Locality-sensitive hashing and its applications

Paolo Ferragina

University of Pisa

ACM Kanellakis Award 2013

A frequent issue

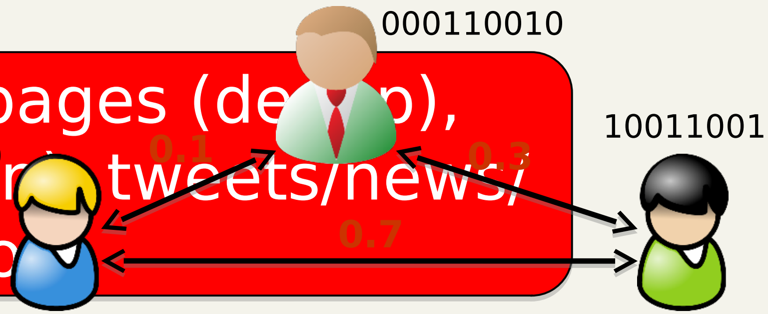
Given U users, described with a set of **d** features, the goal is to find (the largest) group of *similar users*

Features = Personal data, preferences, purchases, navigational behavior, search behavior, followers/ing, ...

- A feature is typically a numerical value: binary or real

Similarity(u1,u2) is a function that, taken the set of features of users u1 and u2, returns a value in [0,1]

Users could also be Web pages (dedup), products (recommendation), tweets/news/ search results (visualization)



Solution #1

Try all *groups of users* and, for each group, check the (average) similarity among all its users.

Sim computations $\cong$ $2^U \times U^2$

In the case of Facebook this is > $2^{1billion} \times (10^9)^2$

If we limit groups to have a size **≤ L** users

Sim computations $\cong$ $\boxed{U^L \times L^2}$

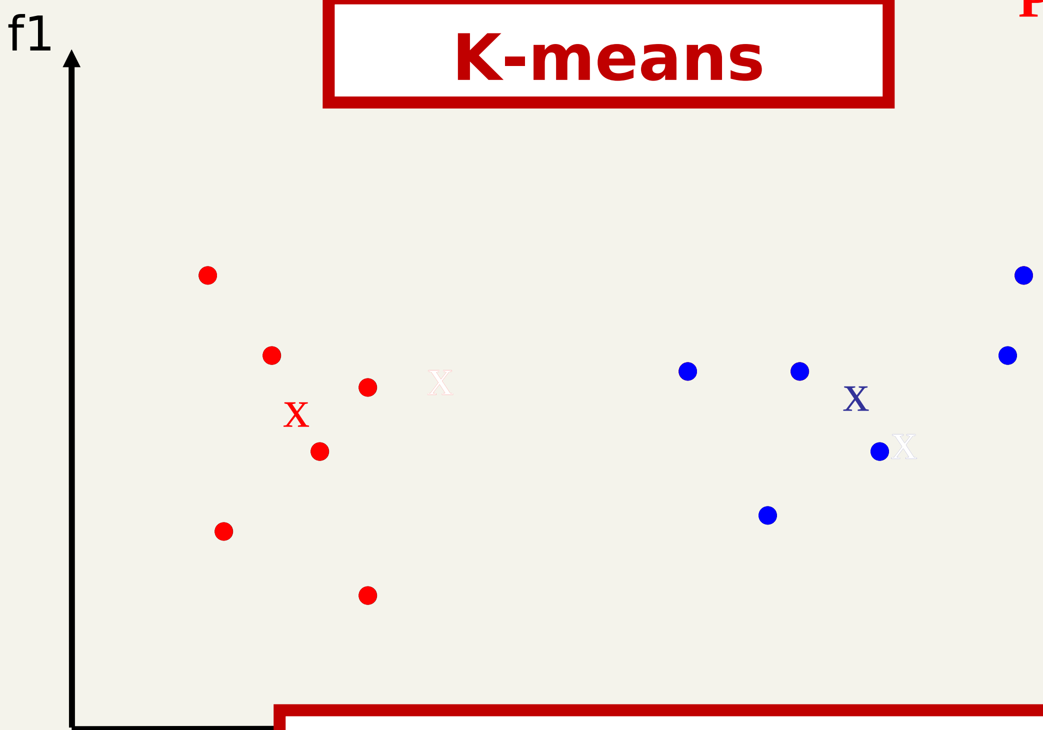
(Even if 1ns/sim and L=10, it is > $(10^9)^{10}/10^9$ secs > 10^{70} years)

No faster CPU/GPU, multi-cores,… could help !

Solution #2: introduce approximation

Interpret every user as a point in a **d**-dim space, and then apply a ***clustering*** algorithm

K-means



Pick K=2 centroids at random

Compute clusters

Re-determine centroids

Re-compute clusters

Re-determine centroids

Re-compute clusters

Converged!

Each iteration takes ≅ K × U

Solution #2: few considerations

- Cost per iteration = **K × U,** #iterations is typically small
- **What about optimality ?** It is locally optimal
  [recently, some researchers showed how to introduce some guarantee]
- **What about the Sim-cost ?** Comparing users/points costs Θ(d) in time and space [notice that d may be bi/mi
- **Wha** **³ < Uᴸ**
  [≅y

In T time, we can manage U = $T^{1/3}$ users

Using s-faster CPU ≈ using sT time an old CPU

→ we can manage $(s*T)^{1/3} = s^{1/3} T^{1/3}$ users

Solution #3: introduce randomization

Generate a **fingerprint** for every user that is **much shorter** than **d** and allows to transform *similarity* into ***equality*** *of fingerprints*.

- ✓ It is ***randomized***, correct ***with high probability***
- ✓ It guarantees ***local access*** to data, which is good for speed in disk/distributed setting

ACM Kanellakis Award 2013

A warm-up problem

- Consider vectors *p,q* of **d** *binary* features

- Hamming distance

  D(p,q)= #bits where *p* and *q* differ

- Define hash function *h* by choosing a set *I* of *k* random coordinates

  h(p) = projection of vector *p* on *I's* coordinates

Example: P*ick I={1,4} (k=2), then h(p=**0**10**1**1) =01*

A key property

Pr[*picking x s.t. p[x]=q[x]*]= (d - D(*p*,*q*))/d

$$\Pr[h(p)=h(q)] = \left(1-\frac{D(p,q)}{d}\right)^k$$

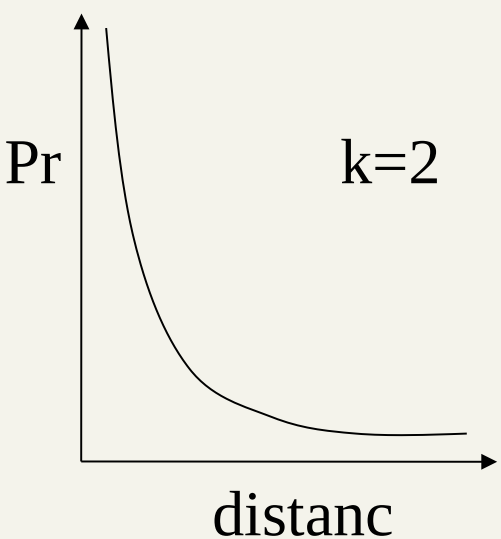
p versus q

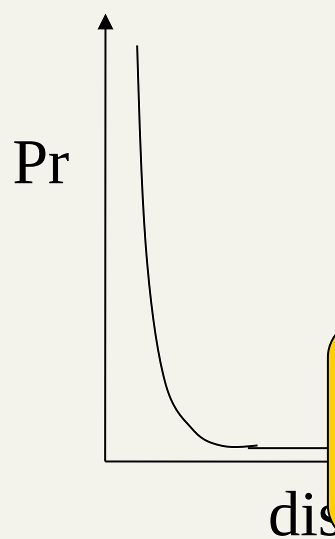
\# = D(*p*,*q*)

\# = d - D(*p*,*q*)

= S^k
where s is the similarity between p and q

We can vary the probability by changing





Larger k
Smaller False Positive
What about False Negatives?

Reiterate L times

Larger L
Smaller False Negatives

1) Repeat **L times** the **k-projections** $h_i(p)$
2) We set $g(p) = < h_1(p), h_2(p), \ldots, h_L(p)>$ Sketch(p)
3) Declare «p matches q» if **at least** one $h_i(p)=h_i(q)$

Example:

*We set k=2, L=3, let p = 01**0**01 and q = 01**1**01*

•*I1 = {3,4}, we have $h_1(p)$ = 00* and *$h_1(q)$=10*

•*I2 = {1,3}, we have $h_2(p)$ = 00* and *$h_2(q)$=01*

•*I3 = {1,5}, we have $h_3(p)$ = 01* and *$h_3(q)$=01*

p and q declared to match !!

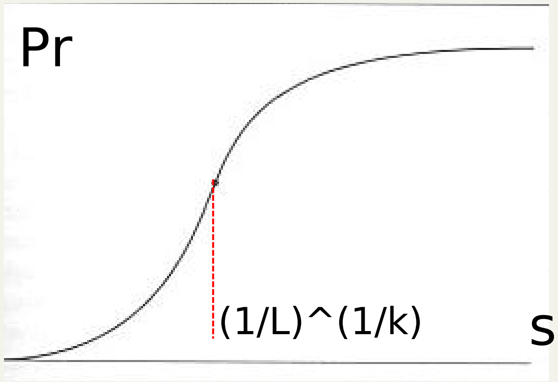
Measuring the error probability

$$\Pr[h_i(p) = h_i(q)] = (1 - \frac{D(p,q)}{d})^k = s^k$$

The g() consists of L independent hashes h_i

Pr[g(p) matches g(q)] = 1 - Pr[h_i(p) ≠ h_i(q), *∀i=1, ..., L*]

$$\Pr\left(g(p) \cong g(q)\right) = 1 - \left(1 - s^k\right)^L$$

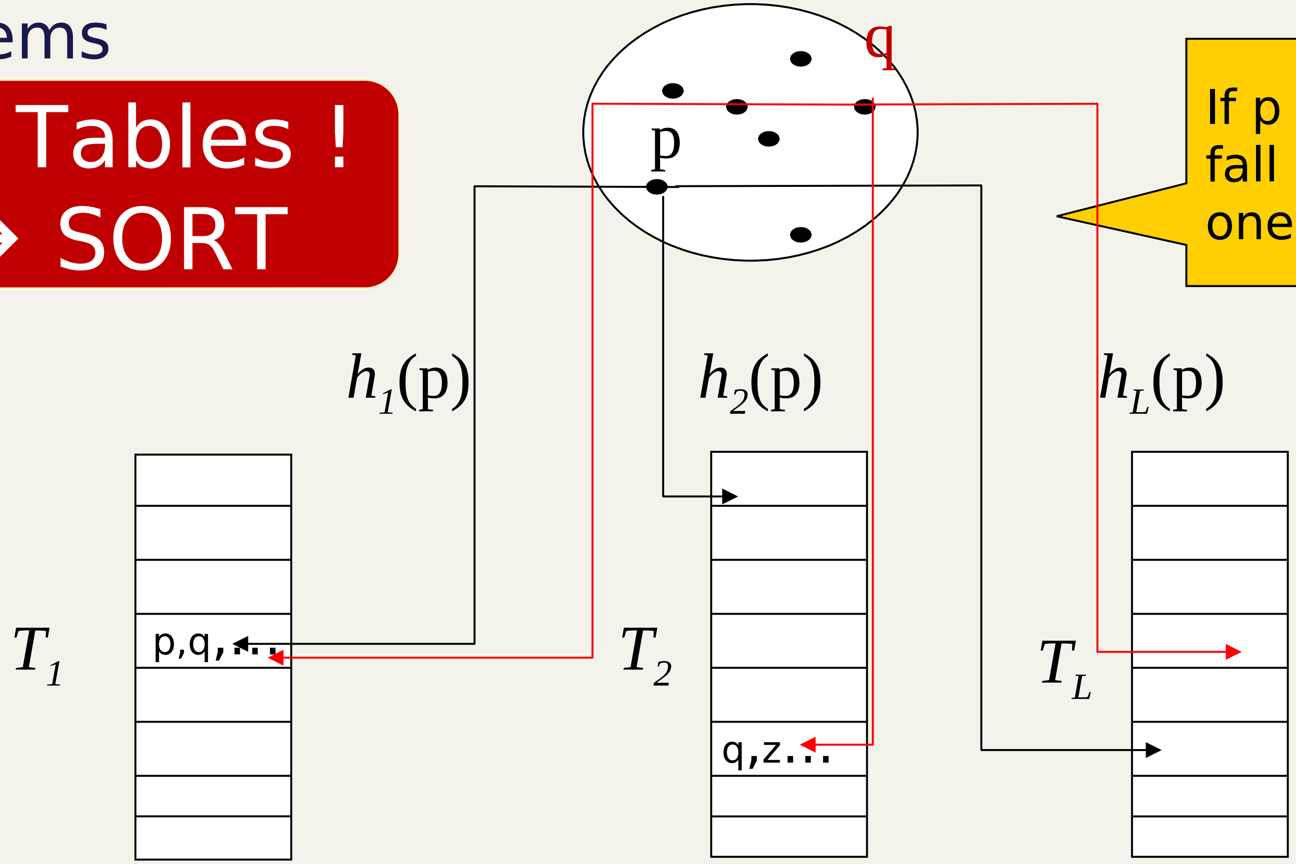


The case: Groups of similar items

Buckets provide the candidate similar items

«Merge» similar sets over L rounds if they share items

No Tables !
→ SORT

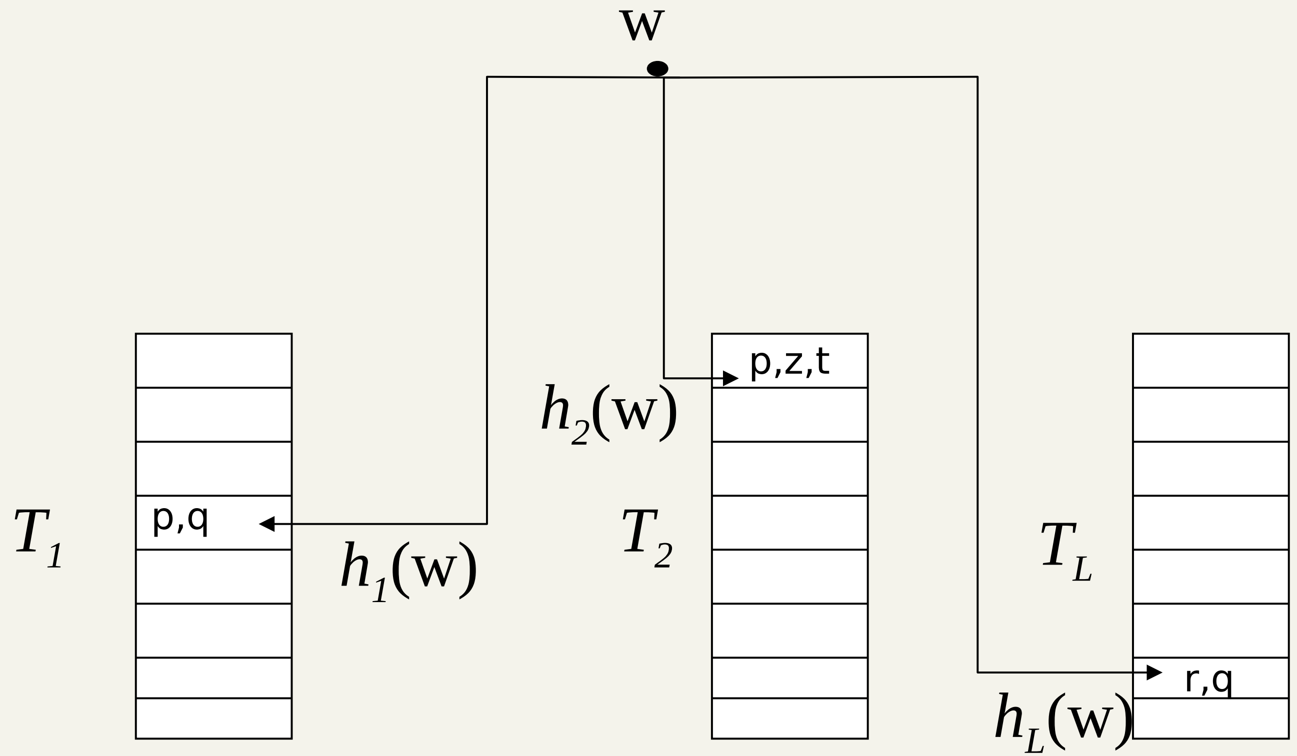


If p ≈ q, then they fall in at least one same bucket

The case of *on-line query*

Given a query w, find the similar indexed vectors:

check the vectors in the buckets $h_j(w)$ for all $j=1,\ldots,L$



LSH *versus* K-means

- **What about optimality ?** K-means is locally optimal [LSH finds correct clusters with high probability]

- **What about the Sim-cost ?** K-means compares vectors of d components [LSH compares very short (sketch) vectors]

- **What about the cost per iteration?** Typically K-means requires few iterations, each costs **K** × **U** × **d** [LSH sorts U short items, few scans]

- **What about K ?** In principle
…, U [LSH does not need to know t

You could apply K-means over LSH-sketch vectors !!