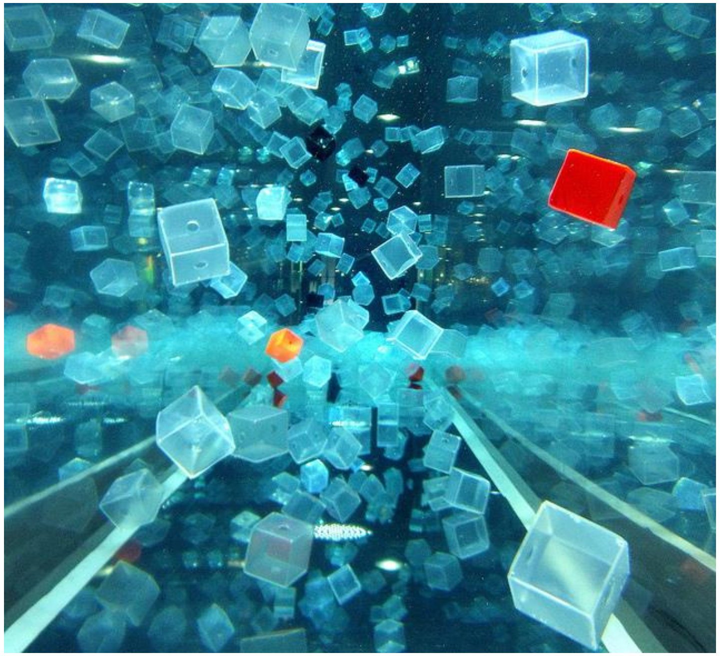
# Finding Similar Sets



http://www.csd.uoc.gr/~hy562

University of Crete

# A small workshop first

https://www.menti.com/altfoogpsu8z

# Motivation

- Many Web-mining problems can be expressed as finding “similar” sets:
  - Pages with similar words, e.g., for classification by topic
  - Netflix users with similar tastes in movies for recommendation systems
    - Dual: movies with similar sets of fans
  - Images of related things

- The best techniques depend on whether you are looking for items that are very similar or only somewhat similar
  - Special cases are easy, e.g., identical documents, or one document contained character-by-character in another
  - General case, where many small pieces of one document appear out of order in another, is very hard

# Finding Similar Documents

- Applications: Given a body of documents, find pairs of documents with a lot of text in common, e.g.:
  - Mirror Web sites, or approximate mirrors
    - Application: Don’t want to show both in a search
  - Plagiarism, including large quotations
  - Similar news articles at many news sites
    - Application: Cluster articles by “same story”

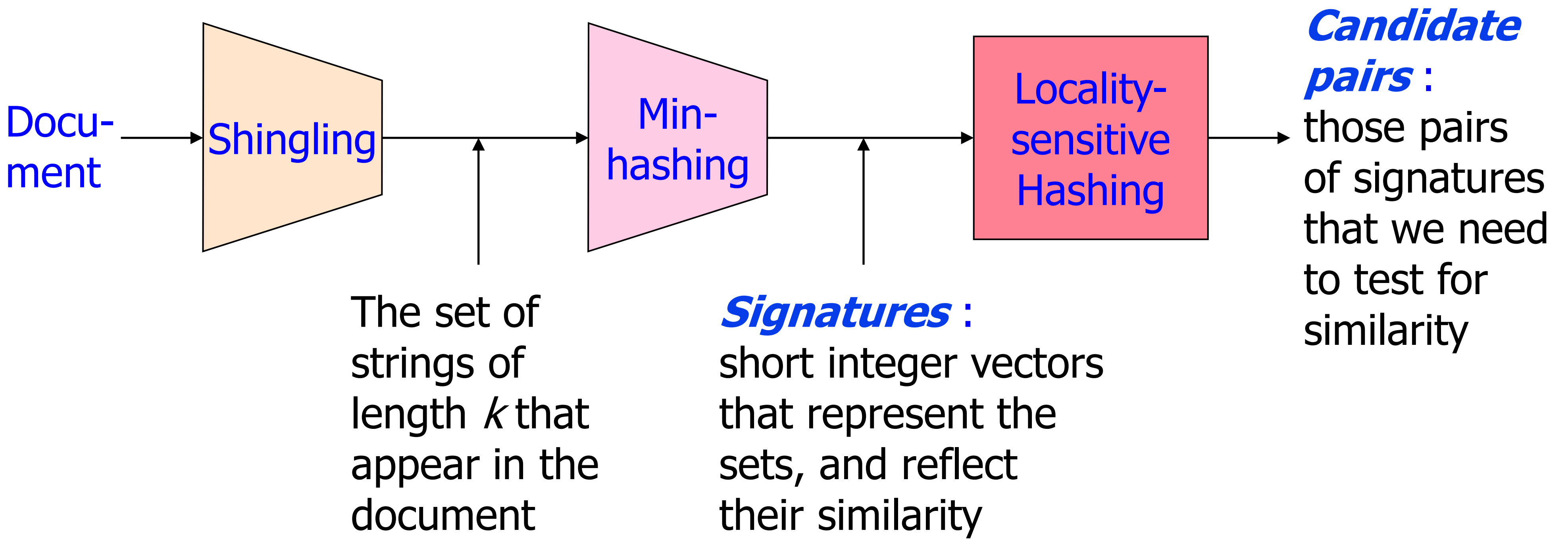
- Simple IR approaches are not suited:
  - Document = set of words appearing in document
  - Document = set of “important” words

  Why? we need to account for ordering of words!

# Main Issues

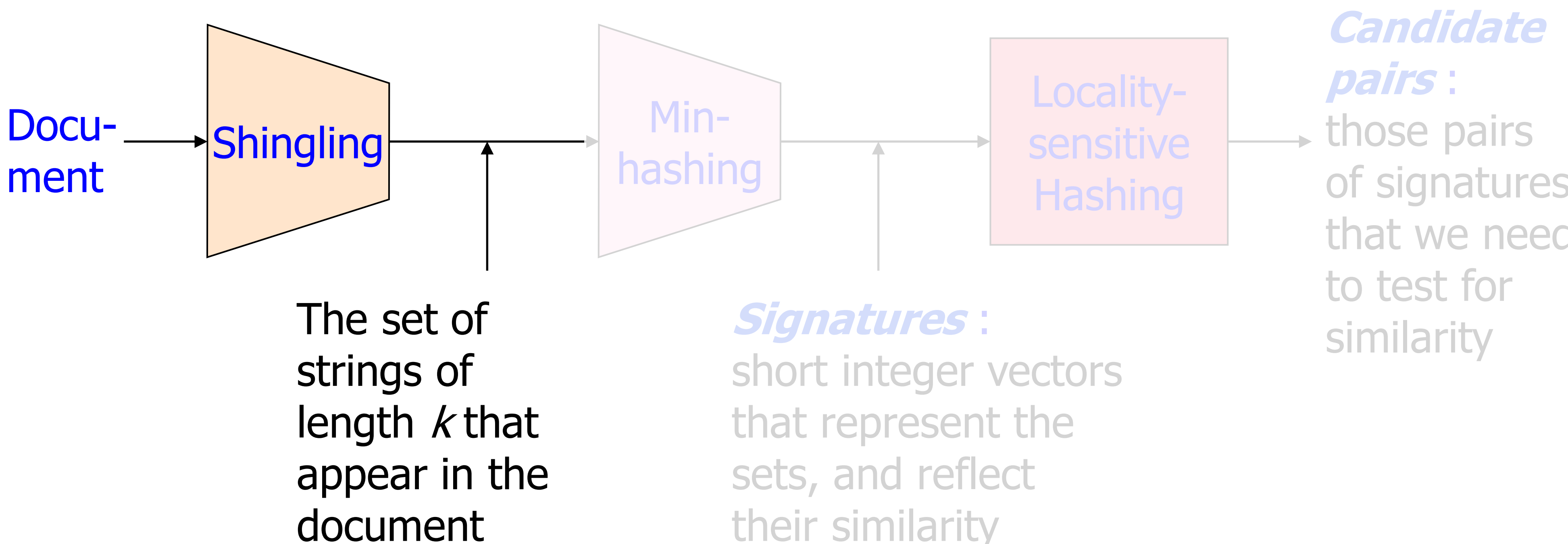
- What is the right representation of the document when we check for similarity?
  - E.g., representing a document as a set of characters will not do (why?)
- When we have billions of documents, keeping the full text in memory is not an option
  - We need to find a shorter representation
- How do we do pairwise comparisons of billions of documents?
  - If exact match was the issue it would be ok, can we replicate this idea?

# Three Essential Techniques for Detecting Similar Documents




- Shingling: convert documents, emails, etc., to *sets*
- Min-hashing: convert *large sets to short signatures*, while preserving similarity
- Locality-sensitive hashing: focus on *pairs of signatures likely to be similar*

# Shignling


Docu-
ment
Shingling
Min-
hashing
Locality-
sensitive
Hashing
Candidate pairs : those pairs of signatures that we need to test for similarity
The set of strings of length k that appear in the document
Signatures : short integer vectors that represent the sets, and reflect their similarity

# Shingles

- A k-shingle (or k-gram) for a document is a sequence of k characters (or words) that appears in the document
  - Represent a document by its set of k-shingles
- Example: doc=“abcab”.
  - Set of 2-shingles
  - {ab, bc, ca}
  - Alternative:
  - Bag of 2-shingles = {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
  - What if two documents differ by a word?
    - Affects only k-shingles within distance k from the word
  - What if we reorder paragraphs?
    - Affects only k-shingles that cross paragraph boundaries

# Shingle Size

- Is k=2 a good choice for a shingle size?

- Example:
  - doc1 = “abcab”. 2-shingles = {ab, bc, ca}
  - doc2 = “cabc”. 2-shingles = {ab, bc, ca}

- Careful decision: you must pick k to be
  - large enough, or most documents will have most shingles in common
  - not too large, or most documents will have no shingles in common

  k = 5 is OK for short documents

  k = 10 is better for long documents

# Basic Data Model: Sets

- Many similarity problems can be couched as finding subsets of some universal set that have significant intersection

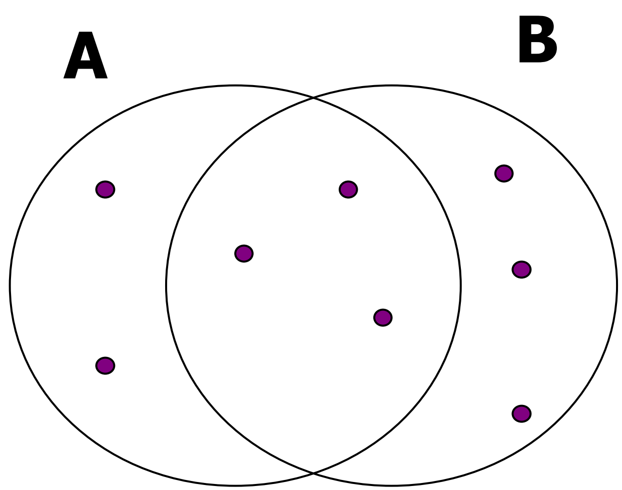
- Examples:
  - Documents represented by their sets of shingles
  - Similar customers or products

- Each document is a 0/1 vector in the space of k-shingles
  - Each unique shingle is a dimension
  - Vectors are very sparse

- Interpret set intersection as bitwise AND, and set union as bitwise OR

# Jaccard Similarity of Sets

- The *Jaccard similarity* of two sets is the size of their intersection divided by the size of their union
  - $Sim\ (C_1,\ C_2) = |C_1 \cap C_2| / |C_1 \cup C_2|$




**3 in intersection**
**8 in union**

**Jaccard similarity = 3/8**

# From Sets to Boolean Matrices

- Rows = elements (shingles) of the universal set
- 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 sets of their rows with 1
- Typical matrix is sparse
  - Sparse matrices are usually better represented by the list of places where there is a non-zero value
  - But the Boolean matrix picture is conceptually useful

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 |

# Example: Jaccard Similarity of Columns

1-shingles

| | $C_1$ | $C_2$ | | |
|---|---|---|---|---|
| a | 0 | 1 | | * |
| b | 1 | 0 | | * |
| c | 1 | 1 | * | * |
| d | 0 | 0 | | |
| e | 1 | 1 | * | * |
| f | 0 | 1 | | * |

$\text{Sim}(C_1, C_2) = \mathbf{2/5 = 0.4}$

**C1 = “bce”**
**C2 = “acef”**

# Shingles: Compression Option

- How about space overhead?
  - Each character can be represented as a byte
  - One k-shingle requires k bytes

- To compare a pair of 9-shingles we need to compare 9 bytes
- To improve efficiency, we can compress long shingles:
  - hash them to (say) 4 bytes, and
  - represent a document by the set of hash values of its k-shingles

(aaabbbccc)(abcabcabc) → h(aaabbbccc)h(abcabcabc)

18 bytes → 8 bytes

- Working Assumption: Two documents with shared hash values will almost always have shingles in common.

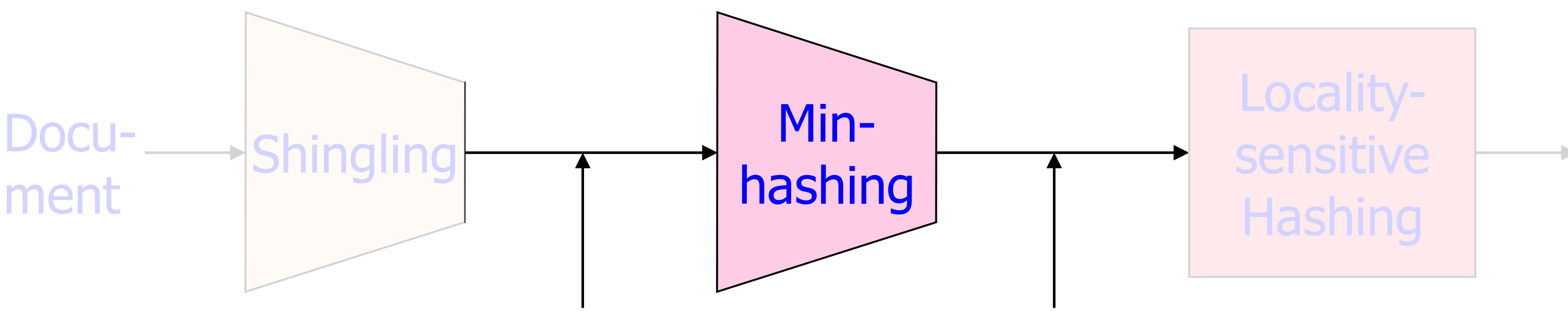
# Outline: Finding Similar Columns

- Naïve approach:
  - ❶ Compute **signatures** of documents = small summaries of columns
  - ❷ Examine pairs of signatures to find similar columns
    - Requirement: similarities of signatures and columns are related
  - ❸ Optional: check that columns with similar signatures are really similar
- This scheme works but …
  - What if the set of signatures (or k-shingles) is too large to fit in the memory?
  - Or the number of documents is too big?
- Idea: Hash a document (column) to a single (small-size) value and similar documents to the same value!
  - Warning: These methods can produce *false negatives*, and even *false positives* (if the above optional check is not made)

# Signatures

- Key idea: "hash" $h(\cdot)$ each column $C$ to a small signature, such that:
  1. $h(C)$ is small enough that we can fit a signature in main memory for each column
  2. $Sim(C_1, C_2)$ is approximated by the "similarity" of $h(C_1)$ and $h(C_2)$
- By hashing columns into buckets we expect that "most" pairs of near duplicate documents hash into the same bucket!
- *Goal: Find a hash function $h(\cdot)$ such that:*
  - If $sim(C_1,C_2)$ is high, then with high probability $h(C_1) = h(C_2)$
  - If $sim(C_1,C_2)$ is low, then with high probability $h(C_1) \neq h(C_2)$
- Clearly, the hash function depends on the similarity metric:
  - Not all similarity metrics have a suitable hash function!
  - There is a suitable hashing technique for the Jaccard similarity:
    - It is called Min-Hashing!

# MinHashing




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

The set of strings of length *k* that appear in the document

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

# Minhashing

- History: invented by Andrei Broder in 1997 (AltaVista) to detect near duplicate web pages

- 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**:

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

# MinHashing - Example

2nd element of the permutation is the first to map to a 1 in col $C_1$

**Permutations**

| | | |
|---|---|---|
| 1 | 4 | 3 |
| 3 | 2 | 4 |
| 7 | 1 | 7 |
| 6 | 3 | 6 |
| 2 | 6 | 1 |
| 5 | 7 | 2 |
| 4 | 5 | 5 |

**Input matrix**

| $C_1$ | $C_2$ | $C_3$ | $C_4$ |
|---|---|---|---|
| 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 |

**Signature matrix *M***

| $h(C_1)$ | $h(C_2)$ | $h(C_3)$ | $h(C_4)$ |
|---|---|---|---|
| 2 | 1 | 2 | 1 |
| 2 | 1 | 4 | 1 |
| 1 | 2 | 1 | 2 |

*$h_2(C_3)$=4 (permutation 2, column $C_3$)*

4th element of the permutation is the first to map to a 1 in $C_3$

# Surprising Property

- The probability (over all permutations of the rows) that $h(C_1)=h(C_2)$ is the same as $Sim(C_1, C_2)$:
  - $Pr[h_\pi(C_1) = h_\pi(C_2)] = sim(C_1,C_2)$
- With multiple signatures (i.e, permutations or hash functions) we get a good approximation
- Use several independent hash functions to create a signature of a column
  - The similarity of signatures is the fraction of the hash functions in which they agree
  - Because of this MinHash property, the similarity of columns is the same as the expected similarity of their signatures

# Why?

- Given columns $C_1$ and $C_2$, rows may be classified as:

| | $C_1$ | $C_2$ |
|---|---|---|
| a | 1 | 1 |
| b | 1 | 0 |
| c | 0 | 1 |
| d | 0 | 0 |

- Let $A$ = # rows of type *a*, $B$ = # rows of type *b*, $C$ = # rows of type *c*
- Look down the permuted columns $C_1$ and $C_2$ until we see a 1
  - If it's a type-a row, then $h(C_1)=h(C_2)$
  - If it's a type-b or type-c row, then $h(C_1)\neq h(C_2)$
  - Then: $Pr[h(C_1)=h(C_2)] = A/(A+B+C)$
- Note $Sim(C_1,C_2) = A/(A+B+C)$
  - **Then:** $Pr[h(C_1)=h(C_2)] = Sim(C_1,C_2)$

# MinHashing – Example

**Input matrix**

| $\pi_1$ | $\pi_2$ | $\pi_3$ | $C_1$ | $C_2$ | $C_3$ | $C_4$ |
|---|---|---|---|---|---|---|
| 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 |

# MinHash – False Positive/Negative

- False positive?
  - False positive can be easily dealt with by doing an additional layer of checking (treat minhash as a filtering mechanism)
- False negative?
  - Requiring full match of signature is strict, some similar sets will be lost
- High error rate! Can we do better?

# MinHash Signatures

- Pick (say) 100 random permutations of the rows

- Think of *Sig(C)* as a column vector

- Let $Sig(C)[i] = min(\pi_i(C))$
  according to the *i* th permutation, the number of the first row that has a 1 in column *C*

- Note: The sketch (signature) of column C is small **~400** bytes!
  - We achieved our goal! We "compressed" long bit vectors into short signatures

# Implementation Trick

- Permuting rows even once is prohibitive
- An approximation to permuting rows: pick many hash functions $h_i$
  - Instead of a permutation, use a random hash function that maps row numbers to as many buckets as there are rows
  - Row hashing: ordering under $h_i$ gives a random row permutation!
- One-pass implementation
  - For each column `C` and each hash function $h_i$, keep a "slot" `M(i,C)` for the min-hash value
    - all slots initialized to infinity
  - Intent: `M(i,C)` will become the smallest value of $h_i$`(r)` for which column `C` has 1 in row `r`
    - i.e., $h_i$`(r)` gives order of rows for `i`-th permutation

# Implementation

```
M(i,C) = ∞
for each row r
    for each column c
      if c has 1 in row r // Scan rows looking for 1s
         for each hash function h_i do
            if h_i(r) < M(i,C) then
                  M(i,C):= h_i(r);
```

**How to pick a random hash function h(x)?**
**Universal hashing:**

$h_{a,b}(x)=((a\cdot x+b) \ mod \ p)$ mod $N$ where:

a,b … random integers

p … prime number (p > N)

# Example

| Row | $C_1$ | $C_2$ |
|---|---|---|
| $r_1$ | 1 | 0 |
| $r_2$ | 0 | 1 |
| $r_3$ | 1 | 1 |
| $r_4$ | 1 | 0 |
| $r_5$ | 0 | 1 |

Jaccard=1/5

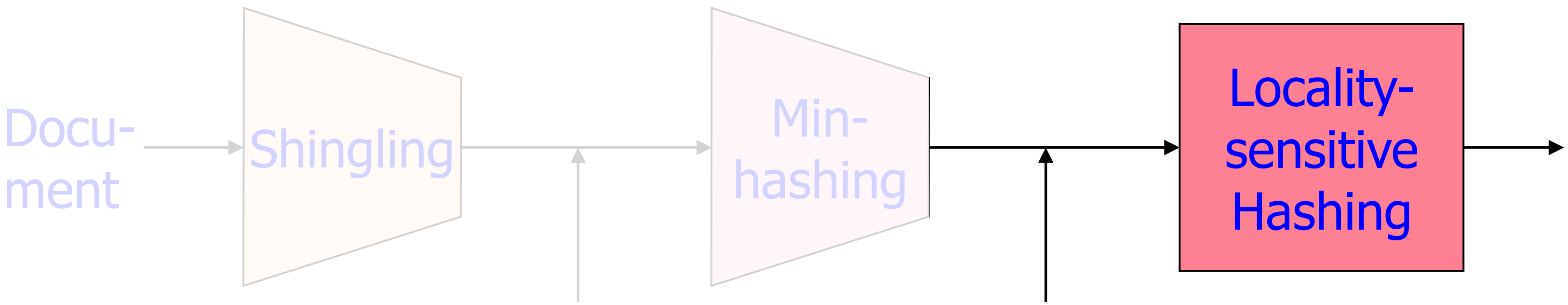
$h_1(x) = x \bmod 5$
$h_2(x) = 2x+1 \bmod 5$

| | Sig1 | Sig2 | |
|---|---|---|---|
| | M(1,1) | | |
| $h_1(1) = 1$ | 1 | ∞ | M(1,2) |
| $h_2(1) = 3$ | 3 | ∞ | M(2,2) |
| | M(2,1) | | |
| $h_1(2) = 2$ | 1 | 2 | |
| $h_2(2) = 0$ | 3 | 0 | |
| $h_1(3) = 3$ | 1 | 2 | |
| $h_2(3) = 2$ | 2 | 0 | |
| $h_1(4) = 4$ | 1 | 2 | |
| $h_2(4) = 4$ | 2 | 0 | |
| $h_1(5) = 0$ | 1 | 0 | |
| $h_2(5) = 1$ | 2 | 0 | |

# So far …

- Represent a document as a set of hash values (of its k-shingles)
- Transform set of k-shingles to a set of minhash signatures
- Use Jaccard to compare two documents by comparing their signatures
- Is this method (i.e., transforming sets to signature) necessarily "better"?

# Locality-Sensitive Hashing




The set of strings of length *k* that appear in the document

***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

# Finding Similar Pairs

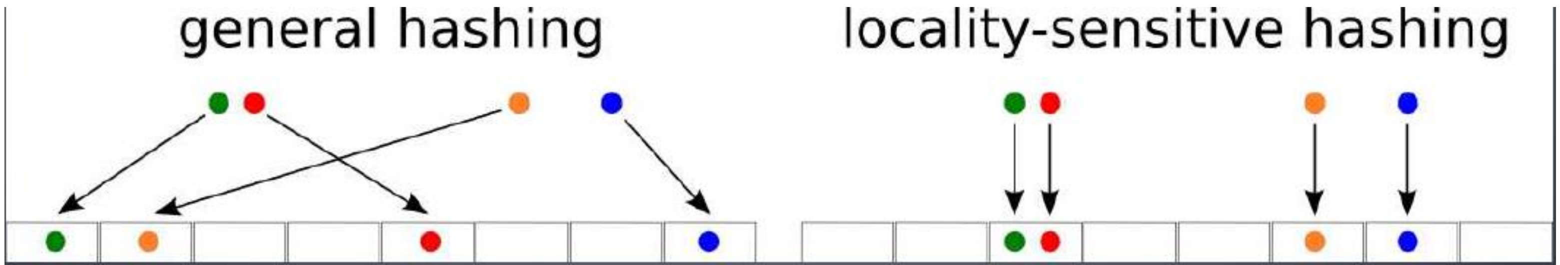
- While the signatures of all columns may fit in main memory, comparing the signatures of all pairs of columns is quadratic in the number of columns

- Naïve solution
  - For each document, compare with the other N-1 documents
    - N-1 comparisons for each document
  - Requires N*(N-1)/2 comparisons

- Example:
  - $10^7$ documents implies ~ $10^{14}$ document-comparisons
  - At 1 μs/comparison $10^8$ (~ 3 years!)

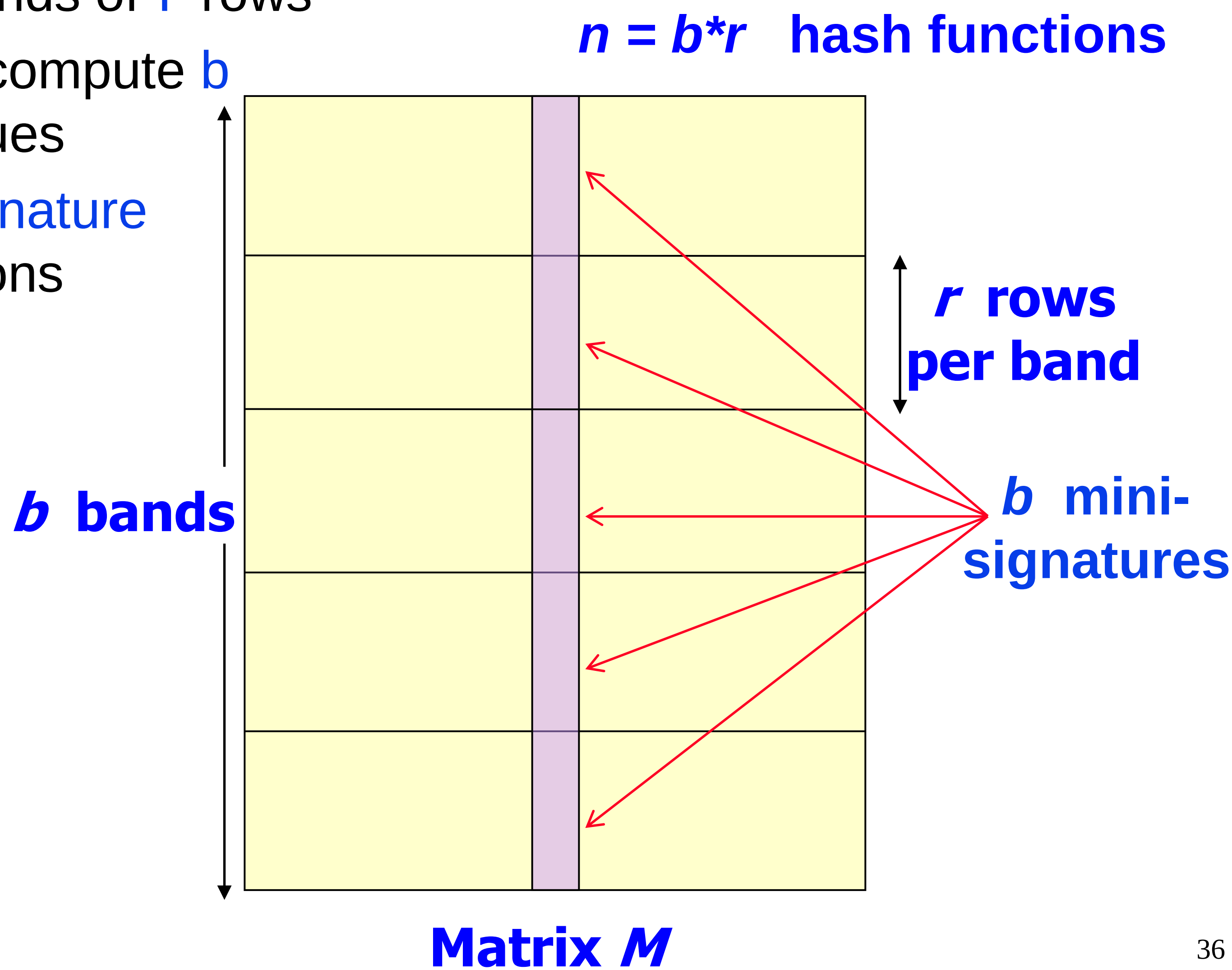
# Locality-Sensitive Hashing

- A function `f(x,y)` tells whether or not `x` and `y` is a candidate pair: a pair of elements whose similarity must be evaluated
- With only one hash function on one entire column of signature, likely to have many *false negatives (i.e., missed similar pairs)*
- Key idea: Apply the hash function on the columns of signature matrix M multiple times, each on a partition of the column (i.e., for a few rows only)
  - Arrange that (only) similar columns are likely to hash (i.e., with high probability) to the same bucket
  - Each pair of columns that hashes at least once into the same bucket is a candidate pair



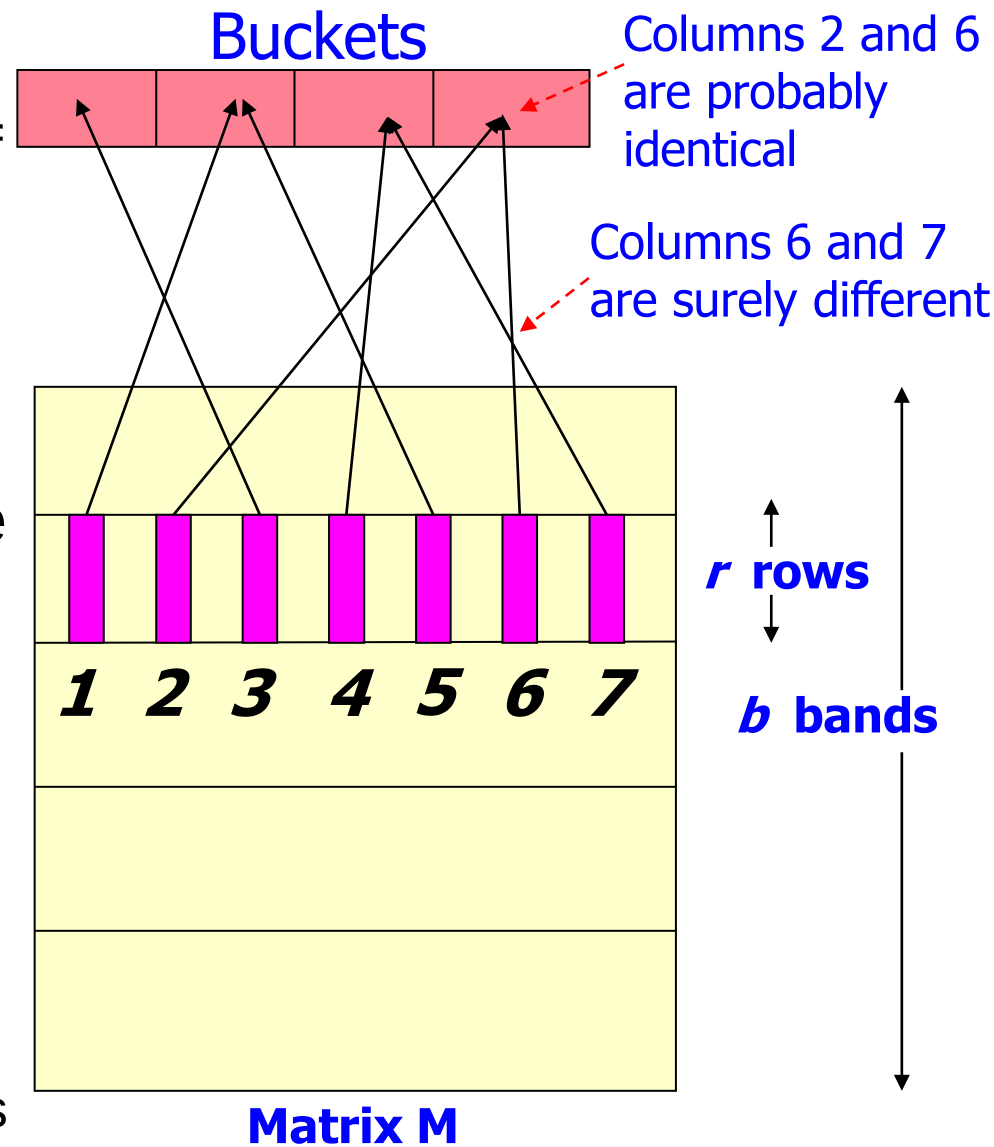
# Partition Into Bands

- Divide matrix `M` into b bands of `r` rows
  - For each document, compute b sets of r minhash values
  - Each set is a mini-signature with r minhash functions



# Partition into Bands

- For each band, hash its portion of each column to a hash table with k buckets
  - larger k => fewer collisions (false positives)
- Candidate column pairs are those that hash to the same bucket for **at least one** band
- Tune b and r to catch *most similar pairs*, but few *non-similar pairs*
  - Intuitively:
    - larger b for lower sim thresholds
    - smaller b for larger sim thresholds

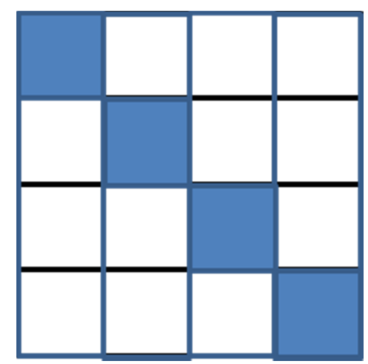


# Simplifying Assumption

- There are enough buckets that columns are unlikely to hash to the same bucket unless they are *identical* in a particular band
  - Hereafter, we assume that "same bucket" means "identical in that band"
  - Assumption needed only to simplify analysis, not for correctness of algorithm
- Finding all pairs within a bucket becomes computationally cheaper!
  - Declare all pairs within a bucket to be "matching" *(faster but noisy)*

  OR

  - Perform pair-wise comparisons for those documents that fall into the same bucket *(slower but more accurate)*
    - Much smaller than pair-wise over all documents



# Example: Effect of Bands

- Suppose $10^5$ columns of M (100k docs)
- Signatures of 100 integers (total rows in M)
- If each integer requires 4 bytes, we only need $10^2*4*10^5$ = 40MB of memory!

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

- $5*10^9$ pairs to compare… this can take a while
- Choose 20 bands of 5 integers/band…

# Analysis of the Banding Technique

- Find pairs with similarity at least $s = 0.8$. Set $b=20$, $r=5$
- Assume: $sim(C_1, C_2) = 0.8$
  - Since $sim(C_1, C_2) \geq s$, 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* identical in any of the 20 bands: $(1-0.328)^{20} = 0.00035$
  - i.e., about 1 in 3000 similar documents are false negatives (we miss them)

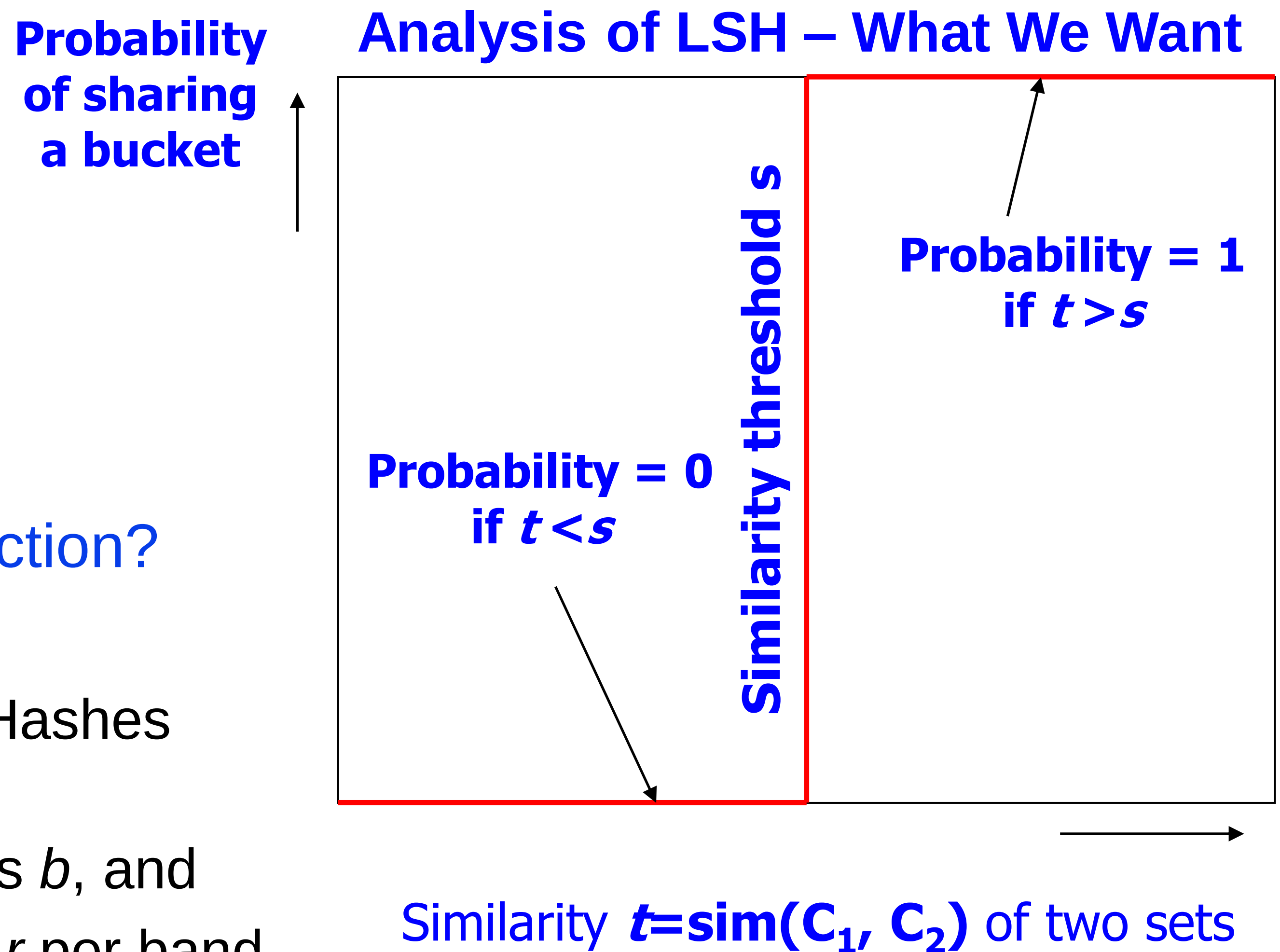
- We would find 99.965% pairs of truly similar documents

# Analysis of the Banding Technique

- Find pairs with similarity at least $s = 0.8$. 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 $s$

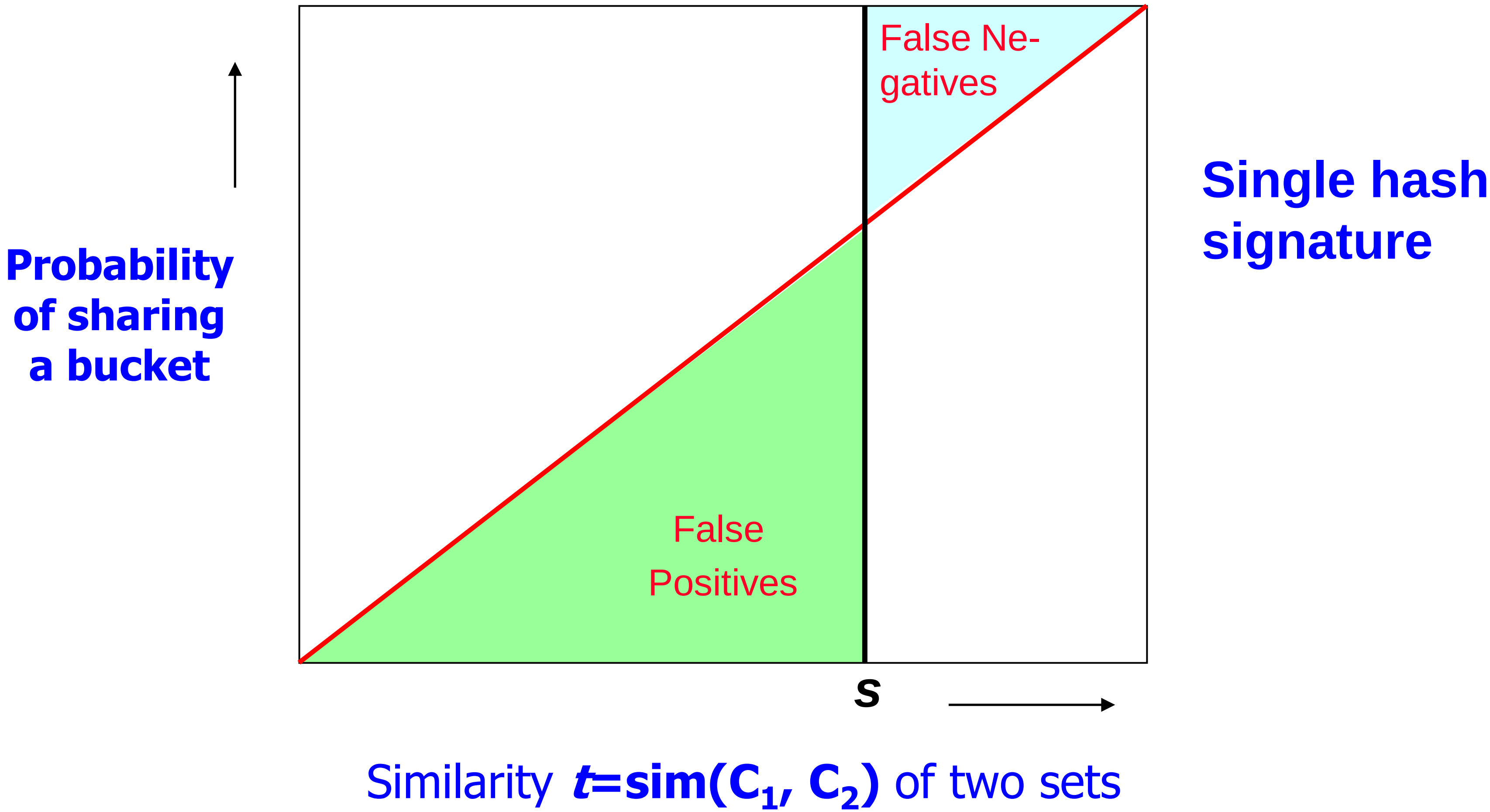
# LSH Involves a Tradeoff




- How to get a step-function?
- 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

# One Band of One Row

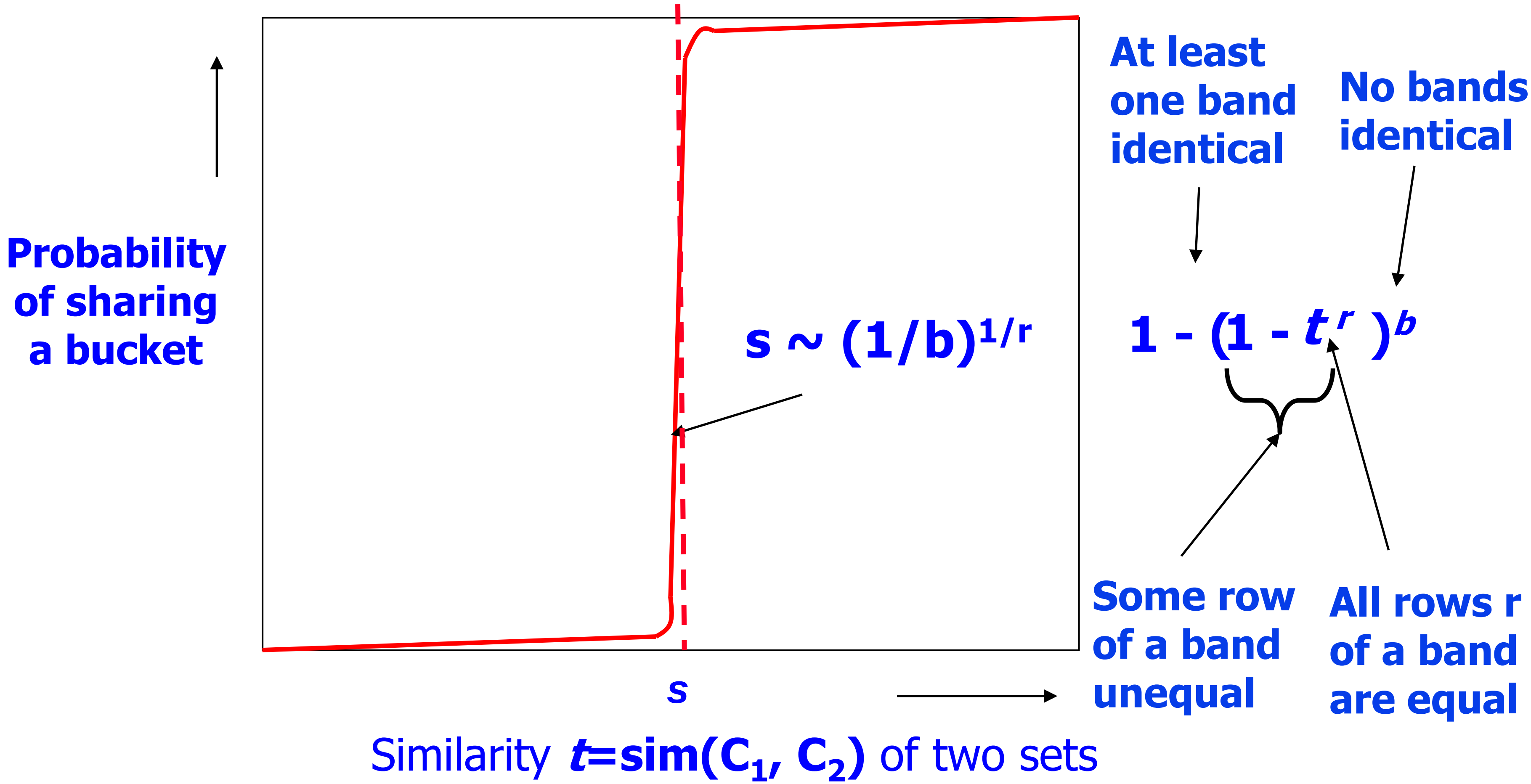



- Remember:

  $\Pr[h_\pi(C_1) = h_\pi(C_2)] = sim(C_1, C_2)$

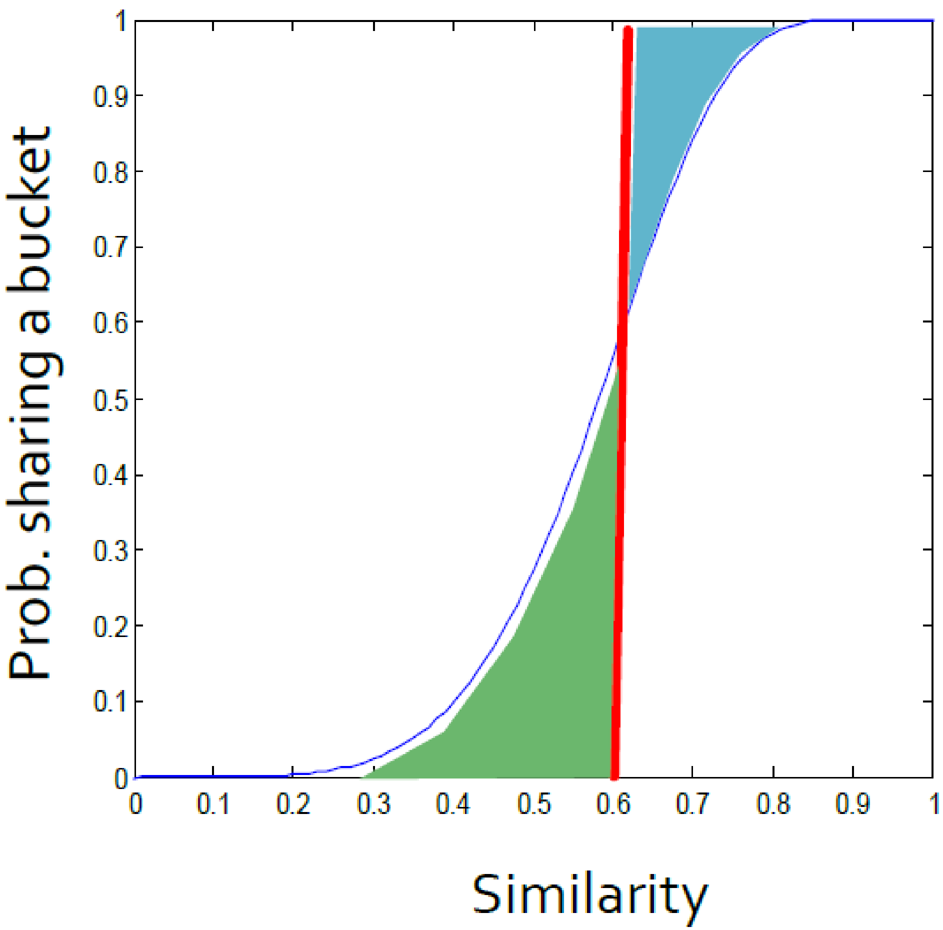
# *b* Bands of *r* Rows

- The S-curve is where the "magic" happens



# Picking *r* and *b*: The S-Curve

- Picking r and b to get the best S-curve

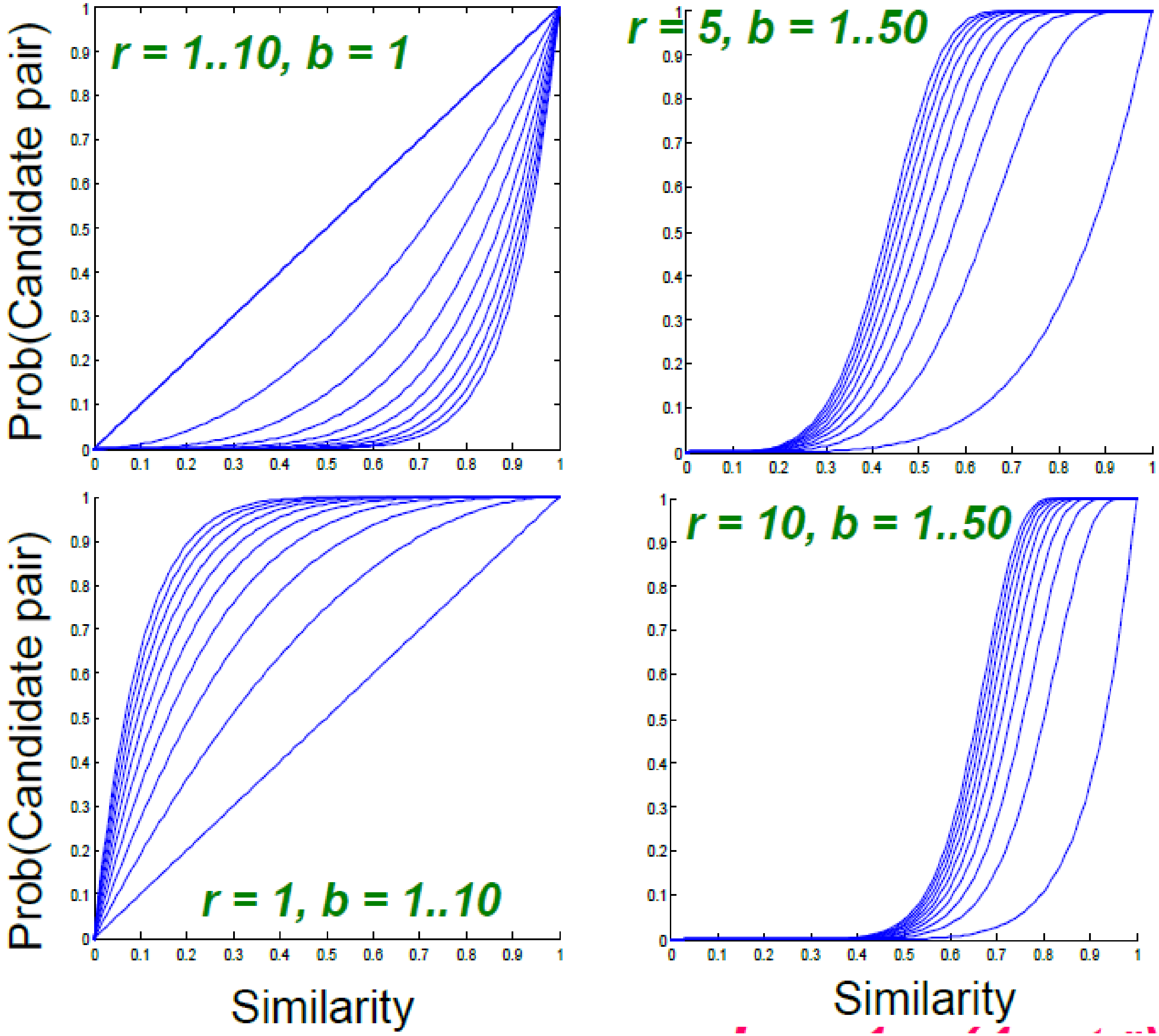



**Blue area**: *False Negative* rate
These are pairs with `sim > s` but the X fraction won't share a band and they will never become candidates. This means we will never consider these pairs for (slow/exact) similarity calculation!

**Green area**: *False Positive* rate
These are pairs with `sim < s` but we will consider them as candidates. This is not too bad, we will consider them for (slow/exact) similarity computation and discard them.

# S-curves as a Function of *b* and *r*

- Given a fixed threshold `s`
- We want choose `r` and `b` such that the `Pr(Candidate pair)` has a “step” right around `s`



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

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

$s = 0.5\ (\sim 1/20)^{1/5}$

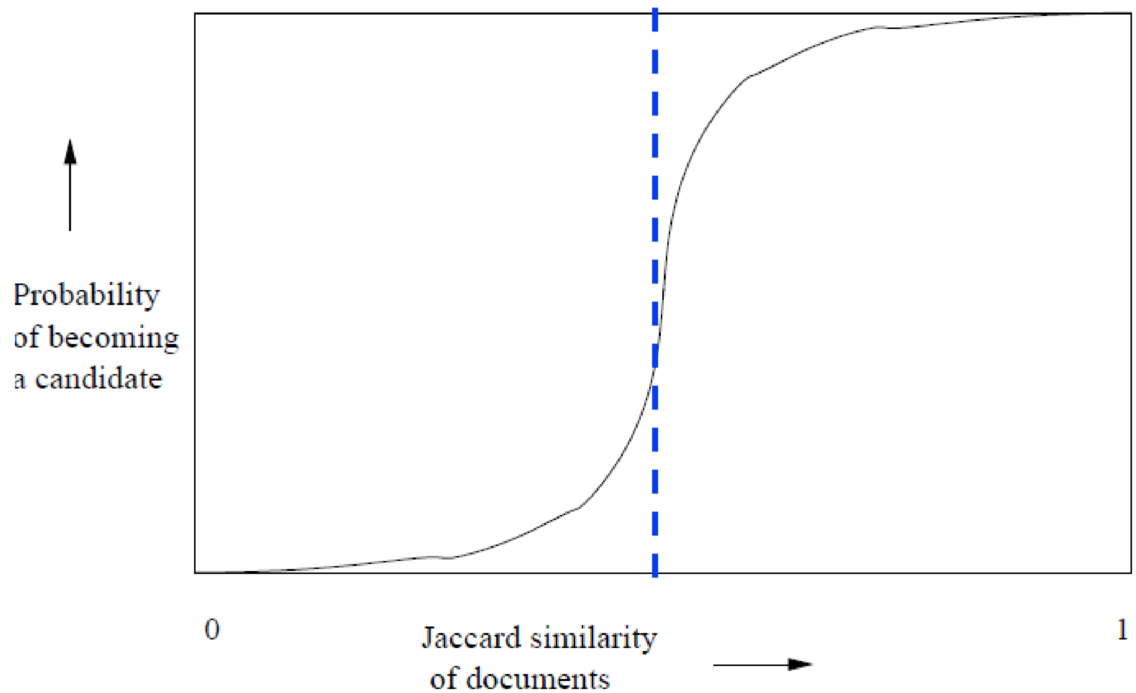
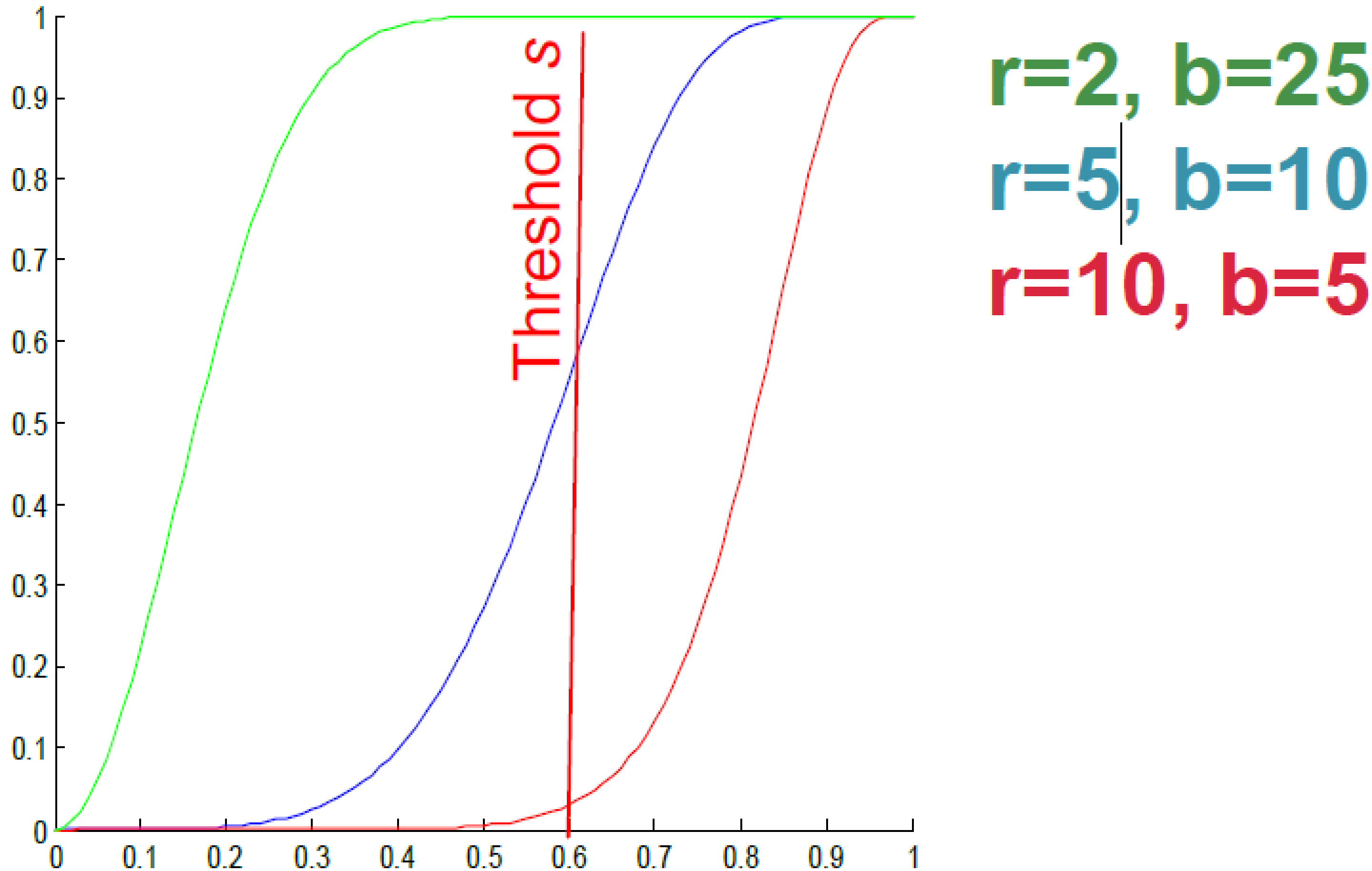



Figure 3.7: The S-curve

**if we had only 20 bands of 5 rows, the number of false negatives would go down, but the number of false positives would go up**

# Picking *r*, *b* to Get Desired Performance

- 50 hash-functions (r * b = 50)



# Limitations of Minhash

- Minhash is great for near-duplicate detection
  - Set high threshold for Jaccard similarity
- Limitations:
  - Jaccard similarity only
  - Set-based representation, no way to assign weights to features
- Random projections:
  - Works with arbitrary vectors using cosine similarity
  - Same basic idea, but details differ
  - Slower but more accurate: no free lunch!

# LSH Generalizations

# Multiple Hash Functions

- For Min-Hashing signatures, we got a Min-Hash function for each permutation of rows

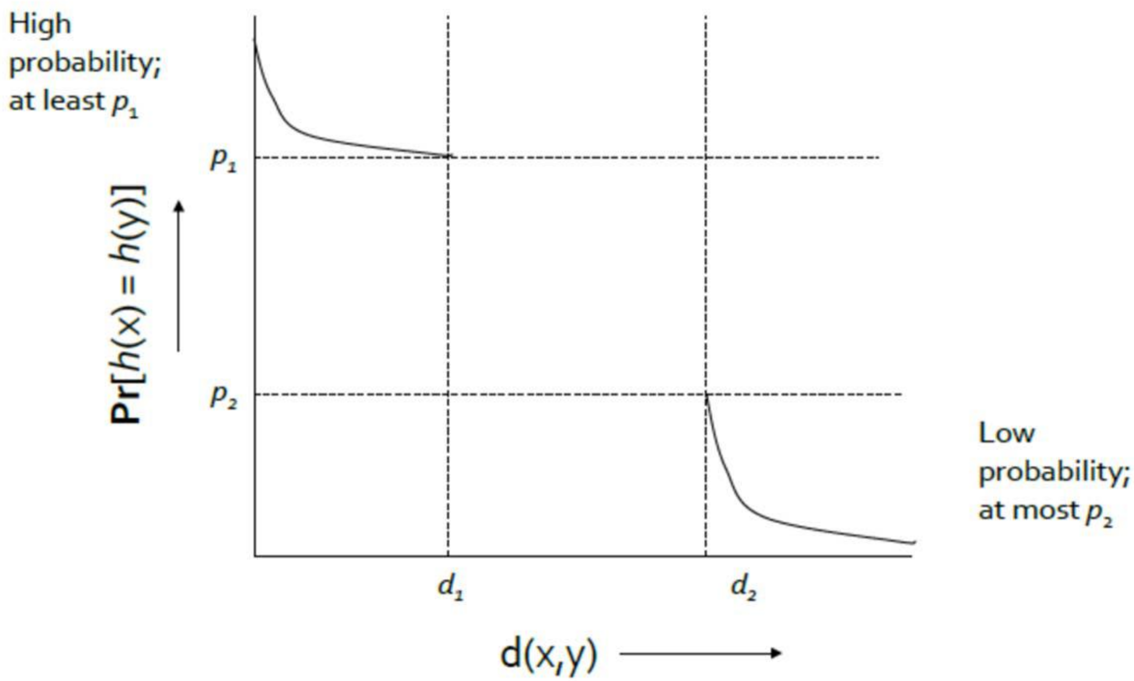
- So far, we have assumed only one hash function (even applied multiple times)
  - Shorthand: `h(x)=h(y)` implies "h says x and y are equal"

- We could have used a family of hash functions
  - A (large) set of related hash functions generated by some mechanism
  - We should be able to efficiently pick a hash function at random from such a family

# Locality-Sensitive (LS) Families

- Consider a space S of points with a distance measure d
- A family **H** of hash functions is said to be $(d_1, d_2, p_1, p_2)$- sensitive if for any x and y in S:
  - If $d(x,y) \le d_1$, then prob over all h in **H** that $h(x)=h(y)$ is at least $p_1$
  - If $d(x,y) \ge d_2$, then prob over all h in **H** that $h(x)=h(y)$ is at most $p_2$

Small distance, high probability of hashing to the same value




Large distance, low probability of hashing to the same value

# Example of LS Family: MinHash

- Let
  - **S** = space of all sets,
  - **d** = Jaccard distance,
  - **H** is family of Min-Hash functions for all permutations of rows
- Minhashing gives a $(d_1, d_2, p_1, p_2)$-sensitive family for any $d_1 < d_2$
  - E.g., **H** is a (1/3, 2/3, 2/3, 1/3)-sensitive family for S and d
  - If distance ≤ 1/3 (i.e., similarity ≥ 2/3), then probability that minhash values agree is ≥ 2/3
  - This is because for any hash function $h \in H$ $Pr(h(x)=h(y))=1-d(x,y)$
- Simply restates theorem about Min-Hashing in terms of distances rather than similarities!

# Example of LS Family: MinHash

- **Claim: Min-hash *H*** is a ($\boxed{1/3,}$ 2/3, $\boxed{2/3,}$ 1/3)-sensitive family for ***S*** and ***d***

If distance < 1/3
(so similarity ≥ 2/3)

Then probability that Min-Hash values agree ≥ 2/3

- For Jaccard similarity, Min-Hashing gives a $(d_1,d_2,(1\text{-}d_1),(1\text{-}d_2))$-sensitive family for any $d_1<d_2$

- Theory leaves unknown what happens to pairs that are at distance between $d_1$ and $d_2$
  - Consequence: No guarantees about fraction of false positives in that range

# Amplifying an LS-family

- Can we reproduce the "S-curve" effect we saw before for any LS family?

- The "banding" technique we learned for signature matrices carries over to this more general setting
  - So we can do LSH with any *(d1, d2, p1, p2)-sensitive* family

- Two constructions:
  - **AND** construction like "rows in a band"
  - **OR** construction like "many bands"

# AND Construction of Hash Functions

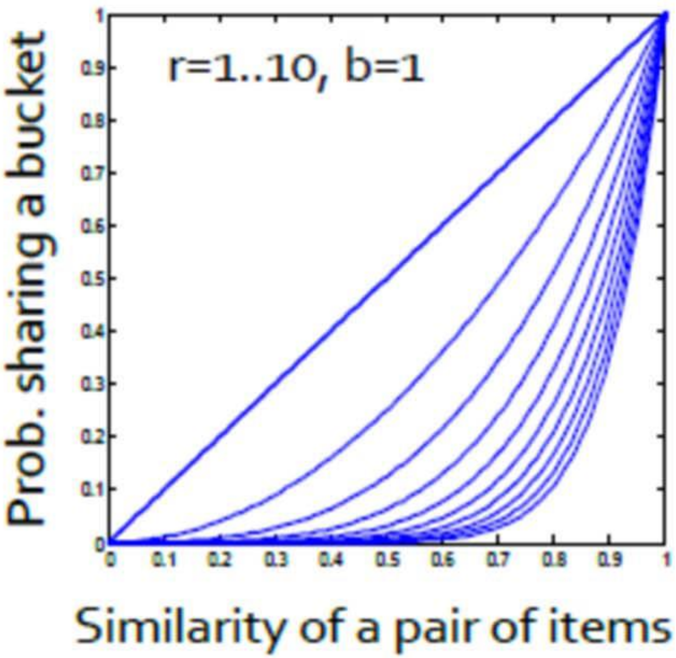
- Given family **H**, construct family **H'** consisting of *r* functions from **H**
- For $h=[h_1, \ldots, h_r]$ in **H'**, $h(x)=h(y)$ if and only if $h_i(x)=h_i(y)$ for all i: $1 \le i \le r$
- Note this has the same effect as "r signatures"
  - x and y are considered a candidate pair if every one of the r rows say that x and y are equal
- Theorem: If **H** is $(d_1, d_2, p_1, p_2)$-sensitive, then **H'** is $(d_1, d_2, p_1^r, p_2^r)$- sensitive
  - That is, for any p, if p is the probability that a member of **H** will declare (x,y) to be a candidate pair, then the probability that a member of **H'** will so declare is $p^r$
  - Proof**:** Use the fact that $\boldsymbol{h_i}$'s are independent

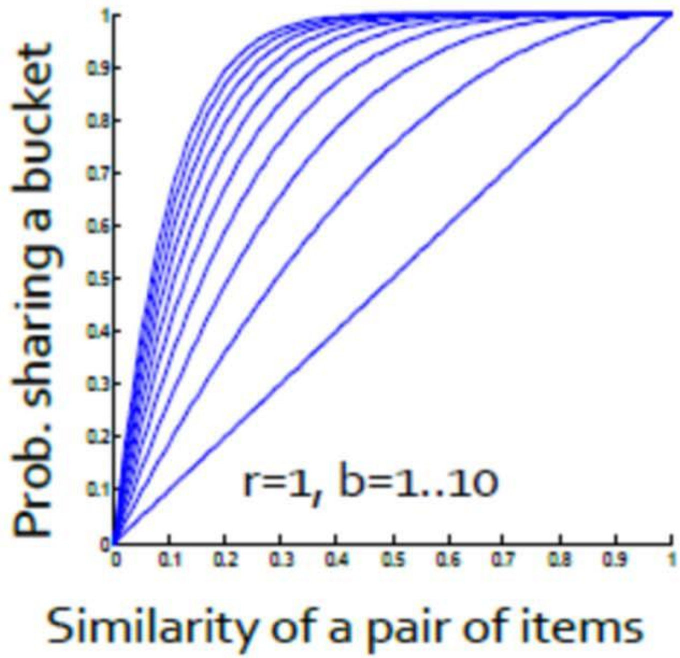
# OR Construction of Hash Functions

- Given family **H**, construct family **H'** consisting of *b* functions from **H**
- For $h=[h_1,\ldots,h_b]$ in **H'**, $h(x)=h(y)$ if and only if $h_i(x)=h_i(y)$ for at least one $i$, $1 \leq i \leq b$
- Mirrors the effect of combining "b bands":
  - x and y become a candidate pair if any set makes them a candidate pair
- Theorem: If **H** is $(d_1, d_2, p_1, p_2)$-sensitive, then **H'** is $(d_1, d_2, 1-(1-p_1)^b, 1-(1-p_2)^b)$-sensitive
  - That is, for any p, if p is the probability that a member of **H** will declare $(x,y)$ to be a candidate pair, then $(1-p)$ is the probability that it will not declare so
  - $(1-p)^b$ is the probability that none of the family $h_1$, $h_b$ will declare $(x,y)$ a candidate pair
  - $1-(1-p)^b$ is the probability that at least one $h_i$ will declare $(x,y)$ a candidate pair, and therefore that **H'** will declare $(x,y)$ to be a candidate pair

# Effect of AND & OR Constructions

- **AND** makes all probabilities shrink, but by choosing r correctly, we can make the *lower probability approach 0* while the higher does not
- **OR** makes all probabilities grow, but by choosing b correctly, we can make the *upper probability approach 1* while the lower does not




$y=1-(1-x^r)^b$

# Composing Constructions: AND-OR Composition

- $r$-way **AND** construction followed by $b$-way **OR** construction
  - Exactly what we did with minhashing
    - If $b$ bands match in all $r$ values hash to same bucket
    - Columns that are hashed into ≥ 1 common bucket -> candidate

- Take points $x$ and $y$ s.t. $Pr[h(x)=h(y)] = p$
  - **H** will make $(x,y)$ a candidate pair with probability $p$

- Construction makes $(x,y)$ a candidate pair with probability $1-(1-p^r)^b$
  - The S-Curve!

# Example

- Example: Take **H** and construct **H'** by the **AND** construction with *r* = 4. Then, from **H'**, construct **H''** by the **OR** construction with *b* = 4

- E.g., transform a (0.2, 0.8, 0.8, 0.2)-sensitive family into a (0.2, 0.8, 0.8785, 0.0064)-sensitive family

| p | $1-(1-p^4)^4$ |
|---|---|
| .2 | .0064 |
| .3 | .0320 |
| .4 | .0985 |
| .5 | .2275 |
| .6 | .4260 |
| .7 | .6666 |
| .8 | .8785 |
| .9 | .9860 |

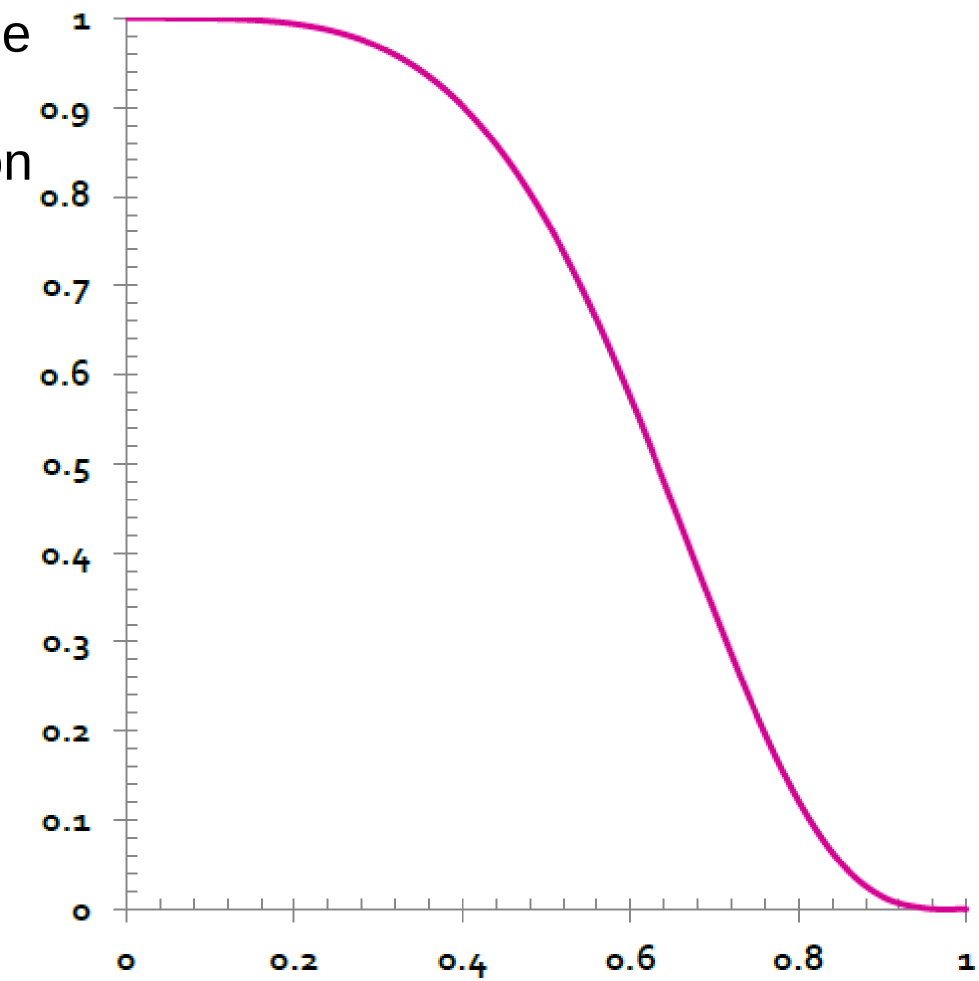
# Composing Constructions: OR-AND Composition

- b-way **OR** construction followed by r-way **AND** construction
- Transforms probability p into $(1-(1-p)^b)^r$
  - The same S-curve, mirrored horizontally and vertically

# Example

- Example: Take **H** and construct **H'** by the **OR** construction with b = 4. Then, from **H'**, construct **H''** by the **AND** construction with r = 4
- E.g., transform a (0.2, 0.8, 0.8, 0.2)-sensitive family into a (0.2, 0.8, 0.9936, 0.1215)-sensitive family

| p | $(1-(1-p)^4)^4$ |
|---|---|
| .1 | .0140 |
| .2 | .1215 |
| .3 | .3334 |
| .4 | .5740 |
| .5 | .7725 |
| .6 | .9015 |
| .7 | .9680 |
| .8 | .9936 |



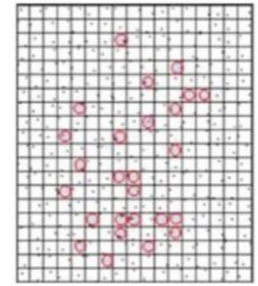
# Cascading Constructions

- Example: Apply the (4,4) OR-AND construction followed by the (4,4) AND-OR construction

- Transforms a (.2,.8,.8,.2)-sensitive family into a (.2,.8,.9999996,.0008715)- sensitive family
  - Note this family uses 256 (= 4*4*4*4) of the original hash functions

# Applications of LSH

# An LHS Family for Fingerprint Matching



- Fingerprint can be uniquely defined by its minutiae
- By overlaying a grid on the fingerprint image, we can extract the grid squares where the minutiae are located
- Two fingerprints are similar if the set of grid squares significantly overlap
  - Jaccard distance and minhash can be used, but …
- Let F be a family of functions
  - f ∈ F is defined by, say 3, grid squares such that f returns the same bucket whenever the fingerprint has minutiae in all three grid squares
  - f sends all fingerprints that have minutiae in all three of f's grid points to the same bucket
  - Two fingerprints match if they are in the same bucket

# LSH for Fingerprint Matching

- Suppose probability of finding a minutiae in a random grid square of a random finger is 0.2
- And probability of finding one in the same grid square of the same finger (different fingerprint) is 0.8
- Prob two fingerprints from different fingers match=$(0.2)^3$x $(0.2)^3$= 0.000064
- Prob two fingerprints from the same finger match=$(0.2)^3$x $(0.8)^3$= 0.004096
- Use more functions from F!
- Take 1024 functions and do a OR construction
    - Prob putting the fingerprints from the same finger in at least one bucket is $1 - (1-0.004096)^{1024} = 0.985$
    - Prob two fingerprints from different fingers falling into the same bucket is $1 - (1-0.000064)^{1024} = 0.063$
    - We have 1.5% false negatives and 6.3% false positives
- Using AND construction will
    - Greatly reduce the prob of a false positive
    - Small increase in false-negative rate

# References


- CS9223 – Massive Data Analysis J. Freire & J. Simeon New York University Course 2013
- CS246: Mining Massive Datasets Jure Leskovec, Stanford University, 2014
- CS5344: Big Data Analytics Technology, TAN Kian-Lee, National University of Singapore 2014