Chapter 4: String Matching

PRINCIPLES OF

DATA INTEGRATION

ANHAI DOAN   ALON HALEVY   ZACHARY IVES

MK
MORGAN KAUFMANN

Introduction

- Find strings that refer to same real-world entities
  - “David Smith” and “David R. Smith”
  - “1210 W. Dayton St Madison WI” and “1210 West Dayton Madison WI 53706”
- Play critical roles in many DI tasks
  - Schema matching, data matching, information extraction
- This chapter
  - Defines the string matching problem
  - Describes popular similarity measures
  - Discusses how to apply such measures to match a large number of strings

Outline

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

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
x_1 = Dave Smith
x_2 = Joe Wilson
x_3 = Dan Smith

(a)

Set Y
y_1 = David D. Smith
y_2 = Daniel W. Smith

(b)

Matches
(x_1, y_1)
(x_3, y_2)

(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: overlap, Jaccard, TF/IDF
  - Hybrid: generalized Jaccard, soft TF/IDF, Monge-Elkan
  - Phonetic: Soundex
- 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 $x = x_1 x_2 \cdots x_n$, $y = y_1 y_2 \cdots y_m$
  - $d(i,j)$ = edit distance between $x_1 x_2 \cdots x_i$ and $y_1 y_2 \cdots y_j$, the i-th and j-th prefixes of x and y
- Recurrence equations

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

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

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

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
|      | |
d e e v e
```

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

$c_g = 1$

- alignment score = sum of scores of all correspondences - sum of penalties of all 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,0) > score(a,0)
  - e.g., different amino-acid pairs may have different semantic distance
- Generalizes insertion and deletion into gaps, and generalizes their costs from 1 to $\mathbf{C}_g$

Computing Needleman-Wunch Score with Dynamic Programming

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

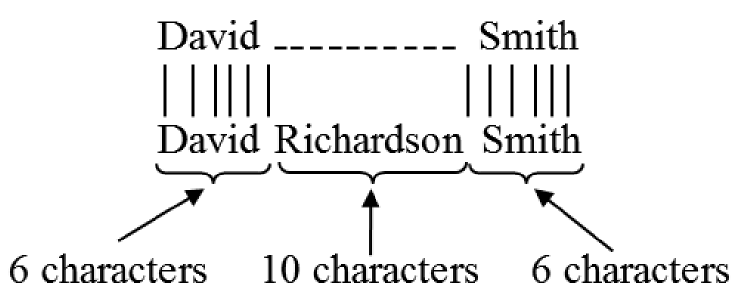
$$s(0,j) = -jc_g$$
$$s(i,0) = -ic_g$$

		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) = $\mathbf{c_0} + (\mathbf{k-1})\mathbf{c_r}$
  - $\mathbf{c_0}$ = cost of opening gap
  - $\mathbf{c_r}$ = cost of continuing gap, $\mathbf{c_0} > \mathbf{c_r}$
- E.g., "David Smith" vs. "David Richardson Smith"
  - $\mathbf{c_0}$ = 1, $\mathbf{c_r}$ = 0.5, alignment cost = 6*2 – 1 - 9*0.5 = 6.5

Computing Affine Gap Score using Dynamic Programming

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

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

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

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

- The notes detail 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: Basic Ideas

- 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

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 $|i\text{-}j| \leq \min\{|x|,|y|\}/2$
  - 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 $jaro(x,y) = 1 / 3[c/|x| + c/|y| + (c - t/2)/c]$, 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

Outline

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

Set-based Similarity Measures

- View strings as sets or multi-sets of tokens
- Use set-related properties to compute similarity scores
- Common methods to generate tokens
  - consider words delimited by space
    - possibly stem the words (depending on the application)
    - remove common stop words (e.g., the, and, of)
    - e.g., given “david smith” → generate tokens “david” and “smith”
  - consider q-grams, substrings of length q
    - e.g., “david smith” → the set of 3-grams are {##d, #da, dav, avi, …, h##}
    - special character # is added to handle the start and end of string

The Overlap Measure

- Let $\mathbf{B}_x$ = set of tokens generated for string x
- Let $\mathbf{B}_y$ = set of tokens generated for string y
- $O(x,y) = |\mathbf{B}_x \cap \mathbf{B}_y|$
  - returns the number of common tokens
- E.g., x = dave, y = dav
  - $\mathbf{B}_x$ = {#d, da, av, ve, e#}, $\mathbf{B}_y$ = {#d, da, av, v#}
  - $O(x,y) = 3$

The Jaccard Measure

- $J(x,y) = |\mathbf{B}_x \cap \mathbf{B}_y| / |\mathbf{B}_x \cup \mathbf{B}_y|$
- 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

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

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

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

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

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

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

$tf(c, z) = 0$

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
tf(b, x) = 1
...
tf(c, z) = 0

idf(a) = 3/3 = 1
idf(b) = 3/1 = 3
idf(c) = 3/1 = 3

	a	**b**	**c**
$\mathbf{v}_x$	2	3	0
$\mathbf{v}_y$	3	0	3
$\mathbf{v}_z$	3	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) / $\sqrt{\sum_{\{t \in T\}} v_d(t)^2}$

Outline

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

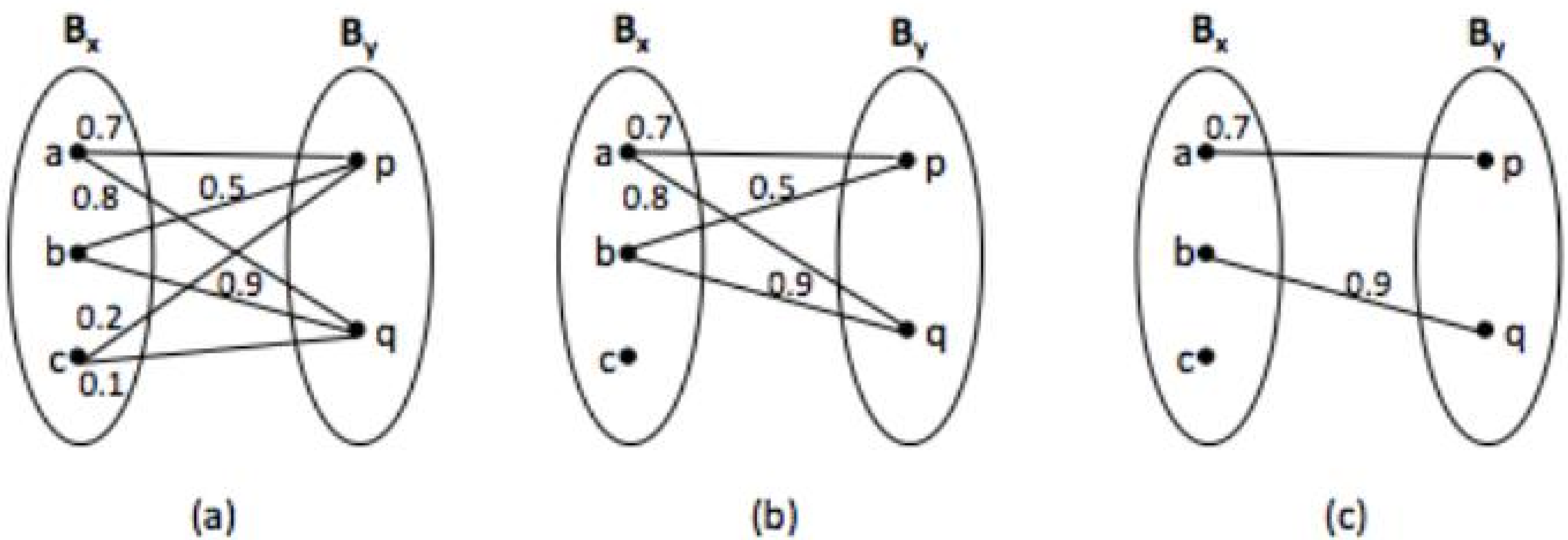
Generalized Jaccard Measure

- Jaccard measure
  - considers overlapping tokens in both x and y
  - a token from x and a token from y must be identical to be included in the set of overlapping tokens
  - this can be too restrictive in certain cases
- Example:
  - matching taxonomic nodes that describe companies
  - “Energy & Transportation” vs. “Transportation, Energy, & Gas”
  - in theory Jaccard is well suited here, in practice Jaccard may not work well if tokens are commonly mispelled
    - e.g., energy vs. eneryg
  - generalized Jaccard measure can help such cases

Generalized Jaccard Measure

- Let $B_x = \{x_1, \ldots, x_n\}$, $B_y = \{y_1, \ldots, y_m\}$
- Step 1: find token pairs that will be in the "softened" overlap set
  - apply a similarity measure s to compute sim score for each pair (x_i, y_j)
  - keep only those score $\geq$ a given threshold α, this forms a bipartite graph G
  - find the maximum-weight matching M in G
- Step 2: return normalized weight of M as generalized Jaccard score
  - $GJ(x,y) = \sum_{(xi,yj) \in M} s(x_i,y_j) / (|B_x| + |B_y| - |M|)$

An Example



- Generalized Jaccard score: (0.7 + 0.9)/(3 + 2 – 2) = 0.53

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", with Apple being mispelt 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)

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

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

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

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

(b)

$$
\begin{aligned}
s(x,y) = \; & v_x(a) \,.\, v_y(a) \,.\, 1 \; + \\
& v_x(b) \,.\, v_y(b') \,.\, 0.8 \; + \\
& v_x(c) \,.\, v_y(c) \,.\, 1
\end{aligned}
$$

(c)

The Monge-Elkan Measure

- Break strings x and y into multiple substrings
  - x = A_1 … A_n , y = B_1 … B_m
- Compute
  - $s(x,y) = 1/n * \sum_{i=1}^{n} max_{j=1}^{m} s'(\mathbf{A}_i,\mathbf{B}_j)$
  - s’ is a secondary sim measure, such as Jaro-Winkler
  - Intuitively, we ignore the order of the matching of substrings and only consider the best match for substrings of x in y
- E.g., match two strings
  “Comput. Sci. and Eng. Dept., University of California, San Diego”
  “Department of Computer Science, Univ. Calif., San Diego”

Outline

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

Phonetic Similarity Measures

- Match strings based on their sound, instead of appearances
- Very effective in matching names, which often appear in different ways that sound the same
  - e.g., Meyer, Meier, and Mire; Smith, Smithe, and Smythe
- Soundex is most commonly used

The Soundex Measure

- Used primarily to match surnames
  - maps a surname x into a 4-letter code
  - two surnames are judged similar if share the same code
- Algorithm to map x into a code:
  - Step 1: keep the first letter of x, subsequent steps are performed on the rest of x
  - Step 2: remove all occurences of W and H. Replace the remaining letters with digits as follows:
    - replace B, F, P, V with 1, C, G, J, K, Q, S, X, Z with 2, D, T with 3, L with 4, M, N with 5, R with 6
  - Step 3: replace sequence of identical digits by the digit itself
  - Step 4: Drop all non-digit letters, return the first four letters as the soundex code

The Soundex Measure

- Example: x = Ashcraft
  - after Step 2: A226a13, after Step 3: A26a13, Step 4 converts this into A2613, then returns A261
  - Soundex code is padded with 0 if there is not enough digits
- Example: Robert and Rupert map into R163
- Soundex fails to map Gough and Goff, and Jawornicki and Yavornitzky
  - designed primarily for Caucasian names, but found to work well for names of many different origins
  - does not work well for names of East Asian origins
    - which uses vowels to discriminate, Soundex ignores vowels

Outline

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

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
  - 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 and overlap measures for now

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

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 $|x \cap y| \geq k$ ➔ any subset $x' \subseteq x$ 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: $\underbrace{\text{\{lake, monona,}}_{x'}$ 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)
  - 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 (see notes)
- 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)

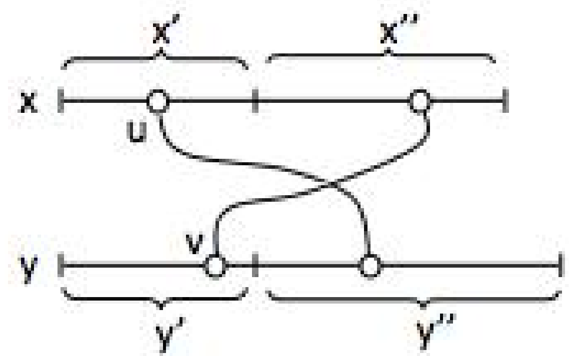
Reordered Set X

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)

- See the notes for applying prefix filtering to Jaccard measure

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} (see notes)
- But we can do better than this. Specifically, we can prove that $O(x,y) \geq [t/(1+t)]*(|x| + |y|) = 4.44$ (see notes)
  - so can immediately discard the above (x,y) pair

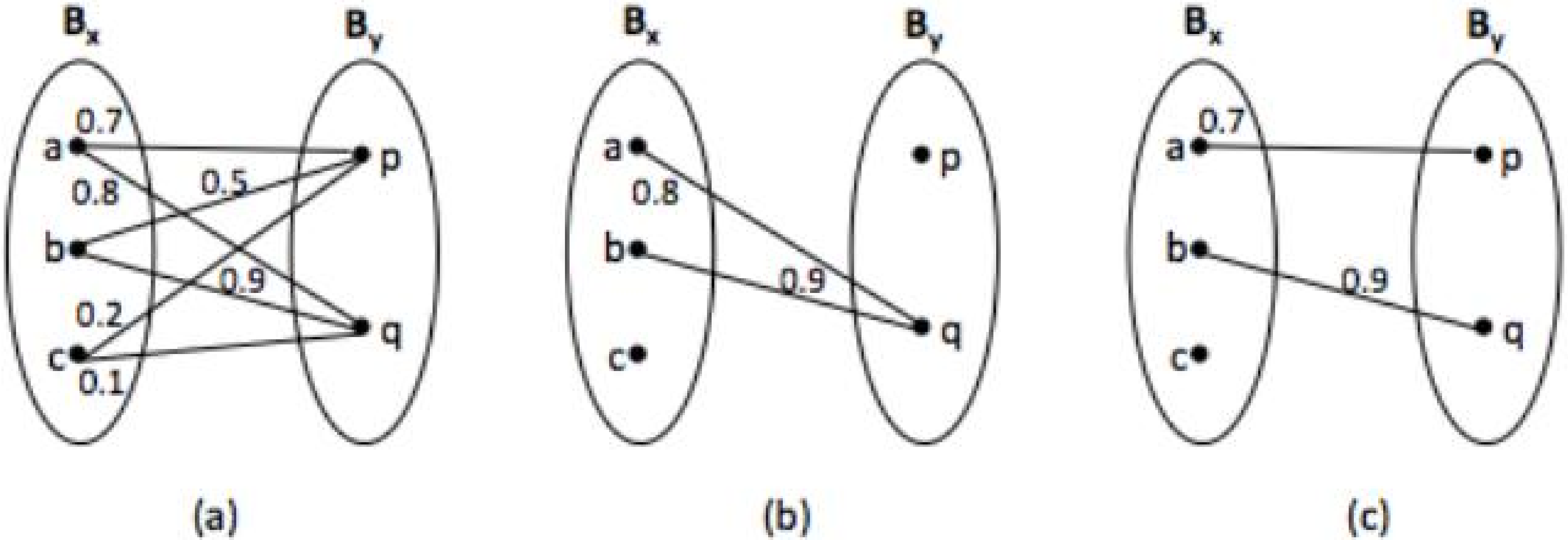
Bound Filtering

- Used to optimize the computation of generalized Jaccard similarity measure
- Recall that
  - $GJ(x,y) = \sum_{(xi,yj) \in M} \mathbf{s}(\mathbf{x}_i, \mathbf{y}_j) / (|\mathbf{B}_x| + |\mathbf{B}_y| - |M|)$
- Algorithm
  - for each (x,y) compute an upper bound UB(x,y) and a lower bound LB(x,y) on GJ(x,y)
  - if UB(x,y) $\leq$ t ➔ (x,y) can be ignored, it is not a match
    if LB(x,y) $\geq$ t ➔ return (x,y) as a match
    otherwise compute GJ(x,y)

Computing UB(x,y) and LB(x,y)

- For each $\mathbf{x}_i \in \mathbf{B}_x$, find $\mathbf{y}_j \in \mathbf{B}_y$ with the highest element-level similarity, such that $\mathbf{s}(\mathbf{x}_i,\mathbf{y}_j) \geq \alpha$. Call this set of pairs $\mathbf{S}_1$.
- For each $\mathbf{y}_j \in \mathbf{B}_y$, find $\mathbf{x}_i \in X$ with the highest element-level similarity, such that $\mathbf{s}(\mathbf{x}_i,\mathbf{y}_j) \geq \alpha$. Call this set of pairs $\mathbf{S}_2$.
- Compute
  - $UB(x,y) = \sum_{(xi,yj) \in S1 \cup S2} \mathbf{s}(\mathbf{x}_i,\mathbf{y}_j) / (|\mathbf{B}_x| + |\mathbf{B}_y| - |\mathbf{S}_1 \cup \mathbf{S}_2|)$
  - $LB(x,y) = \sum_{(xi,yj) \in S1 \cap S2} \mathbf{s}(\mathbf{x}_i,\mathbf{y}_j) / (|\mathbf{B}_x| + |\mathbf{B}_y| - |\mathbf{S}_1 \cap \mathbf{S}_2|)$

Example



- $\mathbf{S}_1$ = {(a,q), (b,q)}, $\mathbf{S}_2$ = {(a,p), (b,q)}
- UB(x,y) = (0.8+0.9+0.7+0.9)/(3+2-3) = 1.65
- LB(x,y) = 0.9/(3+2-1) = 0.225

Extending Scaling Techniques to Other Similarity Measures

- Discussed Jaccard and overlap so far
- To extend a technique T to work for a new similarity measure s(x,y)
  - try to translate s(x,y) into constraints on a similarity measure that already works well with T
- The notes discuss examples that involve edit distance and TF/IDF

Summary

- String matching is pervasive in data integration
- Two key challenges:
  - what similarity measure and how to scale up?
- Similarity measures
  - Sequence-based: edit distance, Needleman-Wunch, affine gap, Smith-Waterman, Jaro, Jaro-Winkler
  - Set-based: overlap, Jaccard, TF/IDF
  - Hybrid: generalized Jaccard, soft TF/IDF, Monge-Elkan
  - Phonetic: Soundex
- Scaling up string matching
  - Inverted index, size/prefix/position/bound filtering

Acknowledgment

- Slides in the scalability section are adapted from http://pike.psu.edu/p2/wisc09-tech.ppt