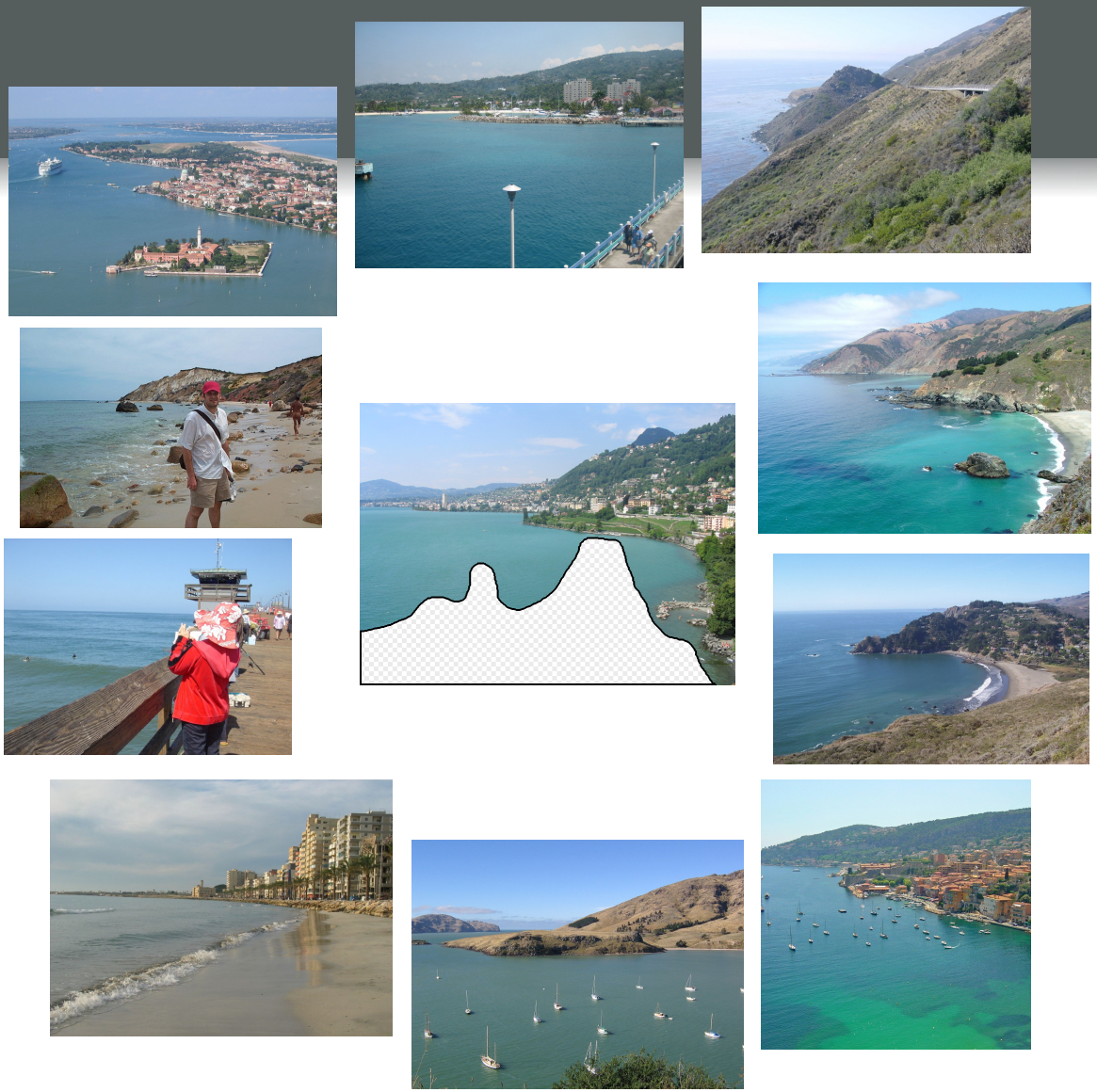









10 nearest neighbors from a collection of 20,000 images

Hays and Efros, SIGGRAPH 2007



10 nearest neighbors from a collection of 2 million images

Hays and Efros, SIGGRAPH 2007

Approximate retrieval

- Input: data set S, distance function d
- Problem 1: Nearest neighbor
  - Given query q in S, find nearest s* in S
- Problem 2: Near-duplicate detection
  - Find all s,s' in S with distance at most ε

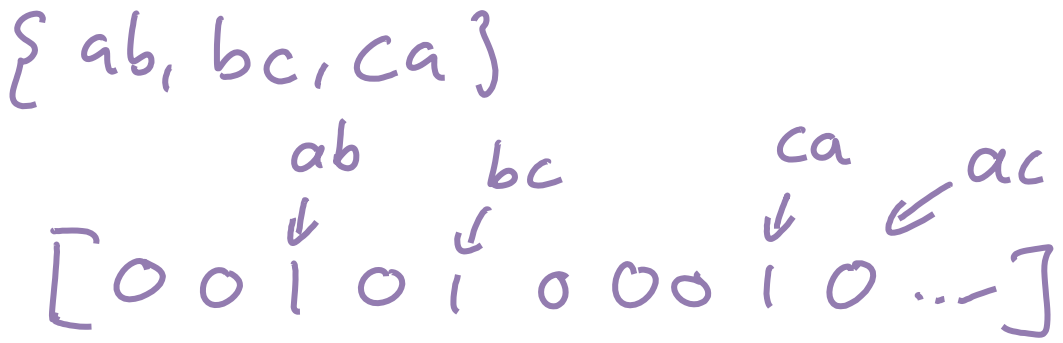
Application: Find similar documents

- Find "near-duplicates" among a large collection of documents
  - Will be key sub-routine in finding nearest neighbors
  - Find clusters in a document collection
  - Detect plagiarism
  - …
- What is the right distance function?

Shingling

- To keep track of word order, extract k-shingles (aka k-grams)
- Document represented as "set of k-shingles"

- Example: *a b c a b*



- Equivalent representation as binary vectors (one dimension per k-shingle)

Shingling implementation

- How large should one choose k?
  - Long enough s.t. they don't occur "by chance"
  - Short enough so that one expects "similar" documents to share some k-shingles
- Storing shingles
  - Want to save space by compressing
  - Often, simply hashing works well (e.g., hash 10-shingle to 4 bytes)

Comparing shingled documents

- Documents are now represented as sets of shingles
- Want to compare two sets
- E.g.: A={1,3,7}; B={2,3,4,7}

Jaccard distance

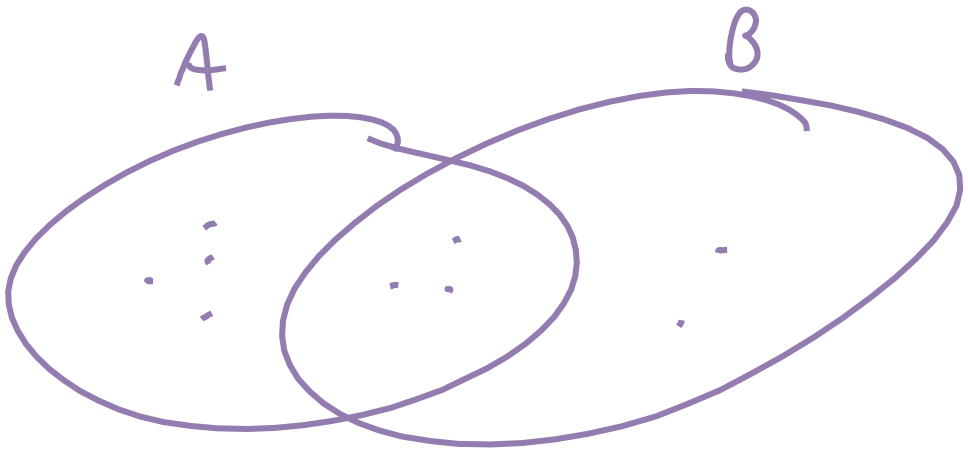
- Jaccard similarity:

$$\mathrm{Sim}(A, B) = \frac{|A \cap B|}{|A \cup B|} \quad \leq 1$$

- Jaccard distance:

$$d(A, B) = 1 - \frac{|A \cap B|}{|A \cup B|}$$

Example



$$Sim(A,B) = \frac{3}{9} = \frac{1}{3}$$

$$d(A,B) = \frac{2}{3}$$

Near-duplicate detection

- Want to find documents that have similar sets of k-shingles
- Naïve approach:
- For i=1:N
  - For j=1:N
    - Compute d(i,j)
    - If d(i,j) < ε then declare near-duplicate

$\mathcal{O}(N^2 D)$

- **Infeasible even for moderately large N** ☹

- **Can we do better??**

Warm-up

- Given a large collection of documents, determine whether there exist **exact** duplicates?

- Compute hash code / checksum (e.g., MD5) for all documents
- Check whether the same checksum appears twice
- Both can be easily parallelized (how?)

Locality sensitive hashing

- **Idea**: Create hash function that maps "similar" items to same bucket

Hashtable

0	1	2	3

- **Key problem**: Is it possible to construct such hash functions??
  - Depends on the distance function
  - Possible for Jaccard distance!! ☺
  - For several other distance functions too!

Shingle Matrix

documents

shingles

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

Min-hashing

- Simple hash function, constructed in the following way:

- Use random permutation π to reorder the rows of the matrix
  - Must use same permutation for all columns C!!
- *h*(*C*) = minimum row number in which permuted column contains a 1

$$h(C) = h_\pi(C) = \min_{i:C(i)=1} \pi(i)$$

Min-hashing example

Input matrix

3	1	0	1	0
4	1	0	0	1
7	0	1	0	1
6	0	1	0	1
1	0	1	0	1
2	1	0	1	0
5	1	0	1	0

⇒ [2 1 2 1]

Min-hashing property

- Want that similar documents (columns) have same hash function value (with high probability)

- Turns out it holds that

$$\Pr[h(C_1) = h(C_2)] = \mathrm{Sim}(C_1, C_2)$$

Proof

C_1	C_2
0	1
1	1
1	1
0	0
0	1
0	0
1	1

a	1	1
b	1	0
c	0	1
d	0	0

$$Sim(C_1, C_2) = \frac{a}{a+b+c}$$

Proof

Consider stepping through rows in π-order
Stop at first 1 in any col.

	C_1	C_2
a	1	1
b	1	0
c	0	1
d	0	0

$$P\left(h(C_1) = h(C_2)\right)$$

$$= P(\text{"stopping at row of type } a\text{"})$$

$$= P(\text{"Type } a \text{ given that we stop"})$$

$$\geq \frac{a}{a+b+c}$$

□

Near-duplicate search with Min-Hashing

- Suppose we would like to find all duplicates with more than 90% similarity
- Apply min-hash function to all documents, and look for candidate pairs (documents hashed to same bucket)

- How many 90%-duplicates will we find? 90%

- How many 90%-duplicates will we miss? 10%

- How can we reduce the number of misses?

Reducing the "misses"

- Apply multiple *independently random* hash functions
- Consider candidate pair of near duplicates if at least one of the functions hashes to same bucket
- What's the probability of a "miss" with k functions?

$$P(\text{"miss with 1 fn"}) = d(C_1, C_2) = 1 - Sim(C_1, C_2)$$

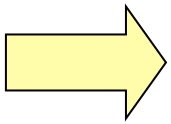
$$P(\text{"miss with } k \text{ fns"}) = d(C_1, C_2)^k$$

- Thus, using multiple independent hash functions can exponentially reduce probability of **misses!**

Min-hash signatures

Input matrix

1	4	3	1	0	1	0
3	2	4	1	0	0	1
7	1	7	0	1	0	1
6	3	6	0	1	0	1
2	6	1	0	1	0	1
5	7	2	1	0	1	0
4	5	5	1	0	1	0



Signature matrix *M*

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

Similarities:

	1-3	2-4	1-2	3-4
Col/Col	0.75	0.75	0	0
Sig/Sig	0.67	1.00	0	0

Implementing min-hashing

- Difficult to randomly permute a data set with a billion rows
- Even representing a permutation of size 10^9 is expensive
- Accessing rows in permuted order is infeasible (requires random access)

Implementing min-hashing

- Directly represent permutation π through hash function h!

$$\pi(i) = h(i) = ai + b \mod n$$

  - Could happen that h(i)=h(j) for i ≠ j, but this is rare for good h
- **Note**: Will use same notation for h(r) and h(C)

$$h(C) = \min_{i:C(i)=1} h(i)$$

- Suppose h(r)<h(s). Then row r appears before s in π
- Why is this useful?
- Can store h very efficiently!
- Allows to process data matrix row-wise!

Example

Row	C1	C2
1	1	0
2	0	1
3	1	1
4	1	0
5	0	1

$h(x) = x \bmod 5$

$h(1)=1, h(2)=2, h(3)=3, h(4)=4, h(5)=0$

$g(x) = 2x+1 \bmod 5$

$g(1)=3, g(2)=0, g(3)=2, g(4)=4, g(5)=1$

$M =$ 

Implementation

Want to iteratively build signature matrix M

For each column *c* and each hash function h_i, use *M*(*i*, *c*) to keep track of smallest value of $h_i(r)$ for which column *c* has 1 in row *r*

- Initialize to ∞

for each column *c*

for each row *r*

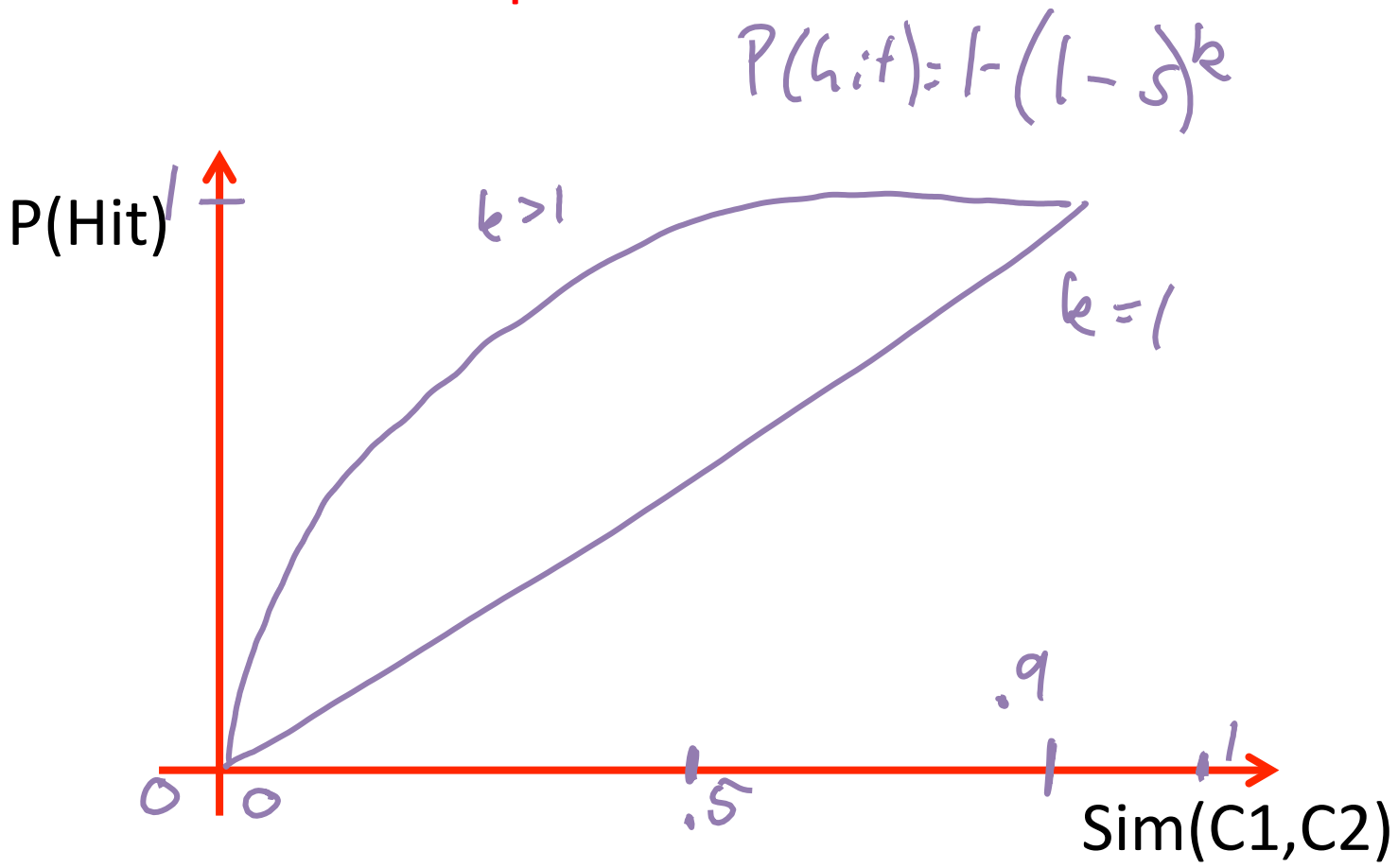
if c has 1 in row *r*

for each hash function h_i **do**

M (*i*, *c*) := min($h_i(r)$, *M*(*i*, *c*));

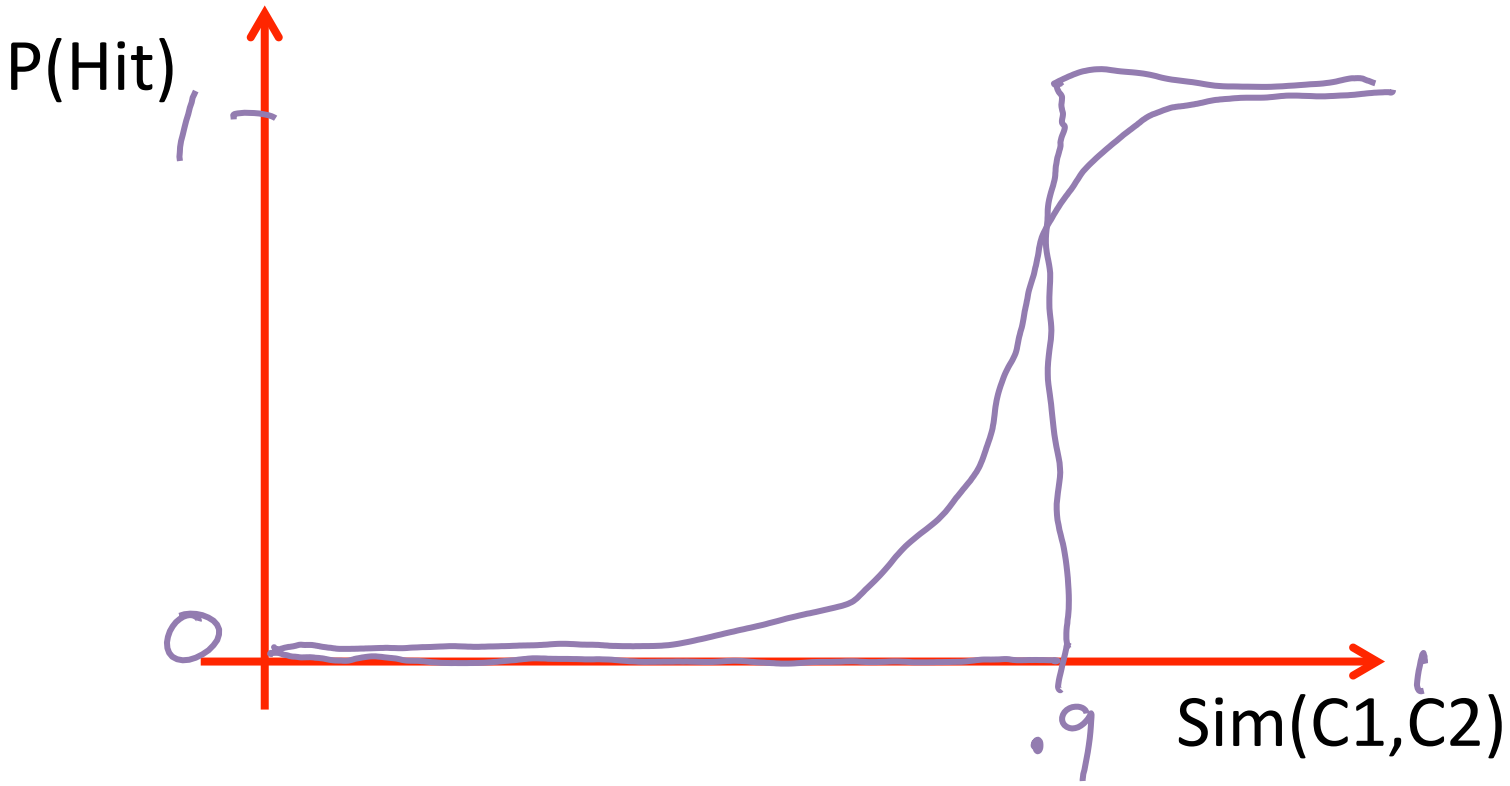
False positives

- Increasing number of hash tables reduces false negative rate ☺
- Also increases false positive rate ☹



False positives

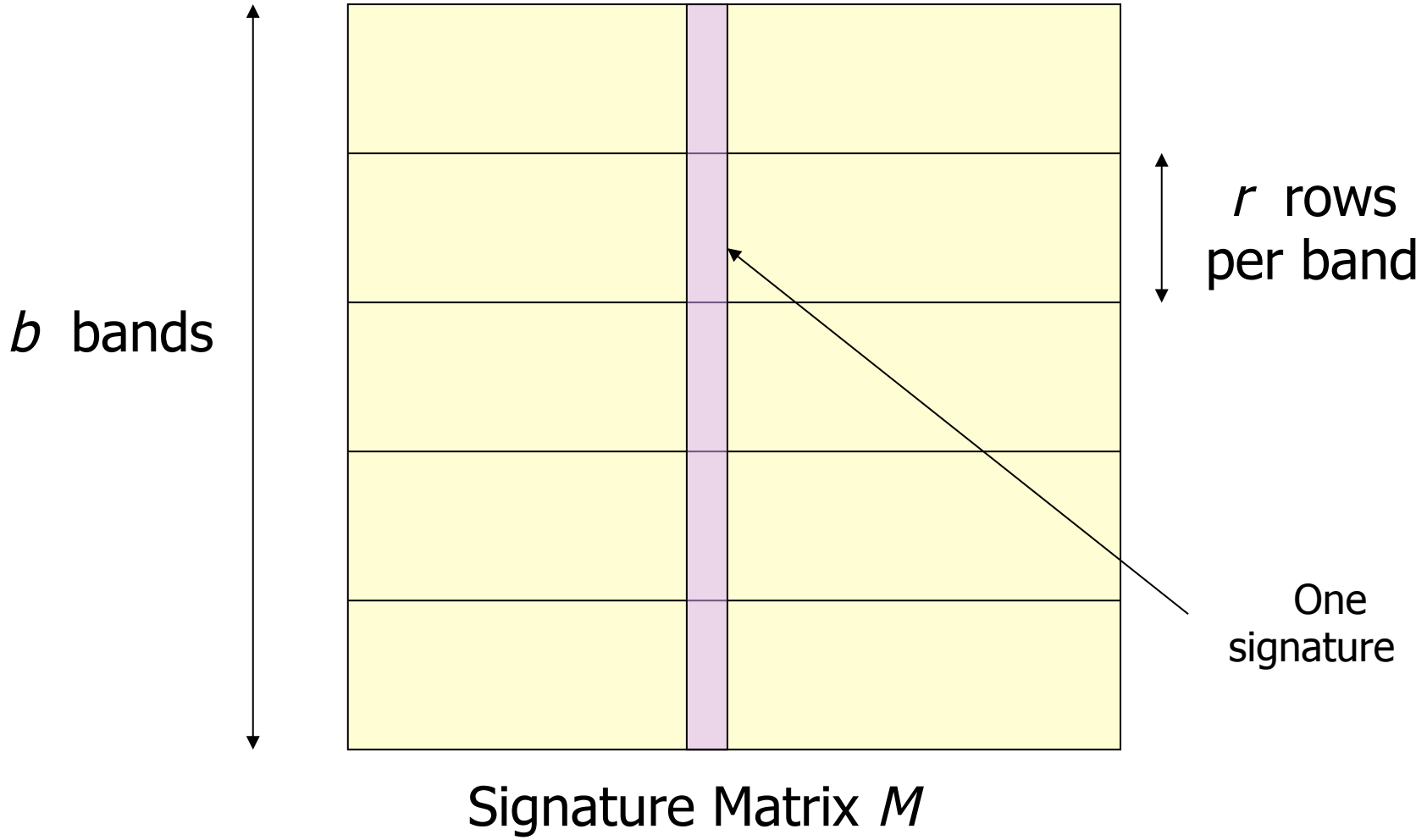
- Ideally want:



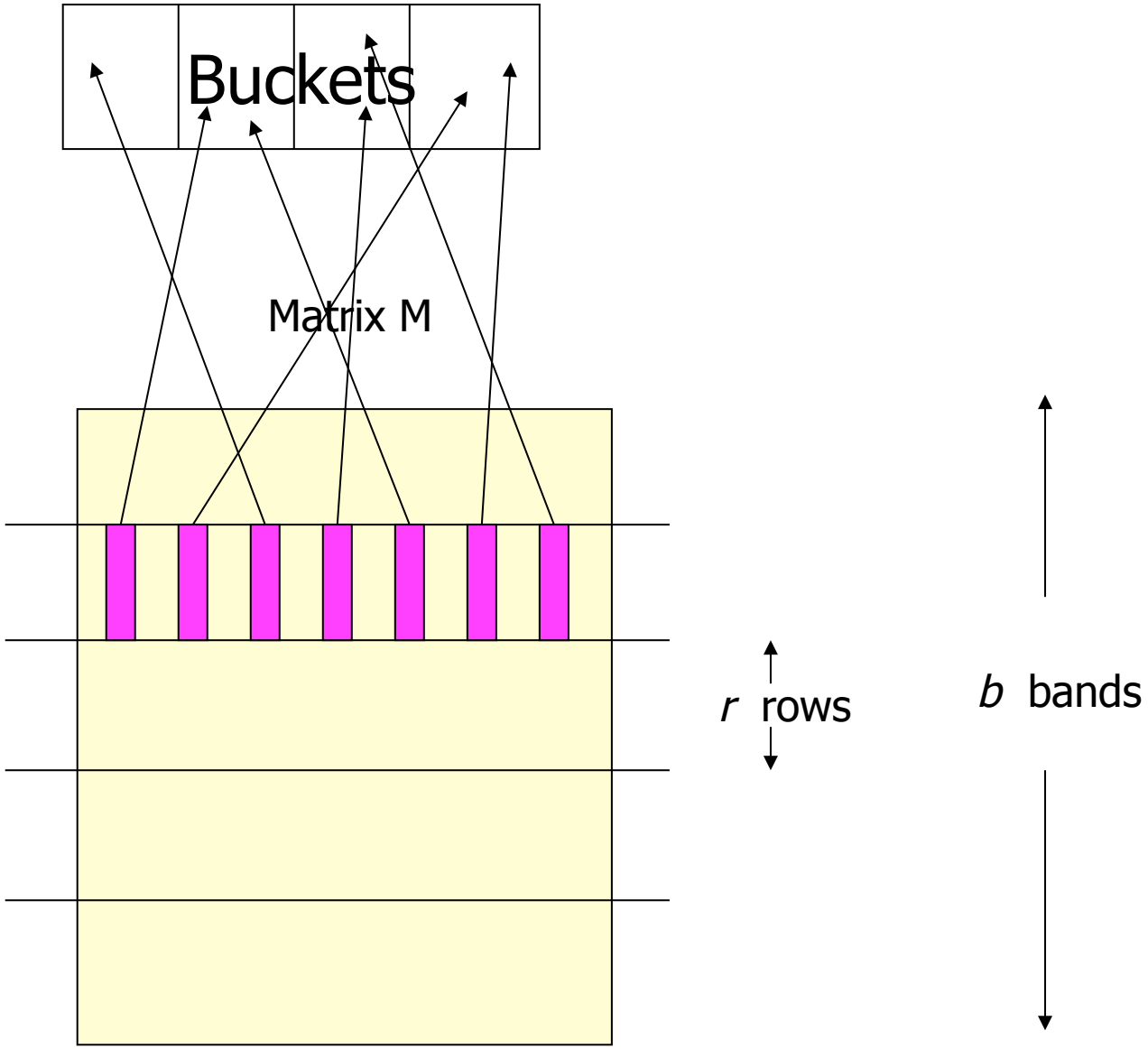
Ingenious trick

- Signature matrix compactly represents similarity between documents
  - Jaccard distance ≈ l1-distance of columns
  - Similar documents have similar signatures
- Naïve approach: Compare any pair of columns to see if their similar
  - Compact representation ➔ faster
  - Still N^2 comparisons ☹
- Will see how to hash columns s.t. with high probability
  - return similar pairs (d(C1,C2) < ε)
  - do not return dissimilar pairs (d(C1,C2) >> ε)

Partitioning the signature matrix



Hashing bands of M



Hashing the signature matrix

- Signature matrix *M* partitioned into *b* bands of *r* rows.
- One hash table per band, independent hash functions
- For each band, hash its portion of each column to its hash table
  - For purpose of analysis, let's assume there's no "false collisions"
  - Doesn't affect correctness of algorithm
- Candidate pairs are columns that hash to the same bucket for at least one band.
- Why is this useful?

Analysis of partitioning

- Suppose columns C1 and C2 have similarity *s*

$C_i = [B_{i,1} \ldots B_{i,b}]$

Consider band j

$P(h(B_{1,j}) = h(B_{2,j})) = s^r$

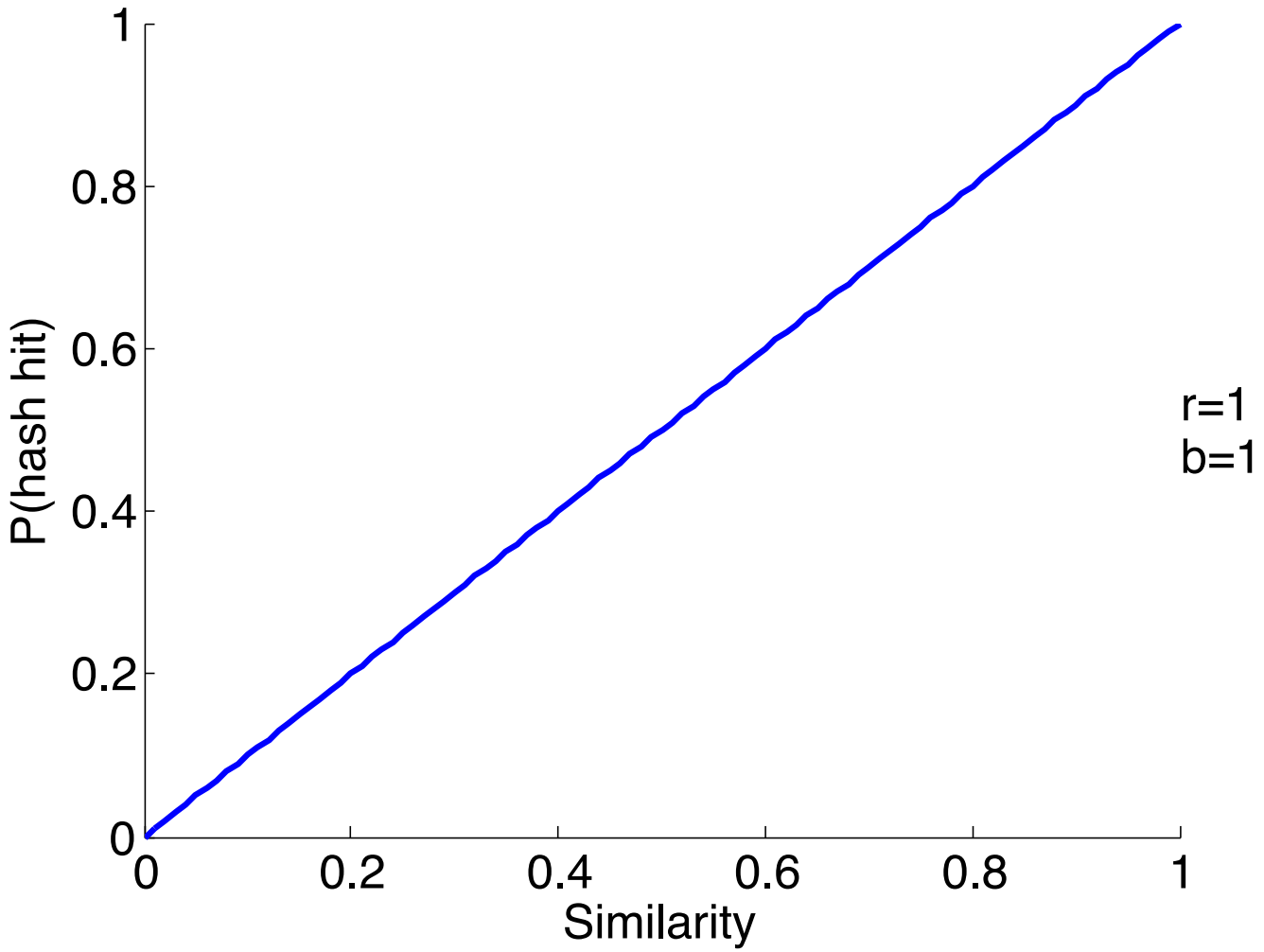
$P(h(B_{1,j}) \neq h(B_{2,j})) = 1 - s^r$

$= P(C_1 \text{ and } C_2 \text{ do } \underline{\text{not}} \text{ collide on band } j)$

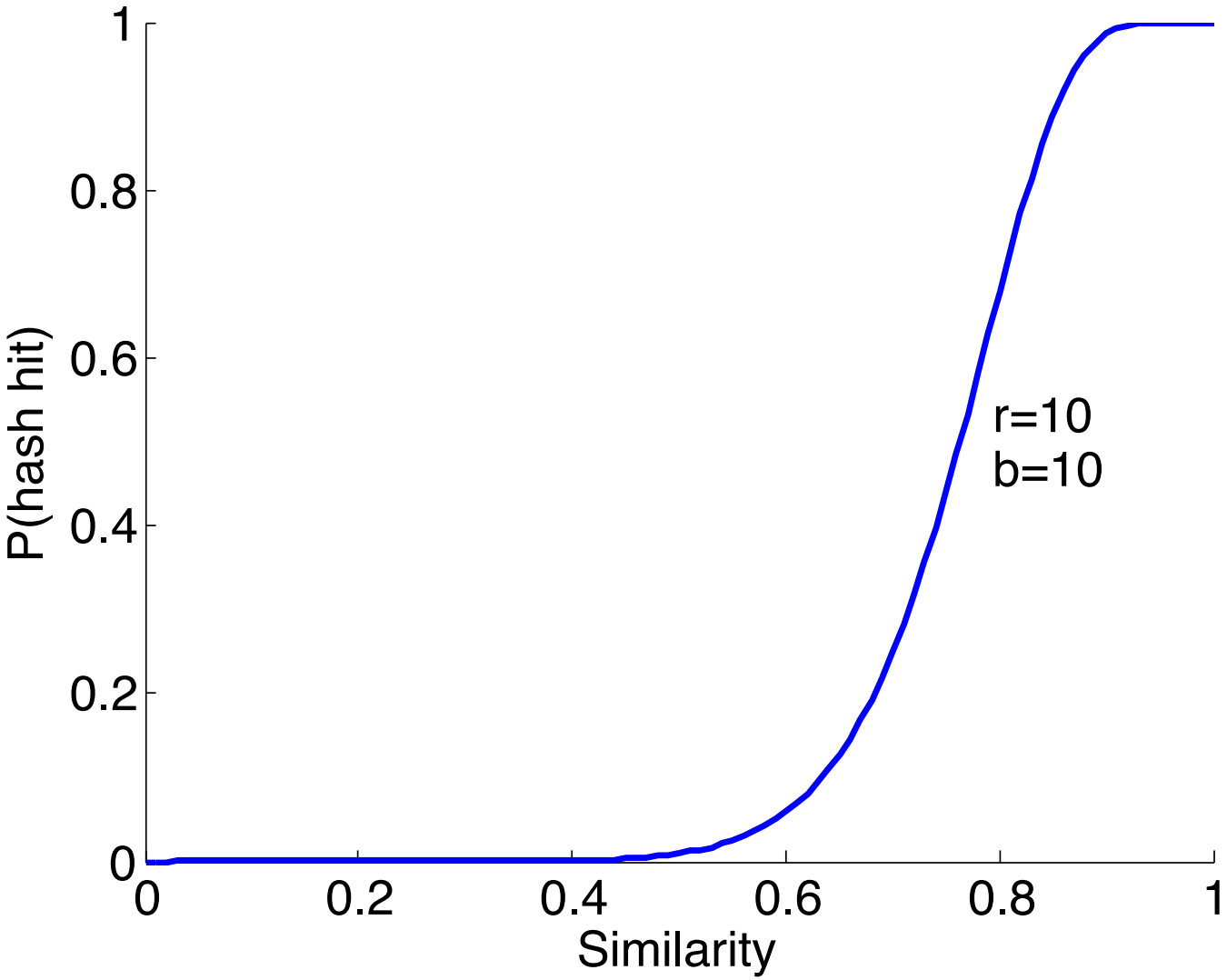
$P(C_1 \text{ and } C_2 \text{ do } \underline{\text{not}} \text{ collide on } \underline{\text{any}} \text{ band}) = (1-s^r)^b$

$P(C_1 \text{ and } C_2 \text{ collide on } \underline{\text{some}} \text{ band}) = 1 - (1-s^r)^b$

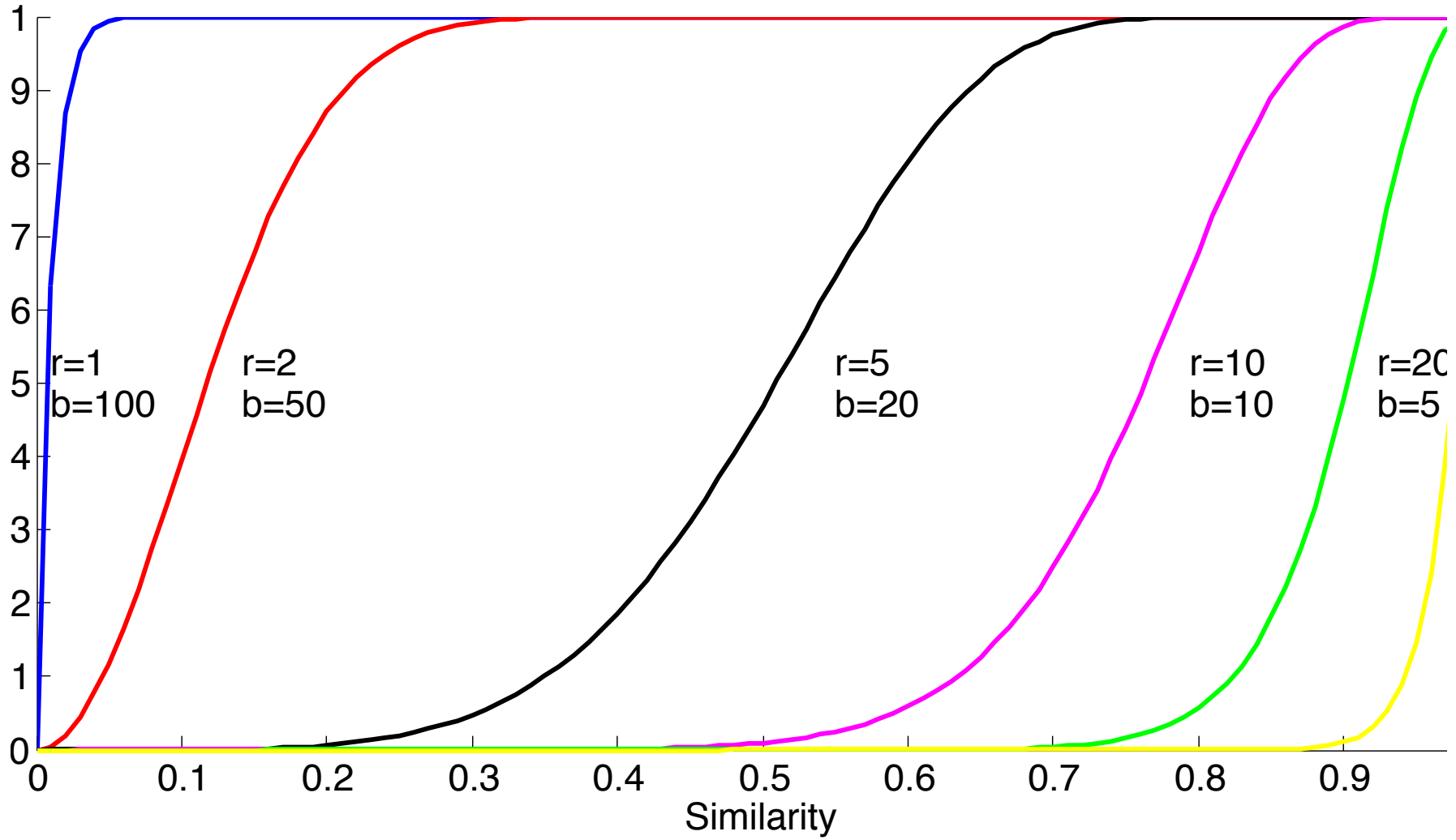
One hash function



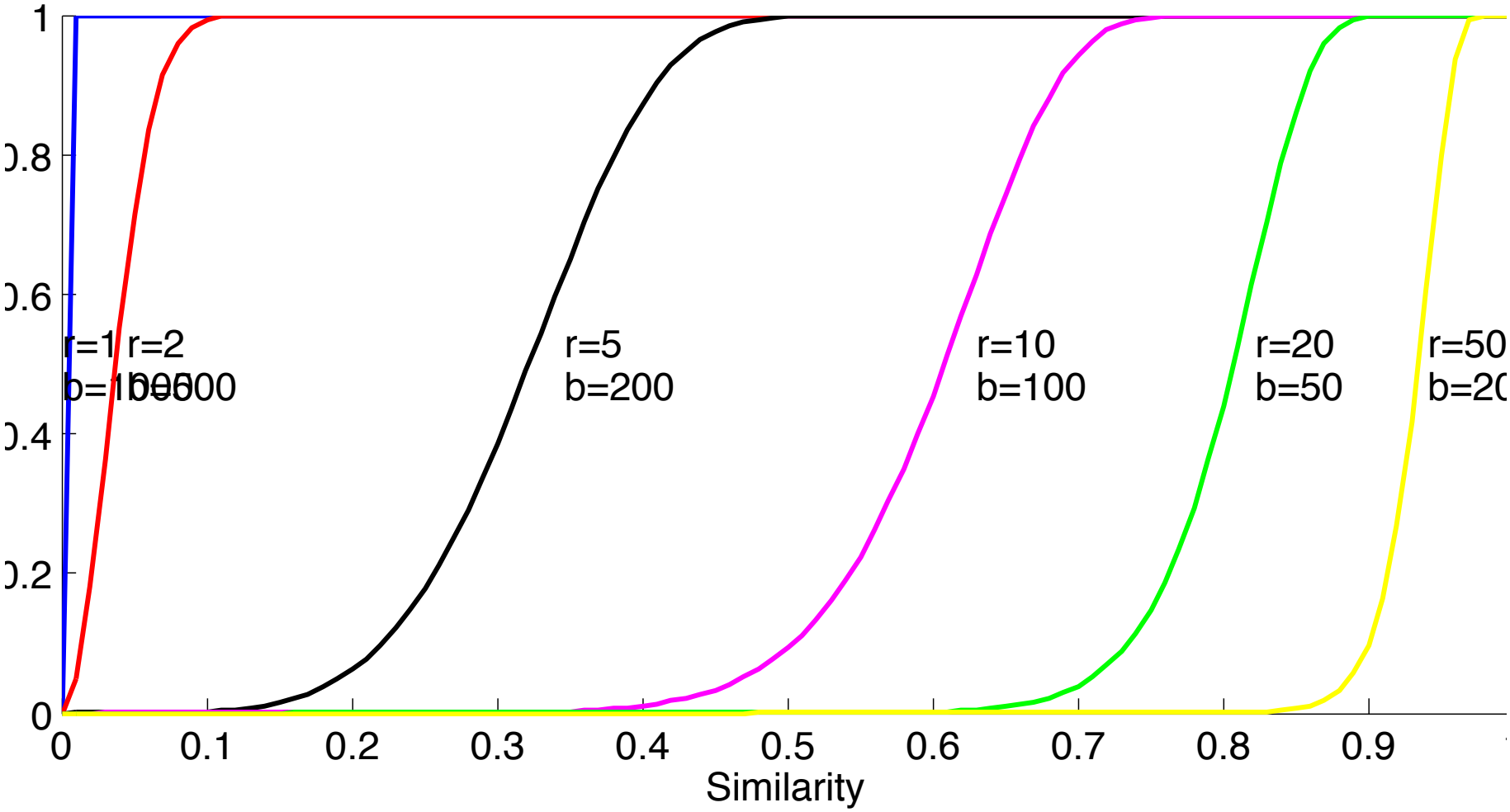
100 hash functions



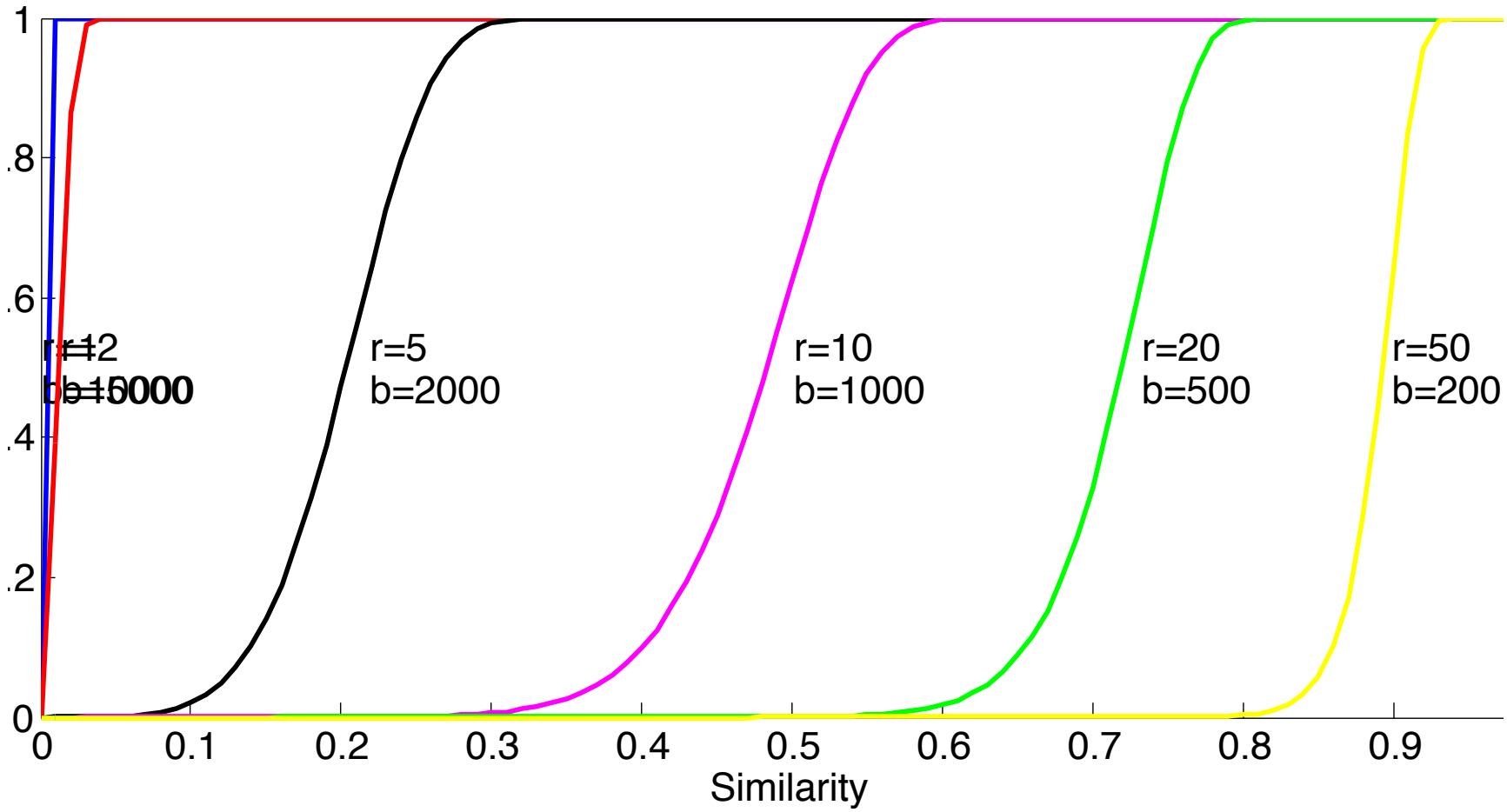
100 hash functions



1000 hash functions



10000 hash functions



Implementation details

- Tune *r* and *b* to achieve desired similarity threshold
- Typically favor
  - few false negatives
  - more false positives
- Do pairwise comparisons of all resulting candidate pairs (in main memory), to eliminate false positives
- Typically also compare the actual documents (needs another pass through the data)

Acknowledgments

- Several slides adapted from the material accompanying the textbook (Anand Rajaraman, Stanford)