7 Data Matching

Data matching is the problem of finding structured data items that describe the same real-world entity. In contrast to string matching (see Chapter 4), where we tried to decide whether a pair of strings refer to the same real-world entity, here the entity may be represented by a tuple in a database, an XML element, or a set of RDF triples. For example, we would like to determine whether the tuples (David Smith, 608-245-4367, Madison WI) and (D. M. Smith, 245-4367, Madison WI) refer to the same person.

The problem of data matching arises in many integration situations. In the simplest case, we may be merging multiple databases with identical schemas, but without a unique global ID, and want to decide which rows are duplicates. The problem is complicated when we need to join rows from sources that have different schemas. Data matching may also arise at query time. For example, a query may often imprecisely refer to a particular data item, e.g., a query asking for the phone number of a David Smith who lives in Madison. We need to employ data matching techniques to decide which tuples in the database match the query. In this chapter we begin by defining the problem of data matching and then describe the techniques for addressing it.

7.1 Problem Definition

We consider the following problem. Suppose we are given two relational tables X and Y. In some cases we will assume that X and Y have identical schemas, but in the general case they will not. We assume that each of the rows in X and Y describes some properties of an entity (e.g., person, book, movie). We say a tuple $x \in X$ *matches* a tuple $y \in Y$ if they refer to the same real-world entity. We call such a pair (x, y) a match. Our goal is to find *all* matches between X and Y. For example, given the two tables X and Y in Figures 7.1(a-b), whose tuples describe properties of people, specifically their names, phone numbers, cities, and states, we want to find the matches shown in Figure 7.1(c). The first match (x_1, y_1) states that (Dave Smith, (608) 395 9462, Madison, WI) and (David D. Smith, 395 9426, Madison, WI) refer to the same real-world person. Of course, while we consider the data matching problem in the context of relational data, it also arises in other data models.

The challenges of data matching are similar in spirit to those of string matching: how to match accurately and to scale the matching algorithms to large data sets. Matching tuples accurately is difficult due to variations in formatting conventions, use of abbreviations,

Principles of Data Integration, DOI: 10.1016/B978-0-12-416044-6.00007-7

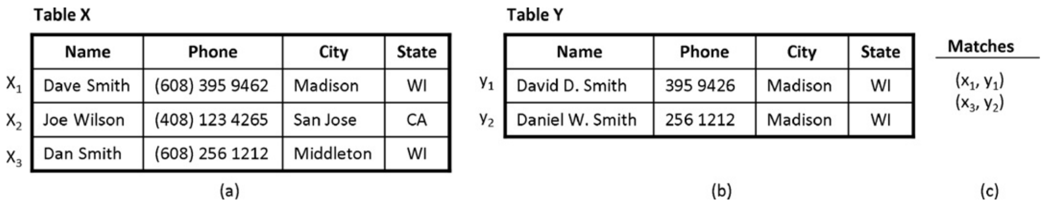


Table X

	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)

FIGURE 7.1 An example of matching relational tuples that describe persons.

shortening, different naming conventions, omission, nicknames, and errors in the data. We could, in principle, treat each tuple as a string by concatenating the fields, and then apply string matching techniques described in Chapter 4. While effective in certain cases, in general it is better to keep the fields apart, since more sophisticated techniques and domain-specific knowledge can then be applied to the problem. For example, when the entities are represented as tuples we can write a rule that states that two tuples match if the names and phones match exactly.

We cover several classes of solutions to the data matching problem. The first kind employs handcrafted rules to match tuples. These techniques typically make heavy use of domain-specific knowledge in domains where the complexity of the rules is manageable. The next kind of solution learns the appropriate rules from labeled examples, using supervised learning. The third kind, clustering does not use training data. Instead, it iteratively assigns tuples to clusters, such that all tuples within a single cluster match and those across clusters do not.

The fourth kind of solution, probabilistic approaches, models the matching domain using a probability distribution, and then reasons with the distribution to make matching decisions. As such, these approaches can naturally incorporate a wide variety of domain knowledge, leverage the wealth of probabilistic representation and reasoning techniques that have been developed in the past two decades, and provide a frame of reference for understanding other matching approaches.

The above approaches match tuple pairs independently. The last kind of approach we consider, known as collective matching, considers correlations among tuples to improve the accuracy of its matching decisions. For example, suppose that “David Smith” is a coauthor of “Mary Jones” and that “D. M. Smith” is a coauthor of “M. Jones.” Then if we have successfully matched “David Smith” with “D. M. Smith,” we should have increased confidence that “Mary Jones” matches “M. Jones.” Collective matching will propagate the result of one matching decision into others in an iterative fashion.

We cover the above matching techniques in the following sections and discuss scaling up in Section 7.7.

7.2 Rule-Based Matching

We begin by covering approaches that employ handcrafted matching rules. For this discussion we assume that we are matching tuples from two tables with the same schema, but generalizing to other contexts is straightforward. A simple yet popular type of rule computes the similarity score between a pair of tuples x and y as a *linearly weighted combination* of the individual similarity scores:

$$sim(x,y) = \sum_{i=1}^{n} \alpha_i \cdot sim_i(x,y) \tag{7.1}$$

where n is the number of attributes in each of the tables X and Y, $s_i(x,y) \in [0,1]$ is a similarity score between the ith attributes of x and y, and $\alpha_i \in [0,1]$ is a prespecified weight that indicates the importance of the ith attribute to the overall similarity score, such that $\sum_{i=1}^{n} \alpha_i = 1$. We declare x and y matched if $sim(x,y) \geq \beta$ for a prespecified threshold β, and not matched otherwise.

Consider matching the tuples of the tables X and Y in Figure 7.1 using a linearly weighted rule. We start by selecting similarity functions for name, phone, city, and state. To match names, we can define a similarity function $s_{name}(x,y)$ that is based on the Jaro-Winkler distance (see Chapter 4). To match phone numbers, we can define a function $s_{phone}(x,y)$ that is based on the edit distance between the phone number of x (after removing the area code because it does not appear in table Y) and the phone number of y. To match cities, we can again use edit distance. Finally, to match states, we can perform an exact match that returns 1 if the two state strings are equivalent and 0 otherwise. Now we can write a rule such as

$$sim(x,y) = 0.3s_{name}(x,y) + 0.3s_{phone}(x,y) + 0.1s_{city}(x,y) + 0.3s_{state}(x,y) \tag{7.2}$$

Intuitively, this rule states that in order to match, it is important that the names, phones, and states match, whereas it is less important that the cities match (perhaps because people often write down the name of the suburb where they live instead of the name of the city, and vice versa). We then select $\beta = 0.7$, say, and declare two persons matched if $sim(x,y) \geq 0.7$, and not matched otherwise.

Note that the decision of the matching system can be used in several ways in the user interface. For example, the system may still let a human judge inspect all matching pairs whose score is between 0.5 and 0.8, even if the threshold is 0.7.

LOGISTIC REGRESSION RULES

A linearly weighted rule has the property that an increase Δ in the value of any individual similarity function s_i will result in a *linear* increase $\alpha_i\Delta$ in the value of the overall similarity function s. This may seem counterintuitive in certain scenarios, where after a certain threshold an increase in s_i should count less (known as the principle of *diminishing*

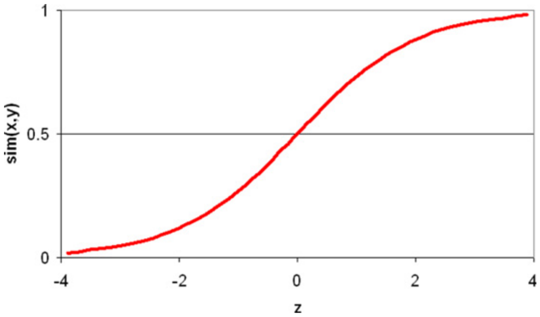


FIGURE 7.2 The general shape of the logistic regression (also known as sigmoid) function.

returns). For example, if $s_{name}(x,y)$ is already 0.95, then it may be the case that the names already very closely match. Thus, any further increase in $s_{name}(x,y)$ should contribute only minimally to $s(x,y)$.

Logistic regression matching rules try to capture this intuition. A logistic regression rule has the form

$$sim(x,y) = 1/(1+e^{-z}) \tag{7.3}$$

where $z = \sum_{i=1}^{n} \alpha_i \cdot sim_i(x,y)$. Here we do not constrain the α_i to be in the range $[0,1]$ and sum to 1. Thus we do not limit the value of z to the range $[0,1]$. Figure 7.2 shows that as z increases from $-\infty$ to $+\infty$, $sim(x,y)$ gradually increases, but minimally so after z has exceeded a certain value, achieving our goal of diminishing returns.

Logistic regression rules are also very useful in situations where firing (producing similarity scores close to 1) of a small proportion of the individual similarity functions is sufficient to achieve a high enough similarity score. Specifically, suppose we have a large number of individual component functions in the similarity score (e.g., 10–15), but only few of them fire for any instance. However, suppose the similarity is such that as long as a reasonable number of them produce high similarity, we are already confident that the two tuples match. In such cases, logistic regression rules ensure that after a reasonable number of individual functions have produced a high similarity score, we have achieved a sufficiently high overall similarity score and the remaining individual functions only contribute in a "diminishing return" fashion to this score.

MORE COMPLEX RULES

Both linearly weighted and logistic regression matching rules are relatively easy to construct and understand. But they do not work in cases where we want to encode more complex matching knowledge, such as two persons match if the names match approximately and *either* the phones match exactly *or* the addresses match exactly. In such cases we must create more complex matching rules, which can take any form and typically involve the individual similarity functions. For example, suppose $e_{phone}(x,y)$ returns true only if the phone numbers of x and y match exactly, and similarly for $e_{city}(x,y)$

and $e_{state}(x,y)$. Then we can encode the above matching knowledge using the following rules:

1. If $s_{name}(x,y) < 0.8$ then return "not matched."
2. Otherwise, if $e_{phone}(x,y) = true$ then return "matched."
3. Otherwise, if $(e_{city}(x,y) = true) \wedge (e_{state}(x,y) = true)$ then return "matched."
4. Otherwise, return "not matched."

Many real-world data matching systems employ such rules, which are often written in a high-level declarative language to make them easier to understand, debug, modify, and maintain (compared to writing them in procedural languages such as Perl and Java).

While very useful, rule-based approaches can be difficult to use. This may be because it is labor intensive to write good matching rules with the appropriate weights, or it is not even clear how to write such rules because the domain is too complex. Learning-based approaches, which we discuss next, address these issues.

7.3 Learning-Based Matching

In this section we describe approaches that use supervised learning to automatically create matching rules from labeled examples. In the next section we will discuss clustering approaches, which are a form of unsupervised learning.

Informally, in supervised learning, we learn a *matching model* M from the training data, then apply M to match new tuple pairs. The training data takes the form

$$T = \{(x_1,y_1,l_1),(x_2,y_2,l_2),\ldots,(x_n,y_n,l_n)\}$$

where each triple (x_i,y_i,l_i) consists of a tuple pair (x_i,y_i) and a label l_i that is "yes" if x_i matches y_i and "no" otherwise.

Given the training data T, we define a set of *features* $f_1,f_2,\ldots,f_m$, each of which quantifies one aspect of the domain judged possibly relevant to matching the tuples. Specifically, each feature f_i is a function that takes a tuple pair (x,y) and produces a numeric, categorical, or binary value. We simply define all features that we think *may* be relevant to matching. The learning algorithm will use the training data to decide which features are actually relevant.

In the next step, we convert each training example (x_i,y_i,l_i) in the set T into a pair

$$(\langle f_1(x_i,y_i),f_2(x_i,y_i),\ldots,f_m(x_i,y_i)\rangle,c_i)$$

where $v_i = \langle f_1(x_i,y_i),\ f_2(x_i,y_i),\ldots,\ f_m(x_i,y_i)\rangle$ is a feature vector that "encodes" the tuple pair (x_i,y_i) in terms of the features, and c_i is an appropriately transformed version of label l_i (e.g., being transformed into "yes"/"no" or 1/0, depending on what kind of matching model we want to learn, as we detail below). Thus, the training set T is converted into a new training set $T' = \{(v_1,c_1),(v_2,c_2),\ldots,(v_n,c_n)\}$.

We can now apply a learning algorithm such as decision trees or support vector machines (SVM) to T' to learn a matching model M, then apply M to the task of matching new tuple pairs. Given a new pair (x,y), we transform it into a feature vector

$$v = \langle f_1(x,y), f_2(x,y), \ldots, f_m(x,y) \rangle$$

then apply model M to v to predict whether x matches y.

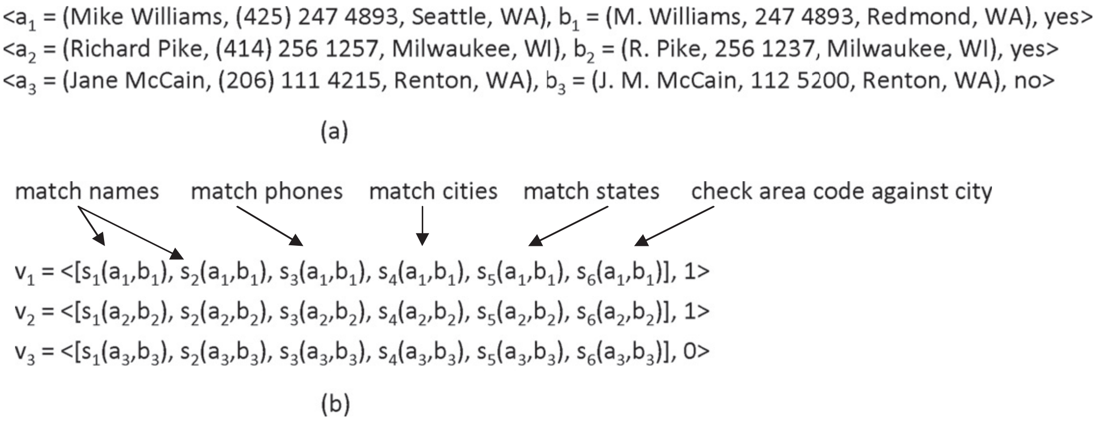
Example 7.1

Consider applying the above algorithm to learning a linearly weighted rule to match the tables X and Y in Figure 7.1 (learning a logistic regression rule can be carried out in a similar fashion). Suppose the training data consist of the three examples in Figure 7.3(a). In practice, a training set typically contains hundreds to tens of thousands of examples.

Suppose also that we have defined six possibly relevant features s_1–s_6. Features $s_1(a,b)$ and $s_2(a,b)$ compute similarity scores based on Jaro-Winkler and edit distance between the person names of tuples a and b, respectively. This is because we do not know *a priori* whether Jaro-Winkler or edit distance works best for this matching scenario, so we "throw" both in and let the learning algorithm decide.

Feature $s_3(a,b)$ computes an edit distance-based similarity measure between the phone numbers (ignoring the area code of a, as this area code does not appear in the phone number of b). Features $s_4(a,b)$ and $s_5(a,b)$ return 1 if the city names and the state names, respectively, match exactly, and 0 otherwise.

Finally, feature $s_6(a,b)$ encodes a heuristic constraint that we believe may be useful in this matching scenario. To explain, consider tuples a_1 and b_1 in Figure 7.3(a). The city of tuple a_1 is Seattle, whereas the city of b_1 is Redmond, an incompatibility. However, the phone number


<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>

(a)

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>

(b)

FIGURE 7.3 (a) The original training data; (b) the transformed training data for learning linearly weighted and logistic regression rules.

425 247 4893 of a_1 has area code 425, which is actually Redmond's area code. Furthermore, the names and phone numbers (without area code) match quite closely. This suggests that a_1 and b_1 may be the same person and that a_1 lives in Redmond, but lists his or her city as the nearby city of Seattle instead. To capture such scenarios, we define feature $s_6(a,b)$ to return 1 if the area code of a is an area code of the city of b, and return 0 otherwise. As we explained earlier, we leave it up to the learning algorithm to decide if this feature can be useful for matching purposes.

After defining the features s_1-s_6, we transform the training examples in Figure 7.3(a) into those in Figure 7.3(b). Here each feature vector v_1-v_3 has six values, as returned by the feature functions. Furthermore, since our goal is to learn a linearly weighted similarity function $s(a,b)$ that returns a *numeric* similarity score between a and b, we have to convert the labels "yes"/"no" into numeric similarity scores. In this case, we simply convert "yes" into 1 and "no" into 0, as shown on the right-hand side of Figure 7.3(b).

Now we are ready to learn from the training set in Figure 7.3(b). Our goal is to learn the weight α_i, $i \in [1,6]$ that would give us a linearly weighted matching rule of the form $s(a,b) = \sum_{i=1}^{6} \alpha_i s_i(a,b)$. To do this, we perform a least-squares linear regression on the data set in Figure 7.3(b). This regression finds the weights α_i that minimize the squared error $\sum_{i=1}^{3}(c_i - \sum_{j=1}^{6} \alpha_j s_j(v_i))^2$, where c_i is the label associated with feature vector v_i (this label is 0 or 1 in this case), and $s_j(v_i)$ is the value of the jth element of feature vector v_i.

All that remains from the learning process is to learn a threshold β, so that given a new tuple pair (a,b), we can predict matched if $s(a,b) \geq \beta$ and not matched otherwise. We can also learn β from the training data, for example, by setting β at the value that lets us minimize the number of incorrect matching predictions.

Example 7.2

As another example to illustrate supervised learning algorithms, consider applying a decision tree learning algorithm to the training examples in Figure 7.3(a). To do this, we first convert these training examples into those in Figure 7.4(a), assuming the same features s_1-s_6. Note that now the labels remain "yes"/"no," because our goal is to learn a decision tree that predicts whether a tuple pair match (rather than learning a function that produces numeric similarity scores, as in the case of linearly weighted and logistic regression rules). Then, after applying a standard decision tree algorithm (such as the well-known C4.5 algorithm) to this training data, we may learn a decision tree such as the one in Figure 7.4(b). This tree states that a tuple pair (a,b) match if the names and the phone numbers match closely, or if the names match closely, the phones do not match very well, but the cities and the states match closely. Note that the tree does not use $s_2(a,b)$ and $s_6(a,b)$. Using the training data, the learning algorithm has decided that these features are irrelevant to making matching decisions.

As described, supervised learning approaches have two major advantages compared to rule-based ones. First, in rule-based approaches we have to *manually* decide if a particular feature is useful for the matching process, a difficult and time-consuming decision.

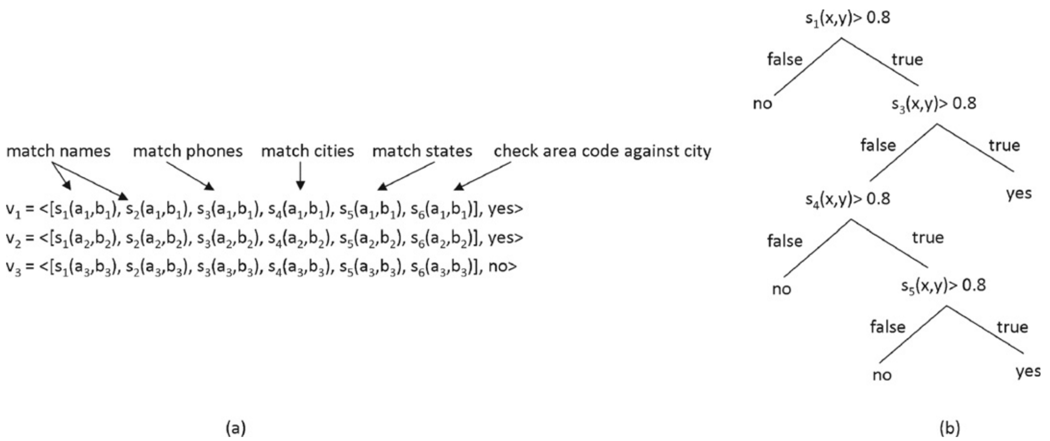

match names
match phones
match cities
match states
check area code against city
v1 = <[s1(a1,b1), s2(a1,b1), s3(a1,b1), s4(a1,b1), s5(a1,b1), s6(a1,b1)], yes>
v2 = <[s1(a2,b2), s2(a2,b2), s3(a2,b2), s4(a2,b2), s5(a2,b2), s6(a2,b2)], yes>
v3 = <[s1(a3,b3), s2(a3,b3), s3(a3,b3), s4(a3,b3), s5(a3,b3), s6(a3,b3)], no>
(a)
s1(x,y)> 0.8
false
true
no
s3(x,y)> 0.8
false
true
s4(x,y)> 0.8
yes
false
true
no
s5(x,y)> 0.8
false
true
no
yes
(b)

FIGURE 7.4 (a) The training data for the decision tree learning algorithm and (b) a sample decision tree learned by the algorithm.

This also limits us to examining only a relatively small set of features. In contrast, learning approaches can *automatically* examine a large set of features to select the most useful ones. The second advantage is that learning approaches can construct very complex “rules” (such as those produced by decision tree and SVM algorithms) that are difficult to construct by hand in rule-based approaches.

Supervised learning approaches, however, often require a large number of training examples, which can be hard to obtain. In the next section we describe unsupervised learning approaches that address this problem.

7.4 Matching by Clustering

In this section we describe the application of clustering techniques to the problem of data matching. Many of the commonly known clustering techniques have been applied to data matching, including agglomerative hierarchical clustering, k-means, and graph-theoretic clustering. Here we focus on agglomerative hierarchical clustering (AHC), a relatively simple yet very commonly used method. We show how AHC works, then use it to illustrate the key ideas that underlie clustering-based approaches to data matching.

Given a set of tuples *D*, the goal of AHC is to partition *D* into a set of clusters, such that all tuples in each cluster refer to a single real-world entity, while tuples in different clusters refer to different entities. AHC begins by putting each tuple in *D* into a single cluster. Then it iteratively merges the two most similar clusters, where similarity is computed using a similarity function. This continues until a desired number of clusters has been reached or until the similarity between the two closest clusters falls below a prespecified threshold. The output of AHC is the set of clusters when the termination condition is reached.

Example 7.3

To illustrate, suppose we want to cluster the six tuples x_1–x_6 shown in the bottom of Figure 7.5(a). We begin by defining a similarity function $s(x,y)$ between the tuples. For instance, if the tuples describe persons and have the same format as those in Figures 7.1(a-b), then we can use the linearly weighted function in Equation 7.2, reproduced here:

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

In iteration 1 we compute $sim(x,y)$ for all tuple pairs and merge the two most similar tuples, which are x_1 and x_2. They have been merged into cluster c_1. In iteration 2 we merge x_4 and x_5 into cluster c_2, and in iteration 3 we merge x_3 and c_2 into cluster c_3. We now have three clusters: c_1, c_3, and x_6 (which is its own cluster), as shown in Figure 7.5(a).

At this point, suppose that no two clusters have a similarity score exceeding β, a prespecified threshold, so we stop and return c_1, c_3, and x_6. This clustering effectively specifies that x_1 and x_2 match, any two tuples of x_3, x_4, and x_5 match, while x_6 does not match any other tuple. Figure 7.5(b) visually depicts the bottom-up merging process and illustrates the "hierarchical" nature of AHC.

To complete the description of the clustering approach we need to explain how to compute a similarity score between two clusters. Methods typically used in practice include the following:

Single link: The similarity score between clusters c and d is the minimal score between all tuple pairs $x_i \in c$ and $y_j \in d$: $s(c,d) = \min_{x_i \in c, y_j \in d} sim(x_i, y_j)$.

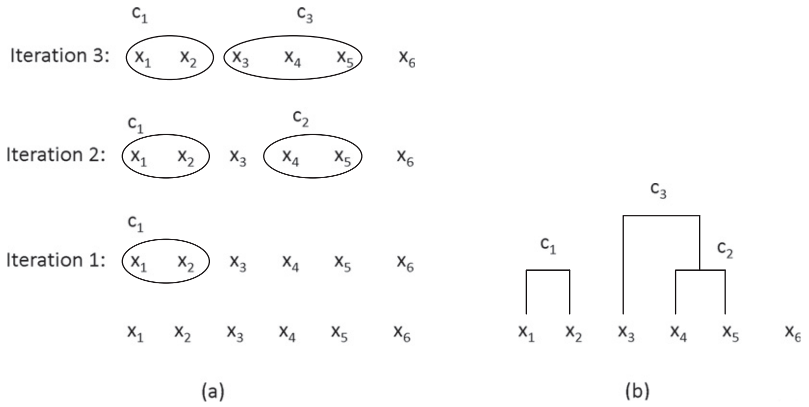


FIGURE 7.5 An example to illustrate agglomerative hierarchical clustering.

Complete link: We compute the maximal score between pairs of tuples in c and d: $s(c,d) = \max_{x_i \in c, y_j \in d} sim(x_i, y_j)$.

Average link: We compute $s(c,d) = \sum_{x_i \in c, y_j \in d} sim(x_i, y_j)/n$, where n is the total number of pairs $x_i \in c$ and $y_j \in d$.

Canonical tuple: We create a canonical tuple that represents each cluster. The similarity between a pair of clusters is defined to be the similarity between their canonical tuples. The canonical tuple is created from the attribute values of the tuples in the cluster. For example, if the names in cluster c are "Mike Williams" and "M. J. Williams," we may merge them to generate the canonical name "Mike J. Williams." If the phone numbers are "425 247 4893" and "247 4893," we may select the longer number "425 247 4893" to be the canonical phone number, and so on.

In general, selecting a good cluster similarity method is application dependent and requires careful consideration from the application builder.

Before continuing to other approaches, it is important to emphasize the new perspectives that the clustering approach introduces to the data matching problem:

1. We view matching tuples as the problem of *constructing entities* (that is, clusters), with the understanding that only tuples within a cluster match.
2. The process is *iterative*: in each iteration we leverage what we have known so far (in the previous iterations) to build "better" entities.
3. In each iteration we *"merge"* all matching tuples within each cluster to construct an "entity profile," then use this profile to match other tuples. This is most clearly seen in the case of generating a canonical tuple, which can be viewed as an entity profile. As such, clustering introduces the novel aspect of merging then exploiting the merged information to help matching.

We will see the same principles appear in other approaches as well.

7.5 Probabilistic Approaches to Data Matching

Probabilistic approaches to data matching model the matching decision as a set of variables over which there is a probability distribution. For example, there would be a variable for whether two person names match and a variable for whether two tuples match. These approaches make matching decisions by reasoning about these variables.

The key benefit of probabilistic approaches to data matching is that they provide a principled framework that can naturally incorporate a wide variety of domain knowledge. In addition, these methods can leverage the wealth of probabilistic representation and reasoning techniques that have been developed in the past two decades in the artificial intelligence and database communities. Finally, probabilistic methods provide a frame of reference for comparing and explaining other matching approaches.

Probabilistic approaches suffer from certain disadvantages. Chief among these is that they often take far longer to run than nonprobabilistic counterparts. Another disadvantage is that it is often hard to understand and debug matching decisions of probabilistic approaches, even though the ability to do so is crucial in real-world applications.

Currently, most probabilistic approaches employ *generative models*, which encode full probability distributions and describe how to generate data that fit the distributions. Consequently, we will focus on these approaches and describe in particular three approaches that employ increasingly sophisticated generative models. In what follows we explain Bayesian networks, a relatively simple type of generative model, then use the ideas underlying Bayesian networks to explain the three approaches.

7.5.1 Bayesian Networks

We begin by describing Baysian networks, a probabilistic reasoning framework on which several of the matching approaches are based. Let $X = \{x_1, \ldots, x_n\}$ be a set of *variables*, each of which models a quantity of interest in the application domain and takes values from a prespecified set. For example, the variable *Cloud* can be *true* or *false*, and *Sprinkler* can be *on* or *off*. In our discussion we consider only variables with discrete values, but continuous variables can also be modeled (see Section 7.7.2). We define a *state* to be an assignment of values to all variables in X. For example, given $X = \{Cloud, Sprinkler\}$, we have four states, one of which is $s = \{Cloud = true, Sprinkler = on\}$.

Let S be the set of all states. A *probability distribution* P is a function that assigns to each state $s_i \in S$ a value $P(s_i) \in [0, 1]$ such that $\sum_{s_i \in S} P(s_i) = 1$. We call $P(s_i)$ the *probability* of s_i. The table in Figure 7.6 specifies a probability distribution over the states of *Cloud* and *Sprinkler* (here t and f are shorthands for *true* and *false*, respectively).

Our goal in reasoning over probabilistic models is to answer queries such as "what is the probability of $A = a$, $P(A = a)$", or "what is the probability of $A = a$ given that we know that the value of B is b, $P(A = a|B = b)$"? In both of these queries, A and B are subsets of the variables in the model. We typically abbreviate the queries as $P(A)$ and $P(A|B)$.

Probability theory gives us precise semantics for such queries. For example, to compute $P(Cloud = t)$ we simply sum over the first two rows of the table in Figure 7.6 to obtain 0.6.

States		**Probabilities**
Cloud	*Sprinkler*	
t	*on*	0.3
t	*off*	0.3
f	*on*	0.3
f	*off*	0.1

FIGURE 7.6 An example of a probability distribution over four states.

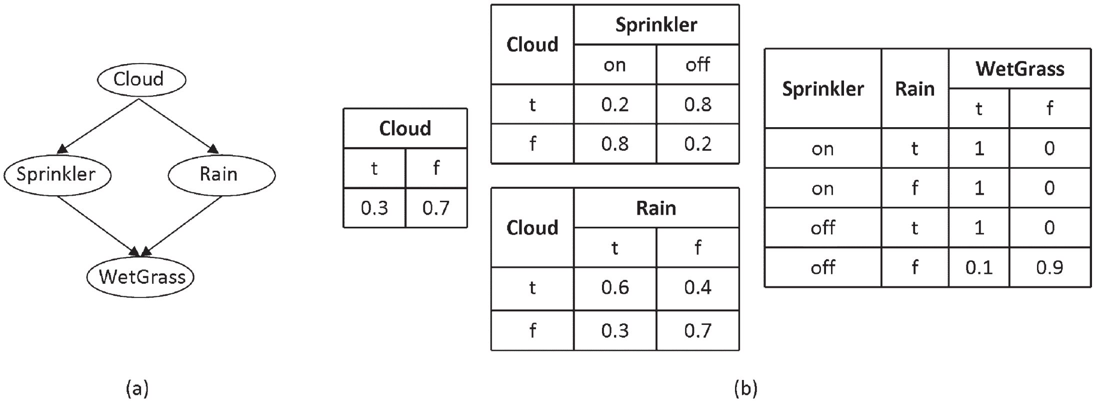
To compute $P(Cloud = t|Sprinkler = off)$, we first compute $P(Cloud = t, Sprinkler = off)$ and then divide it by $P(Sprinkler = off)$ to obtain 0.75.

Unfortunately, it is often impractical to specify a probability distribution by explicitly enumerating *all* states, as we do in Figure 7.6. This is because the number of possible states grows exponentially with the number of variables. Real-world applications often use hundreds or thousands of variables, and thus we cannot enumerate all states. The key idea underlying Bayesian networks is that they provide a compact representation of a probability distribution. We now explain how to represent and reason with Bayesian networks and how to learn them from data.

Representing and Reasoning with Bayesian Networks

A *Bayesian network* is an acyclic directed graph whose nodes denote the variables and whose edges encode probabilistic dependencies among the variables. Figure 7.7(a) shows such a graph over four nodes *Cloud*, *Sprinkler*, *Rain*, and *WetGrass*. Given a Bayesian network G, we say that node A is a parent of node B if there is a directed edge from A to B, and that node C is an ancestor of B if there is a directed path from C to B.

Informally, an edge from node A to node B can be interpreted as stating that B is dependent on A. For example, wet grass is dependent on whether the sprinkler is on and on whether it is raining. The graph G then encodes the following assertion: each node in G is probabilistically independent of its nondescendants given the values of its parents. This assertion is the key for compactly representing a probability distribution using a Bayesian network. For example, in Figure 7.7(a) the value of *WetGrass* is independent of *Cloud* given the values of *Sprinkler* and *Rain*, and the value of *Sprinkler* is independent of *Rain* given the value of *Cloud*. Given the above independence assertion, we can compute the probability distribution specified by G as a product of "local probabilities."



Cloud	
t	f
0.3	0.7

Cloud	Sprinkler	
	on	off
t	0.2	0.8
f	0.8	0.2

Cloud	Rain	
	t	f
t	0.6	0.4
f	0.3	0.7

Sprinkler	Rain	WetGrass	
		t	f
on	t	1	0
on	f	1	0
off	t	1	0
off	f	0.1	0.9

FIGURE 7.7 A sample Bayesian network: (a) the directed acyclic graph and (b) the conditional probability tables (CPTs).

Example 7.4

Consider the graph in Figure 7.7(a). Let C, S, R, and W be shorthands for *Cloud*, *Sprinkler*, *Rain*, and *WetGrass*, respectively, and let a term such as $P(C, S, R, W)$ denote the probability of a state on these four variables, such as $P(C = t, S = on, R = f, W = t)$. Then using the chain rule of probability we have

$$P(C, S, R, W) = P(C) \cdot P(S|C) \cdot P(R|S, C) \cdot P(W|R, S, C)$$

Since a node is probabilistically independent of its nondescendants given its parents, we can simplify $P(R|S, C)$ as $P(R|C)$, and $P(W|R, S, C)$ as $P(W|R, S)$. This gives

$$P(C, S, R, W) = P(C) \cdot P(S|C) \cdot P(R|C) \cdot P(W|R, S)$$

Thus, to compute $P(C, S, R, W)$, we only need to know four *local distributions*: $P(C)$, $P(S|C)$, $P(R|C)$, and $P(W|R, S)$. Figure 7.7(b) shows an example of these local distributions, where $P(C = t) = 0.3$ and $P(S = on|C = t) = 0.2$.

In general, local distributions specify conditional probabilities of a node given its parents and are called *conditional probability tables*, or *CPT*s for short, one CPT per node. Note that to specify a CPT, we need to specify only half of the probabilities in the CPT, the other half can be easily inferred. For example, to specify the CPT of *Cloud*, we only have to specify $P(C = t)$, then infer $P(C = f) = 1 - P(C = t)$.

As described, given a graph and a corresponding set of CPTs, a Bayesian network fully specifies a probability distribution, and does so compactly. For example, the Bayesian network in Figures 7.7(a-b) uses only nine probability statements to specify the CPTs, whereas an explicit representation that enumerates all states would use $2^4 = 16$ statements. In general, a Bayesian network can take exponentially less space than the explicit representation of the probability distribution.

Performing inference with a Bayesian network means computing probabilities such as $P(A)$ or $P(A|B)$ where A and B are subsets of variables. In general, performing exact inference is NP-hard, taking exponential time in the number of variables of the network in the worst case. Data matching approaches address this problem in three ways. First, under certain restrictions on the structure of the Bayesian network, inference is computationally cheaper, and there are polynomial-time algorithms or closed-form equations that will return the exact answers. Second, data matching algorithms can use standard approximate inference algorithms for Bayesian networks that have been described in the literature. Finally, we can develop approximation algorithms that are tailored to a particular domain at hand. We illustrate these inference solutions in Sections 7.5.2–7.5.4, when we describe three representative probabilistic matching approaches.

Learning Bayesian Networks

To use a Bayesian network, current data matching approaches typically require a domain expert to create a directed acyclic graph (such as the one in Figure 7.7(a)), then learn the associated CPTs (e.g., Figure 7.7(b)) from the *training data*. The training data consist of a set of states that we have observed in the world. For example, suppose that on Monday we observed that it was cloudy and raining, that the sprinkler was off, and that the grass was wet; then the tuple $d_1 = (Cloud = t, Sprinkler = off, Rain = t, WetGrass = t)$ forms an observed state of the world. Suppose we also observed $d_2 = (Cloud = t, Sprinkler = off, Rain = f, WetGrass = f)$ on Tuesday, and $d_3 = (Cloud = f, Sprinkler = on, Rain = f, WetGrass = t)$ on Wednesday. Then the tuples d_1–d_3 form a (tiny) set of training data, from which we can learn the CPTs of the graph in Figure 7.7(a). In practice, a training set typically contains hundreds to thousands of such training examples.

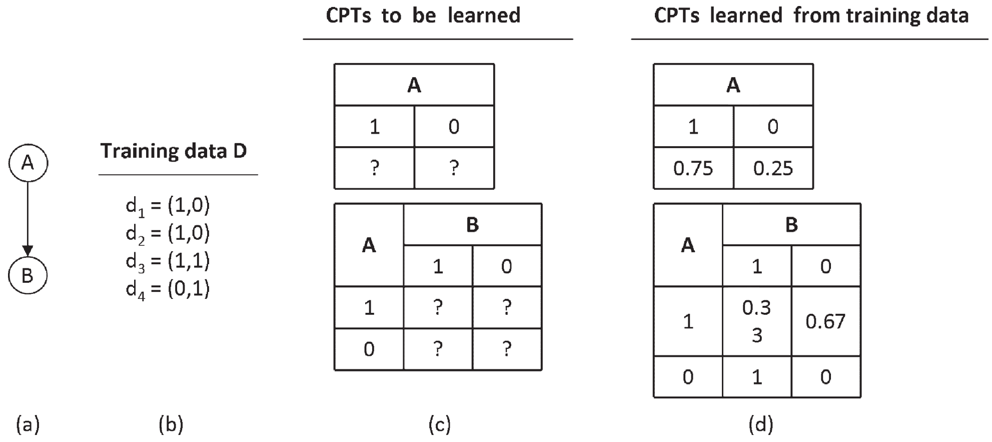
LEARNING WITH NO MISSING VALUES

We begin with the case in which the training examples do not have missing values. We illustrate the learning on the example in Figure 7.8. Given the graph in Figure 7.8(a) and the training data in Figure 7.8(b), we want to learn the CPTs in Figure 7.8(c). Note that a training example such as $d_1 = (1,0)$ specifies $A = 1$ and $B = 0$. Note also that we only have to learn probabilities $P(A = 1), P(B = 1|A = 1)$, and $P(B = 1|A = 0)$. From these we can easily infer the other probabilities of the CPTs.

Let θ denote the probabilities to be learned. In the absence of any other evidence, we want to find a $\theta*$ that maximizes the probability of observing the training data D:

$$\theta* = \arg\max_{\theta} P(D|\theta)$$

It turns out that $\theta*$ can be obtained by simple counting over the training data D. For example, to compute probability $P(A = 1)$, we simply count the number of training examples



Training data D

$d_1 = (1,0)$
$d_2 = (1,0)$
$d_3 = (1,1)$
$d_4 = (0,1)$

(b)

CPTs to be learned

A	
1	0
?	?

A	B	
	1	0
1	?	?
0	?	?

(c)

CPTs learned from training data

A	
1	0
0.75	0.25

A	B	
	1	0
1	0.33	0.67
0	1	0

(d)

FIGURE 7.8 An example of learning the CPTs using training data with no missing values.

where $A = 1$, then divide that number by the total number of examples. In our case, we get $P(A = 1) = 0.75$. To compute $P(B = 1|A = 1)$, we divide the number of examples where $B = 1$ and $A = 1$ by the number of examples where $A = 1$, and so on. Applying this counting procedure to the example in Figures 7.8(a-c), we obtain the CPTs in Figure 7.8(d).

The tricky case is where we do not have sufficient data for certain states. Suppose that the training data D include only the three examples $d_1 - d_3$. In this case it is not clear how to compute $P(B = 1|A = 0)$, because the number of examples where $A = 0$ is 0, and we cannot divide by 0.

To address this problem, we can use a method called *smoothing of the probabilities*. A common smoothing method is the m-estimate of probability, defined as follows. Let X and Y be two variables. Consider computing the m-estimate of a probability that involves both X and Y, say $P(Y = 1|X = 0)$. Suppose we count x examples where $X = 0$ and y examples where $Y = 1$ and $X = 0$; then instead of returning y/x as an estimation of this probability, as before, we return $(y + mp)/(x + m)$, where p is our prior estimate of the probability $P(Y = 1|X = 0)$ and m is a prespecified constant. In the absence of any other information, a common method for choosing p is to assume uniform priors. For example, if Y has two possible values, then we would choose $p = 0.5$. The constant m determines how heavily to weigh p relative to the observed data and can be viewed as augmenting the x actual examples where $X = 0$ with m virtual examples where $x = 0$ and where a p fraction of these examples has $Y = 1$.

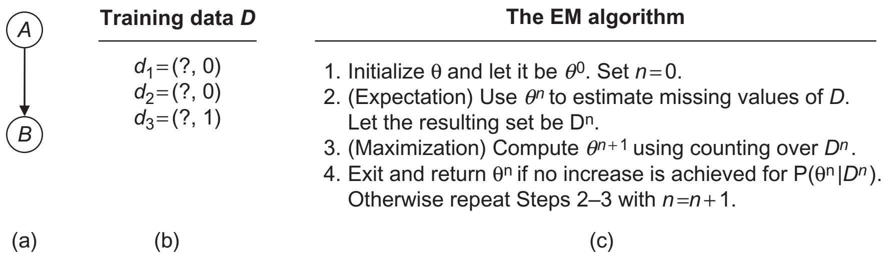
Note that when applying the m-estimate method, we apply it to all probabilities that we want to compute. For example, consider again the case where the training data consist of only three examples $d_1 - d_3$. Assuming $m = 1$ and $p = 0.5$, we have $P(B = 1|A = 1) = (1 + 1 \cdot 0.5)/(3 + 1) = 0.375$, and $P(B = 1|A = 0) = (0 + 1 \cdot 0.5)/(0 + 1) = 0.5$.

LEARNING WITH MISSING VALUES

Training examples may contain missing values. For instance, the example $d = (Cloud = ?, Sprinkler = off, Rain = ?, WetGrass = t)$ has no values for *Cloud* and *Rain*. The values may be missing because we failed to observe a variable (e.g., we slept and did not observe whether it rained). It may also be the case that the variable by its nature is unobservable. For example, if we are werewolves who only venture out of the house on dark moonless nights, then we can observe if it rains, if the sprinkler is on, and if the grass is wet, but we can never tell if the sky is cloudy. Regardless of the reason, missing values cause considerable complications for learning.

We explain how to learn in the presence of missing values using the tiny example in Figures 7.9(a-b). Here the graph is the same as the graph in Figure 7.8(a), but the training data $D = \{d_1, d_2, d_3\}$ has missing values. In particular, the value of variable A is missing from all three examples.

Given the missing values, we cannot proceed as before to use counting over D to compute the CPTs. Instead, we use an expectation-maximization (EM) algorithm (see Figure 7.9(c)). The basic idea behind this algorithm is the following. There are two unknown quantities: (1) θ, the probabilities in the CPTs that we want to learn, and (2) the



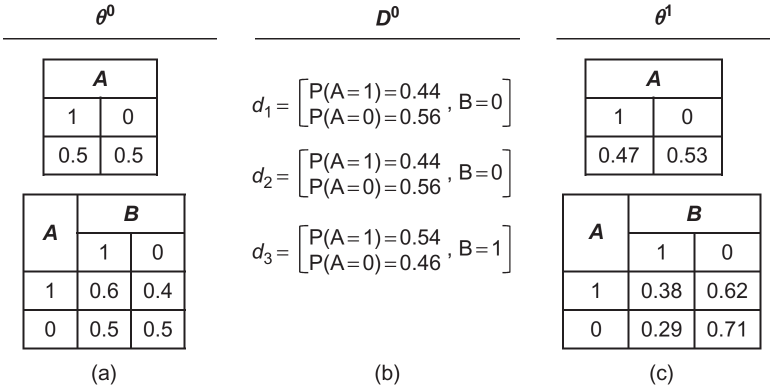
Training data *D*

d_1=(?, 0)
d_2=(?, 0)
d_3=(?, 1)

The EM algorithm

```
1. Initialize θ and let it be $\theta^0$. Set $n=0$.
2. (Expectation) Use $\theta^n$ to estimate missing values of $D$.
   Let the resulting set be $D^n$.
3. (Maximization) Compute $\theta^{n+1}$ using counting over $D^n$.
4. Exit and return $\theta^n$ if no increase is achieved for $P(\theta^n|D^n)$.
   Otherwise repeat Steps 2–3 with $n=n+1$.
```

FIGURE 7.9 Learning the CPTs using training data with missing values.



θ^0

A	
1	0
0.5	0.5

A	*B*	
	1	0
1	0.6	0.4
0	0.5	0.5

D^0

$$d_1 = \begin{bmatrix} P(A=1)=0.44 \\ P(A=0)=0.56 \end{bmatrix}, B=0$$

$$d_2 = \begin{bmatrix} P(A=1)=0.44 \\ P(A=0)=0.56 \end{bmatrix}, B=0$$

$$d_3 = \begin{bmatrix} P(A=1)=0.54 \\ P(A=0)=0.46 \end{bmatrix}, B=1$$

θ^1

A	
1	0
0.47	0.53

A	*B*	
	1	0
1	0.38	0.62
0	0.29	0.71

FIGURE 7.10 Applying the EM algorithm to the example in Figures 7.9(a-b).

missing values in D. The EM algorithm iteratively estimates these two quantities by using one to predict the other and vice versa until a convergence condition is met.

More specifically, we initialize θ to a value θ^0 (Step 1 in Figure 7.9(c)). Then we use θ^0 and the input graph G to estimate the missing values in the training data D. We compute the missing values by computing some conditional probabilities using the Bayesian network specified by θ and G (Step 2).

Let D^0 be the resulting training data (with no missing values). We can now reestimate θ using counting over D^0 (Step 3). Let the resulting θ be θ^1. We repeat Steps 2–3 until a convergence criterion is met, such as the absolute difference between $P(D^n|\theta^n)$ and $P(D^{n+1}|\theta^{n+1})$ is less than a prespecified ϵ (Step 4). Here $P(D^n|\theta^n) = \prod_d P(d|\theta^n)$, where d ranges over all training examples in D_n, and $P(d|\theta^n)$ is the probability of the state specified by d, given the Bayesian network specified by the input graph G and the probabilities θ^n.

Example 7.5

Figure 7.10 illustrates how the above EM algorithm is applied to the example in Figures 7.9(a-b). We start by randomly initializing $P(A=1) = 0.5, P(B=1|A=1) = 0.6$, and $P(B=1|A=0) = 0.5$, then filling in all the other probabilities of the CPTs. Figure 7.10(a) shows the resulting CPTs, denoted as θ^0.

Next, we use θ^0 and the graph in Figure 7.9(a) to estimate the missing values in the training data. Consider tuple $d_1 = (?, 0)$ in Figure 7.9(b). Here the value of variable A is missing. Estimating this value means estimating $P(A=1|B=0)$ and $P(A=0|B=0)$. While estimating these two probabilities is standard inference operation in Bayesian networks, here we can compute them directly using the laws of probabilities. Specifically, we have

$$P(A=1|B=0) = P(B=0|A=1)P(A=1)/P(B=0) \tag{7.4}$$

where

$$P(B=0) = P(B=0|A=1)P(A=1) + P(B=0|A=0)P(A=0) \tag{7.5}$$

Substituting appropriate probabilities from θ^0 in Figure 7.10(a) into Equations 7.4 and 7.5, we obtain $P(A=1|B=0) = 0.44$, and thus $P(A=0|B=0) = 0.56$. Figure 7.10(b) shows the resulting training data D^0 after we have estimated all missing values. Note that each example in D^0 specifies a probability distribution over the values of variable A.

Next, we use the counting algorithm over D^0 to estimate θ^1. This counting is necessarily fractional, given the probabilistic nature of the training examples. For instance, to estimate $P(A=1)$, we count the number of examples where $A=1$. We note that $A=1$ for a 0.44 fraction of d_1, a 0.44 fraction of d_2, and a 0.54 fraction of d_3. So $P(A=1) = (0.44+0.44+0.54)/3 = 0.47$. Similarly, consider computing $P(B=1|A=1) = P(B=1, A=1)/P(A=1)$. We have $B=1$ and $A=1$ in only a 0.54 fraction of d_3. Thus $P(B=1, A=1) = 0.54/3 = 0.18$. We already know that $P(A=1) = 0.47$. Thus, $P(B=1|A=1) = 0.18/0.47 = 0.38$. Figure 7.10(c) shows the resulting θ^1. In the next step, we use θ^1 to estimate the missing values in D, to compute D^1, and so on.

As described, the EM algorithm finds θ that maximizes $P(D|\theta)$, just like the counting approach in the case of no missing values. However, it may not find the globally maximal $\theta*$, converging instead to a local maximum in most cases.

Bayesian Networks as Generative Models

Generative models encode full probability distributions and specify how to generate data that fit such distributions. Bayesian networks are well-known examples of such models. Consider, for example, the familiar Bayesian network in Figures 7.7(a-b). This network specifies that, to generate a state of the world, we begin by selecting a value for variable *Cloud*, according to the CPT $P(Cloud)$ in Figure 7.7(b). Suppose we have selected $Cloud = t$. Next, we select a value for variable *Sprinkler*, according to the probability $P(Sprinkler|Cloud = t)$, then a value for *Rain*, according to the probability $P(Rain|Cloud = t)$. Suppose we have selected $Sprinkler = on$ and $Rain = f$. Then finally we select a value for *WetGrass* according to the probability $P(WetGrass|Sprinkler = on, Rain = f)$.

Taking a perspective on how the data are generated helps guide the construction of the underlying Bayesian network, discover what kinds of domain knowledge can be naturally incorporated into the network structure, and explain the network to the users.

In the rest of this section we will describe three probabilistic approaches to data matching that employ increasingly complex generative models. We begin with an approach that employs a relatively simple Bayesian network that models domain features judged relevant to matching but assumes no correlations among the features. Next, we extend the model to capture such correlations. Finally, we consider matching data tuples embedded in text documents and show how to construct a model that captures the pecularities of that domain.

7.5.2 Data Matching with Naive Bayes

We begin with a relatively simple approach to data matching that is based on Bayesian networks. We formulate the matching problem in terms of probabilistic reasoning by defining a variable M that represents whether two tuples a and b match. Our goal is to compute $P(M|a,b)$. We declare a and b matched if $P(M=t|a,b) > P(M=f|a,b)$, and not matched otherwise.

We assume that $P(M|a,b)$ depends only on a set of features $S_1,\ldots,S_n$, each of which takes a and b as the inputs and returns a discrete value. In the context of matching tuples describing people, example features include whether the two last names match, and the edit distance between the two social security numbers. Given this, we can write $P(M|a,b) = P(M|S_1,\ldots,S_n)$, where S_i is a shorthand for $S_i(a,b)$.

Next, using a well-known law of probability called Bayes' Rule, we have

$$P(M|S_1,\ldots,S_n) = P(S_1,\ldots,S_n|M)P(M)/P(S_1,\ldots,S_n) \tag{7.6}$$

If we assume that the features $S_1,\ldots,S_n$ are independent of one another given M, then we can write $P(S_1,\ldots,S_n|M) = \prod_{i=1}^{n} P(S_i|M)$. Further, observe that the denominator of the right-hand side of Equation 7.6 can be expressed as

$$P(S_1,\ldots,S_n) = P(S_1,\ldots,S_n|M=t)P(M=t) + P(S_1,\ldots,S_n|M=f)P(M=f) \tag{7.7}$$

Thus, it follows that to compute $P(M|S_1,\ldots,S_n)$ (that is, $P(M|a,b)$), we only need to know $P(S_1|M),\ldots,P(S_n|M)$ and $P(M)$.

The model we have just described is shown in the Bayesian network in Figure 7.11(a). The graph models the quantities of interest using the variables M, S_1, S_2, and S_3 (here for concreteness we consider only three variables S_1–S_3). The main simplifying assumption that we made about the nature of data matching is that S_1–S_3 are independent of one another given M. This assumption is often referred to as the *naive Bayes* assumption. The naive Bayes assumption often does not hold in the real world, but making that assumption tends to work surprisingly well in practice. Note that the probabilities $P(S_1|M),\ldots,P(S_n|M)$

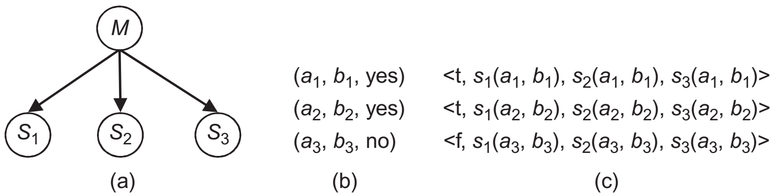

M
S1
S2
S3
(a)
(a1, b1, yes)
(a2, b2, yes)
(a3, b3, no)
(b)
<t, s1(a1, b1), s2(a1, b1), s3(a1, b1)>
<t, s1(a2, b2), s2(a2, b2), s3(a2, b2)>
<f, s1(a3, b3), s2(a3, b3), s3(a3, b3)>
(c)

FIGURE 7.11 Learning the CPTs of the naive Bayesian approach to data matching when we have training data. Note that the "yes" and "no" in part (b) have been converted into the "t" and "f" in part (c) to generate training data (a set of states in this case) for the Bayesian network in part (a).

(a_4, b_4)	$<?, s_1(a_4, b_4), s_2(a_4, b_4), s_3(a_4, b_4)>$
(a_5, b_5)	$<?, s_1(a_5, b_5), s_2(a_5, b_5), s_3(a_5, b_5)>$
(a_6, b_6)	$<?, s_1(a_6, b_6), s_2(a_6, b_6), s_3(a_6, b_6)>$
(a)	(b)

FIGURE 7.12 Learning the CPTs of the naive Bayesian approach to data matching when we have no training data. The set of tuples to be matched (part (a)) is converted into training data with missing values for the Bayesian network in Figure 7.11(a).

and $P(M)$ that we want to learn in order to compute $P(M|a,b)$ form the CPTs of this Bayesian network.

Learning these CPTs is straightforward if we have training data such as the three tuples in Figure 7.11(b), where tuple (a_1, b_1, yes), for example, states that a_1 matches b_1. We simply convert each such tuple into a feature vector. For example, (a_1, b_1, yes) is converted into $\langle t, s_1(a_1, b_1), s_2(a_1, b_1), s_3(a_1, b_1)\rangle$, which represents a state of the world that assigns values t, $s_1(a,b)$, $s_2(a,b)$, and $s_3(a,b)$ to nodes M, S_1, S_2, and S_3, respectively. After converting the tuples in Figure 7.11(b) into the feature vectors in Figure 7.11(c), we can apply learning with no missing values as described in Section 7.5.1 to these feature vectors, to learn the CPTs of the graph in Figure 7.11(a).

Once we have learned the CPTs, we can apply the Bayesian network to match a new pair of tuples. Given a pair of tuples (a_4, b_4), we compute $P(M = t|a_4, b_4)$ and $P(M = f|a_4, b_4)$, using Equations 7.6–7.7, and declare "matched" if $P(M = t|a_4, b_4) > P(M = f|a_4, b_4)$ and "not matched" otherwise. Note that in practice, the denominator $P(S_1, \ldots, S_n)$ in the right-hand side of Equation 7.6 is the same for both cases of $M = t$ and $M = f$. Thus we only need to compare the numerators of the right-hand side of Equation 7.6.

Learning the CPTs becomes more involved when we have no training data. In this case, we convert the tuple pairs to be matched into training data with missing values and apply the EM algorithm as described in Section 7.5.1. To illustrate, suppose we want to match the three tuple pairs in Figure 7.12(a). We first convert them into the three feature vectors in Figure 7.12(b). Each of these vectors specifies a value for the variables M, S_1, S_2, and S_3 of the Bayesian network in Figure 7.11(a). However, since we do not yet know the value for variable M, this value is missing from all three feature vectors. Therefore, we apply the

EM algorithm to these feature vectors to learn the CPTs. After computing the CPTs we can make matching decisions.

7.5.3 Modeling Feature Correlations

The approach we described above was naive in the sense that it assumes no correlation among the features $S_1,\ldots,S_n$, given the "match" variable M. While the naive Bayesian approach works surprisingly well in many domains, there are domains in which accurate matching decisions require that we capture the correlations among the features involved.

The Bayesian network in Figure 7.13(a) attempts to model such correlations by adding directed edges among the feature nodes. Suppose that node S_1 models whether the social security numbers match, and node S_3 models whether the last names match. Then we may want to create an edge from S_1 to S_3, because if the two social security numbers match, then it is very likely that the two last names also match.

The problem with adding such edges is that it may quickly "blow up" the number of probabilities in the CPTs. For example, suppose that on average each node has q parents and d values; then the number of probabilities in the CPTs for the n feature nodes is $O(nd^q)$. In contrast, for the naive Bayesian network in Figure 7.11(a), we need only $2dn$ probabilities ($2d$ for each feature node). The more probabilities we have in the CPTs, the more training data we need to learn accurately, and the longer it takes to learn.

The Bayesian network in Figure 7.13(b) reduces the size of the required CPTs but with some loss of expressive power. Suppose each tuple to be matched has k attributes. Then here we will consider only k features $S_1,\ldots,S_k$, each of which takes as input the values of only one attribute, then outputs a discrete value. For example, if each tuple has only two attributes, *name* and *address*, then we consider two features: the first one compares the two names and the second one compares the two addresses. Note that in contrast, earlier we considered also features that can relate different attributes, such as a zip code with an area code.

Figure 7.13(b) assumes there are four attributes, and shows four features S_1–S_4. For each attribute S_i we introduce a binary variable X_i. The X_i variables model whether the

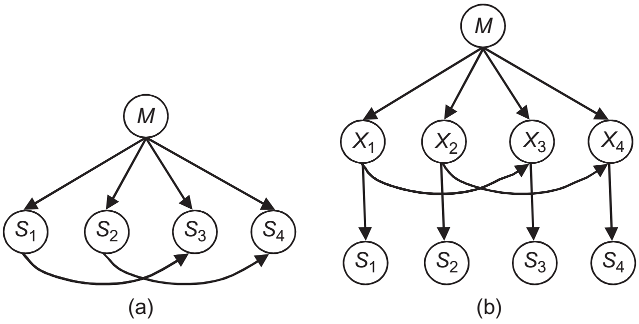


FIGURE 7.13 Sample Bayesian networks of approaches that model feature correlations.

attributes should match, given that the input tuple pair match. For example, if X_1 is about social security numbers, then X_1 models whether the social security numbers match given that the tuples match. It is likely that the value of $P(X_1 = t|M = t)$ would be high in the CPT. Now suppose S_1 returns the edit distance between the social security numbers (i.e., a number between 0 and 9). Then $P(S_1 = 0|X_1 = t)$ and $P(S_1 = 1|X_1 = t)$ would likely be high, to reflect the domain knowledge that if two social security numbers match, then the edit distance between their appearances in the data should be relatively small (i.e., 0 or 1).

In the final step of creating the network in Figure 7.13(b) we model correlations only at the X_1–X_4 level (thus assuming that the S_1–S_4 are independent of one another given the X_1–X_4). For example, suppose again that X_1 is about social security numbers and X_3 is about last names. Then we create an edge from X_1 to X_3 to capture the knowledge that if the social security numbers match, then it is highly likely that the last names also match.

The network in Figure 7.13(b) requires far fewer probabilities in the CPTs. Assume that on average each node has q parents. Then the total number of probabilities required for the nodes X_1–X_k is $O(k2^q)$ (recall that these are binary nodes, with only two values t and f). If each feature node has d values, then we need only $2kd$ probabilities for these nodes. Thus the total is $O(k2^q + 2kd)$, far less than the number of probabilities required for the Bayesian network in Figure 7.13(a). The key to achieving this gain is to push dependency modeling from a level where each node can have many values (d on average) to a level where each node can have only two values. Of course, we pay for this gain by the fact that the graph in Figure 7.13(b) is less expressive in that it cannot model arbitrary features.

A key lesson to take away from this discussion is that constructing a Bayesian network for a matching problem is an "art" that must consider the trade-offs among many factors, including how much domain knowledge can be captured, how accurately we can learn the network, and how efficiently we can do so. The next section shows an even more complex example.

7.5.4 Matching Mentions of Entities in Text

In this section we describe a data matching example that requires a more complex generative model. Instead of considering the problem of matching tuples, as we have done so far, we consider the problem of matching *entity mentions* (e.g., "Michael Jordan," "M. Jordan," and "Mike I. Jordan") in text documents.

PROBLEM DEFINITION

Let D be a set of text documents, and M be a set of entity mentions in D, where each *mention m* is a string that occurs in a document and refers to a real-world entity. Such mentions can be discovered by applying information extraction techniques to the documents. Our goal is to decide, for any two mentions m_i and m_j in M, whether they refer to the same real-world entity. For example, consider the six mentions (the underlined strings) in the three documents d_1–d_3 in Figure 7.14(a). We want to know, for instance, that mention "Michael Jordan" in d_1 matches "Mike" in d_1 and "Michael J. Jordan" in d_2, but not "Prof. M. I. Jordan" in d_3.

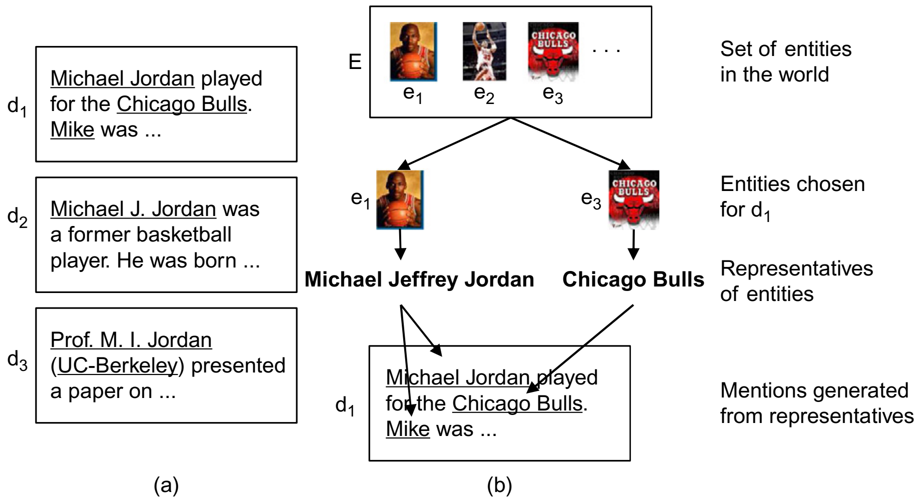

d1
Michael Jordan played for the Chicago Bulls. Mike was ...
d2
Michael J. Jordan was a former basketball player. He was born ...
d3
Prof. M. I. Jordan (UC-Berkeley) presented a paper on ...
(a)
E
e1
e2
e3
. . .
Set of entities in the world
e1
e3
Entities chosen for d1
Michael Jeffrey Jordan
Chicago Bulls
Representatives of entities
d1
Michael Jordan played for the Chicago Bulls. Mike was ...
Mentions generated from representatives
(b)

FIGURE 7.14 (a) A tiny collection of three text documents with person and organization mentions underlined; (b) a generative model to generate the mentions of a document.

THE GENERATIVE MODEL

To develop a matching algorithm, we create a generative model for how entity mentions are generated in text. We assume that the world consists of a set of entities E, such as Michael Jordan the bastketball player, Michael Jordan the researcher, and the Chicago Bulls basketball team. We generate the mentions of a document d in the following steps:

1. We select a number K that will represent the number of entities mentioned in the document d. We select K according to a distribution $P(K)$.
2. We select K entities from E, according to a distribution $P(E)$.
3. For each selected entity e, we select a string r called a *representative* of e, according to a distribution $P(r|e)$.
4. We select a number L that represents the number of mentions of entity e. We select L according to a distribution $P(L)$. Then we perturb r to generate L mentions $m_1, \ldots, m_L$, according to a distribution $P(m|r)$.
5. Finally, we "sprinkle" the mentions $m_1, \ldots, m_L$ in the document d.

Figure 7.14(b) illustrates generating the mentions of document d_1. We first select two entities e_1 and e_3, then two representatives "Michael Jeffrey Jordan" and "Chicago Bulls." Intuitively, each representative is the "full name" of the respective entity. Next, we perturb the representatives to generate mentions. For example, we drop the middle name from the representative "Michael Jeffrey Jordan" to obtain mention "Michael Jordan," drop both middle and last names, then transform the first name into a nickname, to obtain mention "Mike." Finally we sprinkle these mentions into d_1.

As described, the set of parameters θ of our generative model consists of (1) probability distributions $P(K)$ and $P(L)$ that specify how to select the number of entities and the number of mentions of each entity (respectively), (2) a set of entities E and a distribution $P(E)$ that specifies how to select entities, and (3) distributions $P(r|e)$ and $P(m|r)$ that specify how to select representatives and mentions (respectively).

Before we discuss how to learn this model from the data, we show how the model can be used for a data-matching algorithm.

USING THE MODEL TO MATCH MENTIONS

Given a document d with a set of mentions M_d, we want to apply the generative model to match M_d. Intuitively, this means finding the set of entities E_d*, the set of representatives R_d* in d, and the assignments of the mentions in M_d to E_d* and R_d*. We declare two mentions matched if and only if they are assigned to the same entity. Formally, we want to find the most likely E_d* and R_d*, that is:

$$\begin{aligned}(E_d*, R_d*) &= \arg\max_{E_d, R_d} P(E_d, R_d | M_d) \\ &= \arg\max_{E_d, R_d} P(E_d, R_d, M_d) \\ &= \arg\max_{E_d, R_d} P(E_d) P(R_d | E_d) P(M_d | R_d) \qquad (7.8)\end{aligned}$$

Equation 7.8 assumes conditional independence between M and E given R and ignores the probabilities on the number of entities and the number of mentions, as they can be omitted in computing $\arg\max$. Solving Equation 7.8 by considering all possible combinations of E_d and R_d is clearly infeasible. Therefore, we resort to an approximation algorithm.

In the first step, we find R_d* directly by sequentially clustering the mentions M_d. We initialize the clustering algorithm by selecting the longest mention in M_d as the first representative. For each mention $m \in M_d$, we compute $P(m|r)$ for each representative r that has already been created. We use a fixed threshold to decide whether to add m to an existing group that maximizes $P(m|r)$ or to create a new group for m. We always select the longest mention in each group as its representative.

In the second step, once we have found the set of representatives R_d*, we assign each representative $r \in R_d*$ to the entity $e* = \arg\max_e P(e)P(r|e)$. By composing the mapping from mentions to representatives with the mapping from representatives to entities, we have created a mapping from mentions to entities for a single document d.

We can proceed similarly to match a set of new documents $d_1, \ldots, d_n$ with mentions $M_1, \ldots, M_n$. Specifically, once we have assigned mentions to representatives and representatives to entities, then two mentions (from the same document or across documents) match if and only if they refer to the same entity.

Learning the Generative Model

We now describe how to learn the generative model. As before, we distinguish the case in which we have training data from the case in which we do not.

CASE 1: WHEN WE HAVE TRAINING DATA

To simplify our discussion, we assume that $P(K)$ and $P(L)$ are uniform probability distributions over small plausible numeric ranges. For example, we can define $P(K)$ to be a uniform distribution over the range $[1,8]$, on the assumption that each document mentions anywhere between 1 and 8 entities. Given this, we only have to learn the set E and the distributions $P(E), P(r|e)$, and $P(m|r)$.

Let the training data be a set D of documents, and we assume the following. First, we have identified all mentions M of the target entity types (e.g., if we want to match person names and organization names, then we have identified all person and organization mentions in D). Second, we have partitioned M into n clusters such that mentions within each cluster (and only these mentions) refer to a single real-world entity.

Learning E and P(E): Given the training data, we can create n entities $e_1, \ldots, e_n$, one for each cluster, and let this set of entities be E. Assuming that entities are *independently* chosen into a document, we can learn $P(E)$ by computing $P(e_i)$, $i \in [1,n]$, as the number of mentions in the cluster of e_i divided by the total number of mentions in M. The probability of any subset F of E is then $P(F) = \prod_{e \in F} P(e)$.

Learning $P(m|r)$*:* To learn $P(m|r)$, we begin by assuming that each mention m and each representative r are associated with a set of attributes $a_1, \ldots, a_p$. For example, each person mention may have the attributes *title*, *firstName*, *middleName*, and *lastName*. Thus, mention "Michael Jeffrey Jordan" can be converted into a tuple $(title = null, firstName = Michael, middleName = Jeffrey, lastName = Jordan)$.

Suppose $m = (a_1 = v_1, \ldots, a_p = v_p)$ and $r = (a_1 = v'_1, \ldots, a_p = v'_p)$. Then we model $P(m|r)$ as the product of the transformation probabilities from the v'_k's to the v_k's. Specifically,

$$P(m|r) = \prod_{k=1}^{p} P(a_k(m) = v_k | a_k(r) = v'_k)$$

where $P(a_k(m) = v_k | a_k(r) = v'_k)$ is the probability that the kth attribute of m is v_k, given that the corresponding attribute of r is v'_k.

Instead of computing $P(a_k(m) = v_k | a_k(r) = v'_k)$ directly, we distinguish four types of perturbations, and we compute the probability of each one of the perturbation types. We denote the probability that the perturbation of the kth attribute is of type t by $P(k,t)$.

The four types of perturbations are *copy, missing, typical*, and *atypical*. If $v_k = v'_k$, then we say the perturbation type is *copy*. If $v_k = null$, then the type is *missing*. If v_k is obtained from v'_k in a perturbation that is typically expected for the attribute, then we say the type is *typical*; otherwise, it is *atypical*. For example, if the attribute is *firstName*, then abbreviating a first name to the first character is typical, while to the first three characters is not.

To compute the probability of each of the perturbations, we first manually collect typical and atypical perturbations for common attributes. For example, for attributes such as *title*, *firstName*, *middleName*, *lastName*, and *organizationName* we can collect perturbation using multiple data sources such as the U.S. government census and online dictionaries. For other attributes such as *age*, we classify all perturbations that are not *copy* or *missing* as *atypical*.

Given the perturbation types, we can model $P(a_k(m) = v_k | a_k(r) = v'_k)$ as $P(k,t)$, the probability that the perturbation of the kth attribute is of type t. If we know where the representatives are in the training data D, then it is easy to estimate $P(k,t)$. Consider all pairs of the form (r,m) in D, where m is generated from r. Then $P(k,t)$ is the number of pairs where the perturbation of $a_k(r)$ into $a_k(m)$ is of type t, divided by the total number of pairs. (We perform smoothing for perturbation types unseen in the training data, in a way similar to the smoothing discussed in Section 7.5.1.)

Thus we can learn $P(k,t)$ if we know the representatives. The training data D, however, does not specify the representatives, only the mentions and the clusters (i.e., the entities). To address this problem, from the set of all mentions in a document d that correspond to an entity e, we select the one with the largest length as the representative r of e. This heuristic is based on the assumption that a representative r in a document d is the "full name" of an entity e and that all mentions of e in d are obtained by perturbing this full name.

Learning $P(r|e)$*:* The same process can be applied in a similar fashion to learn $P(r|e)$. The only missing piece is how to obtain the attribute values for the entities. Again, from the set of all mentions (across all documents) that refer to an entity e, we can select the one with the largest length as the "full name" of e, then process this full name to obtain the attribute values.

CASE 2: WHEN WE DO NOT HAVE TRAINING DATA

In this case we apply the EM algorithm. Let D be a set of documents and M be a set of mentions in D. We apply the EM algorithm to match M as follows.

1. *Initialization:* We assign an initial (E_d^0, R_d^0) to each document $d \in D$ (i.e., assigning each mention in d to a representative in a set R_d^0 and each representative to an entity in a set E_d^0), as described below. Let $D^0 = \{(E_d^0, R_d^0, M_d) | d \in D\}$, where M_d is the set of mentions in document d. That is, D^0 is the set D where each mention in each document $d \in D$ has been assigned to a representative and each representative has been assigned to an entity.
2. *Maximization:* We compute the parameter θ^{t+1} that maximizes $P(D^t|\theta)$. This amounts to learning the model parameters when we have training data. So we compute θ^{t+1} using the techniques described in Case 1 above.
3. *Expectation:* For each document $d \in D$, we compute (E_d^{t+1}, R_d^{t+1}) that maximizes $P(D^{t+1}|\theta^{t+1})$, where $D^{t+1} = \{(E_d^{t+1}, R_d^{t+1}, M_d) | d \in D\}$. This amounts to using the model

to match mentions, that is, assign mentions to representatives and entities, as described earlier.

4. *Convergence:* Exit if $P(D^t|\theta^t)$ does not increase; otherwise, repeat Steps 2–3.

We note that the above EM algorithm differs from the standard EM algorithm described in Figure 7.9(c) in that in Step 3 it finds the most likely E^d and R^d for each document d, rather than finding probability distributions over all possible E^d and R^d. Since the set of all entities and representatives can be large, finding and storing distributions over them would have been impractical in this case. Note also that once the algorithm terminates, it returns the assignments from mentions to representatives and from representatives to entities. This amounts to solving the mention matching problem.

Finally, we complete our description of the above EM algorithm by describing the initialization step. Given a document d, we compute E^0_d and R^0_d as follows. First, we treat each mention in d as a string, then use the soft TF/IDF string similarity measure (see Chapter 4) to cluster mentions into groups. Next, we select the longest mention in each group as a representative and create an entity (with the same full name) for the group. The sets of all representatives and entities so created become R^0_d and E^0_d, respectively.

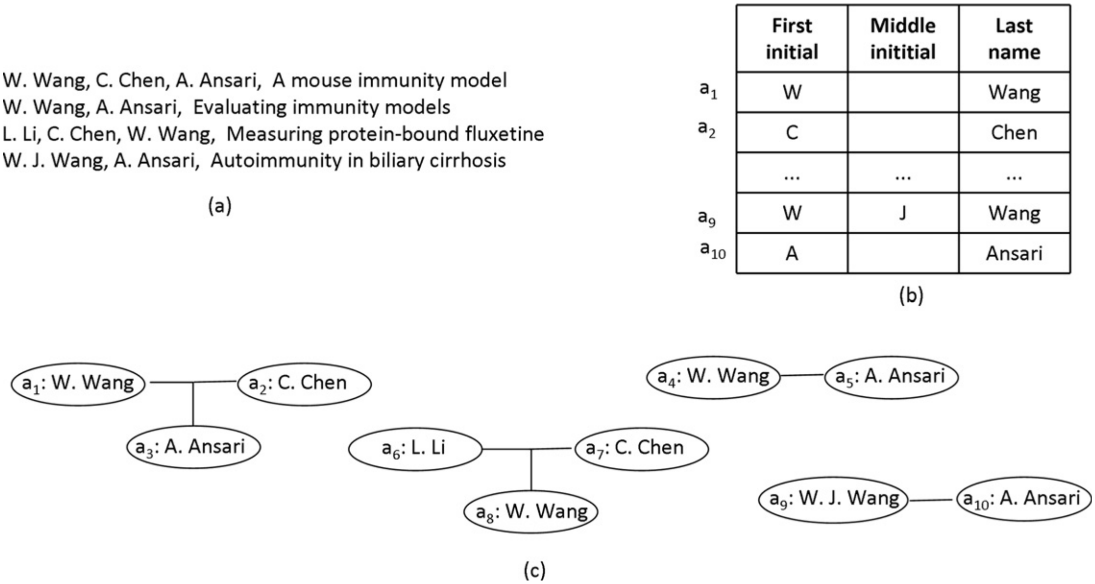
7.6 Collective Matching

The matching approaches that we have discussed so far make independent matching decisions. That is, they decide whether any two tuples a and b match *independently* of whether any two other tuples c and d match. In many real-world scenarios, however, matching decisions are intuitively *correlated*, and by exploiting such correlations we may be able to improve matching accuracy.

To illustrate, suppose we want to match the authors of the four papers listed in Figure 7.15(a). We may start by extracting their names and creating the table shown in Figure 7.15(b), where each tuple in the table lists the first name, middle name, and last name of an author. We can apply one of the methods described so far to match the tuples. However, by doing so we cannot exploit the coauthor relationships that are present in the data, because such relationships are not captured in the table.

To see the potential benefits of such relationships for the matching process, consider Figure 7.15(c). The figure visually depicts the input data as a hypergraph, whose nodes specify the authors, and whose hyperedges connect the coauthors. Suppose we have determined that nodes a_3 : *A. Ansari* and a_5 : *A. Ansari* match, then intuitively that should boost the likelihood that nodes a_1 : *W. Wang* and a_4 : *W. Wang* match, as they share the same name and same coauthor relationship to the same author.

A simple way to exploit the extra information is to add an attribute called *coAuthors* to the tuples in Table 7.15(b). For example, tuple a_1 would have *coAuthors* = {*C. Chen*,*A. Ansari*}; tuple a_4 (for author *W. Wang* in the second paper) would have *coAuthors* = {*A. Ansari*}. We can then apply existing matching methods, but make sure that they take into account the similarity between the *coAuthors* attributes. For example, we



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

(a)

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

(b)

FIGURE 7.15 (a) Four publications that contain a set of author mentions that we want to match, (b) converting the mentions into relational tuples, (c) viewing the mentions as nodes in a hypergraph, whose hyperedges denote coauthor relationships among the mentions.

can compute a Jaccard similarity score between these attributes. Given $coAuthors(a_1) = \{C.\ Chen, A.\ Ansari\}$ and $coAuthors(a_4) = \{A.\ Ansari\}$, we have

$$\begin{aligned} JaccardSim_{coAuthors}(a_1, a_4) &= \frac{|coAuthors(a_1) \cap coAuthors(a_4)|}{|coAuthors(a_1) \cup coAuthors(a_4)|} \\ &= 1/2 \end{aligned} \tag{7.9}$$

This solution, however, can be misleading. Suppose authors $a_3 : A.\ Ansari$ and $a_5 : A.\ Ansari$ share the same name but do not match. In this case the above solution would still assume they match. That is, it still assumes the strings *A. Ansari* in $coAuthors(a_1)$ and $coAuthors(a_4)$ refer to the same author. Thus, it would incorrectly "boost" the similarity score between $a_1 : W.\ Wang$ and $a_4 : W.\ Wang$. In a sense, the source of the problem is that by treating the coauthors as a feature, string equality is already taken as a matching decision.

To avoid this problem, we want to conduct several matching decisions in parallel. Intuitively, we want to match $a_3 : A.\ Ansari$ and $a_5 : A.\ Ansari$, then use that information to help match $a_1 : W.\ Wang$ and $a_4 : W.\ Wang$. But, of course, we also want the opposite, namely, match a_1 and a_4, then use that information to help match a_3 and a_5. This suggests that we may want to match tuples in a *collective* fashion, all at once and iteratively, so that we can leverage the correlations across matching decisions to improve the overall matching accuracy. In what follows we describe two approaches that are based on this intuition. The first approach employs clustering to match tuples, and the second approach extends the probabilistic generative model of Section 7.5.4 to collectively match entity mentions in text documents.

7.6.1 Collective Matching Based on Clustering

We assume that our input is a graph whose nodes specify tuples to be matched and whose edges specify relationships among the tuples. Figure 7.15(c) shows such an input graph, with the edges depicting the coauthor relationship among the nodes. For ease of exposition, we will consider just one type of relationship here, but the approach can be generalized to work with multiple types of relationships.

To match the tuples, we perform agglomerative hierarchical clustering (AHC), exactly as described in Section 7.4, except that here we modify the similarity measure to take into account the correlations among the tuples. Specifically, let A and B be two clusters of nodes. Then we define

$$sim(A,B) = \alpha \cdot sim_{attributes}(A,B) + (1-\alpha) \cdot sim_{neighbors}(A,B)$$

where α is a predefined coefficient. The measure $sim_{attributes}(A,B)$ computes a similarity score between A and B based only on the attributes of the nodes in A and B. Section 7.4 discusses several such scores, such as *single link, complete link*, and *average link*.

The measure $sim_{neighbors}(A,B)$ computes a similarity score between the neighborhood of A and B. Recall that we assume a single relationship R on the edges of the input graph. We define $\mathcal{N}(A)$, the neighborhood of A, to be the bag of the cluster IDs of all nodes that are in the relationship R with some node in A. For example, suppose cluster A has two nodes a and a', and suppose a is in the coauthor relationship with a node b with cluster ID 3, and that a' is in the coauthor relationship with a node b' with cluster ID 3 and another node b'' with cluster ID 5. Then $\mathcal{N}(A) = \{3,3,5\}$. We define $\mathcal{N}(B)$ similarly. Now we can define $sim_{neighbors}(A,B)$ using $\mathcal{N}(A)$ and $\mathcal{N}(B)$, such as

$$sim_{neighbors}(A,B) = JaccardSim(\mathcal{N}(A),\mathcal{N}(B)) = \frac{|\mathcal{N}(A) \cap \mathcal{N}(B)|}{|\mathcal{N}(A) \cup \mathcal{N}(B)|} \tag{7.10}$$

Contrast the above equation with Equation 7.9. Both equations define Jaccard similarity measures over the "neighborhoods." However, Equation 7.9 exploits only the "strings" in the neighborhoods, whereas the above equation exploits the matching decisions in the neighborhoods. More specifically, Equation 7.10 is based on similarity of cluster IDs, which encode matching decisions because two tuples that share the same cluster ID must match.

Note that in Equation 7.10 we can replace the Jaccard measure with whatever similarity measure deemed appropriate for the problem at hand (see Section 7.7.2 for other similarity functions that compare neighborhoods). Note also that the notion $\mathcal{N}(A)$ as defined above technically covers only the "neighhood of radius 1" of A. We can also define higher order neighborhoods and incorporate them into the definition of $sim_{neighbors}(A,B)$.

Figure 7.16 illustrates how AHC works on the input graph in Figure 7.15(c). Initially, we place each node into a singleton cluster, resulting in ten clusters c_1–c_{10} shown in

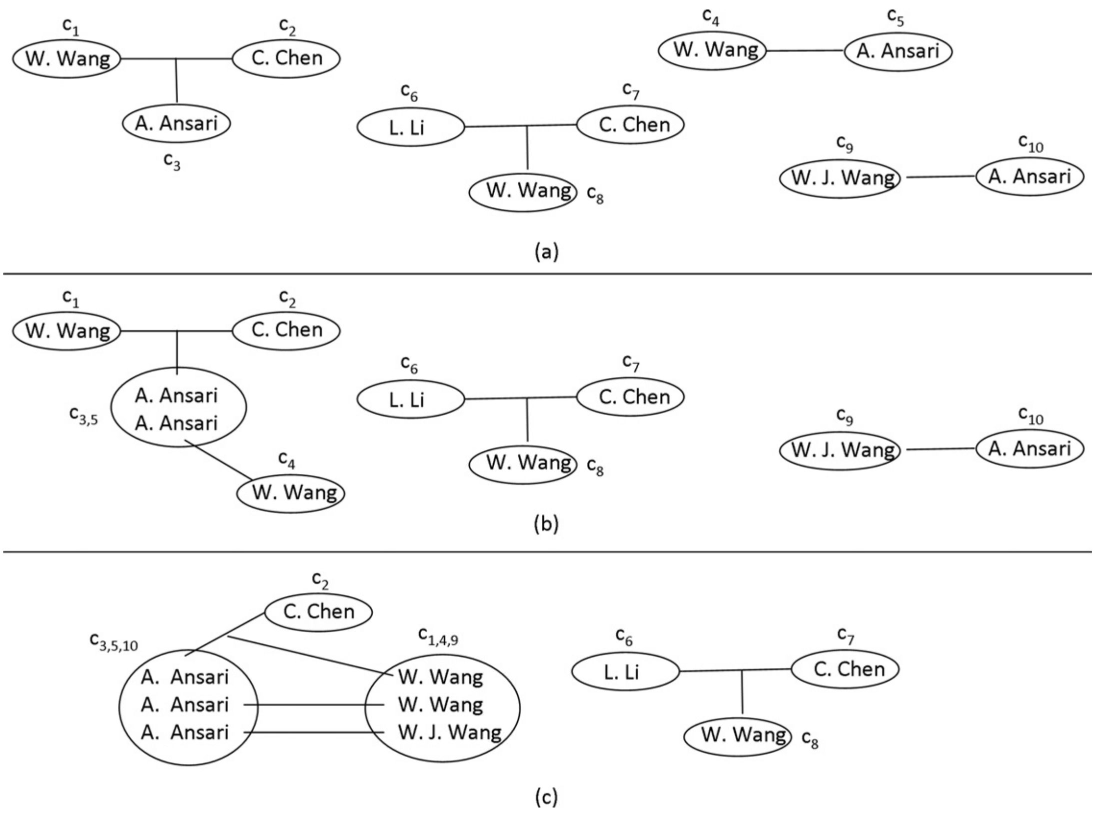


FIGURE 7.16 An example of collectively matching author mentions using clustering.

Figure 7.16(a). Next, we compute cluster similarities and merge the two most similar clusters, c_3 and c_5, into cluster $c_{3,5}$ in Figure 7.16(b). Next, we recompute cluster similarities. In computing the similarities for cluster $c_{3,5}$, we will have to compute $\mathcal{N}(c_{3,5})$, its neighborhood, which is $\{c_1, c_2, c_4\}$, according to Figure 7.16(b). Then we merge the two most similar clusters, and so on. Figure 7.16(c) shows the final clusters.

7.6.2 Collectively Matching Entity Mentions in Documents

In Section 7.5.4 we considered the problem of matching entity mentions in text documents. There, our solution employs a probabilistic generative model. Given a document d, the model selects a set of entities, generates representatives and mentions, then sprinkles the mentions into d.

An important limitation of that model is that it selects the entities of d *independently*: each entity e according to a probability $P(e)$ that does not depend on any other entities.

This independence assumption often does not hold in practice. For example, a document that mentions Michael Jordan and the basketball team Chicago Bulls is more likely to mention the Los Angeles Lakers, another basketball team, than the Los Angeles Philharmonic. Thus, there are often correlations among the entities, and a collective approach that finds and exploits these correlations can help improve matching accuracy.

We now describe a collective matching approach that extends the approach in Section 7.5.4 in a relatively simple fashion. The basic idea is to select entities sequentially, with the probability of selecting an entity depending on which entities have been selected so far. Specifically, given a document d, suppose we have selected the entities $E_d^{i-1} = \{e_1, \ldots, e_{i-1}\}$. Then next we will select entity e_i with the probability $P(e_i|E_d^{i-1})$. We now consider how this new "twist" changes the approach in Section 7.5.4.

We first must learn the model. Let us first consider the case in which we have training data. Let E be the set of all entities that appear in the training data. Instead of learning $P(e)$ for each $e \in E$, as before, now we would have to learn $P(e|S)$, for each $S \subseteq E$. However, when $|E|$ is large we would have to learn too many probabilities, and for many of them we would not have sufficient training data.

We could reduce the number of probabilities we compute by approximating $P(e|S)$ as $\max_{s \in S, s \neq e} P(e|s)$. In this case we still have to learn $|E|^2$ probabilities, possibly a huge number in practice. Consequently, we further approximate $P(e|s)$ as 1 if e and s co-occur in any document in the training data, and as $P(e)$ otherwise. We can now modify the counting procedure in Section 7.5.4 in a relatively straightforward fashion to learn $P(e|s)$ for all e and s in E.

Once we have learned the model, we use it to match mentions. Consider the problem of matching a set of mentions M_d in a new document d. In Section 7.5.4 we cast this problem as finding a set of entities E_d and a set of representatives R_d that maximize $P(E_d)P(R_d|E_d)P(M_d|R_d)$ (see Equation 7.8). To do this, we first find R_d directly, by sequentially clustering the mentions M_d, and then we find E_d that maximizes $P(E_d)P(R_d|E_d)$. Since the entities in E_d are probabilistically independent, this amounts to finding for each $r \in R_d$ the entity e that maximizes $P(e)P(r|e)$, a computationally tractable step.

In the new setting of collective matching, we still seek to find E_d that maximizes $P(E_d)P(R_d|E_d)$. Unfortunately, now the entities in E_d are correlated. So we cannot maximize $P(E_d)P(R_d|E_d)$ by maximizing $P(e)P(r|e)$ for each $r \in R_d$ in isolation, as described above. On the other hand, maximizing $P(E_d)P(R_d|E_d)$ by explicitly enumerating all possible values of E_d would be practically infeasible. Thus, we approximate this step by considering only a relatively small set of promising values for E_d. Specifically, for each $r \in R_d$ we find the top k entities that maximize $P(e)P(r|e)$ (where k is prespecified). Then we consider only those values for E_d that come from the Cartesian product of these top k lists.

In the case where we do not have the training data, we use the EM algorithm as described earlier, except that whenever we have to learn the model and use the model to match mentions, we use the modified algorithm we just described.

7.7 Scaling Up Data Matching

We now turn to the question of how to scale up the approaches to data matching that we described.

7.7.1 Scaling Up Rule-Based Matching

The key challenges in scaling rule-based matching are minimizing the number of tuple pairs to be matched and minimizing the time it takes to match each pair. We describe several techniques to address the first challenge.

Hashing: We scan and hash the tuples into "buckets," then match only tuples within each bucket. This technique works well if the hashing function spreads the tuples over a large number of buckets such that tuples in different buckets are unlikely to match. For example, suppose we are matching house listings that come from multiple real-estate Web sources, but we are confident that the zip codes are correct. In that case, we can hash the listings by zip code and match houses within each bucket to remove duplicate listings.

Sorting: We use a key to sort the tuples, then scan the sorted list and match each tuple with only the previous $(w-1)$ tuples, where w is a prespecified "window size." The key should be strongly *discriminative*, in that using it to sort "brings together" tuples that are likely to match and "pushes apart" tuples that are not likely to match. Example keys include social security numbers, student IDs, last names, and the Soundex value of last names. Sorting clearly employs a stronger heuristic than hashing in that it also requires that tuples that are likely to match must be within a "window" of size w of one another. When this heuristic holds, however, sorting is often faster than hashing because it would match fewer tuple pairs.

Indexing: We index the tuples such that given any tuple a, we can use the index to quickly locate a relatively small set of tuples that are likely to match a. For example, suppose we have built an inverted index on names. Then, given tuple a with the name "Michael Jordan," we can look up the index to find all tuples whose names contain "Michael" and all tuples whose names contain "Jordan." We then match a with only tuples in the union of these two sets, using the heuristic that if two tuples match, their names must share at least one term.

Canopies: We use a computationally cheap similarity measure to quickly group tuples into overlapping clusters called *canopies* (or *umbrella sets*). We then use a different (and typically far more expensive) similarity measure to match tuples within each canopy. For example, if two strings differ by at least 3 in length, then their edit distance cannot be smaller than 3. Thus, we can use length comparison to quickly group tuples into canopies, then use the more expensive edit distance measure to match tuples within each canopy. As another example, we can use the TF/IDF measure to create canopies, then use more expensive similarity measures to match tuples within each canopy. Note that unlike buckets in the hash technique that are disjoint, here the canopies can overlap, and often do.

Using representatives: This technique is applied during the matching process. We assign tuples that have been matched into groups such that tuples within a group match and tuples across groups do not match. Next, we create a representative for each group, either by selecting a tuple in the group, or by "merging" tuples in the groups. When we consider a new tuple *a*, we only try to match it with the representatives of the groups. The rationale is that if *a* does not match a representative *r*, then it is unlikely to match any tuple in the group that contains *r*. A variation of this technique is often used to match Web-scale data, as we describe later in this section.

Combining the techniques: Each of the techniques we described above uses a heuristic to "bring together" candidate tuple pairs, then match only those. However, using just a single heuristic runs the risk of missing tuple pairs that should be matched but are not. For example, if we only hash and match houses based on their zip codes, then we may miss houses that match but happen to have different zip codes due to spelling mistakes (e.g., 53705 versus 53750). To minimize this risk, we can conduct multiple runs, each of which uses a different heuristic. For example, in addition to hashing house listings based on zip code, we may also hash separately based on agent name, relying on the heuristic that each house is typically represented by just one agent. The final set of matching tuples is then the union of the results of all runs.

The above techniques can be also combined in flexible ways, based on the nature of the data set. For example, we may hash houses into buckets using zip codes, then sort houses within each bucket using street names, before matching them using a sliding window. As another example, we can use hashing, sorting, or indexing to speed up the application of the cheap similarity measure in the canopy technique.

Once we have minimized the number of tuple pairs to be matched, we must minimize the time it takes to match each pair. If we use a simple rule-based approach, such as matching individual attributes then combining their scores using weights, as described in Equation 7.1, then we can use the technique of "short circuiting" the matching process: we stop the computation of the similarity score if it is already so low that the tuple pair cannot match, even if all the remaining attributes match perfectly. We discuss other optimization methods in Section 7.7.2.

7.7.2 Scaling Up Other Matching Methods

Learning, clustering, and probabilistic approaches to data matching face similar scalability challenges to rule-based approaches and can benefit from the techniques described above. Collective matching algorithms incorporate many aspects of the above matching approaches, and hence often benefit from the scalability techniques we have just described.

Probabilistic approaches, however, raise additional scalability challenges. In particular, if a probabilistic model has too many parameters, then it is difficult to learn it efficiently, and we would also need a large number of training data to learn it accurately. One approach to scale probabilistic matching methods is to make independence assumptions to reduce the number of parameters (as illustrated in Sections 7.5.2–7.5.4). Of course, such assumptions may affect matching accuracy, and the trade-offs must be evaluated carefully on a case-by-case basis. Once the model is learned, inference with the model can also often be very time consuming. One could use approximate inference algorithms or simplify the model so that these probabilities can be computed using closed form equations (e.g., see the naive Bayesian approach in Section 7.5.2). When no training data is available, we have to use unsupervised approaches such as the EM algorithm described in Section 7.5.1, which can be expensive. To address this problem, we can "truncate" the EM algorithm by computing only the maximum-likelihood probabilities instead of the entire expected distributions (see Section 7.5.4) and by initializing the EM algorithm as accurately as possible, to "encourage" faster convergence.

Finally, we note that large-scale data matching often employs parallel processing. For example, we can hash the tuples into buckets, then match within each bucket in parallel. As another example, Web-scale applications often match tuples against a taxonomy of entities (e.g., a product or Wikipedia-like concept taxonomy), in a parallel fashion. Two tuples are then declared matched if they get assigned to the same taxonomic node. The taxonomy is typically maintained semi-automatically. Matching against a taxonomy is a variant of the approach of using representatives described earlier in this section.

Bibliographic Notes

Data matching has had a long history, dating back to the late 1950s (see [143, 205, 248, 351, 353, 453, 573] for recent books, surveys, and tutorials). In 1959 Newcombe et al. [458] introduced the record linkage problem and suggested many matching ideas: use of Soundex to handle spelling errors, blocking to reduce the number of tuple comparisons, multiple blocking rules to increase the number of matches found, and estimating match probabilities using independence assumptions. This is perhaps the first well-known work in data matching.

In 1969 Fellegi and Sunter followed up the above work with an influential theory of record linkage [224]. Among others, the theory introduces what is roughly the naive Bayesian matching model described in Section 7.5.2. Later, Winkler et al. [572, 574, 575] extended the model in substantial ways to capture additional domain knowledge and proposed using the EM algorithm to learn the model.

These early works originated from the statistics community. Starting in the 1980s, however, data matching also received increasing attention in the database, data mining, and AI communities. Initial approaches assumed tuples that represent the same entity in different databases share a common key (e.g., the Multi-Base project by Dayal [165], [168]). The Pegasus project [19] asked the users to specify the equivalence among object instances (e.g., as a table that maps local object IDs to global object IDs). Monge and Elkan [443] proposed matching whole tuples as strings, using a variant of the Smith-Waterman string similarity measure (see Chapter 4). Dey et al. employed linearly weighted and logistic regression rules to match tuples [175, 176]. Hernandez and Stolfo [300, 301] described a rule-based approach that sorts and matches the tuples using a sliding window. Other works that employ handcrafted rules include [119, 390, 484, 564]. Lawrence, Giles, and Bollacker [249] in particular described handcrafting rules to match citations in the well-known Citeseer system. See Tejada, Knoblock, and Minton [545] for further discussion of rule-based approaches.

Learning matching rules from training data using methods such as decision tree, naive Bayes, and support vector machines (SVMs) was considered in Tejada, Knoblock, and Minton [545] and Sarawagi and Bhamidipaty [507]. Bilenko and Mooney [85, 86] used small corpora of hand-labeled examples to learn appropriate string similarity measures for each attribute pair, then applied SVMs to learn how to combine these measures. Other learning approaches include [141, 243, 479].

McCallum, Nigam, and Ungar [418] described an agglomerative hierarchical clustering approach to data matching and introduced the idea of applying a cheap similarity measure to quickly group tuples into overlapping clusters called canopies. Cohen and Richman [144] show how to make generic clustering methods adaptive to the problem at hand using training.

Probabilistic approaches (which dated back to the work of Newcombe, Fellegi, Sunter, and others, as described above) saw a resurgence in the mid-1990s, and have since become quite popular. Koller and Friedman [348] cover in detail graphical probabilistic models, including Bayesian networks, variables with continuous values, and the EM algorithm. The two matching approaches that use Bayesian networks in Sections 7.5.2–7.5.3 come from Ravikumar and Cohen [494] (which also discusses the case of variables with continuous values), while the generative model for matching entity mentions in text in Section 7.5.4 builds on the work of Li, Morie, and Roth [385, 386]. Generative models encode *full* probability distributions, and learning such models often means learning a large number of probabilities. This is difficult and often requires a large amount of training data. Consequently, recent work (e.g., [156, 416, 524]) has explored using *discriminative models* such as conditional random fields (CRF) [363], which encode only the probabilities that are necessary for matching (e.g., the probability of a label given a tuple pair), and try to learn them directly from the data.

Collective approaches to data matching emerged in the early 2000s. The rule-based approach described in Section 7.6.1 comes from Bhattacharya and Getoor [82]. Other rule-based collective approaches include the Semex system by Dong and Halevy [188], work by Ananthakrishna, Chaudhuri, and Ganti [31], and by Kalashnikov, Mehrotra, and Chen [338]. Probabilistic collective matching approaches include Pasula et al. [475], Li, Morie, and Roth [385], McCallum and Culotta [156], Singla and Domingos [524], and Bhattacharya and Getoor [81].

Several works have considered matching data in nonrelational formats. The Semex system [188] matches entities in personal information management settings, where attributes are frequently missing. The work [485, 567] matches entities represented as complex XML elements. Li, Morie, and Roth [385, 386] match entity mentions in text documents, as described in Section 7.5.4.

It is often the case that different parts of the data have different degrees of heterogeneity and hence may require different matching methods. Consequently, a matching workflow may employ multiple matchers, each for a different part of the data. Papers that consider this idea include [170, 520]. The work [182, 546] shows that even matching the same part of the data also benefits from using multiple matchers, each exploiting a different heuristic.

Work that exploits domain constraints to improve matching accuracy includes [126, 521]. Work that discusses active learning for data matching includes [35, 507]. Bhattacharya and Getoor [83] discuss data matching at run-time. Several works have considered data matching in the context of data cleaning, data matching operators, and declarative data matching languages [66, 100, 124, 491, 584]. The important challenges of distributed processing, secondary storage, negative information, evolving matching rules, and pay-as-you-go data matching are discussed in the SERF project [65, 66, 568].