DSE 203

DAY 3: “MATCHING” – A FUNDAMENTAL CONSTRUCT

The Problem

Table X

	Name	Phone	City	State
X_1	Dave Smith	(608) 395 9462	Madison	WI
X_2	Joe Wilson	(408) 123 4265	San Jose	CA
X_3	Dan Smith	(608) 256 1212	Middleton	WI

(a)

Table Y

	Name	Phone	City	State
y_1	David D. Smith	395 9426	Madison	WI
y_2	Daniel W. Smith	256 1212	Madison	WI

(b)

Matches
(x_1, y_1)
(x_3, y_2)

(c)

- Other variations
  - Tables X and Y have different schemas – we will see this
  - Match tuples within a single table X – finding near-duplicates
  - The data is not relational, but XML or RDF – cross model matches

What is “Matching”?

- A ***match*** is a *correspondence or association* between individual structures in different data sources
  - Value matching
  - Tuple matching
  - Schema matching
- The goal of matching is to find correspondences and not to “clean”/rectify

Problem Description

- Given two sets of strings X and Y
  - Find all pairs $x \in X$ and $y \in Y$ that refer to the same real-world entity
  - We refer to (x,y) as a match
  - Example

Set X	Set Y	Matches
x_1 = Dave Smith	y_1 = David D. Smith	(x_1, y_1)
x_2 = Joe Wilson	y_2 = Daniel W. Smith	(x_3, y_2)
x_3 = Dan Smith		
(a)	(b)	(c)

- Two major challenges: accuracy & scalability

Accuracy Challenges

- Matching strings often appear quite differently
  - Typing and OCR errors: David Smith vs. Davod Smith
  - Different formatting conventions: 10/8 vs. Oct 8
  - Custom abbreviation, shortening, or omission:
    Daniel Walker Herbert Smith vs. Dan W. Smith
  - Different names, nick names: William Smith vs. Bill Smith
  - Shuffling parts of strings: Dept. of Computer Science, UW-Madison vs. Computer Science Dept., UW-Madison

Accuracy Challenges

- Solution:
  - Use a similarity measure $s(x,y) \in [0,1]$
    - The higher $s(x,y)$, the more likely that x and y match
  - Declare x and y matched if $s(x,y) \geq t$
  - Distance measure/cost measure have also been used
    - Same concept
    - But smaller values ➔ higher similarities

Scalability Challenges

- Applying s(x,y) to all pairs is impractical
  - Quadratic in size of data
- Solution: apply s(x,y) to only most promising pairs, using a method FindCands
  - For each string $x \in X$
    use method FindCands to find a candidate set $Z \subseteq Y$
    for each string $y \in Z$
    if $s(x,y) \geq t$ then return (x,y) as a matched pair
  - We discuss ways to implement FindCands later

Outline

- Problem description
- Similarity measures
  - Sequence-based: edit distance, Needleman-Wunch, affine gap, Smith-Waterman, Jaro, Jaro-Winkler
  - Set-based: Jaccard, TF/IDF
  - Hybrid: soft TF/IDF
- Scaling up string matching
  - Inverted index, size filtering, prefix filtering, position filtering, bound filtering

Edit Distance

- Also known as Levenshtein distance
- d(x,y) computes minimal cost of transforming x into y, using a sequence of operators, each with cost 1
  - Delete a character
  - Insert a character
  - Substitute a character with another
- Example: x = David Smiths, y = Davidd Simth, d(x,y) = 4, using following sequence
    - Inserting a character d (after David)
    - Substituting m by i
    - Substituting i by m
    - Deleting the last character of x, which is s

Edit Distance

- Models common editing mistakes
  - Inserting an extra character, swapping two characters, etc.
  - So smaller edit distance ➔ higher similarity
- Can be converted into a similarity measure
  - s(x,y) = 1 - d(x,y) / [max(length(x), length(y))]
  - Example
    - s(David Smiths, Davidd Simth) = 1 – 4 / max(12, 12) = 0.67

Computing Edit Distance using Dynamic Programming

- Define $x = x_1x_2\cdots x_n$, $y = y_1y_2\cdots y_m$
  - $d(i,j)$ = edit distance between $x_1x_2\cdots x_i$ and $y_1y_2\cdots y_j$,
    the i-th and j-th prefixes of x and y

- Recurrence equations

$$d(i,j) = \min \begin{cases} d(i-1,j-1) & \text{if } x_i = y_j \quad \text{// copy} \\ d(i-1,j-1) + 1 & \text{if } x_i <> y_j \quad \text{// substitute} \\ d(i-1,j) + 1 & \text{// delete } x_i \\ d(i,j-1) + 1 & \text{// insert } y_j \end{cases}$$

$$d(i,j) = \min \begin{cases} d(i-1,j-1) + c(x_i,y_j) & \text{// copy or substitute} \\ d(i-1,j) + 1 & \text{// delete } x_i \\ d(i,j-1) + 1 & \text{// insert } y_j \end{cases}$$

$$c(x_i,y_j) = \begin{matrix} 0 & \text{if } x_i = y_j, \\ 1 & \text{otherwise} \end{matrix}$$

Example

- x = dva, y = dave

		y0	y1	y2	y3	y4
			d	**a**	**v**	**e**
x0		0	1	2	3	4
x1	**d**	1	0	1		
x2	**v**	2				
x3	**a**	3				

		y0	y1	y2	y3	y4
			d	**a**	**v**	**e**
x0		0	1	2	3	4
x1	**d**	1	0	1	2	3
x2	**v**	2	1	1	1	2
x3	**a**	3	2	1	2	2

x = d – v a
| | | |
y = d a v e

substitute a with e
insert a (after d)

- Cost of dynamic programming is O(|x||y|)

Needleman-Wunch Measure

- Generalizes Levenshtein edit distance
- Basic idea
  - defines notion of alignment between x and y
  - assigns score to alignment
  - return the alignment with highest score
- Alignment: set of correspondences between characters of x and y, allowing for gaps

```
d - - v a
|     | |
d e e v e
```

Scoring an Alignment

- Use a score matrix and a gap penalty
- Example

	d	v	a	e
d	2	-1	-1	-1
v	-1	2	-1	-1
a	-1	-1	2	-1
e	-1	-1	-1	2

$c_g = 1$

```
d - -  v a
|      | |
d e e v e
```

- alignment score = sum of scores of all correspondences - sum of penalties of all mismatches and gaps
  - e.g., for the above alignment, it is 2 (for d-d) + 2 (for v-v) -1 (for a-e) -2 (for gap) = 1
  - This is the alignment with the highest score, it is returned as the Needleman-Wunch score for dva and deeve.

Needleman-Wunch Generalizes Levenshtein in Three Ways

- Computes similarity scores instead of distance values
- Generalizes edit costs into a score matrix
  - allowing for more fine-grained score modeling
  - e.g., score(o,o) > score(a,o)
- Generalizes insertion and deletion into gaps, and generalizes their costs from 1 to $\mathbf{C}_g$

Computing Needleman-Wunch Score with Dynamic Programming

$$s(i,j) = \max \begin{cases} s(i-1,j-1) + c(xi,yj) \\ s(i-1,j) - c_g \\ s(i,j-1) - c_g \end{cases}$$

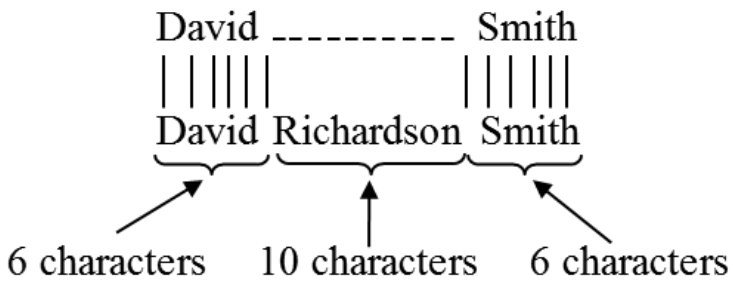
$$s(0,j) = -jc_g$$
$$s(i,0) = -ic_g$$

		d	e	e	v	e
	0	-1	-2	-3	-4	-5
d	-1	2	1	0	-1	-2
v	-2	1	1	0	2	1
a	-3	0	0	0	1	1

d - - v a
d e e v e

The Affine Gap Measure: Motivation

- An extension of Needleman-Wunch that handles longer gap more gracefully
- E.g., “David Smith” vs. “David R. Smith”
  - Needleman-Wunch well suited here
  - opens gap of length 2 right after “David”
- E.g.,



  - Needlement-Wunch not well suited here, gap cost is too high
  - If each char corrspondence has score 2, $\mathbf{c}_g = 1$, then the above has score $6*2 - 10 = 2$

The Affine Gap Measure: Solution

- In practice, gaps tend to be longer than 1 character
- Assigning same penalty to each character unfairly punishes long gaps
- Solution: define cost of opening a gap vs. cost of continuing the gap
  - cost (gap of length k) = $\mathbf{c}_o + (\mathbf{k-1})\mathbf{c}_r$
  - $\mathbf{c}_o$ = cost of opening gap
  - $\mathbf{c}_r$ = cost of continuing gap, $\mathbf{c}_o > \mathbf{c}_r$
- E.g., "David Smith" vs. "David Richardson Smith"
  - $\mathbf{c}_o = 1$, $\mathbf{c}_r = 0.5$, alignment cost = 6*2 – 1 - 9*0.5 = 6.5

Computing Affine Gap Score using Dynamic Programming

$$s(i,j) = \max \{M(i,j), I_x(i,j), I_y(i,j)\}$$

$$M(i,j) = \max \begin{cases} M(i-1,j-1) + c(x_i,y_j) \\ I_x(i-1,j-1) + c(x_i,y_j) \\ I_y(i-1,j-1) + c(x_i,y_j) \end{cases}$$

$$I_x(i,j) = \max \begin{cases} M(i-1,j) - c_o \\ I_x(i-1,j) - c_r \end{cases}$$

$$I_y(i,j) = \max \begin{cases} M(i,j-1) - c_o \\ I_y(i,j-1) - c_r \end{cases}$$

- $M(i,j)$
  - Best score between $x_1 \ldots x_i$ and $y_1 \ldots y_j$ given that x_i is aligned with y_j
- $I_x(i,j)$
  - Best score given that x_i is aligned with a gap
- $I_y(i,j)$
  - Best score given that y_j is aligned with a gap

- Assumption: insertion not directly followed by deletion
- The book shows how these equations are derived

The Smith-Waterman Measure: Motivation

- Previous measures consider global alignments
  - attempt to match all characters of x with all characters of y
- Not well suited for some cases
  - e.g., “Prof. John R. Smith, Univ of Wisconsin” and “John R. Smith, Professor”
  - similarity score here would be quite low
- Better idea: find two substrings of x and y that are most similar
  - e.g., find “John R. Smith” in the above case ➔ local alignment

The Smith-Waterman Measure

- Find the best local alignment between x and y, and return its score as the score between x and y
- Makes two key changes to Needleman-Wunch
  - allows the match to restart at any position in the strings (no longer limited to just the first position)
    - if global match dips below 0, then ignore prefix and restart the match
  - after computing matrix using recurrence equation, retracing the arrows from the largest value in matrix, rather than from lower-right corner
    - this effectively ignores suffixes if the match they produce is not optimal
    - retracing ends when we meet a cell with value 0 ➔ start of alignment

Computing Smith-Waterman Score using Dynamic Programming

$$s(i,j) = \max \begin{cases} 0 \\ s(i-1,j-1) + c(xi,yj) \\ s(i-1,j) - c_g \\ s(i,j-1) - c_g \end{cases}$$

$$s(0,j) = 0$$
$$s(i,0) = 0$$

		d	**a**	**v**	**e**
	0	0	0	0	0
a	0	0	2	1	0
v	0	0	1	4	3
d	0	2	1	3	3

Example 2

CGTGAATTCAT (sequence#1)
GACTTAC (sequence #2)

Match score = 5
Mismatch score = -3
Gap penalty score = -4

	-	C	G	T	G	A	A	T	T	C	A	T
-	0	0	0	0	0	0	0	0	0	0	0	0
G	0											
A	0											
C	0											
T	0											
T	0											
A	0											
C	0											

	-	C	G	T	G	A	A	T	T	C	A	T
-	0	0	0	0	0	0	0	0	0	0	0	0
G	0	0	5	1	5	1	0	0	0	0	0	0
A	0	0	1	2	1	10	6	2	0	0	5	1
C	0	5	1	0	0	6	7	3	0	5	1	2
T	0	1	2	6	2	2	3	12	8	4	2	6
T	0	0	0	7	3	0	0	8	17	13	9	7
A	0	0	0	3	4	8	5	4	13	14	18	14
C	0	5	1	0	0	4	5	2	9	18	14	15

$$
\begin{aligned}
M_{1,1} &= \text{Maximum}\,[M_{0,0} + S_{1,1},\ M_{1,0} + W,\ M_{0,1} + W,\ 0] \\
&= \text{Maximum}\,[0(-3), 0 + (-4), 0 + (-4), 0] \\
&= \text{Maximum}\,[-3, -4, -4, 0] \\
&= 0
\end{aligned}
$$

	-	C	G	T	G	A	A	T	T	C	A	T
-	0	0	0	0	0	0	0	0	0	0	0	0
G	0	0	5	1	5	1	0	0	0	0	0	0
A	0	0	1	2	1	10	6	2	0	0	5	1
C	0	5	1	0	0	6	7	3	0	5	1	2
T	0	1	2	6	2	2	3	12	8	4	2	6
T	0	0	0	7	3	0	0	8	17	13	9	7
A	0	0	0	3	4	8	5	4	13	14	18	14
C	0	5	1	0	0	4	5	2	9	18	14	15

```
G A A T T C A
| | | | |   |
G A C T T - A
```

	-	C	G	T	G	A	A	T	T	C	A	T
-	0	0	0	0	0	0	0	0	0	0	0	0
G	0				5							
A	0					10						
C	0						7					
T	0							12				
T	0								17			
A	0								13			
C	0									18		

```
G A A T T - C
| | | | |   |
G A C T T A C
```

The Jaro Measure

- Mainly for comparing short strings, e.g., first/last names
- To compute jaro(x,y)
  - find common characters $\mathbf{x}_i$ and $\mathbf{y}_j$ such that $\mathbf{x}_i = \mathbf{y}_j$ and $|i-j| \leq \min\{|x|,|y|\}/2$
  - intuitively, common characters are identical and positionally "close to each other"
  - if the i-th common character of x does not match the i-th common character of y, then we have a transposition
  - return $\text{jaro}(x,y) = 1 / 3[c/|x| + c/|y| + (c - t/2)/c]$, where c is the number of common characters, and t is the number of transpositions

The Jaro Measure: Examples

- x = jon, y = john
  - c = 3 because the common characters are j, o, and n
  - t = 0
  - jaro(x,y) = 1 / 3(3/3 + 3/4 + 3/3) = 0.917
  - contrast this to 0.75, the sim score of x and y using edit distance
- x = jon, y = ojhn
  - common char sequence in x is jon
  - common char sequence in y is ojn
  - t = 2
  - jaro(x,y) = 0.81

The Jaro-Winkler Measure

- Captures cases where x and y have a low Jaro score, but share a prefix ➔ still likely to match
- Computed as
  - jaro-winkler(x,y) = (1 – PL*PW)*jaro(x,y) + PL*PW
  - PL = length of the longest common prefix
  - PW is a weight given to the prefix

Given the strings s_1 *MARTHA* and s_2 *MARHTA* we find:

- $m = 6$
- $|s_1| = 6$
- $|s_2| = 6$
- There are mismatched characters T/H and H/T leading to $t = \frac{2}{2} = 1$

We find a Jaro score of:

$$d_j = \frac{1}{3}\left(\frac{6}{6} + \frac{6}{6} + \frac{6-1}{6}\right) = 0.944$$

To find the Jaro-Winkler score using the standard weight $p = 0.1$, we continue to find:

$$\ell = 3$$

Thus:

$$d_w = 0.944 + (3 * 0.1(1 - 0.944)) = 0.961$$

The Jaccard Measure

- $J(x,y) = |\mathbf{B}_x$ Å $\mathbf{B}_y| / |\mathbf{B}_x$ [$\mathbf{B}_y|$
- E.g., x = dave, y = dav
  - $\mathbf{B}_x$ = {#d, da, av, ve, e#}, $\mathbf{B}_y$ = {#d, da, av, v#}
  - J(x,y) = 3/6
- Very commonly used in practice

The TF/IDF Measure: Motivation

- uses the TF/IDF notion commonly used in IR
  - two strings are similar if they share distinguishing terms
  - e.g., x = Apple Corporation, CA
    y = IBM Corporation, CA
    z = Apple Corp
  - s(x,y) > s(x,z) using edit distance or Jaccard measure, so x is matched with y ➔ incorrect
  - TF/IDF measure can recognize that Apple is a distinguishing term, whereas Corporation and CA are far more common ➔ correctly match x with z

Term Frequencies and Inverse Document Frequencies

- Assume x and y are taken from a collection of strings
- Each string is coverted into a bag of terms called a document
- Define term frequency tf(t,d) = number of times term t appears in document d
- Define inverse document frequency idf(t) = N / $\mathbf{N}_d$, number of documents in collection devided by number of documents that contain t
  - note: in practice, idf(t) is often defined as log(N / $\mathbf{N}_d$), here we will use the above simple formula to define idf(t)

Example

$x = aab \Rightarrow B_x = \{a, a, b\}$

$y = ac \Rightarrow B_y = \{a, c\}$

$z = a \Rightarrow B_z = \{a\}$

$tf(a, x) = 2$

$tf(b, x) = 1$

...

$tf(c, z) = 0$

$idf(a) = 3/3 = 1$

$idf(b) = 3/1 = 3$

$idf(c) = 3/1 = 3$

Feature Vectors

- Each document d is converted into a feature vector $\mathbf{v}_d$
- $\mathbf{v}_d$ has a feature $\mathbf{v}_d(\mathbf{t})$ for each term t
  - value of $\mathbf{v}_d(\mathbf{t})$ is a function of TF and IDF scores
  - here we assume $\mathbf{v}_d(\mathbf{t}) = tf(t,d) * idf(t)$

$x = aab \Rightarrow B_x = \{a, a, b\}$

$y = ac \Rightarrow B_y = \{a, c\}$

$z = a \Rightarrow B_z = \{a\}$

$tf(a, x) = 2$ $\quad$ $idf(a) = 3/3 = 1$

$tf(b, x) = 1$ $\quad$ $idf(b) = 3/1 = 3$

... $\quad$ $idf(c) = 3/1 = 3$

$tf(c, z) = 0$

	a	**b**	**c**
$\mathbf{v}_x$	2	3	0
$\mathbf{v}_y$	1	0	3
$\mathbf{v}_z$	1	0	0

TF/IDF Similarity Score

- Let p and q be two strings, and T be the set of all terms in the collection
- Feature vectors $\mathbf{v}_p$ and $\mathbf{v}_q$ are vectors in the |T|-dimensional space wher each dimension corresponds to a term
- TF/IDF score of p and q is the cosine of the angle between $\mathbf{v}_p$ and $\mathbf{v}_q$
  - s(p,q) = $\sum_{t \in T} v_p(t) * v_q(t) \ / [\sqrt{\sum_{t \in T} v_p(t)^2} * \sqrt{\sum_{t \in T} vq(t)^2}]$

TF/IDF Similarity Score

- Score is high if strings share many frequent terms
  - terms with high TF scores
- Unless these terms are common in other strings
  - i.e., they have low IDF scores
- Dampening TF and IDF as commonly done in practice
  - use v_d(t) = log(tf(t,d) + 1) * log(idf(t)) instead of
    v_d(t) = tf(t,d) * idf(t)
- Normalizing feature vectors
  - v_d(t) = v_d(t) / $\sqrt{\sum_{\{t \in T\}} v_d(t)^2}$

Example: Document Vector

3 documents

d1: “new york times”
d2: “new york post”
d3: “los angeles times”

- Inverse document frequency

TERM	DOC-FREQUENCY	IDF
angeles	1	$\log_2(3/1) = 1.584$
los	1	$\log_2(3/1) = 1.584$
new	2	$\log_2(3/2) = 0.584$
post	1	$\log_2(3/1) = 1.584$
times	2	$\log_2(3/2) = 0.584$
york	2	$\log_2(3/2) = 0.584$

Example: The tf-idf matrix

TERM	DOC-FREQUENCY	IDF
angeles	1	$\log_2(3/1) = 1.584$
los	1	$\log_2(3/1) = 1.584$
new	2	$\log_2(3/2) = 0.584$
post	1	$\log_2(3/1) = 1.584$
times	2	$\log_2(3/2) = 0.584$
york	2	$\log_2(3/2) = 0.584$

	angeles	los	new	post	times	york
d1	0	0	1	0	1	1
d2	0	0	1	1	0	1
d3	1	1	0	0	1	0

TERM	DOC-FREQUENCY	IDF
angeles	1	$\log_2(3/1) = 1.584$
los	1	$\log_2(3/1) = 1.584$
new	2	$\log_2(3/2) = 0.584$
post	1	$\log_2(3/1) = 1.584$
times	2	$\log_2(3/2) = 0.584$
york	2	$\log_2(3/2) = 0.584$

	angeles	los	new	post	times	york	Length
d1	0	0	0.584	0	0.584	0.584	**1.011**
d2	0	0	0.584	1.584	0	0.584	**1.786**
d3	1.584	1.584	0	0	0.584	0	**2.316**

Length of d1 = sqrt(0.584^2+0.584^2+0.584^2)=1.011

Searching in Vector Space

query → q: new new york

- Max frequency of a term (“new”) = 2
- Create the query vector

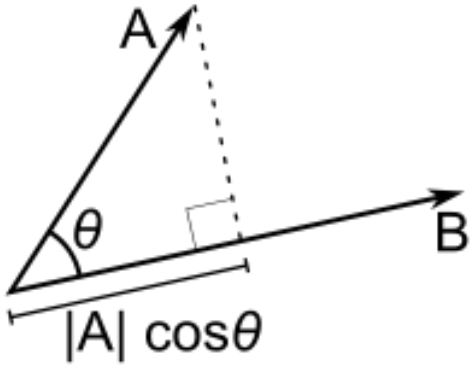
Q [0 0 (2/2)*0.584=0.584 0 (1/2)*0.584=0.292 0]

length(q)=0.652

Similarity Function

- Many possible functions
- Cosine distance

$$similarity = \cos(\theta) = \frac{A \cdot B}{\|A\|\|B\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}}$$



cos (d1,q) = (0.584*0.584+0.584*0.292) / (1.011*0.652) = 0.776
cos (d2,q) = (0.584*0.584) / (1.786*0.652) = 0.292
cos (d3,q) = (0.584*0.292) / (2.316*0.652) = 0.112

Documents can now be sorted according to this score

Query Term Weighting

- Every query term may optionally be associated with a weighting term
  - Q= York times^2 post^5
    - wt(York) = 1/(1+5+2)=1/8= 0.125
    - wt(times) = 2/8 = 0.25
    - wt(post)= 5/8 = 0.625
  - Multiply the query vector with these weights
  - “new york post” ranks first

The Soft TF/IDF Measure

- Similar to generalized Jaccard measure, except that it uses TF/IDF measure as the "higher-level" sim measure
  - e.g., "Apple Corporation, CA", "IBM Corporation, CA", and "Aple Corp",
    - Apple misspelt in the last string
- Step 1: compute close(x,y,k): set of all terms $t \in \mathbf{B}_x$ that have at least one close term $u \in \mathbf{B}_y$, i.e., $s'(t,u) \geq k$
  - s' is a basic sim measure (e.g., Jaro-Winkler), k prespecified
- Step 2: compute s(x,y) as in traditional TF/IDF score, but weighing each TF/IDF component using s'
  - $s(x,y) = \sum_{t \in close(x,y,k)} \mathbf{v}_x(\mathbf{t}) * \mathbf{v}_y(\mathbf{u}^*) * s'(t,u^*)$
  - $u^* \in \mathbf{B}_y$ maximizes $s'(t,u)\ \forall\ u \in \mathbf{B}_y$

An Example

$x = abcd$

$y = aa'b'cd'$

(a)

$B_x = \{a, \ b, \ c, \ d\}$

$1 \quad 0.7 \quad 0.8 \quad 1 \quad 0.6$

$B_y = \{a, \ a', \ b', \ c, \ d'\}$

$close(x, y, 0.75) = \{a, b, c\}$

(b)

$$s(x,y) = v_x(a) \,.\, v_y(a) \,.\, 1 + v_x(b) \,.\, v_y(b') \,.\, 0.8 + v_x(c) \,.\, v_y(c) \,.\, 1$$

(c)

Scalability Challenges

- Applying s(x,y) to all pairs is impractical
  - Quadratic in size of data
- Solution: apply s(x,y) to only most promising pairs, using a method FindCands
  - For each string $x \in X$
    use method FindCands to find a candidate set $Z \subset Y$
    for each string $y \in Z$
    if $s(x,y) \geq t$ then return (x,y) as a matched pair
  - This is often called a blocking solution
  - Set Z is often called the umbrella set of x
- We now discuss ways to implement FindCands
  - using Jaccard measure

Inverted Index over Strings

- Converts each string $y \in Y$ into a document, builds an inverted index over these documents
- Given term t, use the index to quickly find documents of Y that contain t

Example

Set X

1: {lake, mendota}
2: {lake, monona, area}
3: {lake, mendota, monona, dane}

Set Y

4: {lake, monona, university}
5: {monona, research, area}
6: {lake, mendota, monona, area}

(a)

Terms in Y	ID Lists
area	5
lake	4, 6
mendota	6
monona	4, 5, 6
research	5
university	4

(b)

Limitations

- The inverted list of some terms (e.g., stop words) can be very long ➔ costly to build and manipulate such lists
- Requires enumerating all pairs of strings that share at least one term. This set can still be very large in practice.

Size Filtering

- Retrieves only strings in Y whose sizes make them match candidates
  - given a string $x \in X$, infer a constraint on the size of strings in Y that can possibly match x
  - uses a B-tree index to retrieve only strings that satisfy size constraints
- E.g., for Jaccard measure $J(x,y) = |x \cap y| / |x \cup y|$
  - assume two strings x and y match if $J(x,y) \geq t$
  - can show that given a string $x \in X$, only strings y such that $|x|/t \geq |y| \geq |x|*t$ can possibly match x
    - $|x|$ is the number of tokens in x

Example

Set X

1: {lake, mendota}
2: {lake, monona, area}
3: {lake, mendota, monona, dane}

Set Y

4: {lake, monona, university}
5: {monona, research, area}
6: {lake, mendota, monona, area}

(a)

Terms in Y	ID Lists
area	5
lake	4, 6
mendota	6
monona	4, 5, 6
research	5
university	4

(b)

- Consider x = {lake, mendota}. Suppose t = 0.8
- If $y \in Y$ matches x, we must have
  - $2/0.8 = 2.5 \geq |y| \geq 2 * 0.8 = 1.6$
  - no string in Set Y satisfies this constraint ➔ no match

Prefix Filtering

- Key idea: if two sets share many terms ➔ large subsets of them also share terms
- Consider overlap measure $O(x,y) = |x \cap y|$
  - if $|x \cap y|$ ¸ k ➔ any subset $x' \subset x$ of size at least $|x| - (k - 1)$ must overlap y
- To exploit this idea to find pairs (x,y) such that $O(x,y) \geq k$
  - given x, construct subset x' of size $|x| - (k - 1)$
  - use an inverted index to find all y that overlap x'

Example

x: $\underbrace{\{\text{lake, monona}}_{x'}\text{, area}\}$

y: {lake, mendota, monona, area}

(a)

Set X

1: {lake, mendota}
2: {lake, monona, area}
3: {lake, mendota, monona, dane}

Set Y

4: {lake, monona, university}
5: {monona, research, area}
6: {lake, mendota, monona, area}
7: {dane, area, mendota}

(b)

Terms in Y	ID Lists
area	5, 6, 7
lake	4, 6
mendota	6, 7
monona	4, 5, 6
research	5
university	4
dane	7

(c)

- Consider matching using $O(x,y) \geq 2$
- x_1 = {lake, mendota}, let x_1' = {lake}
- Use inverted index to find $\{y_4, y_6\}$ which contain at least one token in x_1'

Selecting the Subset Intelligently

- Recall that we select a subset x' of x and check its overlap with the entire set y
- We can do better by selecting a particular subset x' and checking its overlap with only a particular subset y' of y
- How?
  - impose an ordering O over the universe of all possible terms
    - e.g., in increasing frequency
  - reorder the terms in each $x \in X$ and $y \in Y$ according to O
  - refer to subset x' that contains the first n terms of x as the prefix of size n of x

Selecting the Subset Intelligently

- How? (continued)
  - We can prove that if $|x \cap y| \geq k$, then x' and y' must overlap, where x' is the prefix of size $|x| - (k - 1)$ of x and y' is the prefix of size $|y| - (k - 1)$ of y
  - Algorithm
    - reorder terms in each $x \in X$ and $y \in Y$ in increasing order of their frequencies
    - for each $y \in Y$, create y', the prefix of size $|y| - (k - 1)$ of y
    - build an inverted index over all prefixes y'
    - for each $x \in X$, create x', the prefix of size $|x| - (k - 1)$ of x, then use above index to find all y such that x' overlaps with y'

Example

university < research
< dane < area
< mendota < monona < lake

(a)

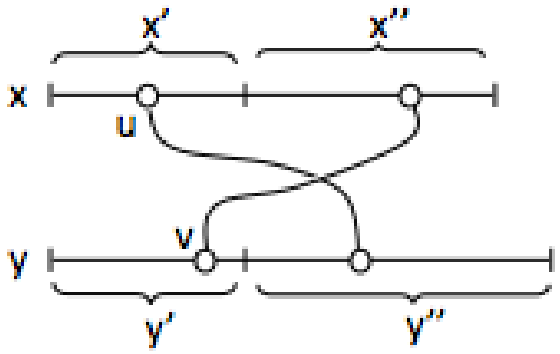
Reordered Set X

1: {mendota, lake}
2: {area, monona, lake}
3: {dane, mendota, monona, lake}

Reordered Set Y

4: {university, monona, lake}
5: {research, area, monona}
6: {area, mendota, monona, lake}
7: {dane, area, mendota}

(b)



(c)

- x = {mendota, lake} ➔ x' = {mendota}

Example

Terms in Y	ID Lists
area	5, 6, 7
mendota	6
monona	4, 6
research	5
university	4
dane	7

(a)

Terms in Y	ID Lists
area	5, 6, 7
lake	4, 6
mendota	6, 7
monona	4, 5, 6
research	5
university	4
dane	7

(b)

- For Jaccard measure
  - $J(x,y) \geq t \Leftrightarrow O(x,y) \geq \alpha = \frac{t}{1+t}(|x| + |y|)$

Position Filtering

- Further limits the set of candidate matches by deriving an upper bound on the size of overlap between x and y
- e.g., x = {dane, area, mendota, monona, lake}
  y = {research, dane, mendota, monona, lake}
- Suppose we consider $J(x, y) \geq 0.8$
  - in prefix filtering we consider x' = {dane, area} and y' = {research, dane}
- If x'' is the rest of x after x' (similarly y")
  - $O(x, y) \leq |x' \cap y'| + \min(|x''|, |y''|)$
  - In this case O(x,y) = 4.44

Tuple Matching

- Problem definition
- Rule-based matching
- Learning- based matching
- Matching by clustering
- Probabilistic approaches to matching
- Collective matching
- Scaling up data matching

Rule-based Matching

- The developer writes rules that specify when two tuples match
  - typically after examining many matching and non-matching tuple pairs, using a development set of tuple pairs
  - rules are then tested and refined, using the same development set or a test set
- Many types of rules exist
  - linearly weighted combination of individual similarity scores
  - logistic regression combination
  - more complex rules

Linearly Weighted Combination Rules

- Compute the sim score between tuples x and y as a linearly weighted combination of individual sim scores
  - sim(x,y) = $\sum_{i=1}^{n} \alpha_i * sim_i(x, y)$
  - n is number of attributes in each table
  - **s_i(x,y)** is a sim score between the i-th attributes of x and y
  - $\alpha_i \in [0,1]$ is a pre-specified weight that indicates the important of the i-th attribute to sim(x,y), such that $\sum_{i=1}^{n} \alpha_i = 1$
- We declare x and y matched if sim(x,y) $\geq \beta$ for a pre-specified β, and not matched otherwise
  - in another variation: declare x and y matched if sim(x,y) $\geq \beta$, not matched if sim(x,y) $< \gamma$, and subject to human review

Example

Table X

	Name	Phone	City	State
X_1	Dave Smith	(608) 395 9462	Madison	WI
X_2	Joe Wilson	(408) 123 4265	San Jose	CA
X_3	Dan Smith	(608) 256 1212	Middleton	WI

(a)

Table Y

	Name	Phone	City	State
y_1	David D. Smith	395 9426	Madison	WI
y_2	Daniel W. Smith	256 1212	Madison	WI

(b)

Matches

(x_1, y_1)
(x_3, y_2)

(c)

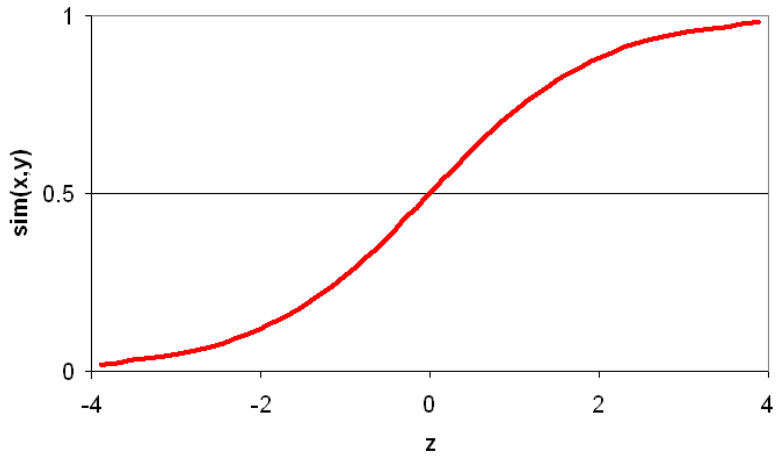
- sim(x,y) =
  $0.3s_{name}(x,y) + 0.3s_{phone}(x,y) + 0.1s_{city}(x,y) + 0.3s_{state}(x,y)$
  - $s_{name}(x,y)$: based on Jaro-Winkler
  - $s_{phone}(x,y)$: based on edit distance between x's phone (after removing area code) and y's phone
  - $s_{city}(x,y)$: based on edit distance
  - $s_{state}(x,y)$: based on exact match; yes ➔ 1, no ➔ 0

Pros and Cons

- Pros
  - conceptually simple, easy to implement
  - can learn weights α_i from training data
- Cons
  - an increase δ in the value of any $\mathbf{s}_i$ will cause a linear increase $\alpha_i * \delta$ in the value of s
  - in certain scenarios this is not desirable,
  - after a certain threshold, an increase in $\mathbf{s}_i$ should count less (i.e., "diminishing returns" should kick in)
  - e.g., if $\mathbf{s}_{name}(\mathbf{x},\mathbf{y})$ is already 0.95 then the two names already very closely match
    - so any increase in $\mathbf{s}_{name}(\mathbf{x},\mathbf{y})$ should contribute only minimally

Logistic Regression Rules

- address the diminishing-returns problem
- sim(x,y) = 1 / (1 + e^{-z}), where z = $\sum_{i=1}^{n} \alpha_i * sim_i(x, y)$
  - here \alpha_i are not constrained to be in [0,1] and sum to 1
  - so z goes from -∞ to +∞, in which case sim(x,y) gradually increases, but minimally so after z has exceeded a certain value ➔ ensuring diminishing returns



Logistic Regression Rules

- Are also very useful in situations where
  - there are many "signals" (e.g., 10-20) that can contribute to whether two tuples match
  - we don't need all of these signals to "fire" in order to conclude that the tuples match
  - as long as a reasonable number of them fire, we have sufficient confidence
- Logistic regression is a natural fit for such cases
- Hence is quite popular as a first matching method to try

More Complex Rules

- Appropriate when we want to encode more complex matching knowledge
  - e.g., two persons match if names match approximately and either phones match exactly or addresses match exactly

1. If $\mathbf{s}_{name}(\mathbf{x},\mathbf{y}) < 0.8$ then return "not matched"
2. Otherwise if $\mathbf{e}_{phone}(\mathbf{x},\mathbf{y})$ = true then return "matched"
3. Otherwise if $\mathbf{e}_{city}(\mathbf{x},\mathbf{y})$ = true and $\mathbf{e}_{state}(\mathbf{x},\mathbf{y})$ = true then return "matched"
4. Otherwise return "not matched"

Pros and Cons of Rule-Based Approaches

- Pros
  - easy to start, conceptually relatively easy to understand, implement, debug
  - typically run fast
  - can encode complex matching knowledge
- Cons
  - can be labor intensive, it takes a lot of time to write good rules
  - can be difficult to set appropriate weights
  - in certain cases it is not even clear how to write rules
  - learning-based approaches address these issues

Learning-based Matching

- Here we consider supervised learning
  - learn a matching model M from training data, then apply M to match new tuple pairs
  - will consider unsupervised learning later
- Learning a matching model M (the training phase)
  - start with training data: $T = \{(\mathbf{x_1},\mathbf{y_1},\mathbf{l}_1), \ldots (\mathbf{x}_n,\mathbf{y}_n,\mathbf{l}_n)\}$, where each $(\mathbf{x}_i,\mathbf{y}_i)$ is a tuple pair and $\mathbf{l}_i$ is a label: "yes" if $\mathbf{x}_i$ matches $\mathbf{y}_i$ and "no" otherwise
  - define a set of features $\mathbf{f}_1, \ldots, \mathbf{f}_m$, each quantifying one aspect of the domain judged possibly relevant to matching the tuples

Learning-based Matching

- Learning a matching model M (continued)
  - convert each training example $(\mathbf{x}_i,\mathbf{y}_i,\mathbf{l}_i)$ in T to a pair $(<\mathbf{f}_1(\mathbf{x}_i,\mathbf{y}_i), \ldots, \mathbf{f}_m(\mathbf{x}_i,\mathbf{y}_i)>, \mathbf{c}_i)$
    - $\mathbf{v}_i = <\mathbf{f}_1(\mathbf{x}_i,\mathbf{y}_i), \ldots, \mathbf{f}_m(\mathbf{x}_i,\mathbf{y}_i)>$ is a feature vector that encodes $(\mathbf{x}_i,\mathbf{y}_i)$ in terms of the features
    - $\mathbf{c}_i$ is an appropriately transformed version of label l_i (e.g., yes/no or 1/0, depending on what matching model we want to learn)
  - thus T is transformed into T' = $\{(\mathbf{v}_1,\mathbf{c}_1), \ldots, (\mathbf{v}_n,\mathbf{c}_n)\}$
  - apply a learning algorithm (e.g. decision trees, SVMs) to T' to learn a matching model M

Learning-based Matching

- Applying model M to match new tuple pairs
  - given pair (x,y), transform it into a feature vector
    - $v = \langle \mathbf{f}_1(\mathbf{x},\mathbf{y}), \ldots, \mathbf{f}_m(\mathbf{x},\mathbf{y}) \rangle$
  - apply M to v to predict whether x matches y

Example:
Learning a Linearly Weighted Rule

<a_1 = (Mike Williams, (425) 247 4893, Seattle, WA), b_1 = (M. Williams, 247 4893, Redmond, WA), yes>
<a_2 = (Richard Pike, (414) 256 1257, Milwaukee, WI), b_2 = (R. Pike, 256 1237, Milwaukee, WI), yes>
<a_3 = (Jane McCain, (206) 111 4215, Renton, WA), b_3 = (J. M. McCain, 112 5200, Renton, WA), no>

match names match phones match cities match states check area code against city

v_1 = <[$s_1(a_1,b_1)$, $s_2(a_1,b_1)$, $s_3(a_1,b_1)$, $s_4(a_1,b_1)$, $s_5(a_1,b_1)$, $s_6(a_1,b_1)$], 1>

v_2 = <[$s_1(a_2,b_2)$, $s_2(a_2,b_2)$, $s_3(a_2,b_2)$, $s_4(a_2,b_2)$, $s_5(a_2,b_2)$, $s_6(a_2,b_2)$], 1>

v_3 = <[$s_1(a_3,b_3)$, $s_2(a_3,b_3)$, $s_3(a_3,b_3)$, $s_4(a_3,b_3)$, $s_5(a_3,b_3)$, $s_6(a_3,b_3)$], 0>

- $\mathbf{s}_1$ and $\mathbf{s}_2$ use Jaro-Winkler and edit distance
- $\mathbf{s}_3$ uses edit distance (ignoring area code of a)
- $\mathbf{s}_4$ and $\mathbf{s}_5$ return 1 if exact match, 0 otherwise
- $\mathbf{s}_6$ encodes a heuristic constraint

Example:
Learning a Linearly Weighted Rule

- Goal: learn rule s(a,b) = $\sum_{i=1}^{6} \alpha_i s_i(a,b)$
- Perform a least-squares linear regression on training data

v_1 = <[$s_1(a_1,b_1)$, $s_2(a_1,b_1)$, $s_3(a_1,b_1)$, $s_4(a_1,b_1)$, $s_5(a_1,b_1)$, $s_6(a_1,b_1)$], 1>

v_2 = <[$s_1(a_2,b_2)$, $s_2(a_2,b_2)$, $s_3(a_2,b_2)$, $s_4(a_2,b_2)$, $s_5(a_2,b_2)$, $s_6(a_2,b_2)$], 1>

v_3 = <[$s_1(a_3,b_3)$, $s_2(a_3,b_3)$, $s_3(a_3,b_3)$, $s_4(a_3,b_3)$, $s_5(a_3,b_3)$, $s_6(a_3,b_3)$], 0>

to find weights α_i that minimizes the squared error

$$\sum_{i=1}^{3} (c_i - \sum_{j=1}^{6} \alpha_j s_j(v_i))^2$$

- c_i is the label associated with v_i

Example: Learning a Decision Tree

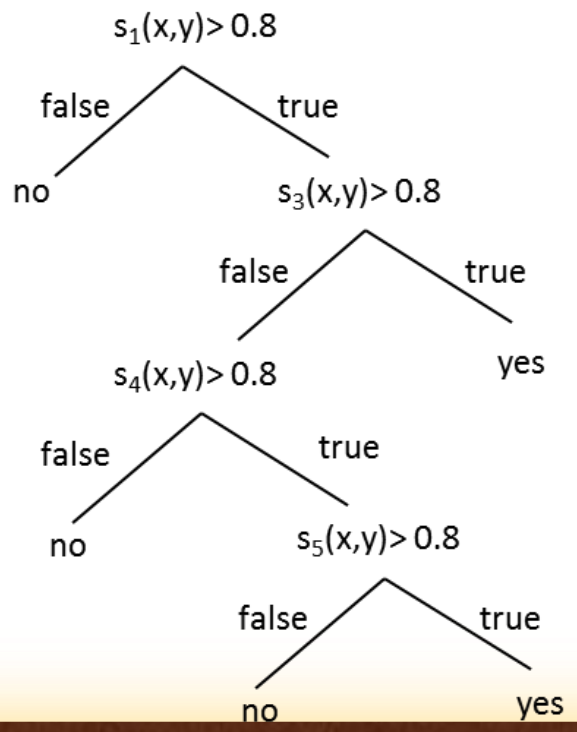
match names match phones match cities match states check area code against city

$v_1 = <[s_1(a_1,b_1), s_2(a_1,b_1), s_3(a_1,b_1), s_4(a_1,b_1), s_5(a_1,b_1), s_6(a_1,b_1)], \text{yes}>$

$v_2 = <[s_1(a_2,b_2), s_2(a_2,b_2), s_3(a_2,b_2), s_4(a_2,b_2), s_5(a_2,b_2), s_6(a_2,b_2)], \text{yes}>$

$v_3 = <[s_1(a_3,b_3), s_2(a_3,b_3), s_3(a_3,b_3), s_4(a_3,b_3), s_5(a_3,b_3), s_6(a_3,b_3)], \text{no}>$

Now the labels are yes/no, not 1/0



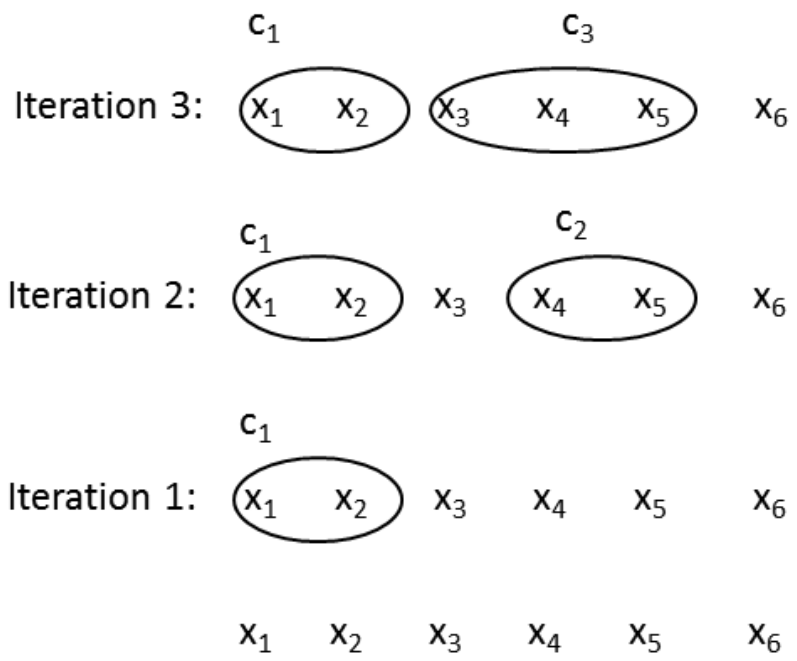
The Pros and Cons of Learning-based Approach

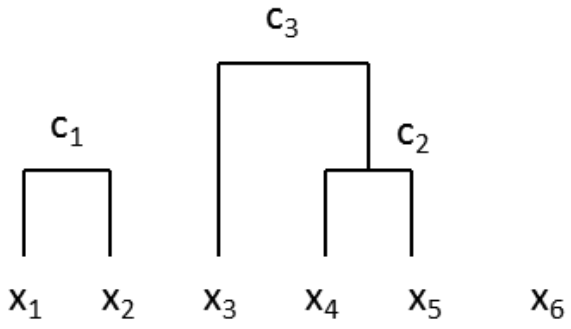
- Pros compared to rule-based approaches
  - in rule-based approaches must manually decide if a particular feature is useful ➔ labor intensive and limit the number of features we can consider
  - learning-based ones can automatically examine a large number of features
  - learning-based approaches can construct very complex "rules"
- Cons
  - still require training examples, in many cases a large number of them, which can be hard to obtain
  - clustering addresses this problem

Matching by Clustering

- Many common clustering techniques have been used
  - agglomerative hierarchical clustering (AHC), k-means, graph-theoretic, …
  - here we focus on AHC, a simple yet very commonly used one
- AHC
  - partitions a given set of tuples D into a set of clusters
    - all tuples in a cluster refer to the same real-world entity, tuples in different clusters refer to different entities
  - begins by putting each tuple in D into a single cluster
  - iteratively merges the two most similar clusters
  - stops when a desired number of clusters has been reached, or until the similarity between two closest clusters falls below a pre-specified threshold

Example





- $\mathrm{sim}(x,y) =$
  $0.3s_{name}(x,y) + 0.3s_{phone}(x,y) + 0.1s_{city}(x,y) + 0.3s_{state}(x,y)$

Computing a Similarity Score between Two Clusters

- Let c and d be two clusters
- Single link: $s(c,d) = \mathbf{min}_{x_i \in c,\, y_j \in d}\ \mathbf{sim}(x_i, y_j)$
- Complete link: $s(c,d) = \mathbf{max}_{x_i \in c,\, y_j \in d}\ \mathbf{sim}(x_i, y_j)$
- Average link: $s(c,d) = [\sum_{x_i \in c,\, y_j \in d}\ \mathbf{sim}(x_i, y_j)]\ /$ [# of (x_i, y_j) pairs]
- Canonical tuple
  - create a canonical tuple that represents each cluster
  - sim between c and d is the sim between their canonical tuples
  - canonical tuple is created from attribute values of the tuples
    - e.g., "Mike Williams" and "M. J. Williams" ➔ "Mike J. Williams"
    - (425) 247 4893 and 247 4893 ➔ (425) 247 4893

Key Ideas underlying the Clustering Approach

- View matching tuples as the problem of constructing entities (i.e., clusters)
- The process is iterative
  - leverage what we have known so far to build "better" entities
- In each iteration merge all matching tuples within a cluster to build an "entity profile", then use it to match other tuples ➔ merging then exploiting the merged information to help matching
- These same ideas appear in subsequent approaches that we will cover

Collective Matching

- Matching approaches discussed so far make independent matching decisions
  - decide whether a and b match independently of whether any two other tuples c and d match
- Matching decisions hower are often correlated
  - exploiting such correlations can improve matching accuracy

An Example

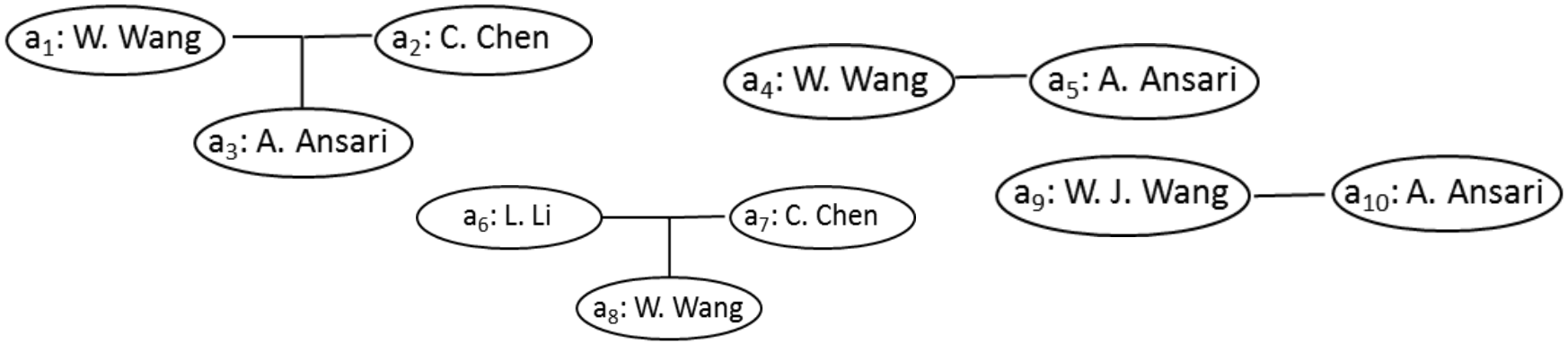
W. Wang, C. Chen, A. Ansari, A mouse immunity model
W. Wang, A. Ansari, Evaluating immunity models
L. Li, C. Chen, W. Wang, Measuring protein-bound fluxetine
W. J. Wang, A. Ansari, Autoimmunity in biliary cirrhosis

	First initial	Middle inititial	Last name
a_1	W		Wang
a_2	C		Chen
	...	...	...
a_9	W	J	Wang
a_{10}	A		Ansari

- Goal: match authors of the four papers listed above
- Solution
  - extract their names to create the table above
  - apply current approaches to match tuples in table
- This fails to exploit co-author relationships in the data

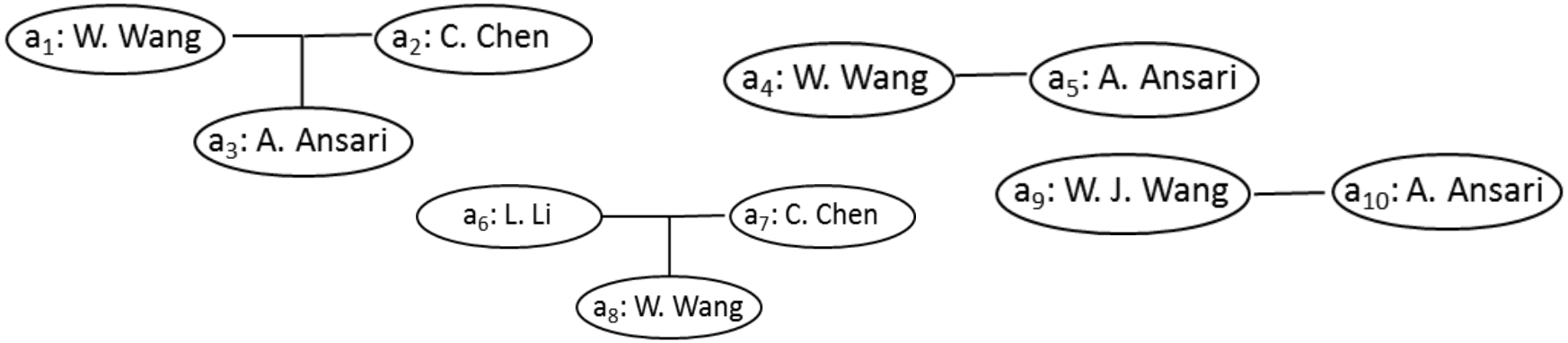
An Example (cont.)

W. Wang, C. Chen, A. Ansari, A mouse immunity model
W. Wang, A. Ansari, Evaluating immunity models
L. Li, C. Chen, W. Wang, Measuring protein-bound fluxetine
W. J. Wang, A. Ansari, Autoimmunity in biliary cirrhosis



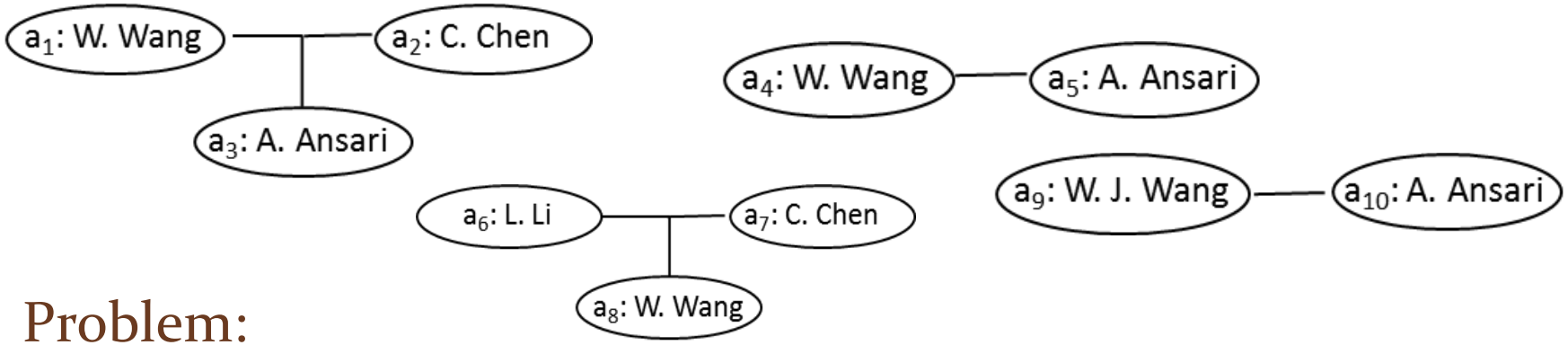
- nodes = authors, hyperedges connect co-authors
- Suppose we have matched a_3 and a_5
  - then intuitively a_1 and a_4 should be more likely to match
  - they share the same name and same co-author relationship to the same author

An Example (cont.)



- First solution:
  - add coAuthors attribute to the tuples
    - e.g., tuple a_1 has coAuthors = {C. Chen, A. Ansari}
    - tuple a_4 has coAuthors = {A. Ansari}
  - apply current methods, use say Jaccard measure for coAuthors

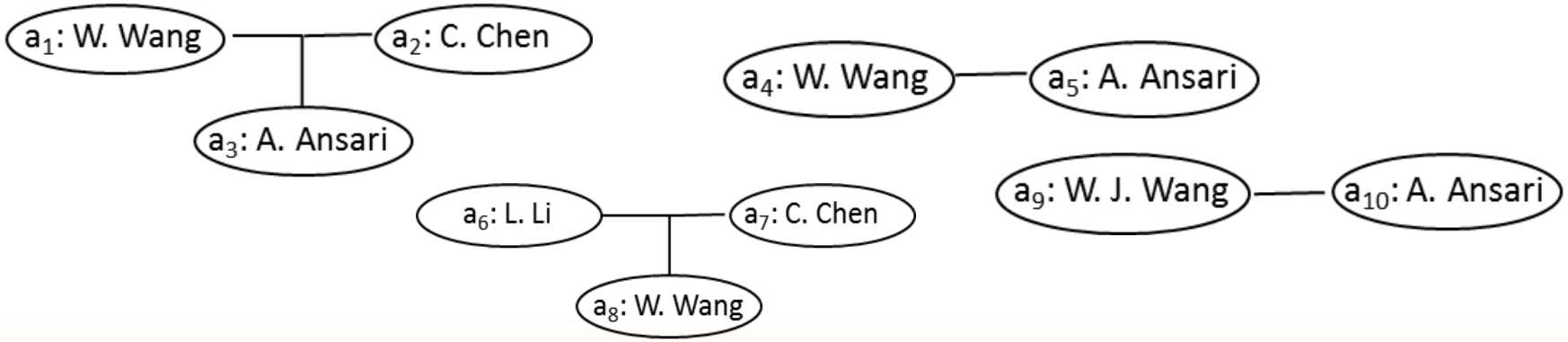
An Example (cont.)



- Problem:
  - suppose $\mathbf{a}_3$: A. Ansari and $\mathbf{a}_5$: A. Ansari share same name but do not match
  - we would match them, and incorrectly boost score of $\mathbf{a}_1$ and $\mathbf{a}_4$
- How to fix this?
  - match $\mathbf{a}_3$ and $\mathbf{a}_5$, then use that info to help match $\mathbf{a}_1$ and $\mathbf{a}_4$; do the opposite
  - so should match tuples collectively, all at once and iteratively

Collective Matching using Clustering

- Many collective matching approaches exist
  - clustering-based, probabilistic, etc.
- Clustering-based method
- Assume input is graph
  - nodes = tuples to be matched
  - edges = relationships among tuples



Collective Matching using Clustering

- To match, perform agglomerative hierarchical clustering
  - but modify sim measure to consider correlations among tuples
- Let A and B be two clusters of nodes, define
  - $sim(A,B) = \alpha * \mathbf{sim}_{attributes}(\mathbf{A,B}) + (1 - \alpha) * \mathbf{sim}_{neighbors}(\mathbf{A,B})$
  - α is pre-defined weight
  - $\mathbf{sim}_{attributes}(\mathbf{A,B})$ uses only attributes of A and B, examples of such scores are single link, complete link, average link, etc.
- $\mathbf{sim}_{neighbors}(\mathbf{A,B})$ considers correlations

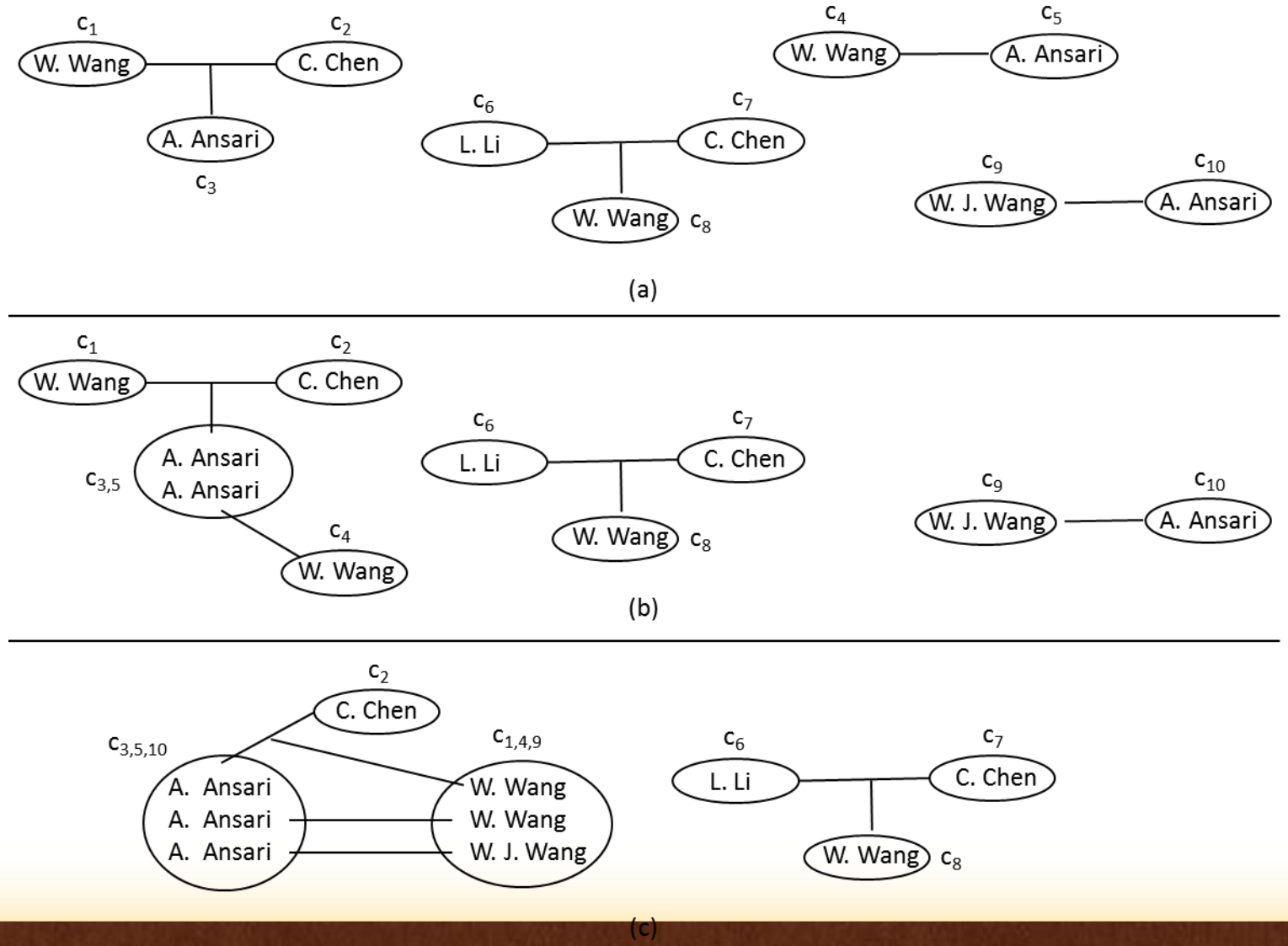
An Example of $\mathbf{sim}_{neighbors}(\mathbf{A,B})$

- Assume a single relationship R on the graph edges
  - can generalize to the case of multiple relationships
- Let N(A) be the bags of the cluster IDs of all nodes that are in relationship R with some node in A
  - e.g., cluster A has two nodes a and a',
    a is in relationship R with node b with cluster ID 3, and
    a' is in relationship R with node b' with cluster ID 3
    and another node b'' with cluster ID 5
    ➔ N(A) = {3, 3, 5}
- Define $\mathbf{sim}_{neighbors}(\mathbf{A,B})$ =
  Jaccard(N(A),N(B)) = |N(A) ∩ N(B)| / |N(A) ∪ N(B)|

An Example of $\mathbf{sim}_{neighbors}(\mathbf{A,B})$

- Recall that earlier we also define a Jaccard measure as
  - $\mathbf{JaccardSim}_{coAuthors}(\mathbf{a,b}) =$
    $|coAuthors(a) \cap coAuthors(b)| \,/\, |coAuthors(a) \cup coAuthors(b)|$
- Contrast that to
  - $\mathbf{sim}_{neighbors}(\mathbf{A,B}) =$
    $Jaccard(N(A),N(B)) = |N(A) \cap N(B)| \,/\, |N(A) \cup N(B)|$
- In the former, we assume two co-authors match if their "strings" match
- In the latter, two co-authors match only if they have the same cluster ID

An Example to Illustrate the Working of Agglomerative Hierarchical Clustering



Scaling up Rule-based Matching

- Two goals: minimize # of tuple pairs to be matched and minimize time it takes to match each pair
- For the first goal:
  - hashing
  - sorting
  - indexing
  - canopies
  - using representatives
  - combining the techniques
- Hashing
  - hash tuples into buckets, match only tuples within each bucket
  - e.g., hash house listings by zipcode, then match within each zip

Scaling up Rule-based Matching

- Sorting
  - use a key to sort tuples, then scan the sorted list and match each tuple with only the previous (w-1) tuples, where w is a pre-specified window size
  - key should be strongly "discriminative": brings together tuples that are likely to match, and pushes apart tuples that are not
    - example keys: soc sec, student ID, last name, soundex value of last name
  - employs a stronger heuristic than hashing: also requires that tuples likely to match be within a window of size w
    - but is often faster than hashing because it would match fewer pairs

Scaling up Rule-based Matching

- Indexing
  - index tuples such that given any tuple a, can use the index to quickly locate a relatively small set of tuples that are likely to match a
    - e.g., inverted index on names
- Canopies
  - use a computationally cheap sim measure to quickly group tuples into overlapping clusters called canopines (or umbrella sets)
  - use a different (far more expensive) sim measure to match tuples within each canopy
  - e.g., use TF/IDF to create canopies

Scaling up Rule-based Matching

- Using representatives
  - applied during the matching process
  - assigns tuples that have been matched into groups such that those within a group match and those across groups do not
  - create a representative for each group by selecting a tuple in the group or by merging tuples in the group
  - when considering a new tuple, only match it with the representatives
- Combining the techniques
  - e.g., hash houses into buckets using zip codes, then sort houses within each bucket using street names, then match them using a sliding window

Scaling up Rule-based Matching

- For the second goal of minimizing time it takes to match each pair
  - no well-established technique as yet
  - tailor depending on the application and the matching approach
  - e.g., if using a simple rule-based approach that matches individual attributes then combines their scores using weights
    - can use short circuiting: stop the computation of the sim score if it is already so high that the tuple pair will match even if the remaining attributes do not match