Mining Massive Datasets

Lecture 7

Artur Andrzejak
http://pvs.ifi.uni-heidelberg.de

RUPRECHT-KARLS-
UNIVERSITÄT
HEIDELBERG

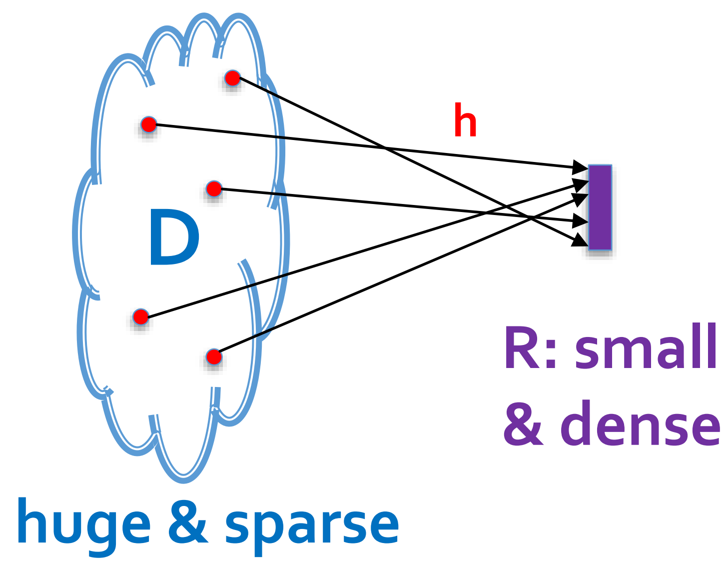
Note on Slides

A substantial part of these slides come (either verbatim or in a modified form) from the book
Mining of Massive Datasets
by Jure Leskovec, Anand Rajaraman, Jeff Ullman
(Stanford University).
For more information, see the website accompanying the book: http://www.mmds.org.

A Word on Hash Functions and Data Structures

Hash Functions /1

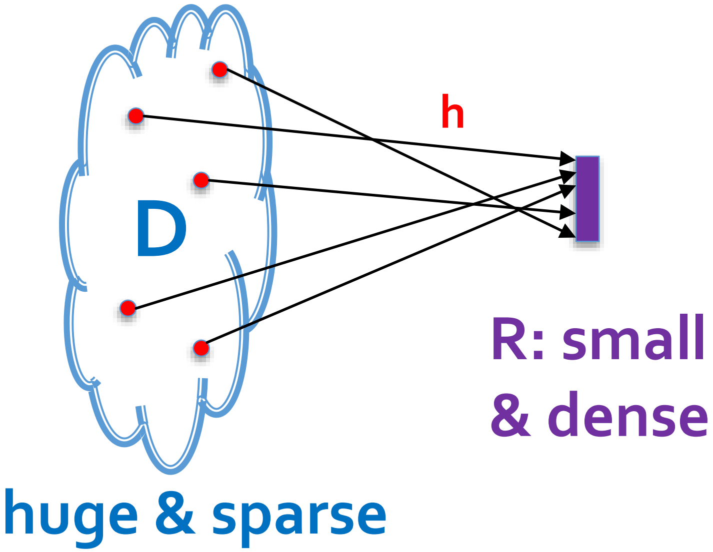
- A **hash function h:D→R** maps objects from a (usually huge) domain space **D** to a (usually small) range **R** of (consecutive) integers (“**buckets**”)
  - E.g. **D** = set of all files in HDFS, **R = [0,…,b-1]**, b=100
- Intuitively, h computes a short **signature** of an object
- **h** is in general not injective => **collisions** possible (i.e. $h(x)=h(y)$ for $x \neq y$)



Hash Functions /2

- Another view: mapping from a <u>sparse</u> storage for D to <u>dense</u> storage (for R)
- A <u>table</u> containing <u>all possible</u> objects in D would be too large, but a table for <u>all buckets</u> in R is possible
- Implementation?

- **x** $\in$D is treated as integer
- **h(x) = x mod b** (b preferably a prime)



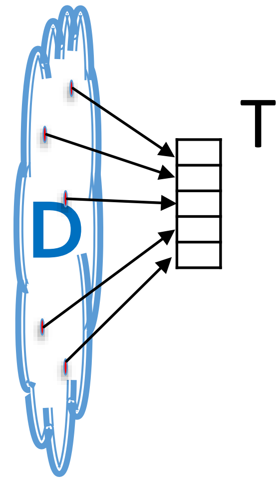
HashSet /HashMap, Dictionary

- Problem of storing and fast search for elements is very common => optimized data structures
- Java
  - HashSet: a set implementation
  - HashMap: for storing pairs <key, value> and fast finding of values by a key
  - TreeMap: also for <key, value>, use balanced trees
- Python, C++ (STL):
  - dictionary, uses hashing

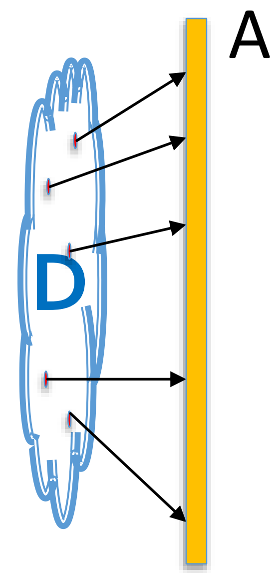
Example: Implementing Sets /1

Given a set **M** ⊆ D, we want to test: *is q ∈ M?*

- Solution 1: sorted table
  - Storage: <u>sort</u> all <u>existing</u> elements by their IDs, store them (or references) in a table T
  - Query for element q: binary search for q in T
    - Assume that q is like a number, e.g. a bit sequence

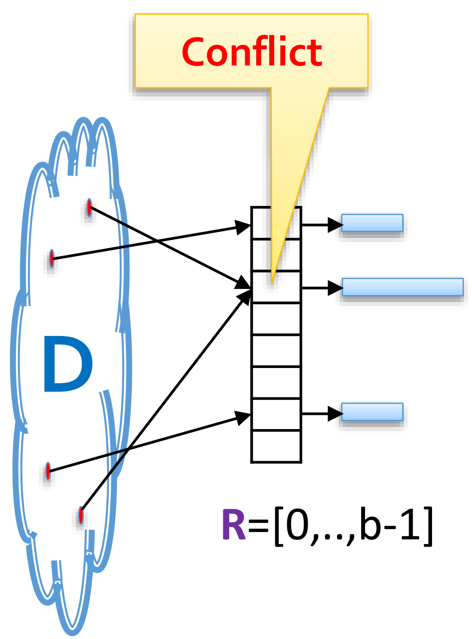


- Solution 2: bit array
  - Storage: Create a huge bit array A <u>covering whole D</u> and set A[x] = 1 iff x ∈ M
  - Query for q: make a <u>lookup</u> A[q]
    - *True* iff q is in the set M
  - <u>Very fast</u> but we need a huge and sparse array (e.g. elements = strings of fixed length)



Example: Implementing Sets /2

- Solution 3: hash table
- Storage and setup:
  - Fix an integer b ≈ c*|**M**| and a hash fn **h**:D→**R**, **R** = [0,..,b-1]
  - Create an array **T** of size b with lists (at first: empty lists)
  - If x ∈ **M**: update list L = **T**[**h**(x)] by adding x (or a ref) to L
- Query for element q:
  - Make a lookup L= **T**[**h**(q)]; if list L not empty, check q ∈ L => yes iff q ∈ **M**
- Runtime, space requirements?
  - Runtime: fast, O(1) for search and inserting
  - Space: b references to lists, b lists, plus approx. size of all elements in M



Finding Similar Items: Locality Sensitive Hashing

Locality Sensitive Hashing

High dim. data	Graph data	Infinite data	Machine learning	Apps
Locality sensitive hashing	PageRank, SimRank	Filtering data streams	SVM	Recommender systems
Clustering	Community Detection	Web advertising	Decision Trees	Association Rules
Dimensionality reduction	Spam Detection	Queries on streams	Perceptron, kNN	Duplicate document detection

Programming in Spark & MapReduce

A Common Metaphor

- Many problems can be expressed as finding "similar" objects:
  - Find near-neighbors in high-dimensional space

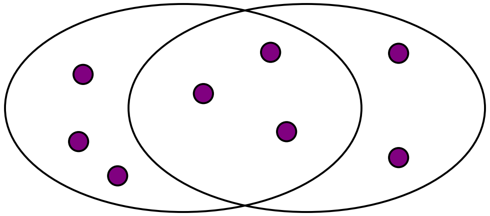
- Examples:
  - Points in the same cluster (clustering)
  - Pages with similar words
    - For duplicate detection, classification by topic
  - Customers who purchased similar products
    - Products with similar customer sets

Problem for Today's Lecture

- **Given: High dimensional data points $x_1, x_2, \ldots$**
  - **For example:** Image is a long vector of pixel colors
    $$\begin{bmatrix} 1 & 2 & 1 \\ 0 & 2 & 1 \\ 0 & 1 & 0 \end{bmatrix} \rightarrow [1\ 2\ 1\ 0\ 2\ 1\ 0\ 1\ 0]$$
- And some distance function $d(x_1, x_2)$
  - Which quantifies the "distance" between x_1 and x_2
- **Goal:** Find **all pairs of data points** (x_i, x_j) that are within some distance threshold $d(x_i, x_j) \leq s$
- **Note:** Naïve solution would take $O(N^2)$ ☹ (N = #data points), see hierarchical clustering
- **MAGIC**: This can be done in $O(N)$!! How?

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 a "distance" metric
- E.g. Jaccard distance/similarity (recall)
  - 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

Task: Finding Similar Documents

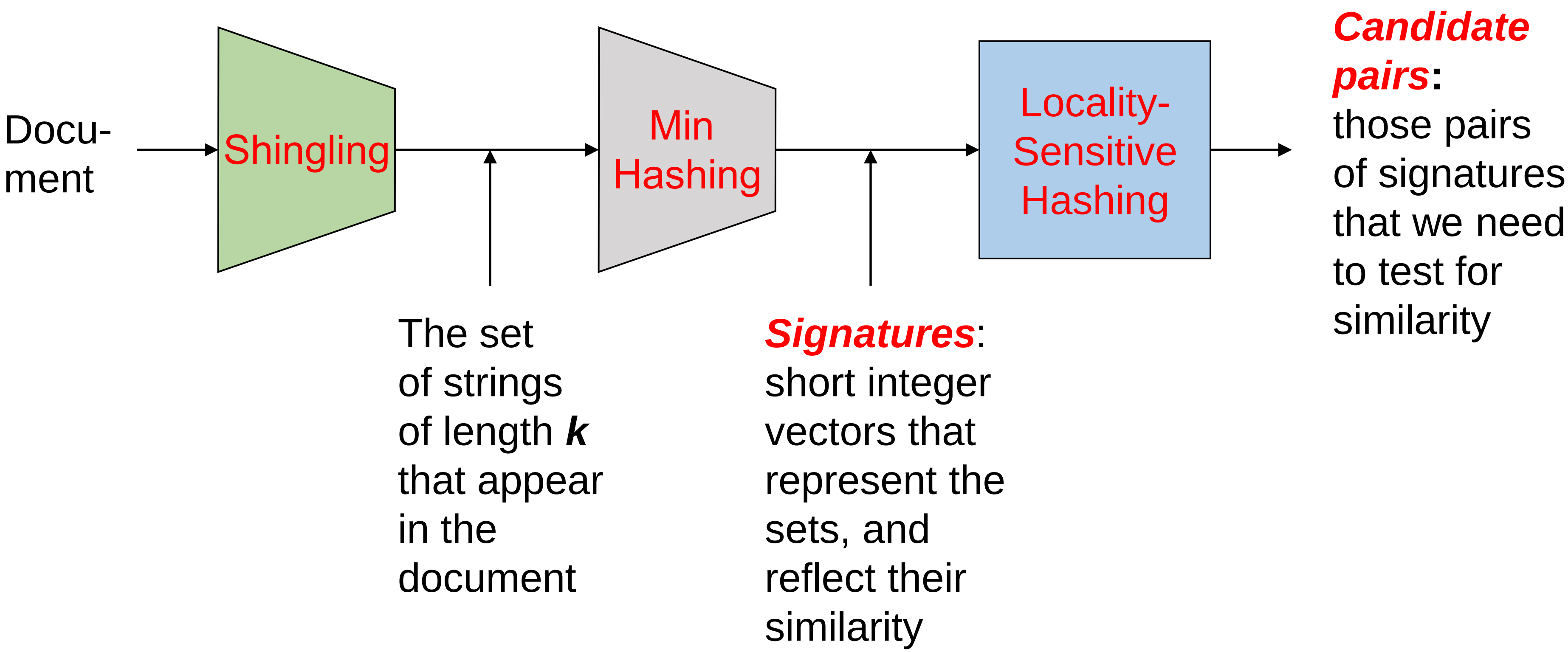
- Goal: Given a large number (N in the millions or billions) of documents, find “near duplicate” pairs
- Applications:
  - Mirror websites, or approximate mirrors
    - Don’t want to show both in search results
  - Similar news articles at many news sites
    - Cluster articles by “same story”
- Problems: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

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:*** Identify pairs of signatures likely to be from similar documents

- We get (few) candidate pairs! => Those can be checked by "expensive" methods

The Big Picture



Shingling

Documents as High-Dim Data

- Step 1: *Shingling:* Convert documents to sets
- Simple approaches:
  - Document = set of words appearing in document
  - Document = set of "important" words
- Don't work well for this application. Why?
- Need to account for ordering of words!
- A different way: Shingles!

Define: Shingles

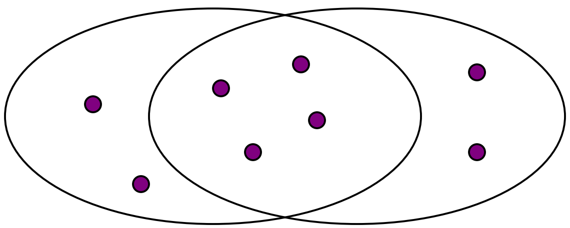
- A ***k*-shingle** (or ***k*-gram**) for a document is a sequence of *k* (consecutive) tokens that appears in the doc
  - Tokens can be characters, words or something else, depending on the application
- Assume for examples: tokens = characters

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

Similarity Metric for Shingles

- Now: doc. D_1 is a set of its k-shingles: $C_1=S(D_1)$
- Equivalently, each document is a 0/1 vector in the space of *k*-shingles
  - Each unique shingle is a dimension
  - Vectors are very sparse
- For sets, a natural similarity measure is the **Jaccard similarity:**

$$sim(D_1, D_2) = |C_1 \cap C_2| / |C_1 \cup C_2|$$



Working Assumption

- Documents that have lots of shingles in common have similar text
  - Even if the text appears in different order
- **Caveat**: You must <u>pick *k* large enough</u>, or most documents will have most shingles
  - $k = 5$ is OK for short documents
  - $k = 10$ is better for long documents

Compressing Shingles (Optional)

- To compress long shingles, we can **hash** them to (say) 4 bytes (value of a hash function)
- => We can represent a document by the set of hash values of its *k*-shingles
- Example: k=2; document D_1= abcab:
  - Set of 2-shingles: $S(D_1)$ = {ab, bc, ca}
  - Set of hash val's of the shingles e.g.: {1, 5, 7}
- **Note:** Two documents could (rarely) appear to have shingles in common, when in fact only the hash-values were shared

Motivation for Minhash/LSH

- Suppose we need to find near-duplicate documents among $\mathrm{N} = 1$ million documents

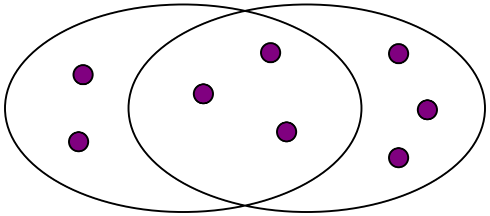
- Naïvely, we would have to compute pairwise Jaccard similarities for every pair of docs
- How long (at 10^6 comparisons/sec)?
  - $N(N-1)/2 \approx 5*10^{11}$ comparisons
  - At 10^5 secs/day and 10^6 comparisons/sec, it would take 5 days

- For $N = 10$ million, it takes more than a year…

MinHashing

Minhashing: Converting **large sets** to **short signatures**, while **preserving similarity**

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
  - Interpret set intersection as bitwise **AND**, and set union as bitwise **OR** => fast
- **Example:** $\mathbf{C_1}$ = 10111; $\mathbf{C_2}$ = 10011
  - Size of intersection **= 3**; size of union **= 4**
  - **Jaccard similarity** (not distance) **= 3/4**
  - **Distance: $\mathbf{d(C_1,C_2)}$ = 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: 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 – (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

Outline: Finding Similar Columns

- **So far:**
  - Documents $\rightarrow$ Sets of shingles
  - Represent sets as Boolean vectors in a matrix
- **Next goal:** Find similar columns from small (similar) signatures
  - Similarity of columns == similarity of signatures

Outline: Finding Similar Columns

- **Next Goal:** Find similar columns from small signatures
- **Naïve approach:**
  - **1) Signatures of columns:** small summaries of columns
  - **2) Examine pairs of signatures** to find similar columns
    - **Essential:** Similarities of signatures and columns are related
  - **3) Optional:** Check that columns with similar signatures are really similar
- **But ..** comparing all pairs of signatures may take too much time: **Job for LSH**
  - Later

Hashing Columns (Signatures)

- **Key idea:** map each column C to a small ***signature*** $h(C)$, such that:
  - **(1)** $h(C)$ is small enough that the signature fits in RAM
  - **(2)** $sim(C_1, C_2)$ is the same as the "similarity" of signatures $h(C_1)$ and $h(C_2)$

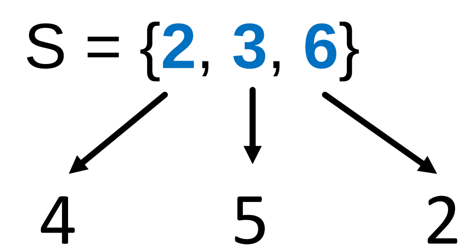
- **Goal: Find a hash function $h(\cdot)$ such that:**
  - If $sim(C_1,C_2)$ is high, then with high prob. $h(C_1) = h(C_2)$
  - If $sim(C_1,C_2)$ is low, then with high prob. $h(C_1) \neq h(C_2)$

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 (for sets C_1, C_2):
  - Not all similarity metrics have a suitable hash function!
- There is a suitable hash function for the Jaccard similarity: it is called **MinHash**

MinHash: Definition

- Def.: Let **h** be a hash function that maps the members of S to distinct integers, and for any set S define **MinHash$_h$**(S) = **h_{min}**(S) to be the minimum value of **h**(x)
- **Example**:
  - Assume S = {**2**, **3**, **6**} and
  - **h**(**2**) = 4, **h**(**3**) = 5, **h**(**6**) = 2



- What is **h_{min}**(S)?
- Of course, **h_{min}**(S) = 2

Min-Hashing: Other Interpretation

- Recall: we represent each set as a Boolean vector **C** (here: S = {**2**, **3**, **6**})
- Assume that a hash function **h** is given by a (random) permutation π of the rows of the Boolean vector
  - **h** fulfills: "… maps the members of S to distinct integers"

	C
1	0
2	1
3	1
4	0
5	0
6	1

Row perm.

$\pi(1) = 3$
$\pi(2) = 4$
$\pi(3) = 5$
$\pi(4) = 6$
$\pi(5) = 1$
$\pi(6) = 2$

1	
2	1
3	
4	1
5	1
6	

- Then **MinHash**$_h$(S) is the index of the **first row** of the permuted column ***C*** with value **1**
- => Again, $\mathbf{h}_{\pi,\min}(S) = 2$

Example of Other Interpretation

- "Goes to" representation of a permutation:
  - Original row with index **r** goes to row π(**r**)
- Recall: Then **MinHash$_h$**(S) is the index of the first **row** of the permuted column ***C*** with value **1**

1	0	$\pi(1) = 3$		0
2	1	$\pi(2) = 4$	⇒	1
3	0	$\pi(3) = 1$		0
4	1	$\pi(4) = 2$		1

- $h_{min,\,\pi}(C) = 2$

Min-Hashing on Matrices

- Recall:
  - **Rows** = elements (shingles)
  - **Columns** = sets (documents)
    - 1 in row ***e*** and column ***s*** if and only if ***e*** is a member of ***s***
- For each column (= set):
  - We use <u>several</u> (e.g., 100) <u>independent</u> hash functions (that is, permutations) to create a **signature of a column**

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 Example

Value **n** at position **p** means:
"previous row **p** goes to row **n** in the new ordering"

2nd element of the permutation
is the first to map to a 1

Permutation π

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

Input matrix (Shingles x Documents)

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*

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

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

The Min-Hash Property

- **Choose a random permutation π**
- **Claim:** $\Pr[h_\pi(C_1) = h_\pi(C_2)] = sim(C_1, C_2)$
  - I.e. the probability that the minhash-values $h_\pi(C_1)$ of C_1 and $h_\pi(C_2)$ of C_2 agree (under the permutation π)
    is
    exactly the Jaccard-similarity of C_1 and C_2

"Proof": Four Types of Rows

- Given cols 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

a = # rows of type A,
b = # rows of type B,
etc.

- **Note:** Jaccard-sim. is: **$sim(C_1, C_2) = a/(a+b+c)$**
- **Then:** **Pr**$[h(C_1) = h(C_2)] = Sim(C_1, C_2)$
  - Look down the permuted cols C_1 and C_2 until we see a 1
  - There are a+b+c rows at which we can stop
  - The prob. that we stopped at type-A row is a/(a+b+c)
    - If it's a type-*A* row, then $h(C_1) = h(C_2)$; $h(C_1) \neq h(C_2)$ for types B/C

MH-Property: Detailed Proof

0	0
0	0
1	**1**
0	0
0	1
1	0

- **Detailed proof:**
  - Let **X** be a set (of shingles), $\boldsymbol{y} \in \boldsymbol{X}$ an element
  - **Then: Pr[π(y) = min(π(X))] = 1/|X|**
    - It is equally likely that any $\boldsymbol{y} \in \boldsymbol{X}$ is mapped to the ***min***
  - Let ***y*** be s.t. $\pi(y) = \min(\pi(C_1 \cup C_2))$
  - **Then either:** $\pi(y) = \min(\pi(C_1))$ if $y \in C_1$, **or**
    $\pi(y) = \min(\pi(C_2))$ if $y \in C_2$

One of the two cols had to have 1 at position ***y***

  - So the prob. that **both** are true is the prob. $\boldsymbol{y} \in C_1 \cap C_2$
  - $\mathbf{Pr[\min(\pi(C_1))=\min(\pi(C_2))]=|C_1 \cap C_2|/|C_1 \cup C_2|}$
    $= \mathit{sim}(C_1, C_2)$

Similarity for Signatures

- We know: **Pr[h_π(C$_1$) = h_π(C$_2$)] = *sim*(C$_1$, C$_2$)**
- We are interested in ***sim*(C$_1$, C$_2$)**
  - => We need to estimate **Pr[h_π(C$_1$) = h_π(C$_2$)]**
- But testing **h_π(C$_1$) = h_π(C$_2$)** gives us only true or false!

- **Idea (compare with estimating $\pi/4$):**
  - **Given a random variable X with values 0, 1 and a distribution (1-p, p) : sample X many times to estimate p!**
  - => Use many different min-hash functions and compute the fraction of cases in which they agree

- Definition: the *similarity of two signatures* is the fraction of the hash functions in which they agree
- This approximates ***sim*(C$_1$, C$_2$)**

Min-Hashing Example

Permutation π

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

Input matrix (Shingles x Documents)

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*

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

Min-Hash Signatures

- **Pick K=100 random permutations of the rows**
- Think of ***sig*(C)** as a column vector
  - ***sig*(C)[i] =** according to the *i*-th permutation, the index of the first row that has a 1 in column *C*

$$\boldsymbol{sig(C)[i] = \min(\pi_i(C))}$$

- **Note:** The sketch (signature) of document *C* is small **~100 bytes!**

- We achieved our goal! We "compressed" long bit vectors into short signatures

Implementation /1

- Permuting rows even once is prohibitive
- Approximate permutation by hash functions h_i
  - $h_i(x)=[((a\cdot x+b) \bmod p) \bmod N]+1$
  - a, b: random ints, p: prime (p > N), N: #rows in the matrix
  - h_i is possibly not injective, but errors are rare => OK
- “Goes to” representation of a permutation:
  - Original row with index **r** goes to row h_i(**r**)

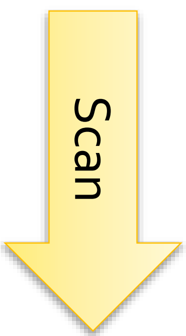
1	**a**	$h_i(1) = 3$		**c**
2	b	$h_i(2) = 4$		d
3	c	$h_i(3) = 1$		a
4	d	$h_i(4) = 2$		b

Implementation /2

- Pick about **K = 100** hash functions h_i
- One-pass implementation
  - For each column C and hash-func. h_i keep a "slot" ***sig(C)[i]*** for the min-hash value
  - Initialize all ***sig(C)[i]*** $= \infty$
  - **Scan rows looking for 1s**
    - If row q has 1 in column C, then for each h_i (i=1..100):
      - If $h_i(q) <$ ***sig(C)[i]***, then ***sig(C)[i]*** $\leftarrow h_i(q)$

Implementation /3

- **Scan rows looking for 1s**
  - If row ***q*** has 1 in column ***C***, then for each h_i:
    - If $h_i(q) <$ ***sig(C)[i]***, then ***sig(C)[i]*** $\leftarrow h_i(q)$
- Example: fixed ***C*** and h_i



	C		*sig(C)[i]*	*sig(C)[i]*
1	**0**	$h_i(1) = 3$		∞
2	1	$h_i(2) = 4$		4
3	1	$h_i(3) = 1$		1
4	0	$h_i(4) = 2$		1

Intuitively: find smallest value $h_i(r)$ for a row r with C(r) = 1

Thank you.

Questions?