Outline

- Locality-Sensitive Hashing
- Applications of Locality-Sensitive Hashing
- Distance Measures
- Locality-Sensitive Functions
- Methods for High Degrees of Similarity

Applications of LSH

- Matching Newspaper Articles
- Entity Resolution
- Matching Fingerprints

Matching Newspaper Articles

- **Problem:** the same article, say from associated press, appears on the web site of many newspapers, but looks quite different
    - each newspaper surrounds the article with logos, ads, links to other articles …
    - a newspaper may also `crop' the article
- LSH substitute: candidates are articles of similar length
- Observe that news articles have a lot of stop words, while ads do not
    - "I recommend that you buy XXX for your laundry." vs "Buy XXX"
    - Define a shingle to be a stop word and the next two following words
    - By requiring each shingle to have a stop word, they biased the mapping to pick more shingles from the articles than from ads

Entity Resolution

- The **entity-resolution** problem is to examine a collection of records and determine which refer to the same entity
  - Entities could be people, events, etc.
  - We want to merge records if their values in corresponding fields are similar
- Company A and B want to find out what customers they share
  - Each company had about 1 million records
  - Records had name, address, and phone. Could be different for the same person

Entity Resolution

- **Step 1:** Design a measure (`score') of how similar records are
  - e.g., deduct points for small misspellings (`Jeffrey' vs. 'Jeffery') or same phone with different area code
- **Step 2:** Score all pairs of records that the LSH scheme identified as candidates; report high scores as matches
  - 3 hash functions: exact values of name, address, phone
  - compare iff records are identical in at least one
  - miss similar records with a small differences in all three fields

Entity Resolution

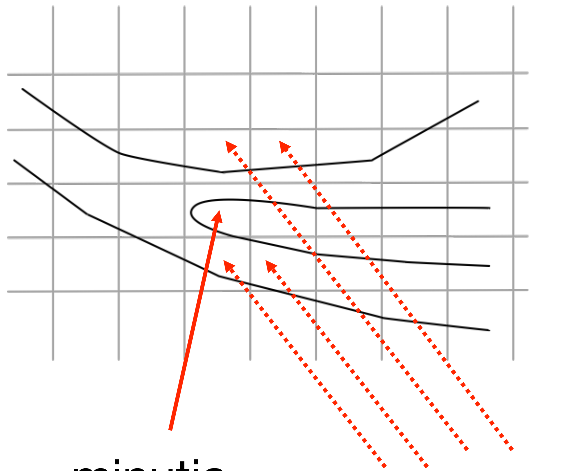
- How do we hash strings such as names so there is one bucket for each string?
  - **Idea:** sort the strings instead
  - Another option was to use a few million buckets, and compare all pairs of records within one bucket
- **Validation** of results
  - identical records has a creation date difference of 10 days
  - only looked for records created within 90 days of each other, so bogus matches has a 45-day average
  - looking at the pool of matches with a fixed score, compute the average time-difference x, fraction (45 - x)/35 of them were valid matches

Validation of Results

- Any field not used in the LSH could be used to **validate**, provided corresponding values were closer for true matches than false
  - e.g., if records has a height field, we would expect true matches to be close, false matches to be the **average difference** for random people

Matching Fingerprints

- Represent a fingerprint by the set of positions of minutiae (features of a fingerprint, e.g., points where two ridges come together or a ridge ends)
- Place a grid on a fingerprint
  - Normalize so identical prints will overlap
- Set of grid squares where minutiae are located represents the fingerprint
  - Possibly, treat minutiae near a grid boundary as if also present in adjacent grid points



- **Problem:** finding similar sets of grid squares that have minutiae
  - rows: grid squares; columns: fingerprints sets. Not sparse!

Matching Fingerprints

- No need to minhash, since the number of grid squares is not too large
- Represent each fingerprint by **a bit-vector** with one position for each square
  - 1 in only those positions whose squares have minutiae
  - Pick 1024 sets of 3 grid squares randomly
  - For each set of three squares, two fingerprints that **each** have 1 for **all three** squares are candidate pairs
    - each set of three squares creates one bucket
    - fingerprints can be in many buckets

Matching Fingerprints

- Why make sense?
  - Suppose typical fingerprints have minutiae in **20%** of the grid squares
  - Suppose fingerprints from the same finger agree in at least **80%** of their squares
  - Prob. two random fingerprints each have 1' in three given squares = $((0.2)(0.2))^3 = 0.00064$ six independent event that a grid square has a minutia
  - Prob. two fingerprints from the same finger each have 1's in three given squares = $((0.2)(0.8))^3 = 0.004096$
  - Prob. for at least one of 1024 sets of three points = $1 - (1-0.004096)^{1024} = 0.985$

1.5% false negatives

1st print has a minutia in its square

2nd print also has a minutia in that square

6.3% false positives

For random fingerprints: $1-(1-0.000064)^{1024}=0.063$

Outline

- Locality-Sensitive Hashing
- Applications of Locality-Sensitive Hashing
- Distance Measures
- Locality-Sensitive Functions
- Methods for High Degrees of Similarity

Distance Measures

- Euclidean Distance
- Jaccard Distance
- Cosine Distance
- Edit Distance
- Hamming Distance

Euclidean vs. Non-Euclidean

- A **Euclidean space** has some number of real-valued dimensions and “dense” points
  - there is a notion of “average” of two points
  - a Euclidean distance is based on the locations of points in such a space
- Any other space is **Non-Euclidean**
  - distance measures for non-Euclidean spaces are based on properties of points, but not their “location” in a space

Jaccard Distance

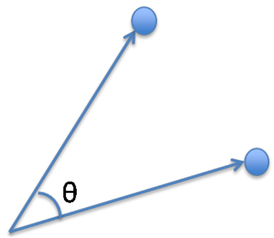
- d is a distance measure (metric) if it is a function from pairs of points to real numbers s.t.:
  1. d(x, y) >= 0
  2. d(x, y) = 0 iff x = y
  3. d(x, y) = d(y, x)
  4. d(x, y) <= d(x, z) + d(z, y) (triangle inequality)
- E.g., L_2 norm, L_1 norm, L_∞ norm, L_r norm …

- Jaccard distance for sets d(x, y) = 1 - sim(x, y):
  1. d(x, y) >= 0 since $|x \cup y| \geq |x \cap y|$
  2. d(x, y) = 0 iff x = y, because $x \cup x = x \cap x = x$
  3. d(x, y) = d(y, x) since union and intersection are symmetric
  4. d(x,y) = 1 - Pr(h(x) = h(y)); $\Pr(h(x) \neq h(y)) \leq \Pr(h(x) \neq h(z)) + \Pr(h(z) \neq h(y))$; whenever $h(x) \neq h(y)$, at least one of h(x) and h(y) must be different from h(z)

Cosine Distance

- d is a distance measure (metric) if it is a function from pairs of points to real numbers s.t.:
  1. d(x, y) >= 0
  2. d(x, y) = 0 iff x = y
  3. d(x, y) = d(y, x)
  4. d(x, y) <= d(x, z) + d(z, y) (triangle inequality)

$$sim(A,B) = \cos(\theta) = \frac{A \cdot B}{\|A\|\|B\|}$$



- Cosine distance = angle between vector x and y:
  1. d(x, y) is in the range of 0 to 180
  2. d(x, y) = 0 iff two vectors are the same direction
  3. the angle between x and y is the same as the angle between y and x
  4. one way to rotate from x to y is to rotate to z and then to y. Sum of the two rotations >= rotation directly from x to y

Edit Distance

- d is a distance measure (metric) if it is a function from pairs of points to real numbers s.t.:
  1. d(x, y) >= 0
  2. d(x, y) = 0 iff x = y
  3. d(x, y) = d(y, x)
  4. d(x, y) <= d(x, z) + d(z, y) (triangle inequality)

  E.g., the edit distance between x = abcde and y = acfdeg is 3. To convert x to y:
  1. Delete b
  2. Insert f after c
  3. Insert g after e

- Edit distance = number of inserts and deletes to change one string into another:
  1. no edit distance can be negative
  2. two identical strings have an edit distance of 0
  3. a sequence of edits can be reversed, with each insertion becoming a deletion, and vice versa
  4. one way to turn a string s to t is to turn s into u and then turn u into t

Hamming Distance

- d is a distance measure (metric) if it is a function from pairs of points to real numbers s.t.:

  1. d(x, y) >= 0
  2. d(x, y) = 0 iff x = y
  3. d(x, y) = d(y, x)
  4. d(x, y) <= d(x, z) + d(z, y) (triangle inequality)

E.g., Hamming distance between 10101 and 11110 is 3

- Hamming distance = number of positions in which two bit vectors differ:

  1. cannot be negative
  2. d(x, y) = 0 iff vectors are identical
  3. symmetric
  4. if x and z differ in m components, and z and y differ in n components, then x and y cannot differ in more than m + n components

Outline

- Locality-Sensitive Hashing
- Applications of Locality-Sensitive Hashing
- Distance Measures
- Locality-Sensitive Functions
- Methods for High Degrees of Similarity

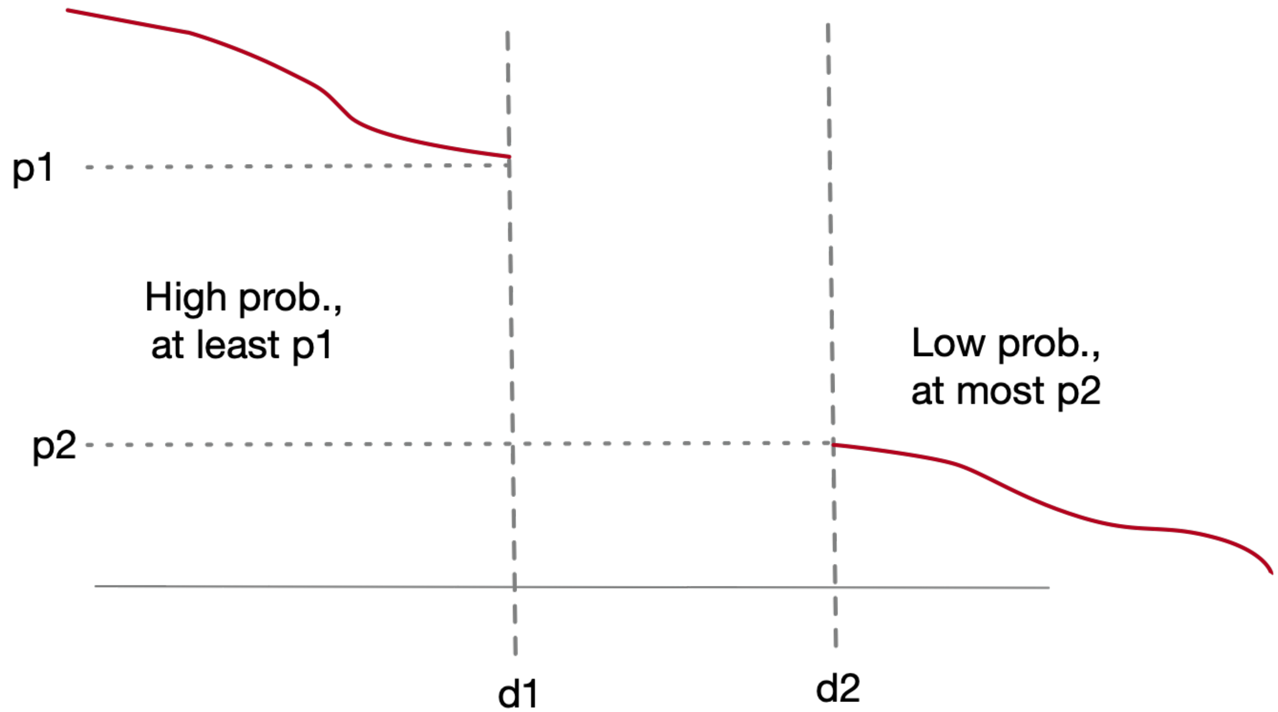
Locality-Sensitive Functions

- Locality-Sensitive Hashing (LSH) Families
- LSH Family for Jaccard Distance
- LSH Family for Cosine Distance
- LSH Family for Euclidean Distance

Hash Functions Decide Equality

- A hash function h takes two elements x and y, and returns a decision whether x and y are candidates for comparison
  - E.g., the family of minhash functions computes minhash values and returns yes if they are the same
  - Shorthand: h(x) = h(y) means h returns yes for the pair of x and y
- Suppose we have a space S of points with a distance measure d
- A family **H** of hash functions is said to be (d1, d2, p1, p2)-sensitive if for any x and y in S:
  - if d(x, y) <= d1, prob. over all h in **H** that h(x)=h(y) is at least p1
  - if d(x, y) >= d2, prob. over all h in **H** that h(x)=h(y) is at most p2

LS Families



Expect the distance between d1 and d2 very small, the distance between p1 and p2 very large

LSF for Jaccard Distance

- S = sets, d = Jaccard distance, **H** is formed from minhash functions for all permutations
- Pr(h(x)=h(y)) = 1 - d(x, y) = sim(x, y)
- **H** is a (1/3, 2/3, 2/3, 1/3)-sensitive family for S and d

if distance <=1/3
(similarity >= 2/3)

Prob. that minhash
values agree >= 2/3)

Steepen the S-curve
with “bands”
technique!
AND construction
like “rows in a band.”
OR construction like
“many bands.”

For Jaccard similarity, minhashing gives
us a (d1, d2, (1-d1), (1-d2))-sensitive
family for any d1 < d2

AND/OR of Hash Functions

- Given family **H**, construct family **H'** whose members each consist of r functions from **H**

  r rows in a single band!

  - AND: For $h = \{h_1, \ldots, h_r\}$ in **H'**, $h(x) = h(y)$ iff $h_i(x) = h_i(y)$ for all i
  - Theorem: if **H** is (d1, d2, p1, p2)-sensitive, then **H'** is (d1, d2, $(p1)^r$, $(p2)^r$)-sensitive. **each member of H is independently chosen**
  - OR: For $h = \{h_1, \ldots, h_b\}$ in **H'**, $h(x) = h(y)$ iff $h_i(x) = h_i(y)$ for some i

    combining b bands

  - Theorem: if **H** is (d1, d2, p1, p2)-sensitive, then **H'** is (d1, d2, $1-(1-p1)^b$, $1-(1-p2)^b$)-sensitive. **at least one band say yes**
  - AND makes all prob. shrink, but by choosing r correctly, we can make the lower prob. (p2) approach 0 while the higher does not
  - OR makes all prob. grow, but by choosing b correctly, we can make the upper prob. (p1) approach 1 while the lower does not

Composing Constructions

- As for the signature matrix, we can use the AND construction followed by the OR construction
  - or vice-versa
  - or any sequence of AND's and OR's alternating
- AND-OR Composition: Each of the two prob. p is transformed into $1-(1-p^r)^b$
  - "S-curve"
  - E.g., r=4, b=4

p	$1-(1-p^4)^4$
0.2	0.0064
0.3	0.032
0.4	0.0985
0.5	0.2275
0.6	0.426
0.7	0.6666
0.8	0.8785
0.9	0.986

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

Composing Constructions

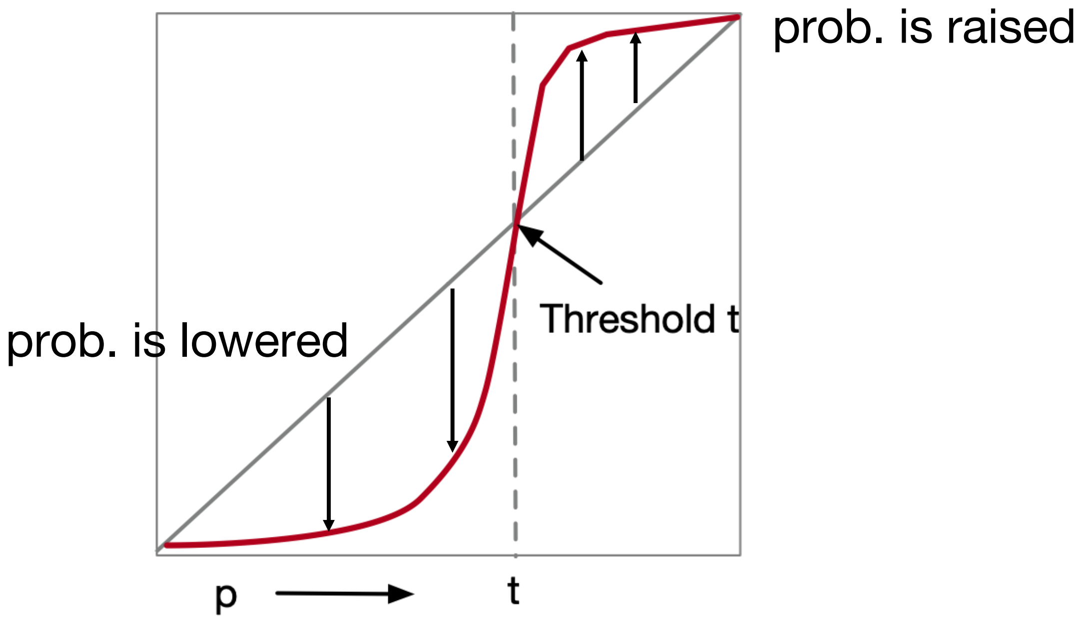
- OR-AND Composition: Each of the two prob. p is transformed into $(1-(1-p)^b)^r$
  - the same S-curve, mirrored horizontally and vertically
  - E.g., b=4, r=4

p	$(1-(1-p)^4)^4$
0.1	0.014
0.2	0.1215
0.3	0.3334
0.4	0.574
0.5	0.7725
0.6	0.9015
0.7	0.9680
0.8	0.9936

E.g., Transforms a (0.2,0.8,0.8,0.2)-sensitive family into a (0.2,0.8,0.9936,0.1215)-sensitive family

General Use of S-Curves

- For each S-curve $1-(1-p^r)^b$, there is a threshold t, for which $1-(1-t^r)^b=t$
- Above t, prob. are increased, below t, they are decreased
- You improve the sensitivity (by AND-OR construction) as long as the low prob. (p2) is less than t, and the high prob. (p1) is greater than t

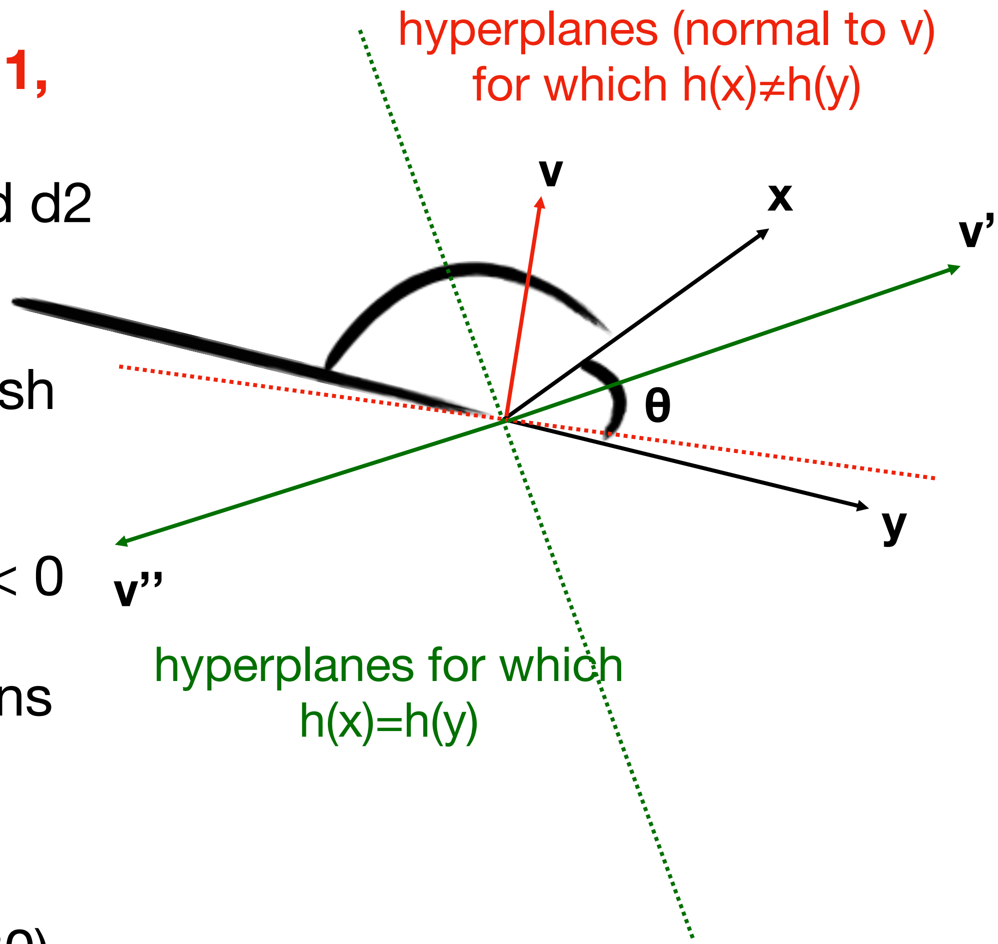


Cascading Construction

- E.g., apply the (4,4) OR-AND construction followed by the (4,4) AND-OR construction
  - 256 minhash functions
  - Transforms a (0.2,0.8,0.8,0.2)-sensitive family into a (0.2,0.8,0.9999996,0.0008715)-sensitive family

LSH Family for Cosine Distance

- For cosine distance, there is a technique analogous to minhashing for generating a **(d1, d2, (1-d1/180),(1-d2/180))-sensitive** family for any d1 and d2 — **random hyperplanes**
- Each vector v determines a hash function h_v, with two buckets
- $h_v(x) = +1$ if $v \cdot x > 0$; $= -1$ if $v \cdot x < 0$
- LS-family **H** = set of all functions derived from any vector
- Claim: $Pr[h(x)=h(y)]=1$ - (angle between x and y divided by 180)



Signatures for Cosine Distance

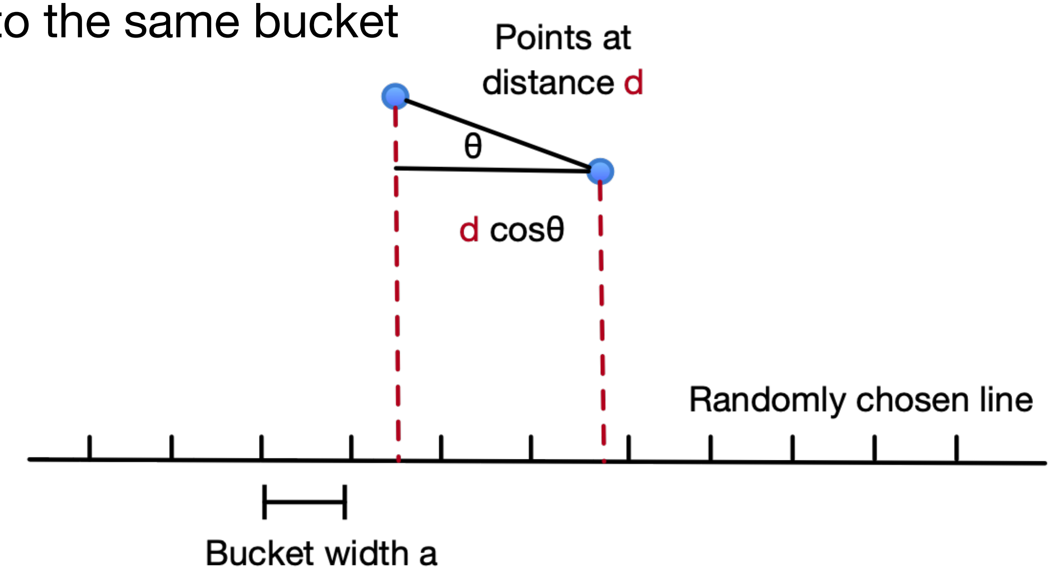
- Pick some number of vectors, and hash your data for each vector
- The result is a signature (sketch) of +1’s and -1’s that can be used for LSH like the minhash signatures for Jaccard distance
- The existence of the LSH-family is sufficient for amplification by AND/OR
- We need not pick from among all possible vectors v to form a component of a sketch
- It suffices to consider only vectors v consisting of +1 and -1 components

LSH Family for Euclidean Distance

- Idea: hash functions correspond to randomly chosen lines
- Partition the line into buckets of size a
- Hash each point to the bucket containing its projection onto the line
- Nearby points are always close; distant points are rarely in same bucket

if d >> a, θ must be close to 90° for there to be any chance points go to the same bucket

if d << a, the chance the points are in the same bucket is at least 1 - d/a



LSH Family for Euclidean Distance

- If points are >= 2a apart, 60<=θ<=90 for there to be a chance that the points go in the same bucket
- at most 1/3 prob. that the randomly chosen hash function returns yes
- If points are <= a/2 apart, there is at least 1/2 chance they share a bucket
- Yields a (a/2, 2a, 1/2, 1/3)-sensitive family of hash functions

Outline

- Locality-Sensitive Hashing
- Applications of Locality-Sensitive Hashing
- Distance Measures
- Locality-Sensitive Functions
- Methods for High Degrees of Similarity

Methods for High Degree of Similarity

Until now, LSH technique works well when the Jaccard similarity $<=$ 80%. When sets are at a very high Jaccard similarity, we have other techniques with no false negatives.

- Length-Based Filtering
- Prefix-Based Indexing
- Position/Prefix-Based Indexing
- Suffix Length

Setting: Sets as Strings

- Represent sets by strings (lists of symbols):
  - order the universal set
  - represent a set by the string of its elements in sorted order
  - if the universal set is k-shingles, there is a natural lexicographic order
  - think of each shingle as a single symbol
    - e.g., the 2-shingling of abcad {ab, bc, ca, ad} is represented by the list [ab, ad, bc, ca]
  - a better way: order words lowest-frequency-first
    - index documents based on the early words in their lists

Jaccard and Edit Distance

- Suppose two sets have Jaccard distance J and are represented by strings s1 and s2. Let the LCS (least common sequence) of s1 and s2 have length C and the edit distance be E. Then:
  - $1 - J = C/(C+E)$
  - $J = E/(C+E)$

works because these strings never repeat a symbol, and symbols appear in the same order

Length-Based Indexes

- Create an index on the length of strings

set A —> string A, length L
set B —> string B, length M
A is Jaccard distance J from B only if L(1-J) <= M <= L/(1-J)
E.g. if 1-J = 90% (Jaccard similarity), them M is between 90% and 111% of L

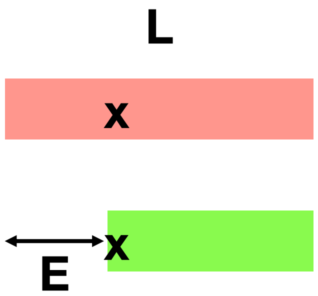
1-J = M/L
M = L(1-J)

1-J =L/M
M = L/(1-J)

Given a string of length L, **we only need to look for candidates in the range L(1-J) to L/(1-J)**

Prefix-Based Indexing

- If two strings are 90% similar, they must share some symbol in their prefixes whose length is just above 10% of the length of each string
- We can base an index on symbols in just **the first $\lfloor JL+1 \rfloor$ positions** of a string of length L



Extreme case: 2nd string has none of the first E symbols of the 1st string, but they agree thereafter

- If two strings do not share any of the first E symbols, then J >= E/L
- Thus, E = JL, but any larger E is impossible
- Index E + 1 positions

Prefix-Based Indexing

- Think of a bucket for each possible symbol
- Each string of length L is placed in the bucket for each of its first $\lfloor JL+1 \rfloor$ positions
- Given a probe string s of length L, with J the limit on Jaccard distance:

for (each symbol **a** among the first $\lfloor JL+1 \rfloor$ positions of **s**)
    look for other strings in the bucket for a;

- E.g., let J=0.2
  - String abcdef is indexed under **a** and **b**
  - String acdfg is indexed under **a** and **c**
  - String bcde is indexed under **b**
  - If search for strings similar to cdef, we need look only in the bucket for **c**

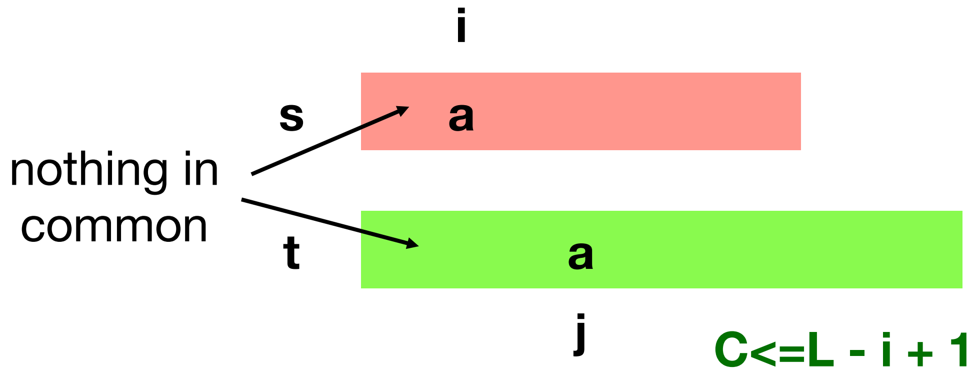
Positions/Prefixes-Based Indexing

- Consider the strings s = acdefghijk and t = bcdefghijk, and assume SIM = 0.9. What are the buckets do s and t placed in?
  - s is indexed under a and c; t is indexed under b and c
- Can we do less comparison?
  - Since c is the second symbol of both, we know there will be two symbols, a and b in this case, that are in the union of the two sets but not in the intersection
  - Even if s and t are identical from c to the end, their intersection is 9 symbols and their union is 11; thus SIM(s, t) = 9/11, which is less than 0.9

Positions/Prefixes-Based Indexing

- Consider whether the first common symbol appear close enough to the fronts of both strings
- If position i of probe string s is the first position to match a prefix position of string t, and it matches position j, then the edit distance between s and t is at least $i + j - 2$

$E >= i + j - 2$



$C <= L - i + 1$

- The LCS of s and t is no longer than $L-i+1$, where L is the length of s
- If J is Jaccard distance, remember $J = E/(E+C)$
- Thus, $(i+j-2)/(L+j-1) <= E/(E+C) \implies j <= (JL-J-i+2)/(1-J)$

Positions/Prefixes-Based Indexing

- Create a 2-attribute index on (symbol, position)
- If string s has symbol a as the i-th position of its prefix (first $\lfloor JL+1 \rfloor$ positions), add s to the bucket (a, i)
- Given probe string s, we only need to find a candidate once. So we
  - visit positions i of s in numerical order, assuming there have been no matches for earlier positions

```
for (i=1; i<=J*L+1; i++){
        let s have a in position i;
        for (j=1; j<=(J*L-J-i+2)/(1-J); j++)
                compare s with strings in bucket (a, j);
}
```

Positions/Prefixes-Based Indexing

- Suppose J=0.2
- Given probe string adegjkmprz, L=10, and the prefix is ade
- For the i-th position of the prefix, we must look at buckets where j <= (JL-J-i+2)/(1-J)=(3.8-i)/0.8
- For i=1: j<=3; for i=2: j<=2; for i=3: j<=1
- Look in the following buckets: (a, 1), (a, 2), (a, 3), (d, 1), (d, 2), (e, 1)
- Suppose string t is in none of these buckets
- Then the edit distance E is at least 3 (s and t share a, d, e, …)
- The LCS length C cannot be longer than s, i.e., 10
- Thus, J = E/(E+C) >= 3/13 > 0.2
- Need not compare s with t which is not in the six buckets!

Positions/Prefixes/Suffix Length Indexing

- We can add to our index a summary of what follows the positions being indexed. Help us eliminate candidate matches without comparing entire strings
- Idea: index on three attributes:
  - Character at a prefix position
  - Number of that position
  - Length of the suffix = number of positions in the entire string to the right of the given position
- s = acdefghijk, SIM = 0.9, what would the buckets that s is indexed under (symbol, position, suffix length)?
  - (a, 1, 9) and (c, 2, 8)

Positions/Prefixes/Suffix Length Indexing

- Given probe string s, we find string t because its j-th position matches the i-th position of s. The suffices of s and t have lengths k and m respectively
  - **A lower bound** on edit distance E is
    - $i + j - 2$
    - $|k-m|$ = absolute difference of the lengths of the suffixes of s and t
  - **An upper bound** on the length C of the LCS is $1 + \min(k, m)$
- Letting J be Jaccard distance, $J = E/(E+C)$, substitute the above values, we have
  - $j + |k-m| <= [J(i - 1 + \min(k,m)) - i + 2]/(1 - J)$

Positions/Prefixes/Suffix Length Indexing

- Create a 3-attribute index on (symbol, position, suffix-length)
- if string s has symbol a as the i-th position of its prefix, and the length of the suffix relative to that position is k, add s to the bucket (a, i, k)
- Consider string s = abcde with J=0.2
- Prefix length = 2
- Index in: (a, 1, 4) and (b, 2, 3)
- To find candidate matches for a probe string s of length L, with required similarity J, visit the positions of s's prefix in order
- If position i has symbol a and suffix length k, look in index bucket (a, j, m) for all j and m s.t.
  - $j + |k-m| <= [J(i - 1 + \min(k,m)) - i + 2]/(1 - J)$
  - look in (a, 1, 3), (a, 1, 4), (a, 1, 5), (a, 2, 4), (b, 1, 3)
  - for i = 1, note k = 4, we want $j + |4-m| <= [0.2\min(4,m) + 1]/0.8$

Summary

- Three index schemes
  - symbol
  - symbol + position
  - symbol + position + suffix length
- The number of buckets grows as we add dimensions to the index, but the total size of the buckets remains the same
  - because each string is placed in $\lfloor JL+1 \rfloor$ buckets

Reading

- Jure Leskovec, Anand Raj, Jeff Ullman, “Mining of Massive Datasets,” Cambridge University Press, Chapter 3