Chapter 6

String Matching

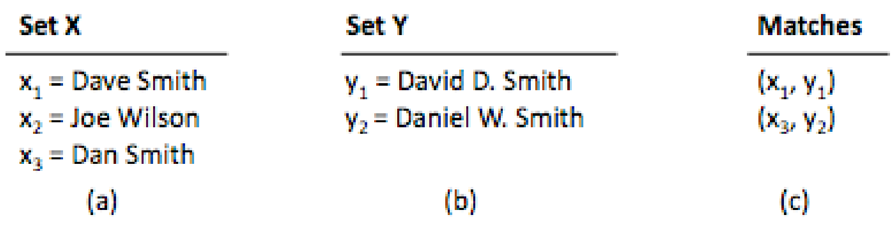
String matching is the problem of finding strings that refer to the same real-world entity. For example, the string "David Smith" in one database may refer to the same person as "David R. Smith" in another database. Similarly, the strings "1210 W. Dayton St, Madison WI" and "1210 West Dayton, Madison WI 53706" refer to the same physical address.

String matching plays a critical role in many data integration tasks, including schema matching, data matching, and information extraction. Consequently, in this chapter we examine this problem in depth. Section 6.1 defines the string-matching problem, and Section 6.2 describes popular similarity measures that can be used to compute a similarity score between any two given strings. Section 6.3 discusses how to efficiently apply such a measure to match a large number of strings.

6.1 Problem Description

The problem we address in this chapter is the following. Given two sets of strings X and Y, we want to find all pairs of strings, (x, y), where $x \in X$, $y \in Y$, and such that x and y refer to the same real-world entity. We refer to such pairs as *matches.* Figure 6.1(a-b) shows two example databases representing sets of persons, and Figure 6.1(c) shows the matches between them. For example, the first match (x_1, y_1) states that strings David Smith and David D. Smith refer to the same real-world person.

Solving the matching problem raises two major challenges: accuracy and scaleup. Matching strings accurately is difficult because strings that refer to the same real-world entity are often very different. The reasons for appearing different include typing and OCR errors (e.g., David Smith is mis-spelled as Davod Smith), different formatting conventions (10/8/2009 vs. Oct 8, 2009), custom abbreviation, shortening



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)

Figure 6.1: An example of matching person names. Column (c) shows the matches between the databases in (a) and (b).

of strings, or omission (Daniel Walker Herbert Smith vs. Dan W. Smith), different names or nicknames (William Smith vs. Bill Smith), and shuffling parts of the string (e.g., Dept. of Computer Science, UW-Madison vs. Computer Science Dept., UW-Madison). Additionally, in some cases the database does not contain enough information to determine whether two strings refer to the same entity (e.g., trying to decide whether an author mentioned on two different publications is the same person).

To address the accuracy challenge, a common solution is to define a *similarity measure* s that takes two strings x and y and returns a score in the range $[0, 1]$. The intention is that the higher the score, the more likely that x and y match. We then say that x and y indeed match if $s(x, y)$ equals or exceeds a pre-specified threshold t. Many similarity measures have been proposed, and we discuss the main ones in Section 6.2.

The second challenge is to apply the similarity metric to a large number of strings. Since string similarity is typically applied to data entries, applying $s(x, y)$ to all pairs of strings in the Cartesian product of sets X and Y would be quadratic in the size of the data, and therefore impractical. To address this challenge, several solutions have been proposed to apply the similarity test only to the most promising pairs. We discuss the key ideas of these solutions in Section 6.3.

6.2 Similarity Measures

A broad range of measures have been proposed to match strings. A similarity measure maps a pair of strings x and y into a number in the range $[0, 1]$ such that a higher value indicates greater similarity between x and y. The terms *Distance* and *cost measures* have also used to describe the same concept, except that smaller values indicate higher similarity.

Broadly speaking, current similarity measures fall into four groups: sequence-based, set-based, hybrid, and phonetic measures. We now describe each one in turn.

6.2.1 Sequence-based similarity measures

The first set of similarity measures we consider view the strings as sequences of characters, and compute a cost of transforming one string into the other. We begin with the basic edit distance, and then consider several more elaborate versions.

Edit distance

The edit distance measure (also known as Levenshtein distance), $d(x, y)$, computes the *minimal* cost of transforming string x to string y. Transforming a string is carried out using a sequence of the following operators: delete a character, insert a character and substitute a character with another.

Example 6.1: The cost of transforming the string $x =$ David Smiths to the string $y =$ Davidd Simth is 4. The required operations are: deleting a d, replacing m by i, replacing i by m, and deleting an s. □

It is not hard to see that the minimal cost of transforming x to y is the same as the minimal cost of transforming y to x (using in effect the same transformation). Thus, $d(x, y)$ is well-defined and symmetric with respect to both x and y.

Intuitively, edit distance tries to capture the various ways people make editing mistakes, such as inserting an extra character (e.g., the second d in davidd), or swapping two characters (e.g., mi in x vs. im in simths). Hence, the smaller the edit distance, the more similar the two strings are.

The edit distance function $d(x, y)$ is converted into a similarity function $s(x, y)$ as follows:

$$s(x, y) = 1 - \frac{d(x, y)}{max(length(x), length(y))}$$

In Example 6.1 the similarity would be

$$s(x, y) = 1 - \frac{4}{max(12, 12)} = 0.67$$

Computing the edit distance: The value of $d(x, y)$ can be computed by a dynamic programming algorithm. Let $x = x_1x_2 \ldots x_n$ and $y = y_1y_2 \ldots y_m$, where the

```
         x = David- Smiths
             | |||||||||||
         y = Davidd Simth-
operations   ccccciccssccd
      cost   0000010011001
```

(a)

$$
\begin{aligned}
d(x,y) &= 4 \\
s(x,y) &= 1 - d(x,y) / \max\{\text{length}(x), \text{length}(y)\} \\
&= 1 - 4 / \max\{12, 12\} \\
&= 0.67
\end{aligned}
$$

(b)

Figure 6.2: (a) Transforming string x to string y using edit operations, and (b) converting the edit distance into a similarity score.

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

(a)

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

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

(b)

Figure 6.3: (a) The recurrence equation for computing edit distance between strings x and y using dynamic programming, and (b) a simplified form of the above equation.

x_i and y_j are characters. Let $d(i, j)$ denote the edit distance between $x_1x_2 \ldots x_i$ and $y_1y_2 \ldots y_j$ (which are the i-th and j-th prefixes of x and y).

The key to designing a dynamic-programming algorithm is to establish a recurrence relation that enables computing d(i,j) from previously computed values of d. Figure 6.3(a) shows the recurrence in our case. When x$_i$ and y$_j$ are the same (first line), then d(i,j) is the same as d(i-1, j-1). When either a substitution, deletion or insertion is needed, the edit distance is incremented by one from the appropriate previous value. Figure 6.3 (b) simplifies the equation in Figure 6.3(a) by merging the first two lines.

Figure 6.4 shows how to implement dynamic programming using the above equation. The figure illustrates computing the distance between the strings dva and dave. We start with the matrix in Figure 6.4(a), where the x_i are listed on the left, and the y_j at the top. Note that we have added x_0 and y_0, two null characters at the start of x and y to simplify the implementation. Specifically, this allows us to quickly fill in the first row and column, by setting $d(i, 0) = i$ and $d(0, j) = j$.

Now we can use the equation in Figure 6.3(b) to fill the rest of the matrix. For example, we have $d(1, 1) = \min\{0+0, 1+1, 1+1\} = 0$. Since this value is obtained by adding 0 to $d(0,0)$, we add an arrow pointing from cell $(1,1)$ to cell $(0,0)$. Similarly, $d(1,2) = 1$ (see Figure 6.4(a)). Figure 6.4(b) shows the entire filled-in matrix. The edit distance between x and y can then be found to be 2, in $(3,4)$, the bottom rightmost cell.

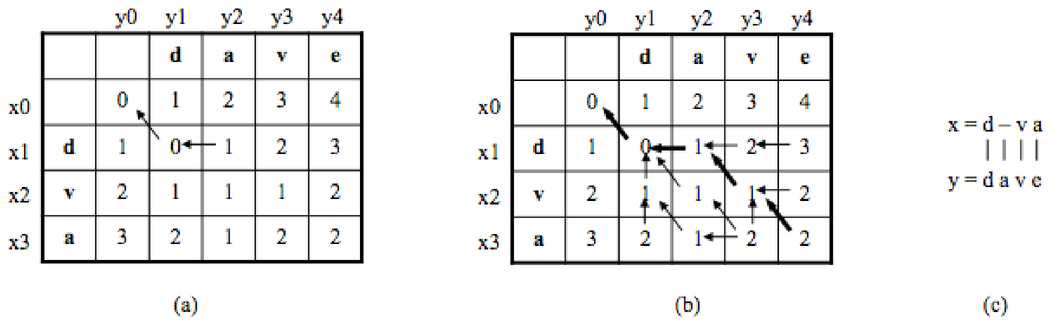


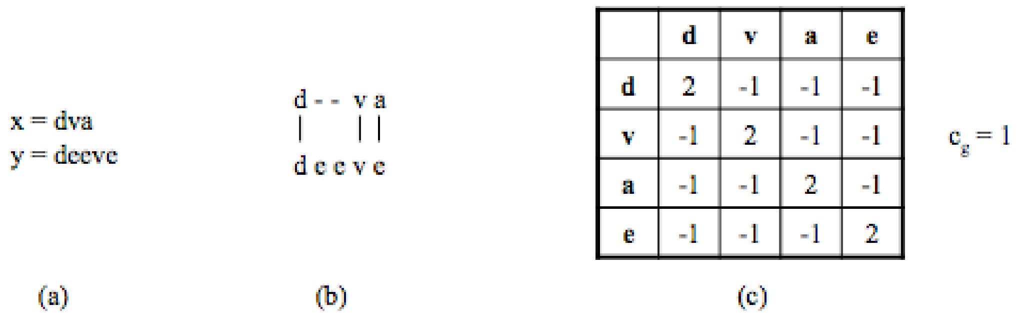
Figure 6.4: Computing the edit distance between dva and dave using the dynamic programming equation in Figure 6.3(b). (a) The dynamic programming matrix after filling in several cells, (b) the filled-in matrix, and (c) finding the sequence of edit operations that transforms dva into dave, by following the bold arrows in Part b.

In addition to computing the edit distance, the algorithm also shows the sequence of edit operations, by following the arrows. In our example, since the arrow goes "diagonal" from $(3,4)$ to $(2,3)$, we know that x_3 (character a) has been copied into or substituted with y_4 (character e). The arrow then goes "diagonal" again, from $(2,3)$ to $(1,2)$. So again x_2 (character v) has been copied into or substituted with y_3 (character v). Next, the arrow goes "horizontal", from $(1,2)$ to $(1,1)$. This means a gap - has been inserted into x, and aligned with a in y (which denotes an insertion operation). This process stops when we have reached cell $(0,0)$. The transformation is depicted in Figure 6.4(c).

The cost of computing the edit distance is $length(x)length(y)$. In practice the lengths of x and y often are roughly the same, and hence we often refer to the above cost as quadratic.

The Needleman-Wunch measure

The Needleman-Wunch measure models the variations between strings more explicitly than a series of transformations. As a result, it can support more elaborate cost functions. First, the measure allows different penalties for variations between the strings.



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

Figure 6.5: An example for the Needleman-Wunch function: (a) two strings x and y, (b) an alignment between x and y, (c) a score matrix and a gap penalty c_g.

For example, the cost of transforming the letter o to the digit 0 (a mistake often made by OCR systems) can receive a smaller penalty than the cost of transforming a to 0. As another example, when matching bioinformatic sequences, different pairs of amino acids have different semantic distances. Second, the measure explicitly models gaps in the alignment of the two strings, and is flexible enough to assign the gap an arbitrary cost.

The Needleman-Wunch measure is computed by assigning a score to each *alignment* between the two input strings, and choosing score of the best alignment. An alignment between two strings x and y is a set of correspondences between the characters of x and y, allowing for gaps. For example, Figure 6.5(b) shows an alignment between the strings $x =$ dva and $y =$ deeve, where d corresponds to d, v to v, and a to e. Note that a gap of length 2 has been inserted into x, and so the two characters ee do not have any corresponding characters in y.

The score of the alignment is computed using a *score matrix* and a *gap penalty.* The matrix assigns a score for a correspondence between every pair of characters and therefore allows to penalize transformations on a case by case basis. Figure 6.5(c) shows a sample score matrix where a correspondence between two identical characters scores 2, and -1 otherwise. A gap of length 1 has a penalty c_g (set to 1 in Figure 6.5). A gap of length k has a linear penalty of kc_g.

Given an alignment A, a score matrix, and a gap penalty, we can then compute the score of x and y given A as the sum of the scores of all matches in A, minus the penalties for the gaps. For example, the score of the alignment in Figure 6.5(b) is

2 (for match d-d) + 2 (for match v-v) - 1 (for match a-e) - 2 (penalty for the gap of length 2) = 1. This is the best alignment between x and y and therefore is the Needleman-Wunch measure between the two.

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

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

(a)

		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

(b)

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

(c)

Figure 6.6: An example for the Needleman-Wunch function: (a) the recurrence equation for computing the similarity score using dynamic programming, (b) the dynamic-programming matrix between dva and deeve using the equation in Part a, and (c) finding the optimal alignment between the above two strings, by following the arrows in Part b.

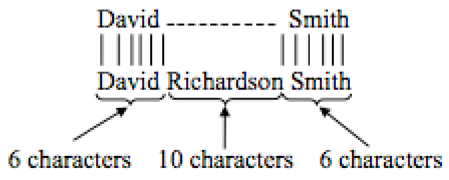
Computing the Needleman-Wunch measure: The Needleman-Wunch score, $s(x, y)$, is computed with a dynamic programming algorithm, using the recurrence equation shown in Figure 6.6(a). We note three differences between the algorithm used here and the one used for computing the edit distance (Figure 6.3(b)). First, here we compute the *max* instead of *min*. Second, we use the score matrix, $c(x_i, y_j)$, in the recurrence equation instead of the unit costs of edit operations. Third, we use the gap cost, c_g, instead of the unit gap cost. Note that when initializing this matrix, we must set $s(i, 0) = -ic_g$ and $s(0, j) = -jc_g$ (instead of i and j as in the edit distance case).

Figure 6.6(b) shows the fully filled-in matrix for x = dva and y = deeve, using the score matrix and gap penalty in Figure 6.5(c). Figure 6.6(c) shows the optimal alignment, depicted by following the arrows in the filled-in matrix of Figure 6.6(b).

The Affine Gap measure

The Affine Gap measure is an extension of the Needleman-Wunch measure that handles longer gaps more gracefully. Consider matching x = David Smith with y = David

R. Smith. The Needleman-Wunch measure can do this match very well, by opening a gap of length 2 in x, right after David , then aligning the gap with R. However, consider matching the same x = David Smith with y' = David Richardson Smith, as shown in Figure 6.7(a). Here the gap between the two strings is 10 character long. Needleman-Wunch does not match very well because the cost of the gap is too high. For example, if each match of a character has a score 2, and that C_g is 1, then the score of the above alignment under Needleman-Wunch is 6*2-10 = 2.



(a)

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

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

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

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

(b)

Figure 6.7: (a) An example of two strings where there is a long gap, (b) the recurrence equations for the affine gap measure.

In practice, gaps tend to be longer than one character. Hence, assigning a uniform penalty to each character in the gap in a sense will unfairly punish long gaps. The Affine Gap measure addresses this problem by distinguishing between the cost of *opening* the gap and the cost of *continuing* the gap. Formally, the mesaure assigns to each gap of length k a cost $c_o + (k-1)c_r$, where c_o is the cost of opening the gap (i.e., the cost of the very first character of the gap), and c_r is the cost of continuing the gap (i.e., the cost of the remaining $k-1$ characters of the gap). Cost c_r is less than c_o, thereby lessening the penalty of a long gap. Continuing with the example in Figure 6.7, assume that $c_o = 1$ and $c_r = 0.5$, the score of the alignment is 6*2 - 1 - 9*0.5 = 6.5, much higher than the score 2 obtained under Needleman-Wunch.

The recurrence relation for computing the Affine Gap measure is a bit more complex. At every stage, we keep track of three values:

- $M(i,j)$: the best score between $x_1 \ldots x_i$ and $y_i \ldots y_j$ given that x_i is aligned with y_j,

- $I_x(i,j)$: the best score given that x_i is aligned to a gap, and

- $I_y(i,j)$: the best score given that y_j is aligned to a gap.

The best score for the cell, $s(i,j)$, is then the maximum of these three scores.

In deriving the recurrence equation for the Affine Gap measure, we make the following assumption about the cost function. We assume that in an optimal alignment, we will never have an insertion followed directly by a deletion, or vice versa. This means we will never have the situation depicted in Figure 6.8 (a) or (b). We can guarantee this property by by setting the cost $-(c_o + c_r)$ to be lower than the lowest mismatch score in the score matrix. Under such conditions, the alignment in Figure 6.8(c)will have a higher score than those to its left.

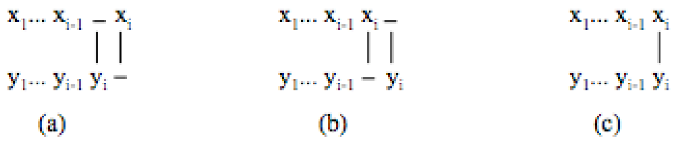


Figure 6.8: By setting the weights in the score matrix appropriately, (c) will always be more optimal than (a) or (b).

Figure 6.9 explains how we derive the equation for $M(i,j)$ in Figure 6.7(b). This equation considers the case where x_i is aligned with y_j (Figure 6.9(a)). Thus, $x_1 \ldots x_{i-1}$ is aligned with $y_1 \ldots y_{j-1}$. This can happen in three ways, as shown in Figures 6.9(b)-(d): x_{i-1} is aligned with y_{j-1}, x_{i-1} is aligned into a gap, or y_{j-1} is aligned into a gap. These three ways give rise to the three cases in the equation for $M(i,j)$ in Figure 6.7(b).

Figure 6.10 shows how to derive the equation for $I_x(i,j)$ in Figure 6.7(b). This equation considers the case that x_i is aligned into a gap (Figure 6.10(a)). This can only happen in three ways, as shown in Figure 6.10(b)-(d): x_{i-1} is aligned with y_j, x_{i-1} is aligned into a gap, or y_j is aligned into a gap. The first two ways give rise to the two cases shown in Figure 6.7(b). The third case cannot happen, due to the assumption we explained above. The equation for $I_y(i,j)$ in Figure 6.7(b) is derived in a similar fashion. The complexity of computing the affine gap measure remains $O(|x||y|)$.

The Smith-Waterman measure

The previous measures considered *global* alignments between the input strings. That is, they attempt to match all characters of x with all characters of y.

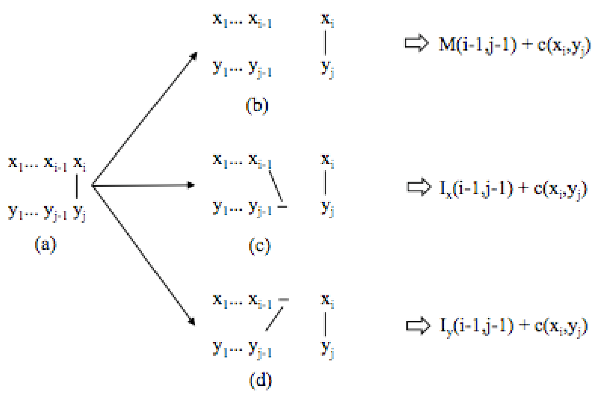


Figure 6.9: The derivation of the equation for $M(i, j)$ in Figure 6.7(b)

Global alignments may not be well suited for some cases. For example, consider the two strings Prof. John R. Smith, Univ of Wisconsin and John R. Smith, Professor. The similarity score based on global alignments will be relatively low. In such a case, what we really want is to find two substrings of x and y that are most similar, and then return the score between the substrings as the score for x and y. For example, here we would want to identify John R. Smith to be the most similar substrings of the above two strings. This means matching the above two strings by ignoring certain prefixes (e.g., Prof.) and suffixes (e.g., Univ of Wisconsin in x and Professor in y). We call this *local alignment*.

The Smith-Waterman is designed to find matching substrings by introducing two key changes to the Needleman-Wunch measure. First, the measure allows the match to re-start at any position in the strings (no longer limited to just the first position). The re-start is captured by the first line of recurrence equation in Figure 6.11(a). Intuitively, if the global match dips below zero, then this line has the effect of *ignoring the prefix* and restarting the match. Similarly, note that all values in the first row and column are zeros, instead of $-ic_g$ and $-jc_g$ as in the Needleman-Wunsch case. Applying this recurrence relation to our example produces the matrix in Figure 6.11(b).

The second key change is that after computing the matrix using the recurrence relation, the algorithm starts retracing the arrows from the largest value in the matrix (4 in our example) rather than starting from the lower-right corner (3 in the matrix). This change effectively ignores suffixes if the match they produce is less optimal. In our example we can read out the most similar substrings substrings of avd and dave, which is av.

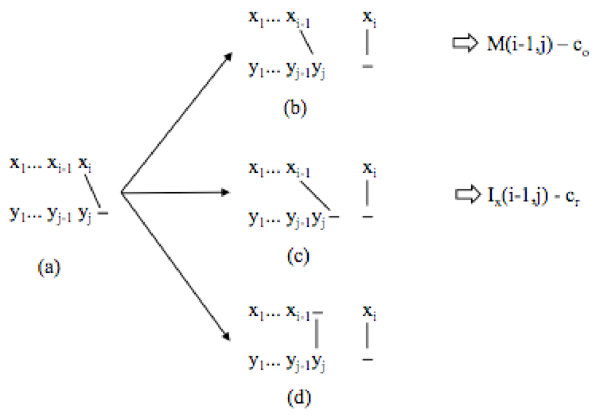


Figure 6.10: The derivation of the equation for $I_x(i, j)$ in Figure 6.7(b).

The Jaro measure

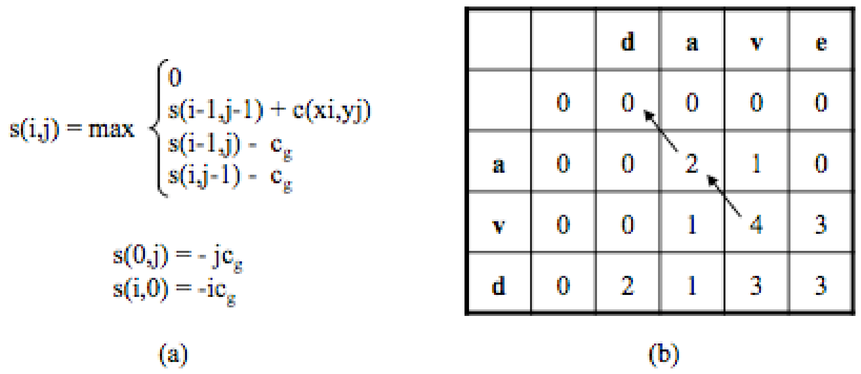
The Jaro measure was developed mainly to compare short strings, such as first and last names. Given two strings x and y, we compute their Jaro score as follows.

- Find the common characters x_i and y_j such that $x_i = y_j$ and $|i-j| \leq \min\{|x|, |y|\}/2$. Intuitively, common characters are those that are identical and are positionally "close to one another". It is not hard to see that the number of common characters x_i in x is the same as the number of common characters y_j in y. Let this number be c.
- Compare the i-th common character of x with the i-th common character of y. If they do not match, then we have a transposition. Let the number of transpositions be t.
- Compute the Jaro score as:

$$jaro(x, y) = 1/3[c/|x| + c/|y| + (c - t/2)/c].$$

As an example, consider $x =$ jon and $y =$ john. We have $c = 3$. The common character sequence in x is jon, and so is the sequence in y. Hence, there is no transposition, and $t = 0$. Thus, $jaro(x, y) = 1/3(3/3 + 3/4 + 3/3) = 0.917$. In contrast, the similarity score according to edit distance would be 0.75.

Now suppose $x =$ jon and $y =$ ojhn. Here the common character sequence in x is jon and the common character sequence in y is ojn. Thus $t = 2$, and $jaro(x, y) = 1/3(3/3 + 3/4 + (3 - 2/2)/3) = 0.81$.



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

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

(a)

		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

(b)

Figure 6.11: An example for the Smith-Waterman measure: (a) the recurrence equation for computing the similarity score using dynamic programming, and (b) the dynamic-programming matrix between avd and dave using the equation in Part a.

The cost of computing the Jaro distance is $O(|x||y|)$, due to the cost of finding common characters.

The Jaro-Winkler measure

The Jaro-Winkler measure tempers the Jaro measure by adding more weight to a common prefix. Specifically, the measure introduces two parameters: PL, which is the length of the longest common prefix between the two strengths, and PW, which is the weight to give the prefix. The measure is computed using the following formula:

$$jaro - winkler(x, y) = (1 - PL * PW) * jaro(x, y) + PL * PW.$$

6.2.2 Set-based similarity measures

The previous class of measures considered the strings as sequences of characters. We now describe similarity measures that view the strings as sets or multi-sets of tokens, and use set-related properties to compute similarity scores.

There are many ways to generate tokens from the input strings. A common method is to consider the words in the string (as delimited by space character) and possibly stem the words. Common stop words (e.g., the, and, of) are typically excluded. For example, given the string david smith we may generate the set of tokens {david, smith}.

Another common type of token is q-grams, which are substrings of length q that are present in the string. For example, the set of all 3-grams of david smith is {#da, dav, avi, ..., ith, th#}. Note that we have appended the special character # to the start and the end of the string, to handle 3-grams in these positions.

We discuss several set-based similarity measures. The bibliographic notes contain points to others that have been proposed in the literature. In our discussion we assume that each string has been converted into a *set* of tokens. We note however that these measures have also been considered for the multi-set case, and what we discuss below can be generalized to that case.

The overlap measure

Let B_x and B_y be the sets of tokens generated for strings x and y, respectively. The overlap measure returns the number of common tokens $O(x, y) = |B_x \cap B_y|$.

Consider $x =$ dave and $y =$ dav, then the set of all 2-grams of x is $B_x =$ {#d, da, av, ve, e#} and the set of all 2-grams of y is $B_y =$ {#d, da, av, v#}. So $O(x, y) = 3$. As such, the overlap measure computes a cost, which can be easily converted into a similarity score.

The Jaccard measure

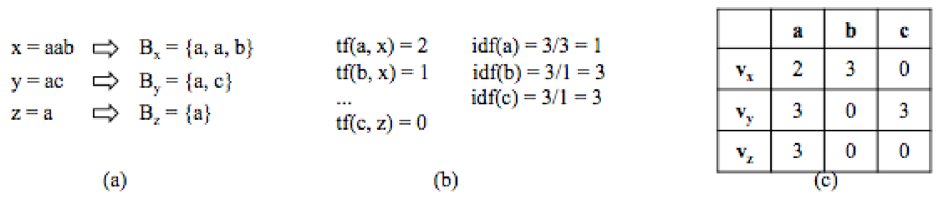
Continuing with the above notation, the Jaccard similarity score between two strings x and y is $J(x, y) = |B_x \cap B_y|/|B_x \cup B_y|$.

Again, consider $x =$ dave with $B_x =$ {#d, da, av, ve, e#}, and $y =$ dav with B_y = {#d, da, av, v#}. Then $J(x, y) = 3/6$.

The TF/IDF measure

This measure employs the notion of TF/IDF score commonly used in information retrieval (IR) to find documents that are relevant to keyword queries. The intuition underlying the TF/IDF measure is that two strings are similar if they contain common distinguishing terms. For example, consider the three strings $x =$ Apple Corporation, CA, $y =$ IBM Corporation, CA, and $z =$ Apple Corp. The edit distance and Jaccard measure would match x with y as $s(x, y)$ is higher than $s(x, z)$. However, the TF/IDF measure is able to recognize that Apple is a distinguishing term, whereas Corporation and CA are far more common, and thus would correctly match x with z.

When discussing the TF/IDF measure, we assume that the pair of strings being matched is taking from a bigger collection of strings. Figure 6.12(a) shows a tiny such collection of three strings $x =$ aab, $y =$ ac, and $z =$ a. We convert each string into a



	a	b	c
v$_x$	2	3	0
v$_y$	3	0	3
v$_z$	3	0	0

Figure 6.12: In the TF/IDF measure (a) strings are converted into bags of terms, (b) TF and IDF scores of the terms are computed, then (c) these scores are used to compute feature vectors.

bag of terms. Using IR terminology, we refer to such a bag of terms as a *document*. For example, we convert string $x = \mathsf{aab}$ into document $B_x = \{\mathsf{a}, \mathsf{a}, \mathsf{b}\}$.

We now compute for every term t and document d the *term frequency* (TD) and the *inverse document frequency* (IDF):

- the term frequency, $tf(t, d)$, is the number of times t occurs in d. For example, since a occurs twice in B_x, we have $tf(\mathsf{a}, x) = 2$.

- the inverse document frequency, $idf(t)$, is the total number of documents in the collection divided by the number of documents that contain t. For example, since a appears in all three documents in Figure 6.12(a), we have $idf(\mathsf{a}) = 3/3 = 1$. A higher value of IDF means that the occurrence of the term is more distinguishing.

Figure 6.12(b) shows the TF/IDF scores for the tiny example in Figure 6.12(a).

Next, we represent each document d into a feature vector v_d. The intuition is that two documents will be similar if their corresponding vectors are close to each other. The vector of d has a feature $v_d(t)$ for each term t, and the value of $v_d(t)$ is a function of the TF and IDF scores. Vector v_d thus has as many features as the number of terms in the collection. Figure 6.12(c) shows the three vectors v_x, v_y, and v_z for the three documents B_x, B_y, and B_z, respectively. In the figure, we took a relatively simple score: $v_d(t) = tf(t, d) \cdot idf(t)$. Thus, the score for feature a of v_x is $v_x(\mathsf{a}) = 2 \cdot 1 = 2$, and so on.

Now we are ready to compute the TF/IDF similarity score between any two strings p and q. Let T be the set of all terms in the collection. Then conceptually the vectors v_p and v_q (of the strings p and q) can be viewed as vectors in the $|T|$-dimensional space where each dimension corresponds to a term. The TF/IDF score between p

and q then can be computed as the cosine of the angle between these two vectors:

$$s(p,q) = \frac{\sum_{t \in T} v_p(t) \cdot v_q(t)}{\sqrt{\sum_{t \in T} v_p(t)^2} \cdot \sqrt{\sum_{t \in T} v_q(t)^2}}. \tag{6.1}$$

For example, the TF/IDF score between the two strings x and y in Figure 6.12(a) is $\frac{2 \cdot 3}{\sqrt{2^2+3^2} \cdot \sqrt{3^2+3^2}} = 0.39$, using the vectors v_x and v_y in Figure 6.12(c).

It is not difficult to see from Equation 6.1 that the TF/IDF similarity score between two strings p and q is high if they share many frequent terms (that is, terms with high TF scores), unless these terms also commonly appear in other strings in the collection (in which case the terms have low IDF scores). Using the IDF component, the TF/IDF similarity score can effectively discount the importance of such common terms.

In the above example, we assumed $v_d(t) = tf(t,d) \cdot idf(t)$. This means that if we "double" the number of occurrences of t in document d, then $v_d(t)$ will also double. In practice this is found to be excessive: doubling the number of occurrences of t should increase $v_d(t)$ but not double it. One way of addressing this is to "dampen" the TF and IDF components by a logarithmic factor. Specifically, we can take

$$v_d(t) = log(tf(t,d) + 1) \cdot log(idf(t)).$$

In addition, the vector v_d is often normalized to length 1, by setting

$$v_d(t) = v_d(t) / \sqrt{\sum_{t \in T} v_d(t)^2}.$$

This way, computing the TF/IDF similarity score $s(p,q)$ as in Equation 6.1 reduces to computing the dot product between the two normalized vectors v_p and v_q.

6.2.3 Hybrid similarity measures

We now describe several similarity measures that combine the benefits of sequence-based and set-based methods.

The Generalized Jaccard measure

Recall that the Jaccard mesaure considered the number of overlapping tokens in the input strings x and y. However, a token from x and a token from y had to be identical in order to be considered in the overlap set, which may be restrictive in some cases.

Consider an example of matching the names of nodes of two taxonomies describing divisions of companies. Each node is described by a string, such as Energy and Transportation and Transportation, Energy, and Gas. The Jaccard measure is a promising candidate for matching such strings, because intuitively two nodes are similar if their names share many tokens (e.g., energy and transportation). However, in practice tokens are often mispelt, such as energy vs. eneryg. The generalized Jaccard measure will enable matching in such cases.

As with the Jaccard measure, we begin by converting the string x into a set of tokens $B_x = \{x_1, x_2, \ldots, x_n\}$ and string y into a set of tokens $B_y = \{y_1, y_2, \ldots, y_m\}$. Figure 6.13(a) shows two such strings x and y, with $B_x = \{a, b, c\}$ and $B_y = \{p, q\}$.

The next three steps will determine the set of pairs of tokens that are considered in the "softened" overlap set. First, let s be a similarity measure that returns values in the range $[0, 1]$. We apply s to compute a similarity score for each pair $(x_i \in B_x, y_j \in B_y)$. Continuing with the above example, Figure 6.13(a) shows all six such scores.

Second, we we keep only those scores that equal or exceed a given threshold α. Figure 6.13(b) shows the remaining scores in our example, given $\alpha = 0.5$. The sets B_x and B_y, together with the edges that denote the remaining scores, form a bipartite graph G. In the third step we find the maximum-weight matching M in the bipartite graph G. Figure 6.13(c) shows this matching in our example. The total weight of this matching is 0.7 + 0.9 = 1.6.

Finally, we return the normalized weight of M as the generalized Jaccard score between x and y. To normalize, we divide the weight by the sum of the number of edges in M and the number of "unmatched" elements in B_x and B_y. This sum is $|M| + (|B_x| -| M|) + (|B_y| -| M|) = |B_x| + |B_y| -| M|$. Formally,

$$GJ(x, y) = \frac{\sum_{(x_i, y_j) \in M} s(x_i, y_j)}{|B_x| + |B_y| -| M|}.$$

Continuing with the above example, the generalized Jaccard score is $(0.7 + 0.9)/(3 + 2 - 2) = 0.53$.

The measure $GJ(x, y)$ is guaranteed to be between 0 and 1. It is a natural generalization of the Jaccard measure $J(x, y)$: if we constrain the elements of B_x and B_y to match only if they are identical, $GJ(x, y)$ reduces to $J(x, y)$. We discuss how to compute $GJ(x, y)$ efficiently later in Section 6.3.

The Soft TF/IDF similarity measure

This measure is similar in spirit to the generalized Jaccard measure, except that it uses the TF/IDF measure instead of the Jaccard measure as the "higher-level"

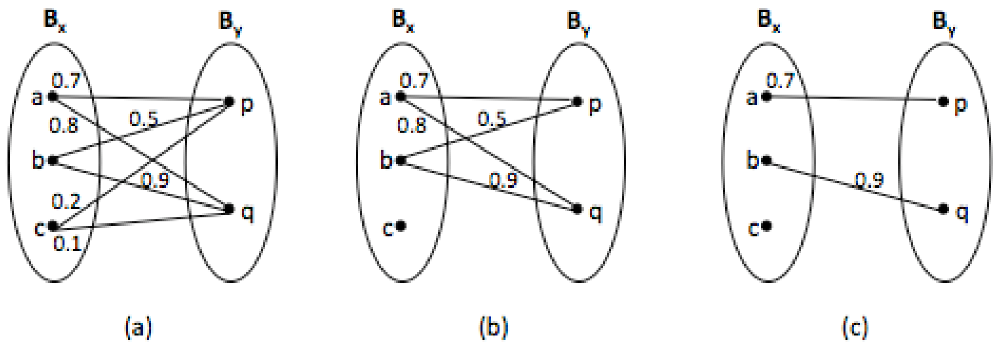


Figure 6.13: An example of computing the generalized Jaccard measure.

similarity measure.

Consider an example with the following three strings x = Apple Corporation, CA, y = IBM Corporation, CA, and z = Aple Corp. To match these strings, we would like to use the TF/IDF measure so that we can discount common terms such as Corporation and CA. Unfortunately in this case the TF/IDF measure does not help us match x with z, because the term Apple in x does not match the mispelt term Aple in z. Thus, x and z do not share any term. As with the generalized Jaccard measure, we would like to "soften" the requirement that Apple and Aple match exactly, and instead require that they be similar to each other.

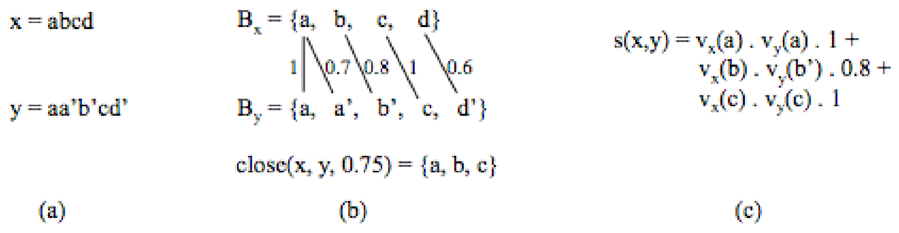


Figure 6.14: An example of computing soft TF/IDF similarity score for two strings x and y.

We compute the soft TF/IDF measure as follows. As with the TF/IDF measure, given two strings x and y, we create the two documents B_x and B_y. Figure 6.14(b) shows the documents created for the strings in Figure 6.14(a).

Next, we compute $close(x, y, k)$ to be the set of all terms in B_x that have at least one close term in B_y. Specifically, $close(x, y, k)$ is the set of terms $t \in B_x$ such that there exists a term $u \in B_y$ that satisfies $s'(t,u) \geq k$, where s' is a basic similarity measure (e.g., Jaro-Winkler) and k is a pre-specified threshold. Continuing with our example in Figure 6.14(b), suppose $s'(a,a) = 1, s'(a,a') = 0.7, \ldots$, as shown in the figure. Then $close(x,y,0.75) = \{a,b,c\}$. Note that $d \in B_x$ is excluded because the closest term to d in B_y is d', but d' is still too far from d (at a similarity score of 0.6).

In the final step, we compute $s(x,y)$ as in the traditional TF/IDF score, but giving a weight to each component of the TF/IDF formula according to the similarity score produced by s'. Specifically,

$$s(x,y) = \sum_{t \in close(x,y,k)} v_x(t) \cdot MaxTermInY(t) \cdot MaxTermScore(t),$$

where $MaxTermInY(t)$ is the term $u \in B_y$ that maximizes the score $s'(t,u)$ and $MaxTermScore(t)$ is this maximal score. Note that v_x and v_y are normalized to length 1 in this case, so that the traditional TF/IDF score can be computed by the dot product of the two vectors. Figure 6.14(c) shows how to compute $s(x,y)$ given the examples in Figures 6.14(a-b).

The Monge-Elkan similarity measure

The Monge-Elkan similarity measure can be effective for domains in which more control is needed over the similarity measure. To apply this measure to two strings x and y, first we break them into multiple substrings, say $x = A_1 \cdots A_n$ and $y = B_1 \cdots B_m$, where the A_i and B_j are substrings. Next, we compute

$$s(x,y) = \frac{1}{n} \sum_{i=1}^{n} \max_{j=1}^{m} s'(A_i, B_j),$$

where s' is a secondary similarity measure, such as Jaro-Winkler. To illustrate the above formula, suppose $x = A_1A_2$ and $y = B_1B_2B_3$. Then

$$s(x,y) = \frac{1}{2}[\max\{s'(A_1,B_1), s'(A_1,B_2), s'(A_1,B_3)\} + \max\{s'(A_2,B_1), s'(A_2,B_2), s'(A_2,B_3)\}].$$

Note that we ignore the order of the matching of the substrings and only consider the best match for the substrings of x in y. Furthermore, we can customize the secondary similarity measure s' to a particular application.

For example, consider matching strings x = Comput. Sci. and Eng. Dept., University of California, San Diego and y = Department of Computer Science, Univ. Calif., San Diego. To employ the Monge-Elkan measure, first we break x and y into substrings such as Comput., Sci., and Computer. Next we must design a secondary similarity measure s' that works well for such substrings. In particular, it is clear that s' must handle matching abbreviations well. For example, s' may decide that if a substring A_i is a prefix of a substring B_j, such as Comput. and Computer, then they match, that is, their similarity score is 1.

6.2.4 Phonetic similarity measures

The similarity measures we discussed so far match strings based on their *appearance*. In contrast, phonetic measures match strings based on their *sound*. These measures have been especially effective in matching names, since names are often spelled in different ways that sound the same. For example, Meyer, Meier and Mire sound the same, as do Smith, Smithe, and Smythe. We describe the Soundex similarity measure that is the most commonly used. We mention extensions of the basic Soundex measure in the bibliographic notes.

Soundex is used primarily to match surnames. It maps a surname x into a four-character code that captures the sound of the name. Two surnames are deemed similar if they share the same code. Mapping x to a code proceeds as follows. We use $x =$ Ashcraft as a running example in our description.

1. Keep the first letter of x as the first letter of the code. The first letter of the code for Ashcraft is A. The following steps are performed on the rest of the string x.

2. Remove all occurrences of W and H. Go over the remaining letters and replace them 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, and R with 6. Note that we do not replace the vowels A, E, I, O, U and Y. Continuing with our example, we convert Ashcraft into A226a13.

3. Replace each sequence of identical digits by the digit itself. So A226a13 becomes A26a13.

4. Drop all the non-digit letters (except the first one, of course). Then return the first four letters as the soundex code. So A26a13 becomes A2613, and the corresponding soundex code is A261.

Thus the soundex code is always a letter followed by three digits, padded by 0 if there are not enough digits. For example, the soundex code for Sue is S000.

As described, the soundex measure in effect "hashes" similar sounding consonants (such as B, F, P and V) into the same digit, thereby mapping similar sounding names into the same soundex code. For example, it maps both Robert and Rupert into R163.

Soundex is not perfect. For example, it fails to map the similar sounding surnames Gough and Goff, or Jawornicki and Yavornitzky (an Americanized spelling of the former), into the same code. Nevertheless, it is a useful tool that has been used widely to match and index names in applications such as census records, vital records, ship's passenger lists and geneology databases. While soundex was designed primarily for Caucasian surnames, it has been found to work well for names of many different origins (such as those appearing in the records of the US Immigration and Naturalization Services). However, it does not work as well for names of East Asian origins, because much of the discriminating power of these names resides in the vowel sounds, which the code ignores.

6.3 Scaling Up String Matching

Once we have selected a similarity measure $s(x, y)$, the next challenge is to match strings efficiently. Let X and Y be two sets of strings to be matched, and t be a similarity threshold. A naive matching solution would be as follows:

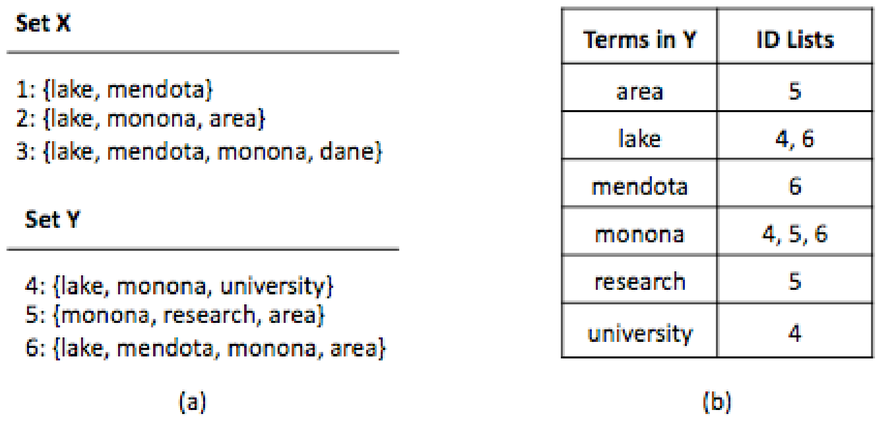
```
for each string $x \in X$
    for each string $y \in Y$
      if $s(x,y) \geq t$, return $(x,y)$ as a matched pair.
```

This $O(|X||Y|)$ solution is clearly impractical for large data sets. A more commonly employed solution is based on developing a method FindCands, that can quickly find the string that *may* match a given string x. Given such a method, we employ the following algorithm:

```
for each string $x \in X$
    use a method FindCands to find a candidate set $Z \subseteq Y$
    for each string $y \in Z$
      if $s(x,y) \geq t$, return $(x,y)$ as a matched pair.
```

This solution, often called a *blocking solution*, takes $O(|X||Z|)$ time, which is much faster than $O(|X||Y|)$ because FindCands is designed so that $|Z|$ is much smaller than $|Y|$. The set Z is often called the *umbrella set* of x. It should contain all true positives (i.e., all strings in Y that can possibly match x) and as few negative positives (i.e., those strings in Y that do not match x) as possible.

Clearly, the method FindCands lies at the heart of the above solution, and many techniques have been proposed for it. These techniques are typically based on indexing or filtering heuristics. We now discuss the basic ideas that underlie several common techniques for FindCands. In the following, we explain the techniques using the Jaccard and overlap measures. Later we discuss how to extend these techniques



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)

Figure 6.15: An example of using an inverted index to speed up string matching.

to other similarity measures.

Inverted index over strings

This technique first converts each string $y \in Y$ into a document $D(y)$, and then builds an inverted index I_Y over these documents. Given a term t, we can use I_Y to quickly find the list of documents created from Y that contain t, and hence the strings $y \in Y$ that contain t.

Figure 6.15(a) shows an example of matching the two sets X and Y. We can scan the set Y to build the inverted index I_Y shown in Figure 6.15(b). For instance, the index shows that term area appears in just document 5, but term lake appears in two documents 4 and 6.

Given a string $x \in X$, the method FindCands uses the inverted index to quickly locate the set of strings in Y that share at least one term with x. Continuing with the above example, given $x = \{$lake, mendota$\}$, we use the index in Figure 6.15(b) to find and merge the ID lists for lake and mendota, to obtain the umbrella set $Z = \{$4, 6$\}$.

This method is clearly much better than naively matching x with all strings in Y. Nevertheless, it still suffers from several limitations. First, the inverted list of some terms (e.g., stop words) can be very long, so building and manipulating such lists are quite costly. Second, this method still requires enumerating all pairs of strings that share at least one term. The set of such pairs can still be very large in practice. The techniques described below develop various ways to address these issues.

Size filtering

This technique retrieves only the strings in Y whose size make them match candidates. Specifically, given a string $x \in X$, we infer a constraint on the size of strings in Y that can possibly match x. The filter uses a B-tree index to retrieve only the strings that fit the size constraints.

To derive the constraint on the size of strings in Y, recall that the Jaccard measure is defined as follows (where $|x|$ refers to the number of tokens in x):

$$J(x, y) = |x \cap y|/|x \cup y|.$$

First, we can show that

$$1/J(x, y) \geq |y|/|x| \geq J(x, y). \tag{6.2}$$

To see why, consider the case where $|y| \geq| x|$. In this case, clearly $|y|/|x| \geq 1 \geq J(x, y)$. So we only have to prove $1/J(x, y) \geq |y|/|x|$, or equivalently that $|x \cup y|/|x \cap y| \geq |y|/|x|$. This inequality is true because $|x \cup y| \geq \max\{|x|, |y|\} = |y|$ and $|x \cap y| \leq \min\{|x|, |y|\} = |x|$. The case where $|y| \leq| x|$ can be proven similarly.

Now let t be the pre-specified similarity threshold. If x and y match, then it must be that $J(x, y) \geq t$. Together with Equation 6.2 this implies that $1/t \geq |y|/|x| \geq t$, or equivalently

$$|x|/t \geq |y| \geq |x| \cdot t \tag{6.3}$$

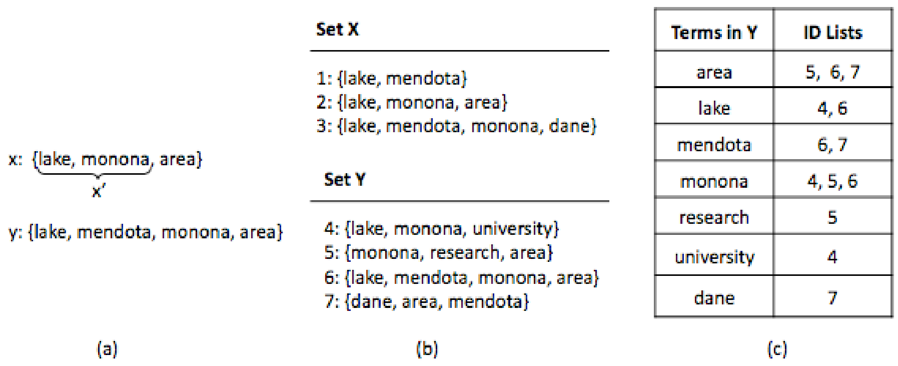
Thus, given a string $x \in X$, we know that only strings that satisfy Equation 6.3 can possibly match x.

To illustrate, consider again the string $x = \{$lake, mendota$\}$ (the first string in set X in Figure 6.15(a)). Suppose $t = 0.8$. Using the above equation, if $y \in Y$ matches x, we must have $2/0.8 = 2.5 \geq |y| \geq 2 \cdot 0.8 = 1.6$. We can immediately see that no string in the set Y in Figure 6.15(a) satisfies this constraint.

Exploiting the above idea, procedure FindCands builds a B-tree index over the sizes of strings in Y. Given a string $x \in X$, it then uses the index to find strings in Y that satisfy Equation 6.3 and returns that set of strings as the umbrella set Z. This technique is effective when there is significant variability in the number of tokens in the strings of X and Y.

Prefix filtering

The basic idea underlying the technique is that if two sets share many terms, then large subsets of them are also must share terms. Using this principle, we can reduce the number of candidate strings that may match a string x.



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}

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

Figure 6.16: An example of using prefix filtering to speed up string matching.

We first explain this technique using the overlap similarity measure and then extend it to the Jaccard measure. Suppose that x and y are strings that have an overlap of tokens $|x \cap y| \geq k$. Then it is easy to see that any subset $x' \subseteq x$ of size at least $|x| - (k-1)$ must overlap y. For example, consider the sets $x = \{$lake, monona, area$\}$ and $y = \{$lake, mendota, monona, area$\}$ in Figure 6.16(a). We have $|x \cap y| \geq 2$ Thus, the subset $x' = \{$lake, monona$\}$ in Figure 6.16(a) overlaps y (as does any other subset of size 2 of x).

We can exploit this idea in procedure FindCands as follows. Suppose we want to find all pairs (x, y) with overlap $O(x, y) \geq k$. Given a particular set x, we construct a subset x' of size $|x| - (k-1)$, and use an inverted index to find all sets y that overlap x'. Figures 6.16(b-c) illustrate this idea. Suppose we want to match strings in the sets X and Y of Figure 6.16(b), using $O(x, y) \geq 2$. We begin by building an inverted index over the strings in set Y, as shown in Figure 6.16(c). Next, given a string such as $x_1 = \{$lake, mendota$\}$, we take the "prefix" of size $|x_1| - 1$, which is $\{$lake$\}$ in this case, and let that be the set x'_1. We use the inverted index to find all strings in Y that contain at least a token in x'_1. This produces the set $\{y_4, y_6\}$ in this case. Note that if we use the inverted index to find all strings in Y that contain at least a token in x, we would end up with $\{y_4, y_6, y_7\}$, a bigger candidate set. Thus, restricting index lookup to just a subset of x can significantly reduce the resulting set size.

Selecting the subset intelligently: So far we arbitrarily selected the subset x' of x and checked its overlap with the entire set y. We can do even better by selecting a *particular subset* x' of x and checking its overlap with only a *particular subset* y' of y. Specifically, suppose we have imposed an ordering $\mathcal{O}$ over the universe of all possible terms. For example, we can order terms in increasing frequency, as computed over

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

(a)

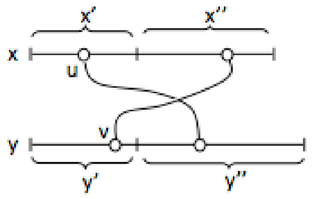
Reordered Set X

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)

Figure 6.17: We can build an inverted index over only the prefixes of strings in Y, then use this index to perform string matching.

the union of sets X and Y of Figure 6.16(b). Figure 6.17(a) shows all terms found in $X \cup Y$ in this order.

We reorder the terms in each set $x \in X$ and $y \in Y$ according to the order $\mathcal{O}$. Figure 6.17(b) shows the reordered sets X and Y. Given a reordered set x, we refer to the subset x' that contains the first n terms of x as the prefix of size n (or the n-th prefix) of x. Consider for example $x_3 = \{$dane, mendota, monona, lake$\}$. The 2nd prefix of x_3 is $\{$dane, mendota$\}$. Given the above, we can establish the following property:

Proposition 6.1: *Let x and y be two sets such that $|x \cap y| \geq k$. Let x' be the prefix of size $|x| - (k-1)$ of x, and let y' be the prefix of size $|y| - (k-1)$ of y. Then x' and y' overlap.* □

Proof: Let x'' be the *suffix* of size $(k-1)$ of x. Clearly $x' \cup x'' = x$. Similarly, let y'' be the suffix of size $(k-1)$ of y. Now suppose $x' \cap y' = \emptyset$. Then we must have $x' \cap y'' \neq \emptyset$ (otherwise $|x \cap y| \geq k$ does not hold). So there exists an element u such that $u \in x'$ and $u \in y''$. Similarly, there exists an element v such that $v \in y'$ and $v \in x''$. Since $u \in x'$ and $v \in x''$, we have $u < v$ in the ordering $\mathcal{O}$. But since $v \in y'$ and $u \in y''$, we also have $v < u$ in the ordering $\mathcal{O}$, a clear contradiction. Thus, x' overlaps y'.□

Given the above property, we can revise procedure FindCands as follows. Suppose again that we consider overlap of at least k $(O(x, y) \geq k)$.

- We reorder the terms in each set $x \in X$ and $y \in Y$ in increasing order of their frequency (as shown in Figure 6.17(a)).
- For each $y \in Y$, we create y', the prefix of size $|y| - (k-1)$ of y.
- We build an inverted index over all the prefixes y'. Figure 6.18(a) shows this index for our example.
- For each $x \in X$, we create x', the prefix of size $|x| - (k-1)$ of x, then use the above inverted index to find all sets $y \in Y$ such that x' overlaps with y'.

Consider the example $x = \{$mendota, lake$\}$, and therefore $x' = \{$mendota$\}$. Using mendota to look up the inverted index in Figure 6.18(a) yields y_6. Thus, FindCands will return y_6 as the sole candidate that may match x. Note that if we check the overlap between x' and the entire y, then y_7 is also returned. Thus, checking the overlap between prefixes can reduce the size of the resulting set. In practice, this reduction can be quite significant.

It is also important to note that the size of the inverted index is much smaller. For comparison purposes, Figure 6.18(b) shows the inverted index for the entire sets $y \in Y$ (reproduced from Figure 6.16(c)). The index we create here does not contain an entry for the term lake, and its index list for mendota is also smaller than the same index list in the entire-sets index.

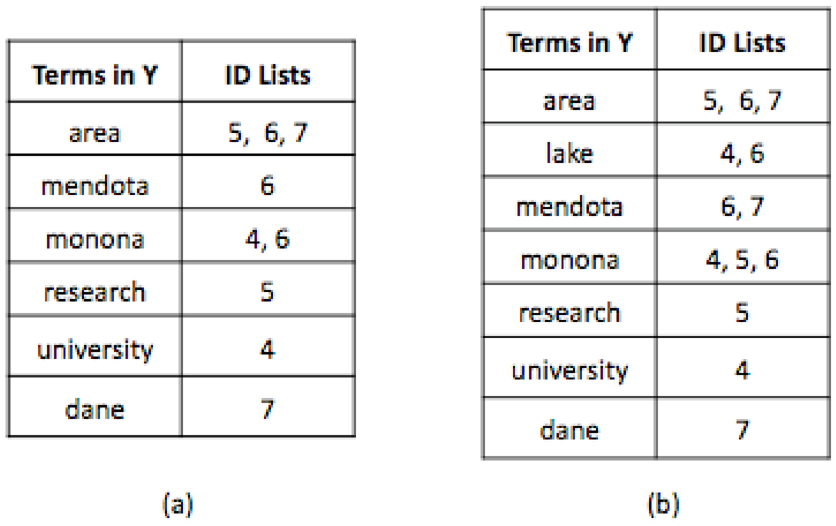
Applying prefix filtering to the Jaccard measure: The following equation enables us to extend the prefix filtering method to the Jaccard measure.

$$J(x,y) \geq t \Leftrightarrow O(x,y) \geq \alpha = \frac{t}{1+t} \cdot (|x| + |y|), \qquad (6.4)$$

The equation shows how to convert the Jaccard measure to the overlap measure, except for one detail. The threshold α as defined above is not a constant, and depends on $|x|$ and $|y|$. Thus, we cannot build the inverted index over the prefixes of $y \in Y$ using α. To address this, we index the "longest" prefixes. In particular, it can be shown that we only have to index the prefixes of length $|y| - \lceil t \cdot |y| \rceil + 1$ of the $y \in Y$, to ensure that we do not miss any correct matching pairs.

Position filtering

Position filtering further limits the set of candidate matches by deriving an upper bound on the size of the overlap between a pair of strings. As an example, consider the two strings $x = \{$dane, area, mendota, monona, lake$\}$ and $y = \{$research, dane, mendota, monona, lake$\}$. Suppose we are considering $J(x,y) \geq 0.8$. The prefix filtering we will



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)

Figure 6.18: The inverted indexes over (a) the prefixes of size $|y| - (k-1)$ of all $y \in Y$, and (b) all $y \in Y$. The former is often significantly smaller than the latter in practice.

index the prefix of length $|y| - \lceil t \cdot |y| \rceil + 1$ of y, which is $y' = \{\text{research, dane}\}$ in this case (because $5 - \lceil 5 \cdot 0.8 \rceil + 1 = 2$). Similarly, the prefix of length $|x| - \lceil t \cdot |x| \rceil + 1$ of x is $x' = \{\text{dane, area}\}$. Since x' overlaps y', in prefix filtering we will return the above pair (x, y) as a candidate pair.

However, we can do better than this. Let x'' be the rest of x, after x', and similarly let y'' be the rest of y, after y'. Then it is easy to see that

$$O(x, y) \leq |x' \cap y'| + min\{|x''|, |y''|\}. \tag{6.5}$$

Applying this inequality to the above example, we have $O(x, y) \leq 1 + min\{3, 3\} = 4$. However, using Equation 6.4 we have $O(x, y) \geq \frac{t}{1+t} \cdot (|x| + |y|) = \frac{0.8}{1+0.8} \cdot (5 + 5) = 4.44$. Hence, we can immediately discard the pair (x, y) from the set of candidate matches. More generally, position filtering combines the constraints from Equation 6.5 and Equation 6.4 to further reduce the set of candidate matches.

Bound filtering

Bound filtering is an optimization for computing the generalized Jaccard similarity measure. Recall from Section 6.2.3 that the generalized Jaccard measure computes the normalized weight of the maximum-weight matching M in the bipartite graph connecting x and y:

$$GJ(x, y) = \frac{\sum_{(x_i, y_j) \in M} s(x_i, y_j)}{|B_x| + |B_y| -| M|}.$$

In the equation, s is a secondary similarity measure, $B_x = \{x_1, x_2, \ldots, x_n\}$ is the set of tokens that corresponds to x, and $B_y = \{y_1, y_2, \ldots, y_m\}$ is the set that corresponds to y.

Computing $GJ(x, y)$ in a straightforward fashion would require computing the maximum-weight matching M in the bipartite graph, which can be very expensive. To address this problem, given a pair (x, y), we compute an upper bound, $UB(x, y)$, and lower bound, $LB(x, y)$, on $GJ(x, y)$. FindCands uses these bounds as follows: if $UB(x, y) \leq t$, then we can ignore (x, y) as it cannot be a match; if $LB(x, y) \geq t$, then we return (x, y) as a match. Otherwise, we compute $GJ(x, y)$.

The upper and lower bounds are computed as follows. First, for each element $x_i \in B_x$, we find an element $y_j \in Y$ with the highest element-level similarity, such that $s(x_i, y_j) \geq \alpha$ (recall that we consider only matches between $x_i \in B_x$ and $y_j \in B_y$ such that $s(x_i, y_j) \geq \alpha$). Let S_1 be the set of all such pairs.

For example, consider the two strings x and y together with the similarity scores between their elements in Figure 6.19(a) (reproduced from Figure 6.13(a)). Figure 6.19(b) shows the set $S_1 = \{(a, q), (b, q)\}$. Note that for element $c \in B_x$, there is no element in B_y such that the similarity score between them equals or exceeds α, which is 0.5 in this case.

Similarity, for each element $y_j \in B_y$, we find an element $x_i \in X$ with the highest element-level similarity, such that $s(x_i, y_j) \geq \alpha$. Let S_2 be the set of all such pairs. Continuing with our example, Figure 6.19(c) shows $S_2 = \{(a, p), (b, q)\}$.

The upper bound for $GJ(x, y)$ is given by the following formula:

$$UB(x, y) = \frac{\sum_{(x_i, y_j) \in S_1 \cup S_2} s(x_i, y_j)}{|B_x| + |B_y| -| S_1 \cup S_2|}$$

Note that the numerator of $UB(x, y)$ is at least as large as that of $GJ(x, y)$, and that the denominator of $UB(x, y)$ is no larger than that of $GJ(x, y)$. The lower bound is given by the following formula:

$$LB(x, y) = \frac{\sum_{(x_i, y_j) \in S_1 \cap S_2} s(x_i, y_j)}{|B_x| + |B_y| -| S_1 \cap S_2|}$$

Continuing with our example, $UB(x, y) = \frac{0.8+0.9+0.7+0.9}{3+2-3} = 1.65$ and $LB(x, y) = \frac{0.9}{3+2-1} = 0.225$.

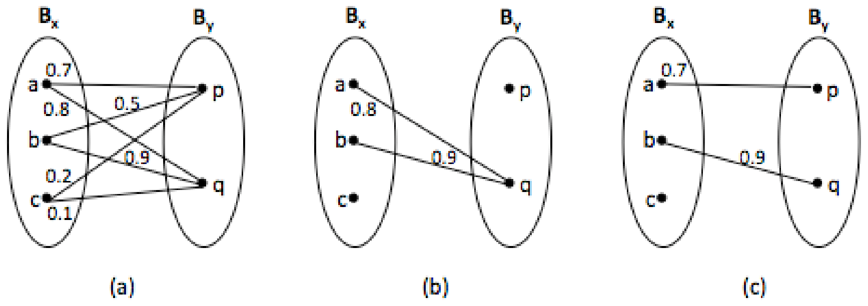


Figure 6.19: An example of computing an upper and lower bound for the generalized Jaccard measure.

Extending Scaling Techniques to other Similarity Measures

So far we have discussed scaling techniques for the Jaccard measure or overlap measure. We now describe how to extend these techniques to work with multiple similarity measures. First, as noted earlier, we can easily prove that

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

by replacing $|x \cup y|$ in $J(x, y)$ with $|x| + |y| - |x \cap y|$. Thus, if a technique works for the overlap measure $O(x, y)$, there is a good chance that we can also extend it to work for the Jaccard measure $J(x, y)$, and vice versa. For example, earlier we described how to extend the prefix filtering technique was originally developed for the overlap measure to the Jaccard measure.

In general, a promising way to extend a technique T to work for a similarity measure $s(x, y)$ is to translate $s(x, y)$ into constraints on a similarity measure that already works well with T. For example, consider edit distance. Let $d(x, y)$ be the edit distance between x and y, and let B_x and B_y be the corresponding q-gram sets of x and y, respectively. Then we can show that

$$d(x, y) \leq \epsilon \Rightarrow O(x, y) \geq \alpha = (\max\{|B_x|, |B_y|\} + q - 1) - q\epsilon.$$

Given the above constraint, we can extend prefix filtering to work with edit distance by indexing the prefixes of size $q\epsilon + 1$.

As yet another example, consider the TF/IDF cosine similarity $C(x, y)$. We can show that

$$C(x, y) \geq t \Leftrightarrow O(x, y) \geq \lceil t \cdot \sqrt{|x||y|} \rceil.$$

Given this, we can extend prefix filtering to work with $C(x, y)$ by indexing the prefixes of size $|x| - \lceil t^2 |x| \rceil + 1$ (this can be further optimized to just indexing the prefixes of size $|x| - \lceil t|x| \rceil + 1$). Finally, the above constraints can also help us extend position filtering to work with edit distance and cosine similarity measures.

6.4 Bibliographic Notes

Durbin et al. [156] provide an excellent description of the various edit distance algorithms, together with HMM-based probabilistic interpretations of these algorithms. Further discussion of string similarity measures and string matching can be found in [113, 163, 353]. The Web site [93] describes numerous string similarity measures and provides open-source implementations.

Edit distance was introduced in [290]. The basic dynamic programming algorithm for computing edit distance is described in [353]. Variations of edit distance include Needleman-Wunsch [354], affine gap [436], Smith-Waterman [408], Jaro [259], and Jaro-Winkler [443].

The Jaccard measure was introduced in [258]. The notion of TF/IDF originated from the Information Retrieval community [320], and TF/IDF-based string similarity measures are discussed in [24, 95, 116, 209, 276]. Soft TF/IDF was introduced in [62, 115]. Generalized Jaccard was introduced in [366]. The Monge-Elkan hybrid similarity measure is introduced in [344].

Cohen et al. empirically compare the effectiveness of string similarity measures over a range of matching tasks [115] (see also [62]).

The Soundex measure was introduced in [391, 392]. Other phonetic similarity measures include New York State Identification and Intelligence System (NYSIIS) [414], Oxford Name Compression Algorithm (ONCA) [199], Metaphone [374], and Double Metaphone [375].

Building inverted indexes to scale up string matching was discussed in [399]. The technique of size filtering was discussed in [27]. Prefix indexes were introduced in [98]. Bayardo et al. [46] discuss how to combine these indexes with inverted indexes to further scale up string matching. On et al. [366] discuss bound filtering, and Xiao et al. [447] discuss position indexes.