Learning to Map between Structured Representations of Data

AnHai Doan

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

University of Washington

2002

Program Authorized to Offer Degree: Computer Science & Engineering

University of Washington
Graduate School

This is to certify that I have examined this copy of a doctoral dissertation by

AnHai Doan

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Co-Chairs of Supervisory Committee:

____
Alon Y. Halevy

____
Pedro M. Domingos

Reading Committee:

____
Alon Y. Halevy

____
Pedro M. Domingos

____
Steven J. Hanks

Date: ______________________________

University of Washington

Abstract

Learning to Map between Structured Representations of Data

by AnHai Doan

Co-Chairs of Supervisory Committee:

Professor Alon Y. Halevy
Computer Science & Engineering

Professor Pedro M. Domingos
Computer Science & Engineering

This dissertation studies *representation matching*: the problem of creating semantic mappings between two data representations. Examples of *data representations* are relational schemas, ontologies, and XML DTDs. Examples of *semantic mappings* include "element location of one representation maps to element address of the other", "contact-phone maps to agent-phone", and "listed-price maps to price * (1 + tax-rate)".

Representation matching lies at the heart of a broad range of information management applications. Virtually any application that manipulates data in different representation formats must establish semantic mappings between the representations, to ensure interoperability. Prime examples of such applications arise in data integration, data warehousing, data mining, e-commerce, bioinformatics, knowledge-base construction, information processing on the World-Wide Web and on the emerging Semantic Web. Today, representation matching is still mainly conducted by hand, in an extremely labor-intensive and error-prone process. The prohibitive cost of representation matching has now become a key bottleneck hindering the deployment of a wide variety of information management applications.

In this dissertation we describe solutions for semi-automatically creating semantic mappings. We describe three systems that deal with successively more expressive data representations and mapping classes. Two systems, LSD and GLUE, find one-to-one mappings such as "address = location" in the context of data integration and ontology matching, respectively. The third system, COMAP, finds more complex mappings such as "name = concatenation(first-name,last-name)". The key idea underlying these three systems is the incorporation of multiple types of knowledge and multiple machine learning techniques into all stages of the mapping process, with the goal of maximizing mapping accuracy. I present experiments on real-world data that validate the proposed solutions. Finally, we discuss how the solutions generalize previous works in databases and AI on creating semantic mappings.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

This dissertation marks the end of a long and eventful journey, which began in a rural area of North-Central Vietnam. There, my parents made tremendous sacrifices to ensure that I had a good education. For this and much more, I am forever in their debt.

After high school, I moved to Hungary, where I met Aiviet Nguyen. We became good friends, and he encouraged me to go to America for my future studies. At the time, there was no diplomatic relationship between the U.S. and Vietnam, the iron curtain in Europe had just barely fallen, and the idea sounded impossible. Nevertheless, I made a try. Fortunately, Dr. Judith Ladinsky at the University of Wisconsin-Madison introduced me to Peter Haddawy, who then took me under his wing. It was with Peter that I first learned how to do research and how to write. He was a fantastic advisor who was warm and caring, and has remained supportive to me until these days. Thank you, Peter, I am very grateful.

After a Masters degree with Peter, I moved to the Ph.D. program at the University of Washington, where I have been extremely lucky to work with several superb advisors: Steve Hanks, Alon Halevy, and Pedro Domingos. They always blow me away with their amazing sharpness, technical depth, knowledge, and communication skills. I learned a lot from them. I especially thank Steve for staying extremely supportive of my work, and for ensuring my continuous funding for the first five years – it helped enormously by allowing me to focus on my studies. I owe a special debt to my two main advisors, Alon and Pedro. Thank you for your guidance on this research topic, and on research and life in general. You are simply the best advisors, period. I thank Pedro for teaching me all that I know about machine learning, for being patient with my questions, and for the many lessons on writing. I thank Alon for guiding my entrance into the database world, for being warm and caring, for putting in encouraging words when I felt down, and for not raising the price you charged for each needless word I used in our papers.

I am grateful to Oren Etzioni, for working with Alon and me in the early state of this research, and for valuable comments later on. I am also extremely grateful to Phil Bernstein, for his feedback during the course of this research, for his support for my research career, and for being on my supervisory committee. I also thank Erhard Rahm for invaluable feedback on part of this research.

This research also benefited tremendously from the many friends at UW. Special thanks to Jayant Madhavan, for countless hours spent discussing schema matching and for your help on the GLUE project. Thank you, Geoff "Squid" "Ba Meo" Hulten and Matt "Octopus" "Ma Do" Richardson, you are ideal office mates. I'm glad I could partially repay your friendship by giving each of you *two* nicknames. Thank you, Oren Zamir, Fred Pighin, Vass Litvanov, Sujay Sparekh, Omid Madani, Adam Carlson, Matthai Phillipose, Markus Mock, and Dave Hsu, for your friendship and support over the years. I owe many thanks to Zack Ives and Rachel Pottinger, fellow "pioneer" database students at UW. I'm indebted to numerous members of the database and AI groups: Dan Weld, Dan Suciu, Corin Anderson, Tessa Lau, Steve Wolfman, Igor Tatarinov, Pradeep Shenoy, Todd Millstein, and many more, for your feedback and support.

As I write these lines, my wife and son have left for Vietnam several days ago. Their absence makes me realize more than ever how much they mean to me. They are the semantics behind all that I do. It is to them that I dedicate this dissertation.

Chapter 1

INTRODUCTION

This dissertation studies *representation matching*: the problem of creating semantic mappings between two data representations. Examples of mappings are “element location of one representation maps to element address of the other”, “contact-phone maps to agent-phone”, and “listed-price maps to price * (1 + tax-rate)”.

We begin this chapter by showing that representation matching is a fundamental step in numerous data management applications. Next, we show that the manual creation of semantic mappings is extremely labor intensive and hence has become a key bottleneck hindering the widespread deployment of the above applications (Sections 1.2-1.3). We then outline our semi-automatic solutions to representation matching (Sections 1.4-1.5). Finally, we list the contributions and give a road map to the rest of the dissertation (Section 1.6-1.7).

1.1 Applications of Representation Matching

The key commonalities underlying applications that require semantic mappings are that they use structured representations (e.g., relational schemas, ontologies, and XML DTDs) to encode the data, and that they employ more than one representation. As such, the applications must establish semantic mappings among the representations, either to enable their manipulation (e.g., merging them or computing the differences [BLN86, BHP00]) or to enable the translation of data and queries across the representations. Many such applications have arisen over time and have been studied actively by the database and AI communities.

One of the earliest such applications is *schema integration*: merging a set of given schemas into a single global schema [BLN86, EP90, SL90, PS98]. This problem has been studied since the early 1980s. It arises in building a database system that comprises several distinct databases, and in designing the schema of a database from the local schemas supplied by several user groups. The integration process requires establishing semantic mappings between the component schemas [BLN86].

As databases become widely used, there is a growing need to *translate data* between multiple databases. This problem arises when organizations consolidate their databases and hence must transfer data from old databases to the new ones. It forms a critical step in *data warehousing* and *data mining*, two important research and commercial areas since the early 1990s. In these applications, data coming from multiple sources must be transformed to data conforming to a single target schema, to enable further data analysis [MHH00, RB01].

During the late 1980s and the 1990s, applications of representation matching arose in the context of *knowledge base construction*, which is studied in the AI community. Knowledge bases store complex types of entities and relationships, using “extended database schemas” called *ontolo-*

gies [BKD+01, HH01, Ome01, MS01, iee01]. As with databases, there is a strong need to build new knowledge bases from several component ones, and to translate data among multiple knowledge bases. These tasks require solving the *ontology matching* problem: find semantic mappings between the involved ontologies.

In the recent years, the explosive growth of information online has given rise to even more application classes that require representation matching. One application class builds *data integration systems* [GMPQ+97, LRO96, IFF+99, LKG99, FW97, KMA+98]. Such a system provides users with a *uniform query interface* to a multitude of data sources. The system provides this interface by enabling users to pose queries against a *mediated schema*, which is a virtual schema that captures the domain's salient aspects. To answer queries, the system uses a set of *semantic mappings* between the mediated schema and the local schemas of the data sources, in order to reformulate a user query into a set of queries on the data sources. A critical problem in building a data-integration system, therefore, is to supply the set of semantic mappings between the mediated- and source schemas.

Another important application class is *peer data management*, which is a natural extension of data integration. A peer data management system does away with the notion of mediated schema and allows peers (i.e., participating data sources) to query and retrieve data directly from each other. Such querying and data retrieval require the creation of semantic mappings among the peers.

Recently there has also been considerable attention on *model management*, which creates tools for easily manipulating models of data (e.g., data representations, website structures, and ER diagrams). Here matching has been shown to be one of the central operations [BHP00, RB01].

The data sharing applications described above arise in numerous real-world domains. Applications in databases have now permeated all areas of our life. Knowledge base applications are often deployed in diverse domains such as medicine, commerce, and military. These applications also play important roles in emerging domains such as e-commerce, bioinformatics, and ubiquitous computing [UDB, MHTH01, ILM+00].

Some recent developments should dramatically increase the need for and the deployment of applications that require mappings. The Internet has brought together millions of data sources and makes possible data sharing among them. The widespread adoption of XML as a standard *syntax* to share data has further streamlined and eased the data sharing process. Finally, the vision of the Semantic Web is to publish data (e.g., by marking up webpages) using ontologies, thus making data that is available on the Internet become even more structured. The growth of the Semantic Web will further fuel data sharing applications and underscore the key role that representation matching plays in their deployment.

Representation matching therefore is truly pervasive. Variations of this problem have been referred to in the literature as *schema matching, ontology matching, ontology alignment, schema reconciliation, mapping discovery, reconciling representations, matching XML DTDs*, and *finding semantic correspondences*.

1.2 Challenges of Representation Matching

Despite its pervasiveness and importance, representation matching remains a very difficult problem. Matching two representations S and T requires deciding if any two elements s of S and t of T *match*, that is, if they refer to the same real-world concept. This problem is challenging for several fundamental reasons:

- The semantics of the involved elements can be inferred from only a few information sources, typically the creators of data, documentation, and associated representation schema and data.

  Extracting semantics information from data creators and documentation is often extremely cumbersome. Frequently, the data creators have long moved, retired, or forgotten about the data. Documentation tends to be sketchy, incorrect, and outdated. In many settings such as when building data integration systems over remote Web sources, data creators and documentation are simply not accessible.

- Hence representation elements are typically matched based on clues in the *schema* and *data*. Examples of such clues include element names, types, data values, schema structures, and integrity constraints. However, these clues are often unreliable. For example, two elements that share the same name (e.g., area) can refer to different real-world entities (the location and square-feet area of the house in this case). The reverse problem also often holds: two elements with different names (e.g., area and location) can actually refer to the same real-world entity (the location of the house).

- The above clues are also often incomplete. For example, the name contact-agent only suggests that the element is related to the agent. It does not provide sufficient information to determine the exact nature of the relationship (e.g., whether the element is about the agent's phone number or her name).

- To decide that element s of representation S matches element t of representation T, one must typically examine *all* other elements of T to make sure there is no other element that matches s better than t. This *global* nature of matching adds substantial cost to the matching process.

- To make matters worse, matching is often *subjective*, depending on the application. One application may decide that house-style matches house-description, another application may decide not. Hence, the user must often be involved in the matching process. Sometimes, even the input of a single user is considered too subjective, and then a whole committee must be assembled to decide on the correct matching [CHR97].

Because of the above challenges, the manual creation of semantic mappings has long been known to be extremely laborious and error-prone. For example, a recent project at the GTE telecommunications company sought to integrate 40 databases that have a total of 27,000 elements (i.e., attributes of relational tables) [LC00]. The project planners estimated that, without the database creators, just finding and documenting the semantic mappings among the elements would take more than 12 person years.

1.3 State of the Art

The high cost of manual mapping has spurred numerous solutions, which fall roughly into two groups:

- The first group *develops standards* (i.e., common vocabularies) that all representations must conform to. This approach eliminates the need for representation matching.

  Standardization can work well for some narrowly defined domains, such as certain business areas. However, it cannot be a general solution to reconciling representations, for several reasons. First, often a domain generates multiple competing standards; this defeats the purpose of having a standard in the first place. Second, as their needs evolve, organizations must extend standards to handle unanticipated data. Extensions from different organizations are generally incompatible with each other. Third, developing standards demands consensus and takes time. This poses a serious problem for newly emerging domains.

  But most importantly, in numerous domains there is always a need to deal with data *originally created for a different purpose*. For example, in data integration, data at the sources is created independently, typically well before the need for integration arises. Such data by nature does not conform to a single domain standard. Hence, the need for representation matching will always remain.

- Since the representation matching problem does not go away, the second solution group seeks to *automate the mapping process*. Because the users must be in the loop, only semi-automatic methods can be considered. Numerous such methods have been developed, in the areas of databases, AI, e-commerce, and the Semantic Web (e.g., [MZ98, PSU98, CA99, LC00, PE95, CHR97, MBR01, MMGR02, MHH00, DR02, Cha00, MFRW00, NM00, MWJ, NM01, RHS01]; see [RB01] for an excellent survey of automatic approaches developed by the database community).

  The proposed approaches have built efficient specialized mapping strategies, and significantly advanced our understanding of representation matching. However, these approaches suffer from two serious shortcomings. First, they typically employ a single matching strategy that exploits only certain types of information and that is often tuned to only certain types of applications. As a result, the solutions have limited applicability and matching accuracy. In particular, they lack modularity and extensibility, and do not generalize well across application domains and data representations. Second, most proposed solutions can discover only 1-1 semantic mappings. They cannot find complex mappings such as "name = concat(first-name, last-name)". This is a serious limitation because complex mappings make up a significant portion of semantic mappings in practice (see Chapter 4 for more detail).

Because no satisfactory solution to representation matching exists, today the vast majority of semantic mappings is still created manually. The slow and expensive manual acquisition of mappings has now become a serious bottleneck in building information processing applications. This problem will become even more critical as data-sharing applications *proliferate* and *scale up*. As mentioned in Section 1.1, the development of technologies such as the Internet, XML, and the Semantic Web will further fuel data-sharing applications, and enable applications to share data across thousands or millions of sources. Manual mapping is simply not possible at such scales.

Hence, the development of *semi-automatic* solutions to representation matching is now truly crucial to building a broad range of information processing applications. Given that representation matching is a fundamental step in numerous such applications, it is important that the solutions be robust and applicable across domains. This dissertation develops such solutions.

1.4 Goals of the Dissertation

The central thesis of this dissertation is that, *for the representation matching problem, we can design a semi-automatic solution that builds on well-founded semantics, that is broadly applicable, and that exploits multiple types of information and techniques to maximize mapping accuracy.*

Specifically, our goals are as follows:

- Develop a *formal framework* for representation matching. The framework should define relevant notions in representation matching, such as semantic mapping, domain constraints, and user feedback. It should explain the behavior of both system and user, and expose the informal assumptions often made by matching solutions.

- Within the above formal framework, develop a solution that has *broad applicability*: the solution should handle a variety of data representations, such as relational tables, XML DTDs, and ontologies, and should discover both 1-1 and complex semantic mappings.

- Design the solution such that it *maximizes matching accuracy* by exploiting a wide range of information. First, whenever possible, the solution should exploit *previous matching activities*: it should be able to "look over the user's shoulder" to learn how to perform mapping, then propose new mappings itself. Second, any single type of syntactic clue is often unreliable, as we have shown in Section 1.2. Hence, the solution should exploit *multiple types of clues*, to achieve high matching accuracy. Third, the solution should utilize *integrity constraints* that appear frequently in application domains. Finally, because the user must be in the loop, the solution should be able to efficiently incorporate user feedback into the matching process.

To achieve these goals, we proceed with the following steps:

1. Develop a formal framework for representation matching (Chapter 2).

2. Develop and evaluate a solution for discovering 1-1 mappings in the context of data integration, for XML data. Design the solution such that it can exploit multiple types of information (Chapter 3).

3. Extend the solution to discovering complex semantic mappings. Evaluate the solution in the context of matching relational representations for data translation (Chapter 4).

4. Extend the solution to finding 1-1 mappings for ontologies, a representation that is more complex than relational and XML ones (Chapter 5).

1.5 Overview of the Solutions

We now outline our solutions to the above problem steps.

1.5.1 Formalizing Representation Matching

In this dissertation we shall consider several types of matching, all of which require solving the following fundamental problem: *given two representations S and T, for each element s of S, find the most similar element t of T, utilizing all available information.* This includes any information about the representations and their domains, such as data instances, integrity constraints, previous matchings, and user feedback.

The solutions that we shall develop for the above problem compute for each element pair $s \in S$ and $t \in T$ a numeric value v. The value v indicates the degree of similarity between s and t; the higher v is, the more similar s and t are. We refer to a tuple (s,t,v) as a *semantic mapping*. Then for each element $s \in S$, the solutions return the semantic mapping that involves s and has the highest similarity value v.

Many works on representation matching [MBR01, BM01, DR02, CA99, BCVB01, LC94, CHR97, RHS01, LG01] have also considered the above problem and solution approach. However, most of them do not define the problem *formally*. They do not make clear what "most similar element" means, nor do they state the implicit assumptions that underlie their solutions.

We believe a formal framework for representation matching is important because it facilitates the evaluation of solutions. It also makes clear to the users what a solution means by a match, and helps them evaluate the applicability of a solution to a given matching scenario. Furthermore, the formalization may allow us to leverage special-purpose techniques for the matching process. An important contribution of this dissertation is that we developed such a framework, which defines the matching problem and explains the solutions that we develop.

Our framework assumes that the user knows a conceptualization of the domain in terms of a representation $\mathcal{U}$, and that he or she knows a similarity measure $\mathcal{S}$ that is defined over the concepts in $\mathcal{U}$. The framework further assumes that the user can map any two concepts s and t of the input representations into semantically equivalent concepts $\mathcal{M}(s)$ and $\mathcal{M}(t)$ in $\mathcal{U}$. In Chapter 2 we discuss the motivations leading to these assumptions.

Given the above assumptions, we can formally state the matching problem as follows: for each s of S, find t that maximizes $\mathcal{S}(\mathcal{M}(\mathrm{s}),\mathcal{M}(\mathrm{t}))$ among all elements of T – in other words, find t such that the similarity value computed by $\mathcal{S}$ between the equivalent concepts in $\mathcal{U}$ is the highest. Thus, when a solution produces a semantic mapping (s,t,v), we can interpret v to be the best estimation of the true similarity value $\mathcal{S}(\mathcal{M}(s),\mathcal{M}(t))$ that the solution could produce.

Notice that in estimating the true similarity values, the solution often has only *partial* knowledge about the representations and their domains. It may know the data types, structures, names, and some data instances associated with representation elements, as well as a set of integrity constraints and some knowledge about $\mathcal{S}$ and $\mathcal{U}$. It must utilize such knowledge and rely largely on the similarities of *syntactic clues* (e.g., element names and data instances) to estimate *semantic similarities* (as represented by $\mathcal{S}$). However, this estimation makes sense only under the assumption that the syntactic similarity is positively and strongly correlated with the semantic similarity. This assumption is frequently made but rarely stated explicitly by previous works in representation matching.

Our framework thus explains that what matching solutions attempt to do is to estimate true similarity values as represented by $\mathcal{S}$. In the next three sections, we study *how* to obtain *good* estimates of true similarity values, in the context of specific matching problems.

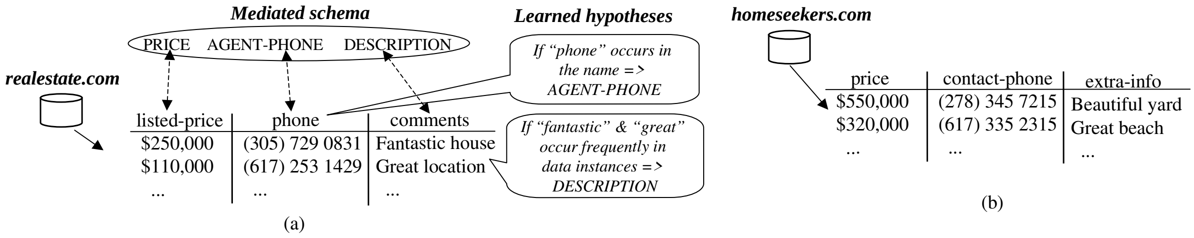


Figure 1.1: Once LSD has trained a set of learners on source *realestate.com* in (a), it can apply the learners to find semantic mappings for source *homeseekers.com* in (b).

1.5.2 1-1 Matching for Data Integration

We begin by considering the basic 1-1 matching problem that is described in the previous section, but in the context of data integration systems. We choose data integration because it is an important data management application and because it provides a general problem setting, the solution of which could be generalized to other applications such as data translation and ontology matching (Chapters 4-5).

Recall from Section 1.1 that a data integration system enables users to retrieve data from a multitude of sources by posing queries on a *mediated schema*. To answer queries, the system must know the semantic mappings between the mediated schema and the schemas of the data sources. Our goal is to develop a solution to semi-automatically create these mappings.

Below we briefly describe the solution as embodied in the LSD system that we have developed. The key idea underlying LSD is that after the schemas of a few data sources have been manually mapped to the mediated schema, it should be able to learn from the manual mappings to successfully propose mappings for subsequent data sources.

Example 1 Consider a data-integration system that helps users find houses on the real-estate market. Suppose that the system has the mediated schema shown in Figure 1.1.a. The mediated schema (which is a simplification of a real one) consists of three elements: PRICE, AGENT-PHONE, and DESCRIPTION.

Suppose further that we have selected source *realestate.com* and manually specified the 1-1 mappings between the schema of this source and the mediated schema. This amounts to specifying the three dotted arrows in Figure 1.1.a. The first arrow, for example, states that source-schema element listed-price matches mediated-schema element PRICE. (We use THIS FONT and this font to refer to the elements of mediated and source schemas, respectively.)

Once we have specified the mappings, there are many different types of information that LSD can glean from the source schema and data to train a set of *learners*. A learner can exploit the *names* of schema elements: knowing that phone matches AGENT-PHONE, it could hypothesize that if an element name contains the word "phone", then that element is likely to be AGENT-PHONE. The learner can also look at example phone numbers in the source data, and learn the *format* of phone numbers. It could also learn from *word frequencies*: it could discover that words such as "fantastic" and "great" appear frequently in house descriptions. Hence, it may hypothesize that if these words appear frequently in the data instances of an element, then that element is likely to be

DESCRIPTION. As yet another example, the learner could also learn from the *characteristics of value distributions*: it can look at the average value of an element, and learn that if that value is in the thousands, then the element is more likely to be price than the number of bathrooms.

Once the learners have been trained, we apply LSD to find semantic mappings for new data sources. Consider source *homeseekers.com* in Figure 1.1.b. A word-frequency learner may examine the word frequencies of the data instances of element extra-info, and recognize that these data instances are house descriptions. Based on these predictions, LSD will be able to predict that extra-info matches DESCRIPTION.□

As described, machine learning provides an attractive platform for finding semantic mappings. However, applying it to our domain raises several challenges. The first challenge is to decide which learners to employ in the training phase. A plethora of learning algorithms have been described in the literature, each of which has strengths in learning different types of patterns. A key distinguishing factor of LSD is that we take a *multi-strategy learning* approach [MT94]: we employ a multitude of learners, called *base learners*, then combine the learners' predictions using a *meta-learner*. An important feature of multi-strategy learning is that our system is *extensible* since we can add new learners that have specific strengths in particular domains, as these learners become available.

The second challenge is to exploit integrity constraints that appear frequently in database schemas and to incorporate user feedback on the proposed mappings in order to improve accuracy. We extended multi-strategy learning to incorporate these. As an example of exploiting integrity constraints, suppose we are given a constraint stating that the value of the mediated-schema element HOUSE-ID is a key for a real-estate entry. In this case, LSD would know not to match num-bedrooms to HOUSE-ID because the data values of num-bedrooms contain duplicates, and thus it cannot be a key. As an example of incorporating user feedback, LSD can benefit from feedback such as "ad-id does not match HOUSE-ID" to constrain the mappings it proposes.

The third challenge arises from the nature of XML data. We built LSD to match both relational and XML data. However, while experimenting with LSD, we realized that none of its learners could handle the hierarchical structure of XML data very well (see Chapter 3). Hence, we developed a novel learner, called the *XML learner*, that handles hierarchical structure and further improves the accuracy of our mappings.

We evaluated LSD on several real-world data integration domains. The results show that with the current set of learners, LSD already obtains predictive accuracy of 71-92% across all domains. The experiments show the utility of multi-strategy learning and of exploiting domain constraints and user feedback for representation matching.

1.5.3 Complex Matching

LSD provides a powerful matching solution that can exploit multiple types of information. However, it discovers only 1-1 semantic mappings such as "DESCRIPTION = comments". Since complex mappings such as "NUM-BATHS = full-baths + half-baths" and "ADDRESS = *concat*(city,zipcode)" are also widespread in practice, we developed the COMAP system, which extends LSD to find both 1-1 and complex mappings.

We explain COMAP using the familiar data-integration setting. Here, for each mediated-schema element s, COMAP considers finding the best semantic mapping (be it 1-1 or complex) over the elements of a given source schema T. To do this, COMAP *quickly* finds a small set of best candidate mappings for s. Next, it "adds" the newly found mappings to source schema T so that they can be treated as additional "composite" elements of T. For instance, suppose T consists of three elements price, city, and state. Suppose further that the best candidate mappings for mediated-schema element ADDRESS are $\{concat(\text{city}, \text{state}), concat(\text{price}, \text{city})\}$. Then $concat(\text{city}, \text{state})$ would be a new "composite" element of T, whose data instances would be obtained by concatenating those of city and state.

Once candidate mappings for all mediated-schema elements have been computed and added to source schema T, GLUE applies LSD (which is modified to fit the complex-matching context) to find 1-1 semantic mappings between the mediated schema and the expanded schema of T. Continuing with the above ADDRESS example, if LSD matches ADDRESS with the composite element that corresponds to the candidate $concat(\text{city}, \text{state})$, then GLUE would return this candidate as the best mapping for ADDRESS.

As described, reducing complex matching to 1-1 matching provides an elegant framework that can utilize all techniques previously developed for 1-1 matching (including our LSD work). However, it raises several challenges. The first challenge is how to efficiently search the vast (often infinite) space of all 1-1 and complex mappings, to find the best candidate mappings. COMAP solves this problem by *breaking up* the search space: it employs multiple *searchers* – each exploits a certain type of information to quickly find a small set of promising candidate mappings – then returns the union of the mappings found by all searchers. Multisearch therefore is a natural extension of multistrategy learning employed in LSD. As such, it provides COMAP with a high degree of modularity and extensibility.

The second challenge is how to implement the searchers and evaluate candidate mappings. COMAP provides a default implementation using *beam search*, and uses machine learning and statistical techniques to evaluate candidate mappings. Naturally, searchers can choose not to use the default implementation if a more suitable technique becomes available.

In Chapter 4 we provide a detailed description of COMAP, as well as of the experiments we conducted with it on real-world data. There, we explain its implementation for matching relational data and the necessary changes to extend it to matching XML data.

1.5.4 Ontology Matching

Together, LSD and COMAP provide a solution that covers both 1-1 and complex matching. We must extend this solution in two important aspects. First, the solution matches only relational and XML representations; we extend it to also match ontologies. Ontologies have proven very popular as representations of data. They play a key role in constructing knowledge bases and marking up data on the proposed Semantic Web [BKD$^+$01, BLHL01]. Hence, ontology matching should be an integral part of any general matching solution.

Second, the solution we developed can exploit a broad range of information, including schema and data information, integrity constraints, past matchings, and user feedback. However, it has not considered exploiting any information that is available about the similarity measure defined over representation elements (i.e., measure $\mathcal{S}$ defined in Section 1.5.1). In many practical settings, the user does know the similarity measure and can supply it as a problem input. Hence, we consider such settings and extend our solution to exploit user-supplied similarity measures, to improve the estimation of true similarity values.

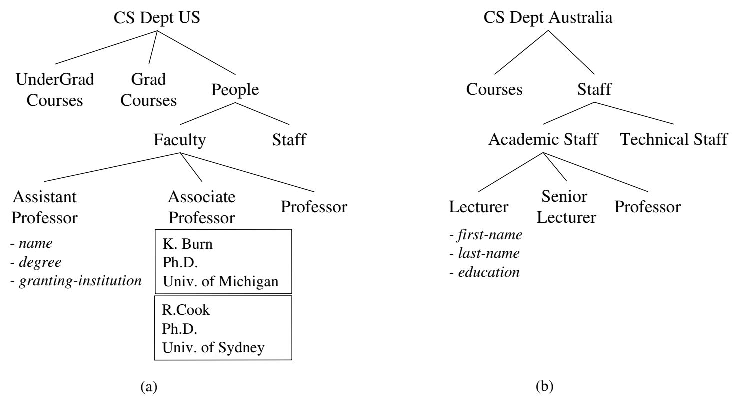


Figure 1.2: Sample ontologies in the CS department domain

We have developed the GLUE system, which extended LSD and COMAP in the two ways described above. The current GLUE focuses on matching *taxonomies*, which are central components of ontologies. A taxonomy is a tree where each node represents a *concept* and each concept is a specialization of its parent. Figure 1.2 shows two sample taxonomies for the CS department domain. Given two taxonomies and a user-defined similarity measure, GLUE finds for each concept node in a taxonomy the most similar concept node in the other taxonomy.

The first challenge GLUE faces is how to compute the similarity of any two concepts in the taxonomies. A key observation we made is that many practical similarity measures can be defined based solely on the *joint probability distribution* of the concepts involved. For example, the well-known *Jaccard* measure [vR79] computes the similarity between two concepts A and B to be $P(A \cap B)/P(A \cup B)$, which can be re-expressed in terms of the joint distribution of A and B.

GLUE assumes that the user-supplied similarity measure also has the above property. Then, instead of attempting to estimate specific similarity values directly, GLUE focuses on computing the joint distributions. After that, it is possible to compute any similarity measure such as the *Jaccard* coefficient as a function of the joint distributions. GLUE therefore has the significant advantage of being able to work with a variety of similarity functions. We apply multistrategy learning as described in LSD to compute the joint distributions of the concepts. We describe this process in detail in Chapter 5.

The second challenge for GLUE is that the taxonomy structure gives rise to matching heuristics that have not been considered in the context of relational and XML data. An example heuristic is that two nodes are likely to match if their parents and descendants also match. Such heuristics occur frequently in practice, and are very commonly used when manually mapping between ontologies. Previous works have exploited only one form or another of such knowledge, in restrictive settings [NM01, MZ98, MBR01, MMGR02].

In the GLUE context we developed a unifying approach to incorporate all such types of heuristic. Our approach is based on *relaxation labeling*, a powerful technique that has been used successfully in vision and image processing [HZ83], natural language processing [Pad98], and hypertext classification [CDI98]. We show that relaxation labeling can be adapted efficiently to our context, and that it can successfully handle a broad variety of heuristics. Chapter 5 describes relaxation labeling and the rest of GLUE in detail. It also describes experiments we conducted on real-world domains to validate GLUE.

1.6 Contributions of the Dissertation

Up to the time of this dissertation, most works have employed only hand-crafted rules to match representations. Several recent works have advocated the use of learning techniques (see Chapter 6 on the related work). However, in general it is not clear how to reconcile the two approaches. There is also an implicit and gradual realization that multiple types of information must be exploited to maximize matching accuracy. However, it is not clear what is a good way to exploit them and combine their effects. Finally, the vast majority of works have considered only 1-1 matching. It is not clear what is a good way to attack the problem of complex matching, and whether a solution that unifies both 1-1 and complex matching can be developed.

- The most important contribution of this dissertation is a solution architecture that provides answers to the above questions. The solution advocates the use of *multiple independent modules*, each exploiting a certain type of information. Meta-learning techniques then utilize training data to find the best way to combine module predictions. As such, the solution provides a unifying framework for previous approaches: its modules can employ rules, learning techniques, or any other techniques deemed most suitable for exploiting the information at hand. The multi-module nature also makes the solution easily extensible and customized to any particular application domain.

  The solution combines both 1-1 and complex matching. Furthermore, it provides a unifying and efficient approach to incorporate a broad range of integrity constraints and domain heuristics. It can utilize previous matching activities and incorporate user feedback, as we show with LSD. It can handle a variety of representations, including relational, XML, and ontologies. Finally, it can also handle a broad range of similarity measures, an ability that is missing from most previous matching solutions.

- Another major contribution of the dissertation is a semantics framework that formally defines representation matching. The framework explains the solutions commonly adopted in practice, and exposes the implicit assumptions these solutions often make.

- The dissertation also makes several contributions to the field of machine learning. It introduces representation matching as an important application of multistrategy learning. It develops the XML Learner, a novel approach that exploits the hierarchical nature of XML data to achieve better classification accuracy than existing learning approaches. Finally, it significantly extends relaxation labeling to address the problem of learning to label interrelated instances.

1.7 Outline

The next chapter describes the representation matching problems that we consider in this dissertation. It elaborates on the ideas outlined in Section 1.5.1. The following three chapters – Chapters

3-5 – describe LSD, COMAP, and GLUE, respectively. They elaborate on the ideas outlined in Sections 1.5.2-1.5.4. Chapter 6 reviews existing solutions and discuss how they relate to ours. Finally, Chapter 7 summarizes the dissertation and discusses directions for future research.

The dissertation is structured so that each chapter is relatively self-contained. The impatient reader can read Chapters 1, 2, and 6 to quickly understand the main ideas and their relation to existing works. The remaining chapters can be read subsequently, as time permits.

Parts of this dissertation have been published in conferences and journals. In particular, the LSD system (Chapter 3) is described in a SIGMOD-2001 paper [DDH01], and the GLUE system (Chapter 5) is described in a WWW-2002 paper [DMDH02]. The key ideas of a multi-strategy learning approach are described in a Machine Learning Journal paper [DDH03].

Chapter 2

PROBLEM DEFINITION

This chapter defines representation matching. We begin by introducing data representations. Next, we describe several specific problems of finding semantic mappings between representations. These problems are important in that they arise frequently in real-world applications. Hence they will be considered in depth in subsequent chapters. Finally, using the above problems as the "spring-boards", we study and develop a formal semantics for representation matching.

2.1 Data Representations

A *data representation* specifies a particular way to structure the data. Prime examples of representations include relational schemas, XML DTDs, ontologies, object-oriented representations, and ER models.

For the purpose of this dissertation, a data representation consists of a finite set of *representation elements* (or *elements* for short). The elements refer to *syntactic constructs* of representations, such as attributes and tables in relational schemas; XML elements and attributes in XML DTDs; and concepts, attributes, and relations in ontologies (see below for more detail). Each element is associated with a universe of *data instances*. An element therefore defines a data type. Given two representations, our goal is to find semantic correspondences between their elements.

Below we introduce several well-known representations. Our goal is to illustrate the notions of representation, element, and data instances. In subsequent chapters we describe these representations in detail.

Relational Schema: A relational schema consists of multiple *tables*, each with a set of *attributes*. The attributes are often referred to as *columns*. Figure 2.1.a shows a relational schema *S* which has one table: LISTINGS. This table has four columns which correspond to the four attributes area, listed-price, agent-address, and agent-name.

Figure 2.1.b shows a relational schema *T* which has two tables. This schema has ten elements: eight attributes (e.g., location) and two tables (e.g., HOUSES). Instances of element location include "Atlanta, GA" and "Raleigh, NC". Instances of element HOUSES include tuples ("Atlanta, GA", "$360,000", "32") and ("Raleigh, NC", "$430,000", "15").

XML DTD: This representation is increasingly used for data exchange across sources. An XML document consists of pairs of matching open- and close-*tags*, enclosing *elements* [XML98]. Each element may also enclose additional sub-elements or uniquely valued attributes. A document contains a unique root element, in which all others are nested. Figure 2.2.a shows a house listing stored as an XML document. In general, an XML element may also have a set of *attributes*. For the purpose of this dissertation we treat its attributes and sub-elements in the same fashion.

Representation S

LISTINGS

area	listed-price	agent-address	agent-name
Denver, CO	$550,000	Boulder, CO	Laura Smith
Atlanta, GA	$370,800	Athens, GA	Mike Brown

(a)

Representation T

HOUSES

location	price	agent-id
Atlanta, GA	$360,000	32
Raleigh, NC	$430,000	15

AGENTS

id	name	city	state	fee-rate
32	Mike Brown	Athens	GA	3%
15	Jean Laup	Raleigh	NC	2%

(b)

Figure 2.1: Two relational representations.

```
<house-listing>
    <location>Miami, FL</location>
    <contact-info>
        <name>Kate Richardson</name>
        <address>47 16th Ave., Miami, FL</address>
    </contact-info>
</house-listing>
```

(a)

```
<!ELEMENT  house-listing  (location?, contact-info)>
<!ELEMENT  location  (#PCDATA)>
<!ELEMENT  contact-info  (name, (address | phone))>
<!ELEMENT  name  (#PCDATA)>
<!ELEMENT  address  (#PCDATA)>
<!ELEMENT  phone  (#PCDATA)>
```

(b)

Figure 2.2: (a) A sample XML document, and (b) the XML DTD that the document conforms to.

We consider XML documents with associated DTDs (Document Type Descriptors). A DTD is a BNF-style grammar that defines legal elements and relationships between the elements. Figure 2.2.b shows a DTD that the XML document in Figure 2.2.a conforms to. This DTD defines six representation elements: house-listing, location, ..., phone. A data instance of location is "$\langle location \rangle$ Miami, FL$\langle / location \rangle$", and a data instance of contact-info is the following text fragment:

```
<contact-info>
    <name>Kate Richardson</name>
    <address>47 16th Ave., Miami, FL</address>
</contact-info>
```

Ontologies: Ontologies are commonly used to construct knowledge bases [BKD$^+$01] and have been proposed as a tool for marking up data on the Semantic Web [BLHL01]. An *ontology* specifies a conceptualization of a domain in terms of concepts, attributes, and relations [Fen01]. The *concepts* are typically organized into a *taxonomy tree* where each node represents a concept and each concept is a specialization of its parent. Figures 2.3.a-b show sample taxonomies for the CS department domain (which are simplifications of real ones).

Each concept in a taxonomy is associated with a set of *instances*. For example, concept Associate-Professor has instances "Prof. Cook" and "Prof. Burn" (Figure 2.3.a). Each con-

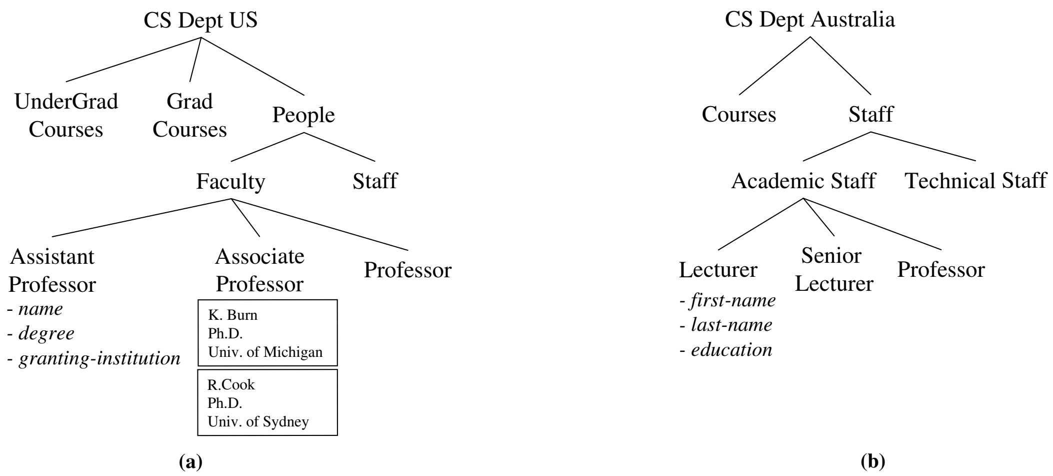


Figure 2.3: Sample ontology representations.

cept is also associated with a set of *attributes*. Concept Associate-Professor in Figure 2.3.a has attributes name, degree, and granting-institution. An *instance* that belongs to a concept has fixed attribute values. Instance "Professor Cook" has values name = "R. Cook", degree = "Ph.D.", and so on. An ontology also defines a set of *relations* among its concepts. For example, a relation AdvisedBy(Student,Professor) might list all instance pairs of Student and Professor such that the former is advised by the latter (this relation is not shown in Figure 2.3).

For the ontology in Figure 2.3.a, representation elements consist of concepts (e.g., UnderGrad-Courses and Grad-Courses), attributes (e.g., name and degree), and relationships (e.g., AdvisedBy(Student,Professor)).

2.2 Representation Matching

We now discuss the problem of finding semantic mappings between data representations. First we describe a basic matching problem that has been considered by many previous works in representation matching. Then we describe three specific matching problems that frequently arise in practice and that extend the above basic problem. We develop solutions for these problems in subsequent chapters.

2.2.1 The Basic 1-1 Matching Problem

Virtually all matching scenarios that have arisen in practice require solving the following fundamental problem:

Problem 1 (1-1 Matching) *Given two representations S and T, for each element s of S, find the most semantically similar element t of T, utilizing all available information (such as data instances, integrity constraints, and user feedback).*

The above problem is often referred to as an *one-to-one (1-1) matching* problem, because it matches each element s with a *single* element t. An instance of this problem is to match representations S and T in Figure 2.1. An example of matchings here would be "element location of S matches element area of T". We refer to such a matching as a *semantic mapping* and denote it as (location,area) or "location = area". (In the rest of this dissertation, we shall use the words "mapping" and "matching" interchangeably.)

Problem 1 is key in that solving it would allow the user (i.e., the application builders) to quickly locate semantically related elements, then examine these elements in detail to discover the exact relationships among them. As such, this problem has been the focus of numerous works in representation matching (e.g., [MZ98, PSU98, CA99, LC00, PE95, CHR97, MBR01, MMGR02, DDH01, DMDH02, DR02, MBR01]).

In the rest of this subsection we shall first define the types of information that can be input to Problem 1. Next, we describe the type of output that we require solutions to this problem to produce. Finally, we discuss a procedure to evaluate such a solution output.

Input Information: In general, the input to Problem 1 can include *any* type of knowledge about the representations to be matched and their domains. The specific types of knowledge that we consider in this dissertation are:

- *Schema information:* This type of input refers to representation elements, their names, textual descriptions, structures, relationships among the elements, and so on.

- *Data instances:* This type of input refers to the instances of representation elements. In some applications such as view integration for schema design, data instances may not be available for matching purposes. However, in numerous other applications, such as data integration and translation, data instances are typically available. Our solutions as presented in Chapters 3-5 work in both cases.

- *Previous matchings:* These matchings refer to the semantic mappings that have been previously created for representations in the same domains.

- *Integrity constraints and domain heuristics:* Constraints and heuristics encode additional knowledge about a domain and the semantic correspondences between schemas in the domain. They are typically specified by the user, only once, at the beginning of the matching process. An example of integrity constraint is "if an element matches house-id then that element is a key". An example of heuristic is "the likelihood that two elements match increases if their neighbors also match".

- *User feedback:* Feedback on the matchability of elements between the representations is another typical source of input. We treat user feedback as temporary integrity constraints that apply only to the current matching scenario.

- *Similarity measures:* As noted in Section 1.5.4 of Chapter 1, in many practical settings the user does have a well-defined notion of similarity. Hence, we consider also the cases where the user can supply such a similarity measure as an input to the matching problem.

We note that most previous approaches in representation matching have considered schema information, integrity constraints, and user feedback as the input to the matching problem. Several recent approaches have also considered the use of data instances and domain heuristics [BM01, BM02, MMGR02, NM01, MBR01]. However, very few approaches have considered the use of previous matchings and similarity measures [RHS01, DR02]. The general matching solution that we shall develop is distinguished in that it can efficiently handle all the above different types of input information.

Solution Output: We require that a solution to Problem 1 produce semantic mappings with *confidence scores* attached. An example of such mappings is (address,location,0.8), which says that address matches location with confidence 0.8. The confidence scores take values in the range [0,1]; the higher a confidence score, the more certain the solution is in its mapping prediction.

In particular, for each element s of representation S, we require the solution to produce a list of k best mappings, sorted in decreasing order of confidence score, where k is a pre-specified small number (say, in the range [1,10]). For example, if $k = 3$, a solution may produce the following output list for element address: $\{(\mathsf{location}, 0.7), (\mathsf{area}, 0.2), (\mathsf{nearby}, 0.1)\}$. Here, a list element such as (location,0.8) is a shorthand for the semantic mapping (address,location,0.8).

We observe that, in practice, many times it is the case that the correct mapping for an element is in the list of k best mappings as produced by a solution, but is not at the top of the list (i.e., is not the mapping with the highest confidence score). If k is small (e.g., under 10), then the user can easily examine the list of k mappings to determine if the correct mapping is present, without spending too much effort. In other words, the *cognitive load* of finding the correct mapping is negligible. Thus, by requiring a solution to return a *small list* of mappings instead of a *single mapping* per element, we increase the accuracy of the solution without imposing much additional burden on the user.

Evaluation of a Solution Output: The above observation leads to the following procedure that we use to evaluate a solution output:

1. We *manually* identify for each element s of S the correct mapping, that is, the most semantically similar element from T.

2. Next, for each element s of S, we examine the list of k best mappings that a matching solution has produced, and judge it to be *correct* if it contains the correct mapping as identified in Step 1.

3. Finally we return the ratio of correct lists over the total number of lists produced as the *matching accuracy* of the solution output.

Many alternative methods to evaluate a solution output have been proposed in the literature (see [DMR02] for a survey). We choose the above method because it is conceptually simple and yet sufficient to quantify the accuracy of a matching solution.

2.2.2 Extensions of the Basic Matching Problem

We have described the basic matching problem in detail, and now are in a position to introduce the specific problems that we shall consider in the rest of the dissertation.

Our ultimate goal is to develop a general solution that can handle a broad range of matching scenarios. As the first step toward this goal, we consider 1-1 matching in the context of data integration. We choose data integration because it is an important application and because it provides a very general problem setting that can be adapted to many other application contexts. Recall from Chapter 1 that a data integration system translates queries and data between a mediated schema and a set of source schemas, using a set of semantic mapping. Our problem then is to find such mappings:

Problem 2 (1-1 Matching for Data Integration) *Given source representations $S_1, S_2, \ldots, S_n$ and a mediated representation T, for each element s of S_i, $i \in [1,n]$, find the most similar element t of T.*

For auxiliary information, here we consider exploiting previous matching activities, data instances, integrity constraints and domain heuristics, and user feedback. In Chapter 3 we develop a solution for this problem, in the context of XML data. (Note that for the purpose of data integration, we may also need to solve the reverse problem of finding mappings from the mediated representation into the source ones. However, a solution to Problem 2 should provide the basis for solving that problem.)

Problem 2 focuses on finding only 1-1 semantic mappings such as "location = area" and "listed-price = price". In practice, however, *complex mappings* such as "address = *concat*(city,state)" also make up a significant portion of mappings. Hence, as the second step toward our goal, we study the problem of finding such mappings. In its simplest form, a complex mapping relates an element in one representation to a formula constructed over the elements of the other representation, using a set of given operators and rules:

Problem 3 (Complex Matching) *Let S and T be two data representations. Let $O = \{o_1, o_2, \cdots, o_k\}$ be a set of operators that can be applied to the elements of T according to a set of rules R to construct formulas. For each element s of S, find the most similar element t, where t can be either an element of T or a formula constructed from the elements of T, using O and R.*

For the above problem, we consider exploiting the same types of auxiliary information as in the problem of 1-1 matching for data integration. To illustrate the notion of formulas, consider representation T in Figure 2.1.a. Suppose set O contains only the operator *concat* (that concatenates its arguments), and set R contains the rule "an element participates at most once in a concatenation". Then examples of formulas are *concat*(location,price), *concat*(city,state), and *concat*(name,city,state). Element agent-address of representation S in Figure 2.1.a would best match formula *concat*(city,state).

In Chapter 4, we shall consider complex matching in the context of relational and XML data. We extend our solution to the problem of 1-1 matching to address this problem. Note that this problem subsumes the basic 1-1 matching problem (Problem 1). The former reduces to the latter if we restrict the set of operators O to be empty.

The solutions to Problems 2-3 cover both 1-1 and complex matching, but only for relational and XML data. As the third, and final, step toward our goal of a generic matching solution, we extend the developed solution to consider ontology matching. This matching scenario arises in many applications, including knowledge base construction and applications on the Semantic Web [BLHL01]. We focus on an important task in ontology matching, which is to match taxonomies of concepts (see Section 2.1 for a definition of taxonomy):

Problem 4 (1-1 Matching for Taxonomies) *Given two taxonomies of concepts S and T, for each concept node s of S, find the most similar concept node t of T.*

The auxiliary information for this problem is the same as that for Problems 2-3, except that we also consider a user-supplied similarity measure as part of the input. As discussed earlier, there are many practical matching scenarios where the user does have a well-defined notion of similarity between the concepts. Hence, to be truly practical, our solution should consider such scenarios and use the supplied similarity measure to obtain better mappings.

2.3 A Semantics for Representation Matching

In the previous section we introduced a set of matching problems to be considered in the dissertation. However, we have defined them only *informally*. In this section, we provide formal definitions for the problems. We have argued in Section 1.5.1 (Chapter 1) that such formalization is important for the purposes of evaluating, comparing, and further developing the solutions.

In what follows we develop formal definitions for the basic 1-1 matching and the complex matching scenarios (Problems 1 and 3). The formal definitions for other scenarios (Problems 2 and 4) follow naturally from the above definitions.

2.3.1 Formal Semantics for 1-1 Matching

We begin by developing several basic notions that underlie the representation matching process:

User Representation $\mathcal{U}$ and Mapping Function $\mathcal{M}$: Virtually all works in representation matching have made the fundamental assumption that the user can accept or reject a given semantic mapping (s,t,v), where s and t are representation elements and v is a confidence score. This capability suggests that the user must *understand* the meaning of elements s and t.

To formalize this notion of "understanding", we assume that the user knows a domain representation $\mathcal{U}$, and that he or she can map any element of representations S and T into a semantically equivalent element in $\mathcal{U}$. We characterize this mapping process with a function $\mathcal{M}$. The notations $\mathcal{M}(s)$ and $\mathcal{M}(t)$ then denote the elements in $\mathcal{U}$ that are equivalent to s and t, respectively.

Intuitively, the two elements s of representation S and $\mathcal{M}(s)$ of representation $\mathcal{U}$ are semantically equivalent if they refer to the same concept in the universe. Formally, let $\mathcal{W}$ be the universe. We define an *interpretation* of a representation to be a function that maps elements of the representation into concepts of $\mathcal{W}$. Then the statement "element s of S is semantically equivalent to element $\mathcal{M}(s)$ of $\mathcal{U}$" means that there exists interpretations I_S for S and $I_{\mathcal{U}}$ for $\mathcal{U}$ such that they map s and $\mathcal{M}(s)$ into the same concept in $\mathcal{W}$. We denote this meaning as $\{I_S, I_{\mathcal{U}}\} \models \{s = \mathcal{M}(s)\}$, using the notations of mathematical logic.

We can now formalize the notion of mapping function $\mathcal{M}$ as follows:

Definition 2 (Mapping Function $\mathcal{M}$) *Let S and T be two given representations, and let $\mathcal{U}$ be a user representation. Function $\mathcal{M}$ pairs elements from S and T with those in $\mathcal{U}$ such that there exist interpretations I_S, I_T, $I_{\mathcal{U}}$ (for S, T, and $\mathcal{U}$, respectively) that satisfy:*

- $\forall s \in S, \{I_S, I_{\mathcal{U}}\} \models \{s = \mathcal{M}(s)\}$

- $\forall t \in T, \{I_T, I_{\mathcal{U}}\} \models \{t = \mathcal{M}(t)\}$

A Similarity Function $\mathcal{S}$ over $\mathcal{U}$: Another fundamental assumption underlying most representation matching works is that the user can judge if two given elements s and t are indeed most similar. This assumption suggests that the user does have a notion of *semantic similarity*. Since the user has a domain representation $\mathcal{U}$, a reasonable way to formalize the above notion is to assume the existence of a user-defined similarity function $\mathcal{S}$ over the concepts of $\mathcal{U}$. Then for any pair of concepts e and f in $\mathcal{U}$, $\mathcal{S}(e, f)$ returns a value that indicates the degree of similarity between e and f. The larger this value, the higher the similarity. Without loss of generality, we can assume that $\mathcal{S}$ takes value in the interval [0,1].

We now show that the above three notions – user representation $\mathcal{U}$, mapping function $\mathcal{M}$, and similarity function $\mathcal{S}$ – are sufficient to formally explain many important aspects of the representation matching process:

- First, we can formally state the basic 1-1 matching problem (Problem 1) as follows: *given two representations S and T, for each element s of S, find element t of T that maximizes* $\mathcal{S}(\mathcal{M}(s), \mathcal{M}(t))$.

- Second, we can provide a conceptual explanation for the working of most matching tools, including those that we shall develop in Chapters 3-5. Obviously, if a matching tool knows $\mathcal{U}$, $\mathcal{M}$, and $\mathcal{S}$, then it can trivially solve the basic 1-1 matching problem. In most practical cases, however, the matching tool does not have access to these entities, which exist only in the user's head. Often, the tool has access to only the *syntactic clues* in the schema and data. Examples of such clues include element names, data types, the relationships among the elements, and data instances. Thus, in solving the above matching problem, we can think about most proposed solutions as trying to *approximate* true similarity values using the syntactic clues: for any two elements s and t, the solutions approximate $\mathcal{S}(\mathcal{M}(s), \mathcal{M}(t))$ using the syntactic clues of s and t.

- Third, we can explain the meaning of the output of a matching tool. We have mentioned that most proposed solutions produce semantic mappings of the form (s, t, v): element s matches element t with confidence score v. In our semantics framework, such a mapping simply means that the best approximation of the similarity $\mathcal{S}(\mathcal{M}(s), \mathcal{M}(t))$ that the tool can compute is v.

- Fourth, our semantics framework makes clear that matching solutions that approximate semantic similarity using syntactic information rely on the following crucial assumption:

  Assumption 1 *The more similar the syntactic clues of two elements s and t, the more semantically similar the two elements are.*

  In other words, semantic similarity is strongly and positively correlated to syntactic similarity. This assumption is implicit in many current matching solutions. Making it explicit – as done in our framework – brings several important benefits. First, it helps the user decide if a particular matching problem satisfies the assumption, and if not, how the problem can be transformed into one that does. Second, it suggests that formal models of the correlation between syntax and semantics can be investigated; exploiting such models, whenever available, may further improve matching accuracy. And finally, it helps to formally explain the input to the matching problem, as shown below.

- Finally, our semantics framework provides a clean definition for the input to matching problems. We argue that many possible input types can be explained as *knowledge about five entities*: user representation $\mathcal{U}$, the functions $\mathcal{M}$ and $\mathcal{S}$, the syntactic clues, and Assumption 1 which relates syntactic similarity to semantic similarity.

  For example, element names and data instances are knowledge about the syntactic clues. The user-defined similarity function which is an input to the taxonomy matching problem (Problem 4) is just the $\mathcal{S}$ function; and integrity constraints (Section 2.2.1) provide knowledge about Assumption 1. As yet another example, suppose representation S has an element contact-phone whose instances are phone numbers without area code: "234 5689", "453 1264", etc. Suppose the user knows that S is related to the Seattle metro area. Hence he or she prefixes all above phone numbers with area code "206" before attempting to match contact-phone with other elements of representation T. In this case, the input "206" provides some knowledge about function $\mathcal{M}$ and user representation $\mathcal{U}$. Specifically, it says that $\mathcal{M}$ pairs contact-phone with the concept "phone numbers of Seattle metro area" in user representation $\mathcal{U}$.

  An important benefit of explaining input types as knowledge about the five developed entities is that it suggests a methodology for input extraction from the user: systematically examine each entity in turn, to find out what the user knows about it. This method enables extracting as much relevant knowledge from the user as possible, for the goal of maximizing matching accuracy.

2.3.2 Formal Semantics for Complex Matching

The semantics framework for 1-1 matching that we have developed in the previous section can be extended to cover complex matching in a straightforward manner. Recall that a complex mapping relates an element s of a representation S to a formula F that is constructed over the elements of another representation T, using a set of operators O and rules R. Hence, the only problem we face here is to define how to map formula F to a semantically equivalent concept in user representation $\mathcal{U}$. In other words, we must extend the mapping function $\mathcal{M}$ so that $\mathcal{M}(F)$ is well-defined.

To do this, we assume that the operators of O and rules of R are well-defined *over the user representation* $\mathcal{U}$. With this assumption, $\mathcal{M}(F)$ can be defined recursively in a trivial manner: if F is an element of representation T, then $\mathcal{M}(F)$ is already well-defined. If $F = o_i(t_1,t_2)$, where o_i is an operator and t_1 and t_2 are elements of T, then $\mathcal{M}(F) = o_i(\mathcal{M}(t_1),\mathcal{M}(t_2))$, and so on.

2.4 Summary

In this chapter we have defined several important and general problems in representation matching. We have also developed a semantic framework that formally defines the above problems and explains many key aspects of representation matching.

In particular, our framework explains that *what* a matching solution actually tries to do is to approximate true similarity values that are defined over the elements of a user representation. In the next three chapters, we develop solutions that show *how* different types of input knowledge can be leveraged and combined to achieve *good approximations* of such similarity values.

Chapter 3

1-1 MATCHING FOR DATA INTEGRATION

The previous chapter defines 1-1 matching as the problem of finding for each element of a representation the most similar element of another given representation. 1-1 matching is the simplest type of matching, and is already extremely useful, because it arises in numerous application contexts. Hence, we begin our study to develop a general matching solution by focusing on this problem. Specifically, we shall consider the problem in the context of data integration, an important data management application. Chapters 4-5 then extend the solution described here to other application contexts (e.g., data translation and ontology matching) and to more complex types of matching.

The chapter is organized as follows. The next section defines the matching problem. Section 3.2 provides an overview of our solution, as embodied by the LSD system. Sections 3.3–3.6 describes LSD in detail. Section 3.7 presents experiments. Section 3.8 discusses the limitations of the current LSD system and Section 3.9 summarizes the chapter.

3.1 Problem Definition

In this section we begin by introducing data integration systems. We then describe the problem of 1-1 matching for data integration, in the context of XML data.

3.1.1 Data Integration

The number of structured data sources available online is growing rapidly. Integrating data across such sources holds the potential to better many aspects of our lives, ranging from everyday activities such as reading news to important tasks such as buying a house. Manually integrating data from multiple sources, however, is extremely labor intensive. Hence, researchers have proposed building *data integration systems* (e.g., [GMPQ$^+$97, LRO96, IFF$^+$99, LKG99, FW97, KMA$^+$98]). Such a system provides a *uniform* query interface to a multitude of data sources, thereby freeing the user from the tedious job of accessing the individual sources, querying them, and manually combining the answers.

Consider for example a data integration system that helps users find houses on the real-estate market (Figure 3.1). The system provides the uniform interface in terms of a *mediated schema*, which is a *virtual* schema that captures the relevant aspects of the real-estate domain. The mediated schema may contain elements such as ADDRESS, PRICE, and DESCRIPTION, listing the house address, price, and description, respectively. The system maintains for each data source a *source schema* that describes the content of the source. *Wrapper programs*, attached to each data source, handle data formatting transformations between the local data model and the data model in the integration system.

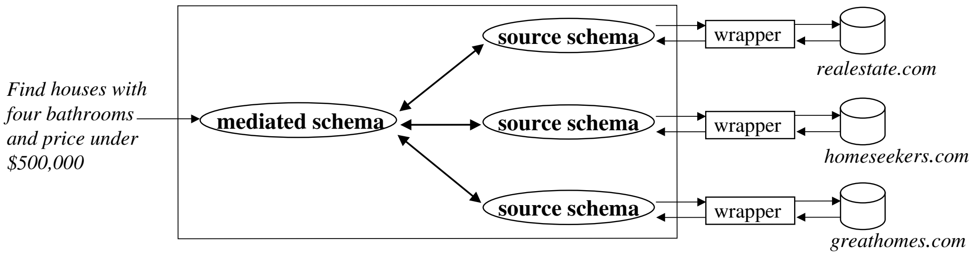


Figure 3.1: A data-integration system in the real-estate domain.

Given a user query formulated in the mediated schema, such as “find houses with 2 bedrooms priced under $200K”, the system translates the query into queries in the source schemas and executes these queries with the help of the wrappers, then combines the data returned by the sources to produce the final answers.

To translate user queries, a data integration system uses *semantic mappings* between the mediated schema and the local schemas of the data sources. Today, such semantic mappings are specified *manually* by the system builder. Our goal in this chapter is to develop a solution to semi-automatically create semantic mappings.

Schema Matching versus Wrapper Learning: Before we proceed, we note the difference between the problem of learning semantic mappings and that of wrapper learning. The key distinction is that wrapper learning [Kus00a, AK97, HGMN$^+$98] focuses on learning *syntax*: learning how to transform a semi-structured file (e.g., HTML) into a structured file (e.g., a set of tuples). In contrast, the problem we study is a *semantic* one: learning the relationship between the elements of a local schema and those of the mediated schema. While the growing number of structured documents (e.g., XML ones) will reduce significantly the need for wrappers, it only emphasizes the need for discovering semantic mappings.

3.1.2 XML Preliminaries

The eXtensible Markup Language standard [XML98] is increasingly being used as a protocol for the dissemination and exchange of information from data sources and stores of all types. Hence, we decided to consider the problem of reconciling schemas in the context of XML data. In addition to encoding relational data, XML can also encode object-oriented data, hierarchical data, and data in structured documents.

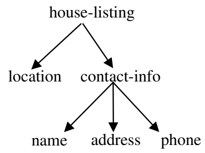
An XML document consists of pairs of matching open- and close-tags, enclosing *elements*. Each element may also enclose additional sub-elements or uniquely valued attributes. We often refer to the *type* of an element by the name of its opening tag. A document contains a unique root element, in which all others are nested. Figure 3.2.a shows a house listing stored as an XML document. In general, an XML element may also have a set of *attributes*. For the purpose of this dissertation we treat its attributes and sub-elements in the same fashion. Some of the attributes may be references

```
<house-listing>
    <location>Miami, FL</location>
    <contact-info>
        <name>Kate Richardson</name>
        <address>47 16th Ave., Miami, FL</address>
    </contact-info>
</house-listing>
```

(a)

```
<!ELEMENT  house-listing  (location?, contact-info)>
<!ELEMENT  location  (#PCDATA)>
<!ELEMENT  contact-info  (name, (address | phone))>
<!ELEMENT  name  (#PCDATA)>
<!ELEMENT  address  (#PCDATA)>
<!ELEMENT  phone  (#PCDATA)>
```

(b)



(c)

Figure 3.2: (a) An XML document, (b) a DTD associated with the previous document, and (c) the visualization of the DTD as a tree.

to other elements in the document, thereby adding more structure to a document. However, we do not yet use this additional structure in our work.

We consider XML documents with associated DTDs (Document Type Descriptors). A DTD is a BNF-style grammar that defines legal elements and relationships between the elements. Figure 3.2.b shows a DTD, which states that a document that conforms to this DTD consists of a house-listing element, which in turn consists of an optional location element and a contact-info element. The location element is a string, and the contact-info element consists of a name element, followed by either an address or phone element.

Our algorithms will make use of a tree representation of a DTD (see Figure 3.2.c). The tree represents information about the nesting and ordering of the elements, according to the DTD. However, the tree does not retain information on whether an element is required or optional (e.g., location), and whether an element can appear more than once. The tree representation also ignores unions in the DTD. It simply puts both of the options in a union as children (e.g., address and phone). We also do not attempt to represent recursion in a DTD with the tree.

3.1.3 Matching XML DTDs

To build a data integration system, the application designer must create a *mediated schema* over which users pose queries. The mediated DTD (or schema; we use the terms interchangeably) captures the aspects of the domain that are relevant to the data integration application. We assume that each of the data sources is associated with a source DTD. Data may be supplied by the source directly in this DTD, or processed through a wrapper that converts the data from a less structured format. Figure 3.3 shows the DTD trees of sample schemas in the real-estate domain.

Given a mediated schema and a source schema, the matching problem that we consider is to find some mapping between the two schema trees. In this chapter we start by considering the restricted case of *one-to-one (1-1) mappings* between tag names in the source DTD and tag names in the mediated DTD. For example, in Figure 3.3, node location matches ADDRESS, area matches COUNTY, and contact matches CONTACT-INFO.

In Chapter 4 we shall consider more complex mappings such as mappings from a tag in one DTD into a set of tags (or an aggregation on a set of tags) in the other (e.g., num_baths in one tree is the sum of full_baths and half_baths in the other). In general, we note that a mapping can be specified using an XML querying/transformation language (e.g., XQuery [Xqu], Quilt [CRF00], XML-QL [DFF+99], or XSLT [XSL99]).

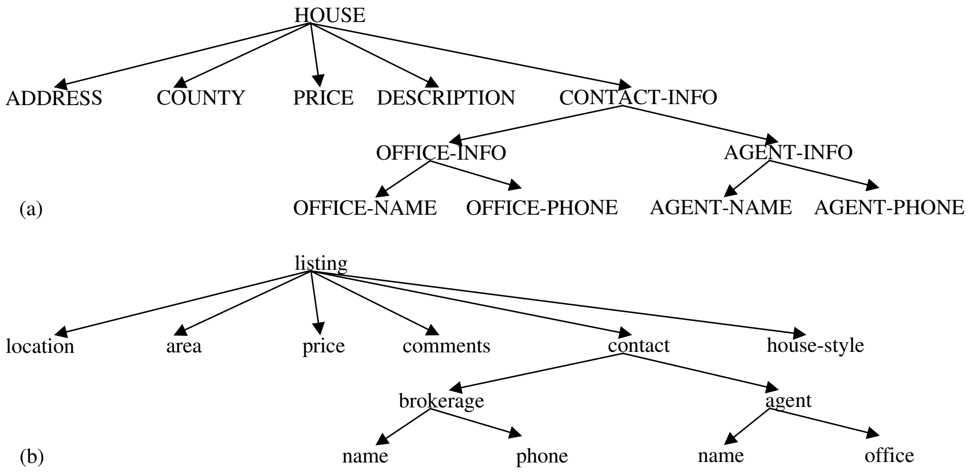


Figure 3.3: Sample DTDs for the real-estate domain: (a) a mediated DTD, and (b) source DTD of *greathomes.com*.

3.1.4 Schema Matching as Classification

Our approach rephrases the problem of finding 1-1 mappings as a *classification* problem: given the mediated-DTD tag names as distinct *labels* $c_1, \ldots, c_n$, we attempt to assign to each source-schema tag a matching label. (If no label matches the source-schema tag, then the unique label OTHER is assigned.) Classification proceeds by training a learner L on a set of *training examples* $\{(x_1, c_1), \ldots, (x_m, c_m)\}$, where each x_i is an object and c_i is the observed label of that object. During the *training phase*, the learner inspects the training examples and builds an *internal classification model* on how to classify objects.

In the *matching phase*, given an object x the learner L uses its internal classification model to predict a label for x. In this dissertation we assume a prediction is a list of labels weighted by *confidence scores*: $\langle LABEL_1 : score_1, LABEL_2 : score_2, \ldots, LABEL_n : score_n \rangle$, where $\sum_{i=1}^{n} score_i = 1$, and $score_i$ is learner L's *confidence score* that x matches $LABEL_i$ (omitted labels have confidence score 0). The higher a confidence score, the more certain the learner is in its prediction. (In machine learning, some learners output a *hard* prediction, which is a single label. However, most such learners can be easily modified to produce confidence-score predictions.) The following examples illustrate the learners employed in our approach:

Example 3 Consider the *Name Learner* which assigns a label to an XML element based on its *name* (see Section 3.4 for more details). Its training set may include the following examples:

```
(``location'', ADDRESS)
(``contact-name'', NAME)
```

The first training example states that an XML element with the name “location” matches label ADDRESS. In the matching phase, given an XML element, such as “$\langle$phone$\rangle$ (235) 143 2726 $\langle$/phone$\rangle$”, the *Name Learner* inspects the name, which is “phone”, and may issue a prediction such as $\langle$ADDRESS:0.1,DESCRIPTION:0.2,AGENT-PHONE:0.7$\rangle$.□

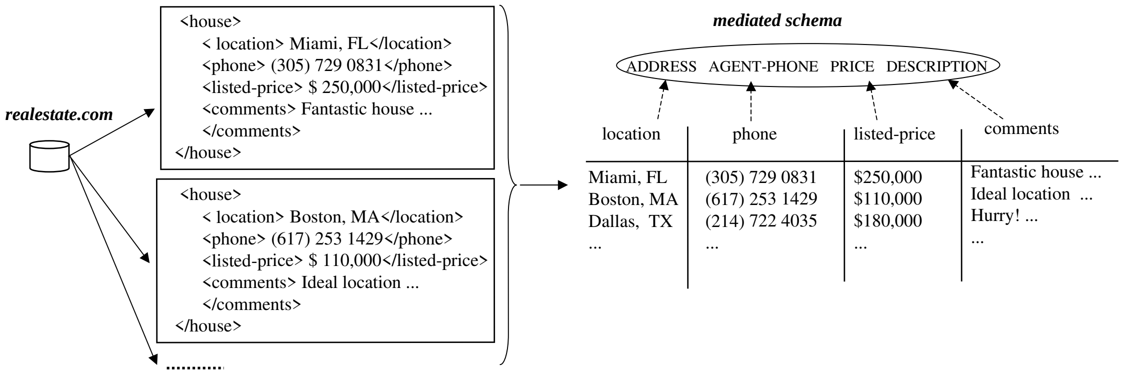


Figure 3.4: Once we have manually specified the 1-1 mappings between the mediated schema and the schema of *realestate.com*, LSD can be trained using the mappings and the data of this source. It then can be applied to match the DTD of *greathomes.com* in Figure 3.3.b.

Example 4 Consider the *Naive Bayes Learner* which assigns a label to an XML element based on its *data value* (see Section 3.4 for more details). Its training set may include the following examples:

```
(``Seattle, WA'', ADDRESS)
(``250K'', PRICE)
```

The first training example states that an XML element with data value “Seattle, WA” matches label ADDRESS. In the matching phase, given an XML element such as “⟨location⟩ Kent, WA ⟨/location⟩”, the Naive Bayes Learner inspects the data value, which is “Kent, WA”, and may issue a prediction such as ⟨ADDRESS:0.7,NAME:0.2⟩.□

3.2 An Overview of Our Approach

We now illustrate the key points of LSD with a simple example, where we apply LSD to the real-estate domain. In the example, we will train LSD using the source *realestate.com* (Figure 3.4), then use the predictions of LSD to match the DTD of source *greathomes.com* in Figure 3.3.b. The mediated DTD is shown in Figure 3.3.a.

3.2.1 Training and Matching

Our overall approach is to first use a set of manually provided mappings to train our learner and then apply the learner’s hypotheses to new sources during the matching phase. An important aspect of our approach is that we use *multiple* learning algorithms, and combine their results using a meta-learner.

The Training Phase: To use *realestate.com* as a training source, first we manually specify all 1-1 mappings between its DTD and the mediated DTD. This is a simple task which amounts to specify-

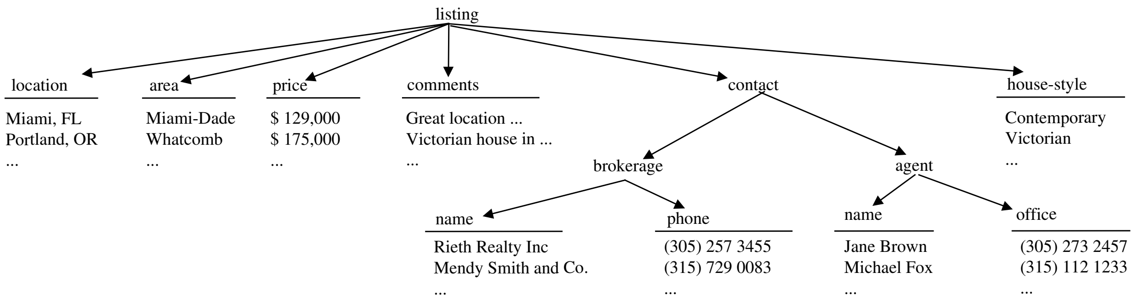


Figure 3.5: The schema of *greathomes.com*, and data coming from the extracted house listings.

ing only the dashed arrows in Figure 3.4. Next, we extract a set of house listings from *realestate.com* (between 20 and 300 houses in our experiments), and we use the listings in conjunction with the source DTD to train the base learners.

In this example, we consider the two base learners introduced in Examples 3-4. The first learner, the *Name Learner,* matches DTD tags using their names, using the full path and allowing synonyms. During the training process, the *Name Learner* inspects the *names* of matching tags, and builds general hypotheses about mapping rules. For example, since location matches ADDRESS (Figure 3.4), it may hypothesize that if an element name contains the word "location", then that element is likely to be ADDRESS.

The second learner, the *Naive Bayes Learner*, matches tag names based on the frequencies of words in corresponding elements' content. Thus during the training process the learner examines the *data instances* in the extracted house listings, to find out how word frequencies relate to element types. It may find, for example, that words such as "fantastic" and "great" appear frequently in house descriptions, but rarely in other element types. So it may construct the hypothesis that if these words appear frequently in the data instances of an element, then that element is likely to be DESCRIPTION.

Next, we train the meta-learner, which combines the predictions of the base learners during the matching phase. Roughly speaking, the meta-learner uses the training data to learn for each pair of mediated tag name and base learner a weight that indicates how much it *trusts* that learner's predictions regarding the mediated-schema tag.

The Matching Phase: Once the base learners and the meta-learner have been trained, we apply LSD to match the schemas of new sources (e.g., *greathomes.com*). First, we extract a set of house listings from the source. Next, for each source-DTD tag, we collect all elements of that tag from the house listings. Figure 3.5 shows the source DTD and the collected instances for the leaf elements of the DTD.

Now we iteratively obtain the matching for each source-DTD tag. Consider the tag office (on the rightmost side of Figure 3.5). We first apply the *Name Learner* to the tag name, expanded with all the tags leading to it from the root of the name (i.e., we apply the *Name Learner* to "listing contact agent office"). The *Name Learner* issues a prediction in the form of a probability distribution over the set of mediated-DTD tags. Then we apply the Naive Bayes learner to the *data instances* of office, and it issues another prediction. The meta-learner then combines the two predictions, weighting them using the weights learned during the training phase. Figure 3.6 shows the predictions returned by the meta-learner. Together, the two base learners have correctly matched 7 out of 13 tags, achieving a matching accuracy of 54%.

3.2.2 Highlights of Our Approach

We now highlight some of the important aspects of our approach and illustrate our innovations in applying multi-strategy learning to this problem domain.

Multiple Learners versus a Single Learner: By itself, the *Name Learner* would correctly match only 4 out of 13 tags, achieving an accuracy of 31%. It does not work well because the tags of some corresponding elements do not share synonyms (e.g., comments and DESCRIPTION), and some tag names are vacuous (e.g., listing), or partial (e.g., office refers to office phone).

The *Naive Bayes Learner* correctly matches 5 out of 13 elements, achieving an accuracy of 38%. It fails to match area with COUNTY, for instance, because the training source (*realestate.com*) does not have elements that match COUNTY, to provide examples of county names. It does not deal very well with nested elements, frequently confusing listing, contact, brokerage, and agent together. It also cannot distinguish between agent phone numbers and office phone numbers.

However, taken together, the two base learners can each contribute some complementary information to the matching process, thus significantly improving the matching accuracy. Consider for example the source tag office. From its full name ("listing contact agent office"), the *Name Learner* can only conclude that this element is about something related to the agent's office. From its data instances (e.g., (206) 273 2457, see Figure 3.5), the Naive Bayes learner can only conclude that this element is about phone numbers; it does not know if the numbers are related to the agent or the brokerage. However, together the two base learners can unambiguously identify this tag name to be about the agent phone numbers.

The above example thus illustrates the benefits of the multiple-learner over the single-learner approach. It also demonstrates that using only schema or data instances is potentially inadequate for the matching process, since in this case we cannot find the correct matching from either the schema nor the data by itself.

An additional advantage of the multiple-learner approach is the ability to extend the system with additional learners as needed. Later we describe a novel learner that exploits information from the structure of an XML document. A large source of important learners are *recognizers* for particular domains. For example, note that in our example, the source tag area is not correctly matched with the tag COUNTY. This is because the source that provided training data did not have elements that match COUNTY, so the learners do not recognize the contents of area to be county names. However, it is easy to build a county-name recognizer, which essentially compares a string with entries in a database of county names.

Incorporating Domain Constraints: Consider again the predictions returned by the meta-learner in Figure 3.6. It is easy to see that taken together they violate several integrity constraints that may be known for this domain. For example, we may have a constraint stating that at most one tag in a source should match the tag HOUSE in the mediated DTD. In Figure 3.6 this constraint is violated because both listing and contact are predicted to match HOUSE. As another example, we may have a constraint stating that if a source tag A matches OFFICE-INFO and source tag B matches AGENT-NAME, then A cannot include B. This constraint is also violated by the predictions of agent

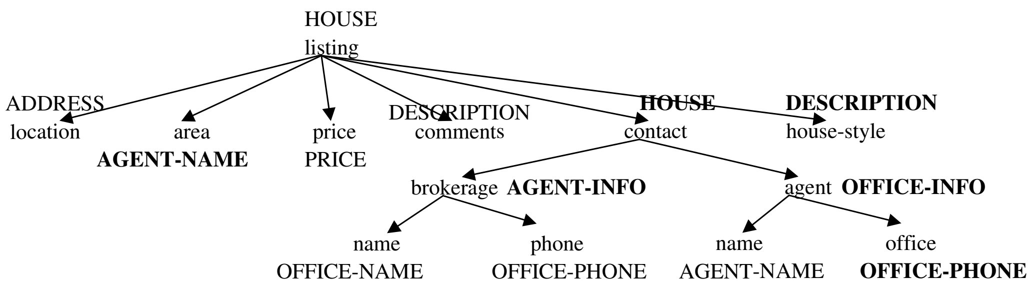


Figure 3.6: The combined predictions of the *Name Learner* and the *Naive Bayes Learner*, on the schema of *greathomes.com*. Incorrect predictions are highlighted in bold font.

and name. Some constraints may be *soft* (i.e., considered only as heuristics): for example, source designers typically put all agent-related elements together, or as close to each other as possible, to form a coherent semantic unit. This proximity constraint is violated by the predictions of name (under agent), brokerage, and area.

Clearly, if these constraints are observed during the matching process, then we could potentially improve the matching accuracy. We therefore added a module to our system that eliminates learners' hypotheses if they violate integrity constraints. On our running example, we would now be able to match 9 out of 13 elements correctly, achieving an accuracy of 69%, as compared to 54% before. Note that constraints can either be known in advance or be given as user feedback and incorporated into future predictions of the system.

The XML Learner: As it turned out, none of the learners we used dealt well with the hierarchical structure of XML documents, and therefore we were unable to correctly classify non-leaf elements in the DTDs. In our example, the Naive Bayes learner frequently confuses the four elements listing, contact, brokerage, and agent together. This is the reason we could not match these elements correctly in Figure 3.6 or even after taking domain constraints into account.

In fact, the machine-learning literature does not offer any methods specifically designed for classifying XML elements with nested structure (see related work in Chapter 6). So we developed a novel learner that can classify XML elements effectively. We incorporated this *XML Learner* as another base learner, and with it we can correctly match on this example all but one element, achieving an accuracy of 85%.

Ambiguous Schema Elements: Clearly, we cannot expect to develop a perfect schema reconciliation system. One reason is that there will always be some limitations to the learning algorithms we use on some domains. However, a second more fundamental reason, is that some matches are inherently ambiguous. In our example, the one tag that was matched incorrectly is house-style. The system matched it with DESCRIPTION. It is not clear whether we should accept this matching. house-style describes the architecture of the house, and thus is indeed a kind of description. However, it is short and describes only the architecture, whereas house descriptions tend to be longer and can describe many different things. That's why initially we decided house-style does not match DESCRIPTION, but OTHER. However, ultimately, this decision is subjective and heavily depends on the particular domain and context for which we are doing data integration.

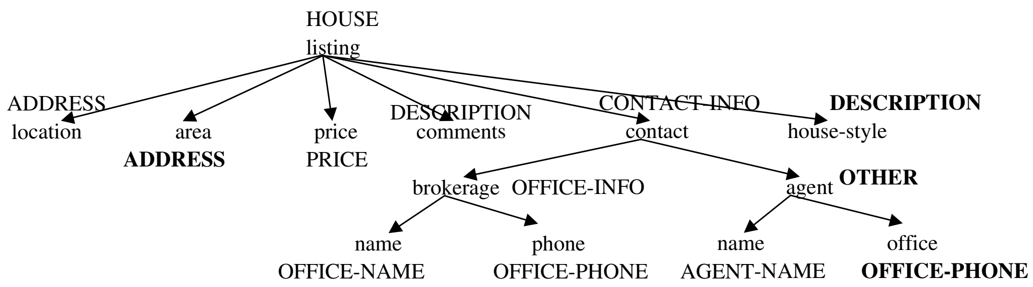


Figure 3.7: The predictions made by the system after incorporating domain constraints, on the schema of *greathomes.com*.

Hence our goal is to develop a system that provides as much accuracy as possible, but still leaving room for human intervention. As this example illustrates, the combination of multiple learners and the consideration of domain constraints can achieve a high level of matching accuracy.

3.3 Multi-Strategy Learning

We now describe LSD in detail. The system consists of four major components: *base learners*, *meta-learner*, *prediction combiner*, and *constraint handler*. It operates in two phases: training and matching (Figure 3.8). In the *training phase* LSD first asks the user to manually specify the mappings for several sources. Second, it extracts some data from each source. Third, it creates training examples for the base learners from the extracted data. Different base learners will require different sets of training examples. Fourth, it trains each base learner on the training examples. The output of training the base learners is a set of internal classification models (i.e., classification hypotheses). Finally, LSD trains the meta-learner. The output of training the meta-learner is a set of weights, one for each pair of base learner and label (i.e., mediated-schema element).

In the *matching phase* the trained learners are used to match new source schemas. Matching a target source proceeds in three steps. First, LSD extracts some data from the source, and creates for each source-schema element a column of XML instances that belong to it. Second, LSD applies the base learners and the meta-learner to the XML instances in the column to obtain predictions, one per instance. Third, LSD combines the instance-level predictions into a column-level prediction using the prediction combiner. Finally, the *Constraint Handler* takes the predictions, together with the available domain constraints, and outputs 1-1 mappings for the target schema. The user can either accept the mappings, or provide some feedback and ask the *Constraint Handler* to come up with a new set of mappings.

This section describes the two phases, the meta-learner, and the prediction combiner. The next section (Section 3.4) describes the base learners except the *XML Learner*. Then Section 3.5 describes the *Constraint Handler*. Finally Section 3.6 wraps up the description of the LSD system by describing the *XML Learner*, a novel base learner we developed to handle nested DTD elements. For the pseudo code of LSD, see Appendix A.

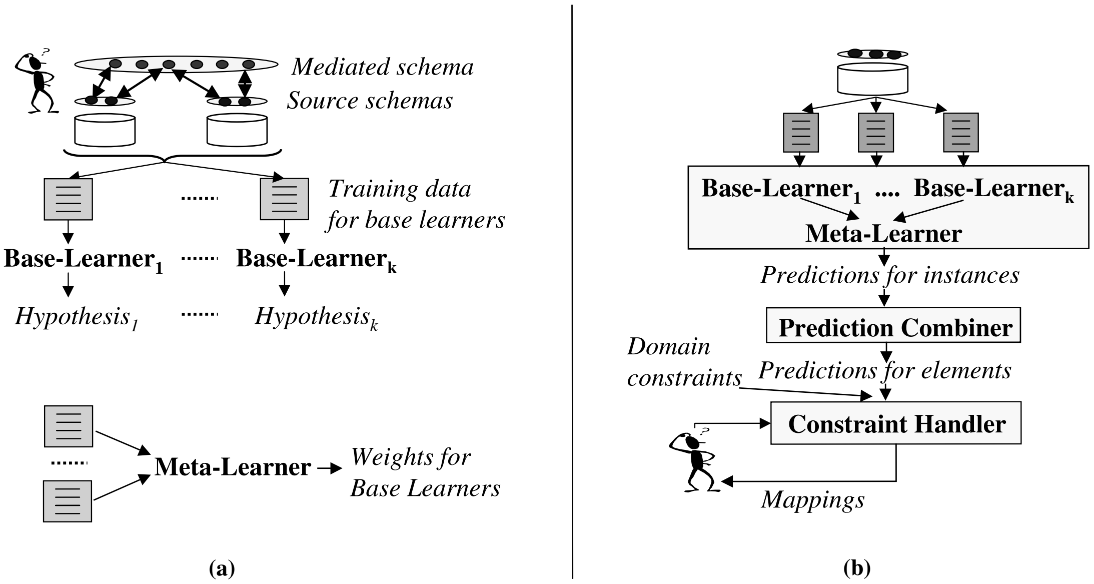


Figure 3.8: The two phases of LSD: (a) training, and (b) matching

3.3.1 The Training Phase

1. Manually Specify Mappings for Several Sources: Given several sources as input, LSD begins by asking the user to specify 1-1 mappings for these sources, so that it can use the sources to create training data for the learners.

Suppose that LSD is given the two sources *realestate.com* and *homeseekers.com*, whose schemas are shown in Figure 3.9.a, together with the mediated schema. (These schemas are simplified versions of the ones we actually used in the experiments.) Then the user simply has to specify the mappings shown in Figure 3.9.b, which says that location matches ADDRESS, comments matches DESCRIPTION, and so on.

Note that in specifying the mappings, the user labels only the *schemas*, not the *data instances* of the sources. This is done only once, at the beginning of the training phase, so the work should be amortized over the subsequent tens or hundreds of sources in the matching phase. Furthermore, once a new source has been matched by LSD and the matchings have been confirmed/refined by the user, it can serve as an additional training source, making LSD unique in that it can directly and seamlessly reuse past matchings to continuously improve its performance.

2. Extract Source Data: Next, LSD extracts data from the sources (20-300 house listings in our experiments). In our example, LSD extracts a total of four house listings as shown in Figure 3.9.c. Here, for brevity we show an XML element such as "⟨location⟩ Miami, FL ⟨/location⟩" as "location: Miami, FL". Each house listing has 3 XML elements. Thus we have a total of 12 extracted XML elements.

3. Create Training Data for each Base Learner: LSD then uses the extracted XML elements,

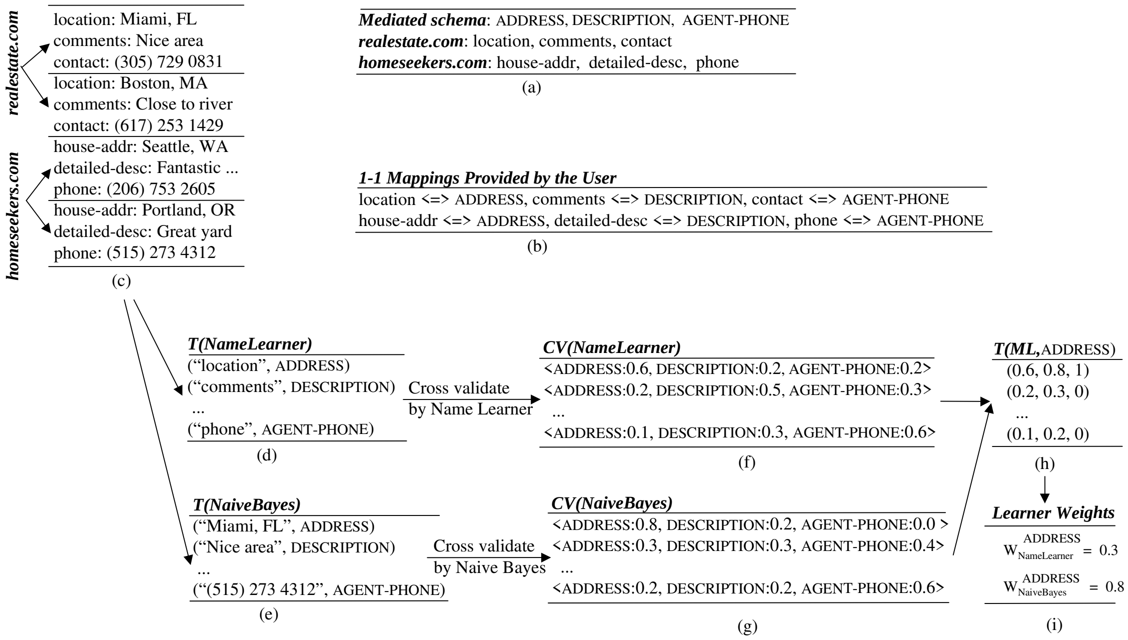


Figure 3.9: An example of creating training data for the base learners and the meta-learner.

together with the 1-1 mappings provided by the user, to create the training data for each base learner. Given a base learner *L*, from each XML element *e* we extract all features that *L* can learn from, then pair the features with the correct label of *e* (as inferred from the 1-1 mappings) to form a training example.

To illustrate, we shall assume that LSD uses only two base learners: the *Name Learner* and the *Naive Bayes Learner* (both are described in detail in Section 3.4). The *Name Learner* matches an XML element based on its *tag name*. Therefore, for each of the 12 extracted XML elements (Figure 3.9.c), its tag name and its true label form a training example. Consider the first XML element, "location: Miami, FL". Its tag name is "location". Its true label is ADDRESS, because the user has manually specified that "location" matches ADDRESS. Thus, the training example derived from this XML element is ("location",ADDRESS). Figure 3.9.d lists the 12 training examples for the *Name Learner*. Some training examples are duplicates, but that is fine because most learners, including the *Name Learner*, can cope with duplicates in the training data.

The *Naive Bayes Learner* matches an XML element based on its *data content*. Therefore, for each extracted XML element, its data content and its true label form a training example. For instance, the training example derived from the XML element "location: Miami, FL" will be ("Miami, FL", ADDRESS). Figure 3.9.e lists the 12 training examples for the *Naive Bayes Learner*.

4. Train the Base Learners: Next, LSD trains each base learner on the training examples created for that learner. Each learner will examine its training examples to construct an internal classification model that helps it match new examples. These models are part of the output of the training phase, as shown at the bottom of Figure 3.8.a.

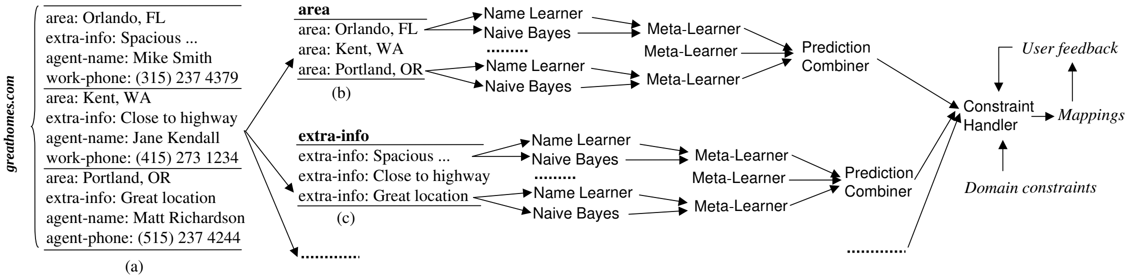


Figure 3.10: Matching the schema of source *greathomes.com*.

5. Train the Meta-Learner: Finally, LSD trains the meta-learner. This learner uses a technique called *stacking* [Wol92, TW99] to combine the predictions of the base learners. Training the meta-learner proceeds as follows [TW99]. First the meta-learner asks the base learners to predict the labels of the training examples. The meta-learner knows the correct labels of the training examples. Therefore it is able to judge how well each base learner performs with respect to each label. Based on this judgment, it then assigns to each combination of label c_i and base learner L_j a weight $W_{L_j}^{c_i}$ that indicates how much it *trusts* learner L_j's predictions regarding c_i. Stacking uses a technique called *cross-validation* to ensure that the weights learned for the base learners do not overfit the training sources, but instead generalize correctly to new ones.

We now describe computing the learner weights in detail. Section 3.3.2 will describe how the meta-learner uses the weights to combine the base learners' predictions.

(a) Apply Base Learners to Training Data: For each base learner L, let $T(L)$ be the set of training examples created for L in Step 3. The meta-learner applies L to predict labels for the examples in $T(L)$. The end result will be a set $CV(L)$ that consists of exactly *one prediction* for each example in $T(L)$.

A naive approach to create $CV(L)$ is to have learner L trained on the *entire* set $T(L)$, then applied to each example in $T(L)$. However, this approach biases learner L because when applied to any example t, it has already been trained on t. *Cross validation* is a technique commonly employed in machine learning to prevent such bias. To apply cross validation, the examples in $T(L)$ are randomly divided into d equal parts $T_1, T_2, \ldots, T_d$ (we use $d = 5$ in our experiments). Next, for each part T_i, $i \in [1, d]$, L is trained on the remaining $(d-1)$ parts, then applied to the examples in T_i.

Figure 3.9.f shows the set CV for the *Name Learner*. Here, the first line is the prediction made by the *Name Learner* for the first training example in $T(NameLearner)$, which is ("location",ADDRESS) in Figure 3.9.d. The second line is the prediction for the second training example, and so on. Figure 3.9.g shows the set CV for the *Naive Bayes Learner*.

(b) Gather Predictions for each Label: Next, the meta-learner uses the CV sets to create for each label c_i a set $T(ML, c_i)$ that summarizes the performance of the base learners with respect to c_i. For each extracted XML element x, the set $T(ML, c_i)$ contains exactly one tuple of the form $\langle s(c_i|x,L_1), s(c_i|x,L_2), \ldots, s(c_i|x,L_k), l(c_i,x)\rangle$, where $s(c_i|x,L_j)$ is the confidence score that x matches label c_i, as predicted by learner L_j. This score is obtained by looking up the prediction that corresponds to x in $CV(L_j)$. The function $l(c_i,x)$ is 1 if x indeed matches c_i, and 0 otherwise.

For example, consider label ADDRESS. Take the first extracted XML element in Figure 3.9.c: "location: Miami, FL". The *Name Learner* predicts that it matches ADDRESS with score 0.6 (see the first tuple of Figure 3.9.f). The *Naive Bayes Learner* predicts that it matches ADDRESS with score 0.8 (see the first tuple of Figure 3.9.g). And its true label is indeed ADDRESS. Therefore, the tuple that corresponds to this XML element is $(0.6, 0.8, 1)$. We proceed similarly with the remaining 11 XML elements. The resulting set $T(ML, \text{ADDRESS})$ is shown in Figure 3.9.h.

(c) Perform Regression to Compute Learner Weights: Finally, for each label c_i, the meta-learner computes the learner weights $W^{c_i}_{L_j}$, $j \in [1,k]$, by performing least-squares linear regression on the data set $T(ML, c_i)$ created in Step (b). This regression finds the learner weights that minimize the squared error $\sum_j \left(l(c_i,x_j) - \sum_t s(c_i|x_j,L_t) * W^{c_i}_{L_t}\right)^2$, where x_j ranges over the entire set of extracted XML elements. The regression process has the effect that if a base learner tends to output a high confidence that an instance matches c_i when it does and a low confidence when it does not, it will be assigned a high weight, and vice-versa.

To continue with our example, suppose applying linear regression to the set $T(ML, \text{ADDRESS})$ yields $W^{ADDRESS}_{NameLearner} = 0.3$ and $W^{ADDRESS}_{NaiveBayes} = 0.8$ (Figure 3.9.i). This means that based on the performance of the base learners on the training sources, the meta-learner will trust Naive Bayes much more than the *Name Learner* in predicting label ADDRESS.

3.3.2 The Matching Phase

Once the learners have been trained, LSD is ready to predict semantic mappings for new sources. Figure 3.10 illustrates the matching process on source *greathomes.com*. We now describe the three steps of this process in detail.

1. Extract & Collect Data: First, LSD extracts from *greathomes.com* a set of house listings (three listings in Figure 3.10.a). Next, for each source-DTD tag, LSD collects all the instances of elements with that tag from the listings. Figures 3.10.b and 3.10.c show the instances for tags area and extra-info, respectively.

2. Match each Source-DTD Tag: To match a source-DTD tag, such as area, LSD begins by matching *each data instance* of the tag. Consider the first data instance: "area: Orlando, FL" (Figure 3.10.b). To match this instance, LSD applies the base learners, then combines their predictions using the meta-learner.

The *Name Learner* will take the instance *name*, which is "area", and issue the prediction:

$$\langle \text{ADDRESS:0.5, DESCRIPTION:0.3, AGENT-PHONE:0.2} \rangle$$

The *Naive Bayes Learner* will take the instance *content*, which is "Orlando, FL", and issue another prediction:

$$\langle \text{ADDRESS:0.7, DESCRIPTION:0.3, AGENT-PHONE:0.0} \rangle$$

The meta-learner then combines the two predictions into a single prediction. For each label, the meta-learner computes a combined score which is the sum of the scores that the base learners give to that label, weighted by the learner weights. For example, assuming learner weights $W^{ADDRESS}_{NameLearner} = 0.3$ and $W^{ADDRESS}_{NaiveBayes} = 0.8$, the combined score regarding the above instance matching label ADDRESS will be $0.3 * 0.5 + 0.8 * 0.7 = 0.71$. Once combined scores have been computed for all three labels, the meta-learner normalizes the scores, and issues the following prediction for the above instance:

⟨ADDRESS:0.7, DESCRIPTION:0.2, AGENT-PHONE:0.1⟩

We proceed similarly for the remaining two instances of area (note that in all cases the input to the *Name Learner* is "area"), and obtain the following two predictions:

⟨ADDRESS:0.5, DESCRIPTION:0.2, AGENT-PHONE:0.3⟩

⟨ADDRESS:0.9, DESCRIPTION:0.09, AGENT-PHONE: 0.01⟩

The *Prediction Combiner* then combines the three predictions of the three data instances into a single prediction for area. Currently, the *Prediction Combiner* simply computes the average score of each label from the given predictions. So in this case it returns

⟨ADDRESS:0.7, DESCRIPTION:0.163, AGENT-PHONE:0.137⟩

3. Apply the Constraint Handler: After the *Prediction Combiner* has computed predictions for all source-DTD tags, the *Constraint Handler* takes these predictions, together with the domain constraints, and outputs the 1-1 mappings. If there are no domain constraints, each source-DTD tag is assigned the label associated with the highest score, as predicted by the prediction converter for that tag. Section 3.5 describes the *Constraint Handler*, together with domain constraints and user feedback.

Remark 3.3.1 Figure 3.10 shows that to match a source-schema tag, say area, each base learner takes as input a *row* in the data column for area. One may ask why not apply each base learner directly to the *entire data column*, because we are ultimately interested in matching *columns*, not *rows*. Obviously, for learners that only use schema information (e.g., the *Name Learner*), there is no difference. For learners that use row data, the difference is between having few examples with a long description for each example, and having many examples with a short description for each on. Machine learning algorithms typically work much better in the latter case. Notice that longer example descriptions imply a potentially larger instance space. (For example, if an example is described by n Boolean attributes, the size of the instance space is 2^n.) Learning is easier when the instance space is small and there are many examples. When the instance space is large and there are few examples, it becomes very hard to tell where the frontiers between different classes are. Thus it is preferable to consider each row as a separate example.

3.4 The Base Learners

The current LSD implementation uses the following base learners (Section 3.6 describes the *XML Learner*).

3.4.1 *The Name Learner*

This learner matches an XML element using its tag name, expanded with synonyms and all tag names leading to this element from the root element.

We first tried to obtain synonyms using a general-purpose dictionary such as *WorldNet* [Wor]. We soon realized, however, that this strategy did not work, for two reasons. First, *WordNet* cannot provide domain-specific synonyms such as "bdrm" for "bedroom" and "comments" for "house-description". And second, for many words *WordNet* provides too many synonyms. Most of these synonyms are irrelevant to the domain at hand and hence only serve to confuse the learner. Based on this experience, we decided to build domain-specific synonym tables. For each experimental domain, we randomly collected 30 listings (e.g., houses or courses). Next, we examined the listings and collected all synonyms we could find. The domain's synonym table consists of these synonyms.

To make predictions, the *Name Learner* uses Whirl, the nearest-neighbor classification system developed by Cohen and Hirsh [CH98]. The *Name Learner* stores all training examples of the form (expanded tag-name,label) that it has seen so far. Then given an XML element t, it computes the label for t based on the labels of all examples in its store that are within a δ similarity distance from t. The similarity distance between any two examples is the TF/IDF distance (commonly employed in information retrieval) between the *expanded tag names* of the examples. See [CH98] for more details.

The *Name Learner* works well on specific and descriptive names, such as price or house_location. It is not good at names that do not share synonyms (e.g., comments and DESCRIPTION), that are partial (e.g., office to indicate office phone), or vacuous (e.g., item, listing).

3.4.2 *The Content Learner*

This learner also uses Whirl. However, the learner matches an XML element using its *data content*, instead of its *expanded tag name* as with the *Name Learner*. Therefore, here each example is a pair (data-content,label), and the TF/IDF distance between any two examples is the distance between their data contents.

This learner works well on long textual elements, such as house description, or elements with very distinct and descriptive values, such as color (red, blue, green, etc.). It is not good at short, numeric elements such as number of bathrooms and number of bedrooms.

3.4.3 *The Naive Bayes Learner*

This learner is one of the most popular and effective text classifiers [DH74, DP97, MN98]. Since some of the ideas underlying Naive Bayes will also be used in the *XML Learner*, we describe it in more detail.

This learner treats each input instance as a *bag of tokens*, which is generated by parsing and stemming the words and symbols in the instance. For example, "RE/MAX Greater Atlanta Affiliates of Roswell" becomes "re / max greater atlanta affili of roswell".

Let $c_1, \ldots, c_n$ be the classes and $d = \{w_1, \ldots, w_k\}$ be an input instance, where the w_j are tokens. Then the *Naive Bayes Learner* assigns d to the class c_d that has the highest posterior probability given d. Formally:

$$c_d = arg\ max_{c_i}\ P(c_i|d)$$

$$= \quad arg\ max_{c_i}\ [P(d|c_i)P(c_i)/P(d)]$$
$$= \quad arg\ max_{c_i}\ [P(d|c_i)P(c_i)].$$

The probabilities $P(c_i)$ and $P(d|c_i)$ are estimated using the training data. $P(c_i)$ is approximated as the portion of training instances with label c_i. To compute $P(d|c_i)$, we assume that the tokens w_j appear in d *independently* of each other given c_i. With this assumption, we have

$$P(d|c_i) = P(w_1|c_i)P(w_2|c_i)\cdots P(w_k|c_i),$$

where $P(w_j|c_i)$ is estimated as $n(w_j,c_i)/n(c_i)$. The number $n(c_i)$ is the total number of token positions of all training instances with label c_i, and $n(w_j,c_i)$ is the number of times token w_j appears in all training instances with label c_i. Even though the independence assumption is typically not valid, the *Naive Bayes Learner* still performs surprisingly well in many domains, including text-based ones (for an explanation, see [DP97]).

It follows from the above description that the *Naive Bayes Learner* works best when there are tokens that are strongly indicative of the correct label, by virtue of their frequencies. For example, it works for house descriptions which frequently contain words such as “beautiful” and “fantastic” – words that seldom appear in other elements. It also works well when there are only weakly suggestive tokens, but many of them. It does not work well for short or numeric fields, such as color, zip code, or number of bathrooms.

3.4.4 The County-Name Recognizer

This recognizer module searches a database (extracted from the Web) to verify if an XML element is a county name. LSD uses this module in conjunction with the base learners when working on the real-estate domain. This module illustrates how recognizers with a narrow and specific area of expertise can be incorporated into our system. Many such entity recognizers can be developed (e.g., recognizers for names, phone numbers, zip codes, addresses, etc.).

3.5 Exploiting Domain Constraints

As we saw in Section 3.2.2, the consideration of domain constraints can improve the accuracy of our predictions. We begin by describing domain constraints, then the process of exploiting these constraints using the *Constraint Handler*.

3.5.1 Domain Integrity Constraints

Domain constraints impose semantic regularities on the schemas and data of the sources in the domain. They are specified only once, at the beginning, as a part of creating the mediated schema, and independently of any actual source schema. Thus, exploiting domain constraints does not require any subsequent work from the user. Of course, as the user gets to know the domain better, constraints can be added or modified as needed.

Table 3.1 shows examples of domain constraints currently used in our approach and their characteristics. Notice that the constraints refer to labels (i.e., mediated-schema elements) and generic source-schema elements (e.g., a,b,c,), and that they are grouped into different types. The idea is that

Table 3.1: Types of domain constraints. Variables a, b and c refer to source-schema elements.

Constraint Types		Examples	Can Be Verified With
Hard Constraints	*Frequency*	At most one source element matches HOUSE. Exactly one source element matches PRICE.	Schema of target source
	Nesting	If **a** matches AGENT-INFO & **b** matches AGENT-NAME, then **b** is nested in **a.** If **a** matches AGENT-INFO & **b** matches PRICE, then **b** cannot be nested in **a**.	,,
	Contiguity	If **a** matches BATHS & **b** matches BEDS, then **a** & **b** are siblings in the schema-tree, and the elements between them (if any) can only match OTHER.	,,
	Exclusivity	There are no **a** and **b** such that **a** matches COURSE-CREDIT & **b** matches SECTION-CREDIT.	,,
	Column	If **a** matches HOUSE-ID, then **a** is a key. If **a** matches CITY, **b** matches OFFICE-NAME, and **c** matches OFFICE-ADDRESS, then **a** & **b** functionally determine **c**.	Schema + data from target source
Soft Constraints	*Binary*	Number of elements that match DESCRIPTION is not more than 3.	Schema, also may need data from target source
	Numeric	If **a** matches AGENT-NAME & **b** matches AGENT-PHONE, then we prefer **a** & **b** to be as close to each other as possible, all other things being equal.	Schema of target source

for any source S in the domain, given a *mapping combination m* that specifies which source-schema element matches which label, we can use the DTD and the extracted data of source S to compute $cost(m,T)$, a cost value that quantifies the extent to which m violates the constraints of type T. Next, the cost of m can be computed based on the costs of violating different constraint types. Finally, the *Constraint Handler* returns the candidate mapping with the least cost.
We distinguish two types of constraints:
Hard Constraints: These are the constraints that the application designer thinks any correct matching combination should satisfy. Let $T_{hard} = \{t_1, \ldots, t_u\}$ be the set of hard constraints. Then we define

$$cost(m, T_{hard}) = \begin{cases} 0 & \text{if } m \text{ satisfies } t_1 \wedge t_2 \wedge \cdots \wedge t_u \\ \infty & \text{otherwise} \end{cases}$$

Table 3.1 shows examples of five types of hard constraints. The first four types, from *frequency* to *exclusivity*, impose regularities that the source schema must conform to. The last type, *column*, imposes regularities that both the source schema and data must conform to.

We can specify arbitrary hard constraints that involve only the schemas, because given any candidate mapping, they can always be checked. Constraints involving data elements cannot always be checked because we have access only to the current source data. (Even when all the data in the source at a given time conforms to a constraint, that still does not mean the constraint holds on the source.) In many cases, however, the few data instances we extract from the source will be enough to find a violation of such a constraint.

Soft Constraints: These are the constraints for which we try to minimize the extent to which they are violated. They can be used to express heuristics about the domain.

We distinguish two types of soft constraints: *binary constraints,* whose cost of violation is 1, and *numeric constraints*, that can have varying cost of violation. Table 3.1 shows examples of binary and numeric soft constraints.

A common example of numeric constraints are *proximity constraints*. A proximity constraint refers to a set of labels, and has the intuitive meaning that we prefer the source-schema elements that match these labels to be *as close to each other as possible*, given all other things being equal.

For example, source designers typically put all agent-related elements together to form a coherent semantic unit. So we may specify a proximity constraint g that refers to the set of labels S $=\{$AGENT-INFO, AGENT-NAME, AGENT-PHONE$\}$. Constraint g says that if elements a, b, and c match the labels in S, then we prefer the combination that minimizes the average distance among a, b, c. The average distance among a set of source elements $e_1, \ldots, e_k$ is $(\sum_{1 \leq i,j \leq k} dist(e_i, e_j))/([k(k-1)/2]$, where $dist(e_i, e_j)$ is the distance between node e_i and node e_j in the source-schema tree.

Note that g applies even when source elements match only a subset of S. For example, if a and b match AGENT-NAME and AGENT-PHONE, and no source element matches AGENT-INFO, then we still prefer the combination that minimizes the average distance between a and b.

3.5.2 The Constraint Handler

The *Constraint Handler* takes the domain constraints, together with the predictions produced by the *Prediction Combiner*, and outputs the 1-1 mappings.

Recall from the previous section that a mapping combination assigns a label to each source-schema element. Conceptually, the *Constraint Handler* searches through the space of possible mapping combinations to find the one with the lowest cost, where cost is defined based on the likelihood of the combination and the degree to which the combination satisfies the domain constraints.

Specifically, let $e_1, e_2, \ldots, e_q$ be the DTD tags of the target source, and $c_1, c_2, \ldots, c_n$ be the class labels, as before. We denote a mapping combination m by $\langle e_1 : c_{i1}, e_2 : c_{i2}, \ldots, e_q : c_{iq} \rangle$ where tag e_j is mapped to label c_{ij}.

Then the cost of mapping combination m is defined as

$$cost(m) = -\alpha * LH(m) + \beta_1 * cost(m, T_1) + \beta_2 * cost(m, T_2) + \cdots + \beta_v * cost(m, T_v), \qquad (3.1)$$

where $LH(m)$ represents the likelihood of m, $cost(m, T_i)$ represents the degree to which m satisfies domain constraints of type T_i, and $\alpha, \beta_1, \ldots, \beta_v$ are the scaling coefficients that represent the trade-offs among the cost components. We now describe these terms in more detail.

The term $LH(m)$ is defined as $log\ conf(m)$, where $conf(m)$ is the confidence score of combination m. This confidence is computed as:

$$conf(m) = s(c_{i1}|e_1, PC) * s(c_{i2}|e_2, PC) * \cdots * s(c_{iq}|e_q, PC),$$

where $s(c_{ij}|e_j, PC)$ is the confidence that source-DTD element e_j matches label c_{ij}, returned by the *Prediction Combiner PC*. The formula for $conf(m)$ effectively assumes that the label assignments of source-schema tags are independent of each other. This assumption is clearly not true, because in many cases the label of a schema tag does depend on the labels of its parents/children. However, we make this assumption to reduce the cost of our search procedure. Note also that the definition of $LH(m)$ coupled with Equation 3.1 implies that we prefer the combination with the highest confidence score, all other things being equal.

LSD uses the A* algorithm [HNR72] to search the space of possible mapping combinations. In the next subsection we describe this search process in more detail.

3.5.3 Adapting the A* Algorithm for the Constraint Handler

The A* algorithm takes as input an *initial state*, a *set of actions* that can be applied to a state to reach new states, a set of *goal states*, and a *path cost* that assigns a cost to a path from the initial state to a goal state. The path cost is typically defined to be the sum of the costs of all actions along the path.

Given the above input, A* searches for the cheapest solution: the path with least cost from the initial state to a goal state. It performs a *best-first* search: start with the initial state, and always select the state with the smallest *estimated cost* for further expansion. The estimated cost of a state n is computed as $f(n) = g(n) + h(n)$, where $g(n)$ is the cost of the path from the initial state to n, and $h(n)$ is a *lower bound* on the cost of the cheapest path from n to a goal state. Hence, the estimated cost $f(n)$ is a lower bound on the cost of the cheapest solution via n. A* terminates when it reaches a goal state, returning the path from the initial state to that goal state.

The A* algorithm is complete and optimal in that if solutions exist (i.e., there exist some paths from the initial state to goal states), then it will find the cheapest solution. Its efficiency (i.e., the number of states that it examines and stores in memory) depends on the accuracy of the heuristic $h(n)$ that estimates the lowest cost necessary to reach a goal state. The closer $h(n)$ is to the actual lowest cost, the fewer states A* needs to examine. In the ideal case where $h(n)$ is the lowest cost, A* marches straight to the goal with the lowest cost.

In what follows we describe adapting A* to constraint handling.

States: Recall from the previous section that the source DTD has q tags: $e_1, e_2, \dots, e_q$ (which we must match with the mediated-DTD tags $c_1, c_2, \dots, c_n$). Hence, we define a state to be a tuple of q elements, where the i-th element specifies the label of tag e_i. This label is either a specific label, such as c_j, or the wildcard *, which represents *any* label. Thus, a state *partially* specifies a mapping combination, and can be interpreted as representing the *set* of mapping combinations that are consistent with the specification. For example, suppose $q = 5$ and $n = 3$, then state $(c_2, *, c_1, c_3, c_2)$ represents three states $(c_2, c_1, c_1, c_3, c_2)$, $(c_2, c_2, c_1, c_3, c_2)$, and $(c_2, c_3, c_1, c_3, c_2)$. We refer to a state as an *abstract state* if it contains wildcards, and an *concrete state* otherwise.

We define the initial state to be the abstract state $(** \dots *)$, which represents all possible mapping combinations, and the goal states to be all concrete states. Our goal then is to search for the goal state (i.e., a mapping combination) that has the lowest cost, as defined in Equation 3.1.

Actions: Given a non-goal state s, we refine it by selecting a wildcard for expansion, then creating new states, each being the same as s, except that the wildcard is replaced with a specific label.

We select the wildcard for expansion as follows. First, we assign a score to each source-DTD tag e_i. The score measures the extent to which the tag participates in likely domain constraints. Currently the score of a tag is approximated with the number of distinct tags that can be nested within that tag, based on the heuristic that the greater the structure below a tag, the greater the probability that the tag is involved in one or more constraints. Next, we order the tags in the source-DTD in decreasing order of their scores. Finally, we examine the tags in the order and selects the first wildcard tag (in the non-goal state s) for expansion.

For example, assume again that the source-DTD has five tags: $e_1, \dots, e_5$. Tag e_1 is the root, tags e_2 and e_3 are children of e_1, and tags e_4 and e_5 are children of e_3. Then the score of e_1 is 4 (because all four other tags are nested within it), of e_2 is 0, of e_3 is 2, and so on. A possible ordering is then e_1, e_3, e_4, e_5, e_2 (with ties being resolved arbitrarily). Suppose state s

is $(c_1 * * c_3 c_2)$, then the second wildcard (which corresponds to e_3) is selected for expansion, and three new states are created: $(c_1 * c_1 c_3 c_2)$, $(c_1 * c_2 c_3 c_2)$, and $(c_1 * c_3 c_3 c_2)$.

Note that the tag ordering is done only once, at the beginning of A*, not at the refinement of each state during the search.

Path Costs: Consider again state s. Suppose we have selected the wildcard at position i (corresponding to tag e_i) for expansion, and have replaced it with label c_j to create a new state s'. Then the cost of the path from s to s' is defined to be $-log\ s(c_j|e_i, PC)$, where $s(c_j|e_i, PC)$ is the confidence score that source-DTD element e_i matches label c_j, as returned by the *Prediction Combiner PC*.

Hence, for any state n, the cost $g(n)$ of the path from the initial state to n will be the sum of costs over all non-wildcard elements of n. Suppose $n = (c_1 c_2 * * c_3)$, then

$$g(n) = -[logs(c_1|e_1, PC) + logs(c_2|e_2, PC) + logs(c_3|e_5, PC)].$$

The cost $h(n)$ of the path from n to a goal state will be an estimated cost $h_1(n)$ over the wildcard elements plus an estimated cost $h_2(n)$ of the extent to which the goal state satisfies the domain constraints.

Suppose again that $n = (c_1 c_2 * * c_3)$, then the estimated cost $h_1(n)$ is

$$h_1(n) = -(log[\max_i s(c_i|e_3, PC)] + log[\max_i s(c_i|e_4, PC)]).$$

Thus, $h_1(n)$ is a lower bound on the cost of expanding all wildcards of n.

The estimated cost $h_2(n)$ is defined as the sum of $cost(n, T_i)$, where $cost(n, T_i)$ measures the extent to which n satisfies constraints of type T_i. This cost measure has already been defined for the case where n is a goal state; it is $cost(m, T_i)$ defined in the previous section. It can be generalized to cover the case of non-goal states as follows. For each non-goal state n, we try to determine for each constraint type T_i if the constraints of that type can be possibly satisfied by *any* goal state in the set of concrete states represented by n. We assume the best-case scenario, in that if we cannot determine the satisfaction of constraints in T_i, then we assume there is a goal state that is represented by n and satisfies T_i.

For example, consider $n = (c_1 c_2 * * c_3)$ and the hard constraint t = "at most one tag matches c_2". Clearly there are goals states that are represented by n and satisfies constraint t, such as goal state $(c_1 c_2 c_3 c_1 c_3)$. Hence, we say n satisfies t. Consider $n' = (c_1 c_2 * * c_2)$. Clearly any goal state represented by n' will violate the hard constraint t. Hence, $cost(n, T_{hard}) = \infty$.

It is trivial to show that the cost $f(n) = g(n) + h(n)$ is a lower bound on the cost of any goal state that is in the set of concrete states represented by n. Recall that the cost of a goal state m is $cost(m)$ as defined in Equation 3.1.

The A* algorithm as adapted to our context will terminate, returning the mapping combination (i.e., a goal state) that best satisfies the domain constraints.

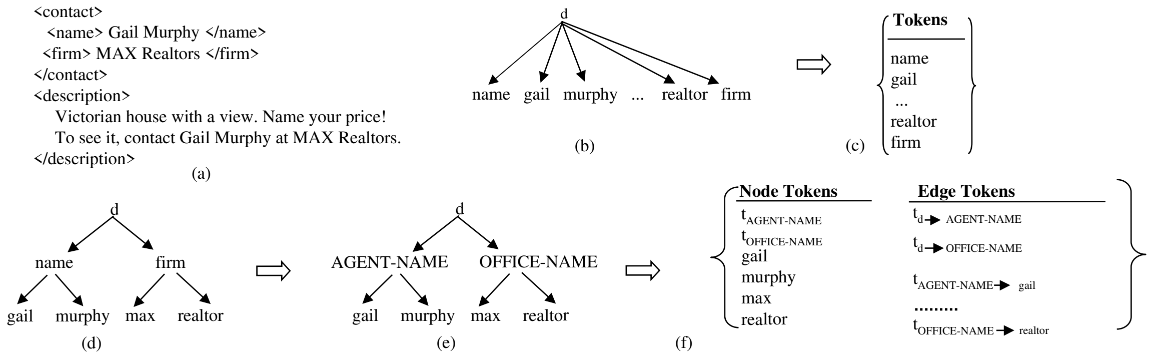


Figure 3.11: (a)-(c) The working of the *Naive Bayes Learner* on the XML element contact in (a); and (d)-(f) the working of the XML learner on the same element.

3.5.4 User Feedback

User feedback can further improve matching accuracy, and is necessary in order to match ambiguous schema elements (see Section 3.2.2). Our framework enables easy and seamless integration of such feedback into the matching process. If the user is not happy with the current mappings, he or she can specify constraints, then ask the *Constraint Handler* to output new mappings, taking these constraints into account. The constraint handler simply treats the new constraints as additional domain constraints, but uses them only in matching the current source.

The user can specify soft binary and numeric constraints, but more probably she will specify hard constraints, such as “ad-id does not match HOUSE-ID” and “brokerage matches OFFICE-INFO”. He or she can greatly aid the system by manually matching a few hard-to-match source elements, as we show empirically in Section 3.7.3. Typically these elements are in the middle of the schema tree and are involved in many domain constraints. Examples are brokerage, agent, and contact in Figure 3.3.b. By matching these elements, the user in effect “anchors” them down, and let the system work around them. One can also say that the user creates “islands of certainty” that help the system match elements around them more effectively.

3.6 Learning with Nested Elements

As we built LSD, we realized that none of our learners can handle the hierarchical structure of XML data very well. For example, the *Naive Bayes Learner* frequently confused instances of classes HOUSE, CONTACT-INFO, OFFICE-INFO, and AGENT-INFO. This is because the learner “flattens” out all structures in each input instance and uses as tokens only the *words* in the instance. Since the above classes share many words, it cannot distinguish them very well. As another example, Naive Bayes has difficulty classifying the two XML elements in Figure 3.11.a. The content matcher faces the same problems.

To address this problem we developed a novel learner that exploits the hierarchical structure of XML data. Our *XML Learner* is similar to the *Naive Bayes Learner* in that it also represents each input instance as a bag of tokens, assumes the tokens are independent of each other given the class, then multiplies the token probabilities to obtain the class probabilities. However, it differs from

Table 3.2: The XML learner algorithm.

XML Classifier - Testing Phase

Input: an XML element E **Output:** a predicted label for E

1. Create T, a tree representation of E. Each node in T represents a sub-element in E.
2. Use LSD (with other base learners) to predict for each non-leaf & non-root node in T a label. Replace each node with its label.
3. Generate the bag of text-, node-, & edge-tokens: B = $\{t_1, t_2, ..., t_k\}$.
4. Return label c that maximizes $P(c) * P(t_1|c) * P(t_2|c) * ... * P(t_k|c)$.

XML Classifier - Training Phase

Input: a set of XML elements S = $\{E_1, E_2, ..., E_n\}$;
the correct label for each E_i and each sub-element in E_i.

Output: the set of all text-, node-, & edge-tokens: V = $\{t_1, t_2, ..., t_m\}$;
probability estimates for tokens $P(t_i|c)$ & classes P(c)

1. For each E_j: (a) Create T_j, its tree representation.
   (b) Replace T_j's root with the generic root t_R; replace each non-root & non-leaf node with its label.
   (c) Create the bag of text-, node-, & edge-tokens B_j for T_j.
2. Compute V, P(c) & $P(t_i|c)$ as with the Naive Bayes learner.

Naive Bayes in one crucial aspect: it considers not only *text* tokens, but also *structure* tokens that take into account the structure of the input instance.

We explain the *XML Learner* by contrasting it to Naive Bayes on a simple example (see Table 3.2 for pseudo code). Consider the XML element contact in Figure 3.11.a. When applied to this element, the *Naive Bayes Learner* can be thought of as working in three stages. First, it (conceptually) creates the tree representation of the element, as shown in Figure 3.11.b. Here the tree has only two levels, with a generic root *d* and the words being the leaves. Second, it generates the bag of text tokens shown in Figure 3.11.c. Finally, it multiplies the token probabilities to find the best label for the element (as discussed in Section 3.4).

In contrast, the *XML Learner* first creates the tree in Figure 3.11.d, which takes into account the element's nested structure. Next, it uses LSD (with the other base learners) to find the best matching labels for all non-leaf and non-root nodes in the tree, and replaces each node with its label. Figure 3.11.e shows the modified tree. Then the *XML Learner* generates the set of tokens shown in Figure 3.11.f. There are two types of tokens: *node tokens* and *edge tokens*. Each non-root node with label l in the tree forms a node token t_l. Each node with label l_1 and its child node with label l_2 form an edge token $t_{l_1 \to l_2}$. Finally, the *XML Learner* multiplies the probabilities of the tokens to find the best label, in a way similar to that of the *Naive Bayes Learner*.

Node and Edge Tokens: Besides the text tokens (i.e., leaf node tokens), the *XML Learner* also deals with structural tokens in form of non-leaf node tokens and edge tokens. It considers non-leaf node tokens because they can help to distinguish between classes. For example, instances of CONTACT-INFO typically contain the token nodes AGENT-NAME and OFFICE-NAME, whereas instances of DESCRIPTION do not. So the presence of the above two node tokens helps the learner easily tell apart the two XML instances in Figure 3.11.a. It considers edge tokens because they can serve as good class discriminators where node tokens fail. For example, the node token AGENT-NAME cannot help distinguish between HOUSE and AGENT-INFO, because it appears frequently in the instances of both classes. However, the edge token d→AGENT-NAME tends to appear only in instances of AGENT-INFO, thus serving as a good discriminator. As yet another example, the presence of the edge WATERFRONT→"yes" strongly suggests that the house belongs to the class with a water view. The presence of the node "yes" alone is not sufficient, because it can also appear under other nodes (e.g., FIREPLACE→"yes").

Learning Weights for the XML learner: Recall from Section 3.3.1 that in order to combine the base learners' predictions, the meta-learner must learn for each base learner a set of weights that indicate the relative accuracy of that learner. Learning weights for the *XML Learner* requires building the LSD version without the *XML Learner*, then using that LSD version in conjunction with other base learners to create training data for the meta-learner. In the current prototype implementation, we considered a simpler way to obtain the weights for the *XML Learner*. Since this learner can be considered an enhanced version of the *Naive Bayes Learner*, we simply substitute the weights of the *Naive Bayes Learner* for the *XML Learner*. The substituted weights underestimate the accuracy of the *XML Learner*, thus is likely to cause some degradation in its performance. However, the degradation comes at the gains of ease in learning the weights. Even with the likely degradation, the *XML Learner* still helps improve accuracy across all experimental domains (see Section 3.7).

3.7 Empirical Evaluation

We have evaluated LSD on several real-world domains. Our goals were to evaluate the matching accuracy of LSD, and the contribution of different system components.

Domains and Data Sources: We report the evaluation of LSD on four domains, whose characteristics are shown in Table 3.3. Both Real Estate I and Real Estate II integrate sources that list houses for sale, but the mediated schema of Real Estate II is much larger than that of Real Estate I (66 vs. 20 distinct tags). Time Schedule integrates course offerings across universities, and Faculty Listings integrates faculty profiles across CS departments in the U.S.

We began by creating a mediated DTD for each domain. Then we chose five sources on the WWW. We tried to choose sources that had more complex structure. Next, since sources on the WWW are not yet accompanied by DTDs, we created a DTD for each source. In doing so, we were careful to mirror the structure of the data in the source, and to use the terms from the source. Then we downloaded data listings from each source. Where possible, we downloaded the entire data set; otherwise, we downloaded a representative data sample by querying the source with random input values. Finally, we converted each data listing into an XML document that conforms to the source schema.

In preparing the data, we performed only trivial data cleaning operations such as removing "unknown", "unk", and splitting "$70000" into "$" and "70000". Our assumption is that the learners LSD employs should be robust enough to deal with dirty data.

Table 3.3: Domains and data sources for experiments with LSD.

Domains	Mediated Schema			Sources	Downloaded Listings	Source Schemas			
	Tags	Non-leaf Tags	Depth			Tags	Non-leaf Tags	Depth	Matchable Tags
Real Estate I	20	4	3	5	502 – 3002	19 – 21	1 – 4	1 – 3	84 – 100 %
Time Schedule	23	6	4	5	704 – 3925	15 – 19	3 – 5	1 – 4	95 – 100 %
Faculty Listings	14	4	3	5	32 – 73	13 – 14	4	3	100 %
Real Estate II	66	13	4	5	502 – 3002	33 – 48	11 – 13	4	100 %

Table 3.3 shows the characteristics of the mediated DTDs, the sources, and source DTDs. The table shows the number of tags (leaf and non-leaf) and maximum depth of the DTD tree for the mediated DTDs. For the source DTDs the table shows the range of values for each of these parameters. The rightmost column shows the percentage of source-DTD tags that have a 1-1 matching with the mediated DTD.

Domain Constraints: Next we specified integrity constraints for each domain. For the current experiments, we specified only *hard constraints*. For each mediated-schema tag, we specified all non-trivial column and frequency constraints that we could find. For each pair of mediated-schema tags, we specified all applicable nesting constraints. Finally, we specified all contiguity and exclusivity constraints that we thought should apply to the vast majority of sources. See Table 3.1 for examples of hard constraints of different types. In general, most constraints we used were frequency, nesting, or column constraints. We specified very few contiguity and exclusivity constraints.

Setting Parameters of the Constraint Handler: To compute the cost of a mapping combination m, the *Constraint Handler* uses the following formula:

$$cost(m) = -\alpha * LH(m) + \beta_1 * cost(m, T_1) + \beta_2 * cost(m, T_2) + \cdots + \beta_v * cost(m, T_v), \qquad (3.2)$$

where $LH(m)$ represents the likelihood of m, $cost(m, T_i)$ represents the degree to which m satisfies domain constraints of type T_i, and $\alpha, \beta_1, \ldots, \beta_v$ are the scaling coefficients that represent the trade-offs among the cost components (see Section 3.5.2).

For our experiments we had to set the parameters α and β_i. Let T_1 represent the class of hard constraints. Since our experiments utilize only hard constraints, the cost formula can be rewritten as

$$cost(m) = -\alpha * LH(m) + \beta_1 * cost(m, T_1). \qquad (3.3)$$

If the mapping combination m satisfies the hard constraints, then $cost(m, T_1) = 0$ (see Section 3.5.2). Thus $cost(m) = -\alpha * LH(m)$. If m does not satisfy the hard constraints, then $cost(m, T_1) = \infty$. Thus $cost(m) = \infty$.

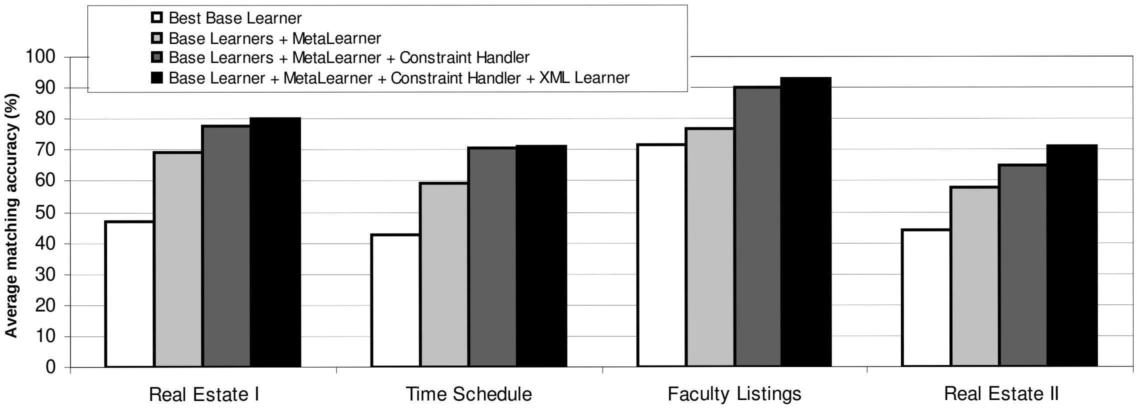


Figure 3.12: Average matching accuracy; experiments were run with 300 data listings from each source; for sources from which fewer than 300 listings were extracted, all listings were used.

The above two observations imply that we do not have to set parameter β_1 and that we can set parameter α to some arbitrary positive value. In our experiments we set α to 1.

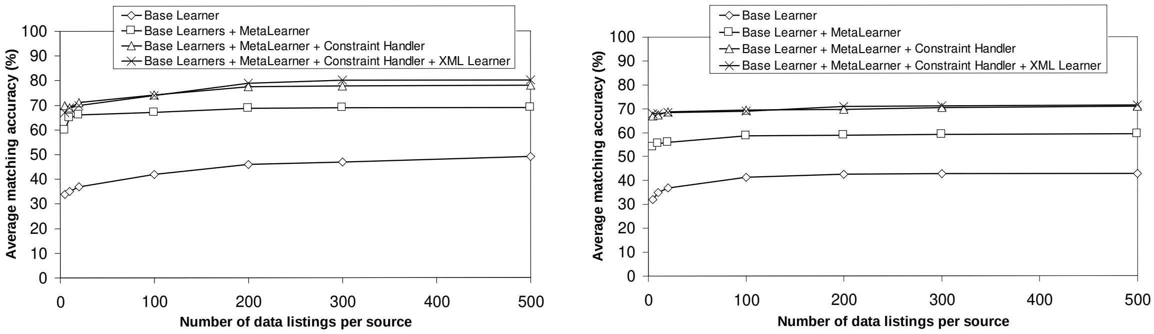
Experiments: For each domain we performed three sets of experiments. First, we measured LSD's accuracy and investigated how sensitive it is to the amount of data available from each source. Second, we conducted lesion studies to measure the contribution of each base learner and the *Constraint Handler* to the overall performance. We also measured the relative contributions of learning from schema elements versus learning from data elements. Third, we measured the amount of user feedback necessary for LSD to achieve perfect matching.

Experimental Methodology: To generate the data points shown in the next three sections, we ran each experiment three times, each time taking a new sample of data from each source. In each experiment on a domain we carried out all ten runs in each of which we chose three sources for training and used the remaining two sources for testing. We trained LSD on the training sources, then applied it to match the schemas of the testing sources. The *matching accuracy of a source* is then defined as the percentage of matchable source-schema tags that are matched correctly by LSD. The *average matching accuracy of a source* is its accuracy averaged over all settings in which the source is tested. The *average matching accuracy of a domain* is the accuracy averaged over all five sources in the domain.

3.7.1 Matching Accuracy

Figure 3.12.a shows the average matching accuracy for different domains and LSD configurations. For each domain, the four bars (from left to right) represent the average accuracy produced respectively by the best single base learner (excluding the XML learner), the meta-learner using the base learners, the domain *Constraint Handler* on top of the meta-learner, and all the previous components together with the *XML Learner* (i.e., the complete LSD system).

The results show that LSD achieves accuracy 71 - 92% across all four domains. In contrast, the best matching results of the base learners (achieved by either the Naive Bayes or the *Name Learner*,



(a) Matching accuracy for Real Estate I

(b) Matching accuracy for Time Schedule

Figure 3.13: The average domain accuracy as a function of the amount of data available per source.

depending on the domain) are only 42 - 72%.

As expected, adding the meta-learner improves accuracy substantially, by 5 - 22%. Adding the *Constraint Handler* further improves accuracy by 7 - 13%. Adding the *XML Learner* improves accuracy by 0.8 - 6.0%. In all of our experiments, the *XML Learner* outperformed the *Naive Bayes Learner* by 3-10%, confirming that the *XML Learner* is able to exploit the hierarchical structure in the data. The results also show that the gains with the *XML Learner* depend on the amount of structure in the domain. For the first three domains, the gains are only 0.8 - 2.8%. In these domains, sources have relatively few tags with structure (4 - 6 non-leaf tags), most of which have been correctly matched by the other base learners. In contrast, sources in the last domain (Real Estate II) have many non-leaf tags (13), giving the *XML Learner* more room for showing improvements (6%).

In Section 3.8 we identify the reasons that prevent LSD from correctly matching the remaining 10 - 30% of the tags.

Performance Sensitivity: Figures 3.13.a-b show the variation of the average domain accuracy as a function of the number of data listings available from each source, for the Real Estate I and Time Schedule domains, respectively. The results show that on these domains the performance of LSD stabilizes fairly quickly: it climbs steeply in the range 5 - 20, minimally from 20 to 200, and levels off after 200. Experiments with other domains show the same phenomenon. LSD thus appears to be robust, and can work well with relatively little data. One of the reasons this observation is important is that we can reduce the running time of LSD if we run it on fewer examples.

3.7.2 Lesion Studies

Figure 3.14.a shows the contribution of each base learner and the *Constraint Handler* to the overall performance. For each domain, the first four bars (from left to right) represent the average accuracy produced by LSD when one of the components is removed. (The contribution of the *XML Learner* is already shown in Figure 3.12.a). The fifth bar represents the accuracy of the complete LSD system, for comparison purposes. The results show that each component contributes to the overall performance, and there appears to be no clearly dominant component.

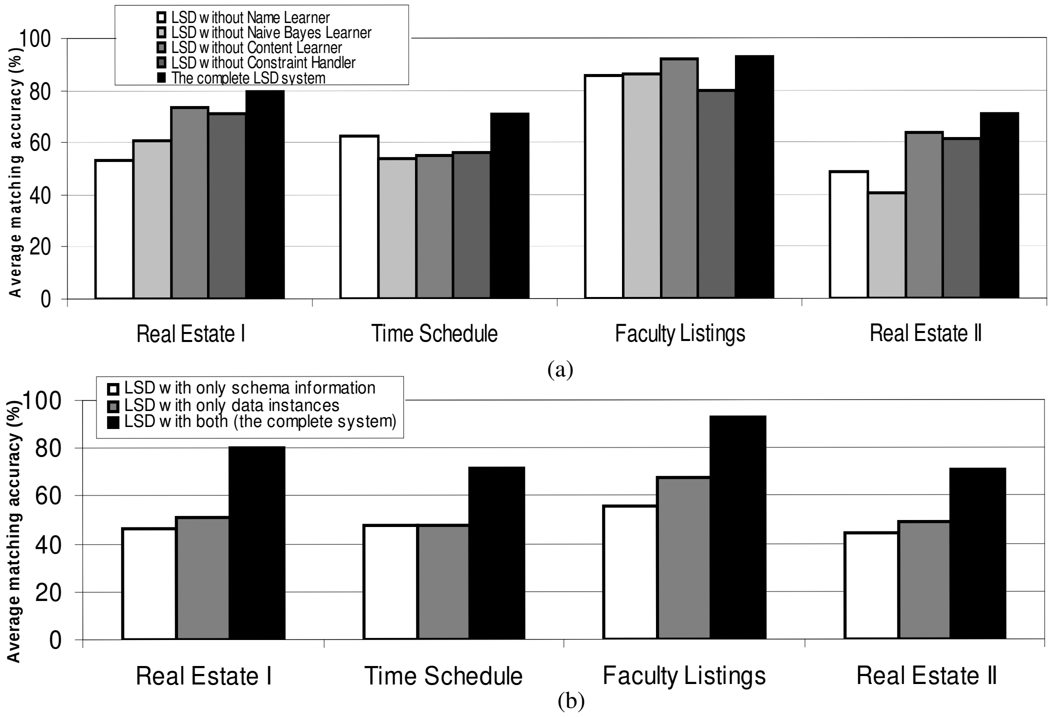


Figure 3.14: The average matching accuracy of LSD versions (a) with each component being left out versus that of the complete LSD system, (b) with only schema information or data instances versus that of the LSD version with both.

Given that most previous work exploited only schema information in the process of schema reconciliation, we wanted to test the relative contribution of learning from schema and learning from data information. In Figure 3.14.b, the first bar of each domain shows the average accuracy of the LSD version that consists of the *Name Learner* and the *Constraint Handler* (with only schema-related constraints). The second bar shows the average accuracy of the LSD version that consists of the *Naive Bayes Learner*, the content matcher, the *XML Learner*, and the *Constraint Handler* (with only data-related constraints). The third bar reproduces the accuracy of the complete system, for comparison purpose. The results show that with the current system, both schemas and data instances make important contributions to the overall performance.

3.7.3 Incorporating User Feedback

We performed experiments on the Time Schedule and Real Estate II domains to measure the effectiveness of LSD in incorporating user feedback. For each domain we carried out three runs. In each run we randomly chose three sources for training and one source for testing. Then we trained LSD using the training sources. Finally we applied LSD and provided feedback to it, in order to achieve the perfect matching on the testing source.

The interaction works as follows. First, we order the tags in the testing source. We use the same order employed by our A* implementation to refine states (i.e., to direct its search through the space of matching combinations, see Section 3.5.3). Then we enter the following loop until every tag has been matched correctly: (1) we apply LSD to the testing source, (2) LSD shows the predicted labels of the tags, in the above mentioned order, (3) when we see an incorrect label, we provide LSD with the correct one, then ask LSD to redo the matching process (i.e., rerun the constraint handler), taking the correct labels into consideration.

The number of correct labels we needed to provide to LSD before it achieved perfect matching, averaged over the three runs, was 3 for Time Schedule and 6.3 for Real Estate II. The average number of tags in the test source schemas for the two domains is 17 and 38.6, respectively. These numbers suggest that LSD can efficiently incorporate user feedback. In particular, it needs only a few equality constraints provided by the user in order to achieve perfect or near-perfect matching.

3.8 Discussion

We now address the limitations of the current LSD system, as well as issues related to system evaluation. The first issue to address is whether we can increase the accuracy of LSD beyond the current range of 71 - 92%. There are several reasons that prevent LSD from correctly matching the remaining 10 - 30% of the tags. First, some tags (e.g., suburb) cannot be matched because none of the training sources has matching tags that would provide training data. This problem can be handled by adding domain-specific recognizers or importing data from sources outside the domain.

Second, some tags simply require different types of learners. For example, course codes are short alpha-numeric strings that consist of department code followed by course number. As such, a format learner would presumably match it better than any of LSD's current base learners.

Finally, some tags cannot be matched because they are simply ambiguous. For example, given the following text in the source "course-code: CSE142 section: 2 credits: 3", it is not clear if "credits" refers to the course- or the section credits. Here, the challenge is to provide the user with a possible *partial* mapping. If our mediated DTD contains a label hierarchy, in which each label (e.g., credit) refers to a concept more general than those of its descendant labels (e.g., course-credit and section-credit) then we can match a tag with the most specific unambiguous label in the hierarchy (in this case, credit), and leave it to the user to choose the appropriate child label.

Efficiency: The training phase of LSD can be done offline, so training time is not an issue. In the matching phase, LSD spends most of its time in the *Constraint Handler* (typically in the range of seconds to 5 minutes, but sometimes up to 20 minutes in our experiments), though we should note that we did not spend any time optimizing the code. Since we would like the process of prediction and incorporating user feedback to be interactive, we need to ensure that the *Constraint Handler*'s performance does not become a bottleneck. The most obvious solution is to incorporate some constraints within some early phases to substantially reduce the search space. There are many fairly simple constraints that can be pre-processed, such as constraints on an element being textual or numeric. Another solution is to consider more efficient search techniques, and to tailor them to our context.

Overlapping of Schemas: In our experiments source schemas overlap substantially with the mediated schema (84-100% of source-schema tags are matchable). This is typically the case for "aggregator" domains, where the data-integration system provides access to sources that offer essentially the same service. We plan to examine other types of domains, where the schema overlap is much smaller. The performance of LSD on these domains will depend largely on its ability to recognize that a certain source-schema tag matches *none* of the mediated-schema tags, despite superficial resemblances.

Performance Evaluation: We have reported LSD's performance in terms of the predictive matching accuracy. Predictive accuracy is an important performance measure because (a) the higher the accuracy, the more reduction in human labor the system would achieve, and (b) the measure facilitates comparison and development of schema matching techniques. The next step is to actually quantify the reduction in human labor that the system achieves. This step has been known to be difficult, due to widely varying assumptions on how a semi-automatic tool such as LSD is used, and has just recently been investigated [MMGR02].

Examining Other Meta-Learning Techniques: The meta-learning technique employed by LSD is conceptually easy to understand and appears empirically to work well. Nevertheless, it is important to evaluate the technique further. Moreover, a wealth of meta learning techniques have been developed in the literature. Examining the suitability of these meta-learning techniques to schema matching is an important area for future research.

3.9 Summary

The problem of finding 1-1 semantic mappings between data representations arises in numerous application domains. In this chapter we have considered the problem in the context of data integration, an important data sharing application. We considered the scenario where sources export data in XML format, and have associated source DTDs. Then the problem is to find mappings between the tags of source DTDs and the mediated DTD.

We have described a solution to the above matching problem that employs and extends machine learning techniques. Our approach utilizes both schema and data from the sources. To match a source-DTD tag, the system applies a set of learners, each of which looks at the problem from a different perspective, then combines the learners' predictions using a meta-learner. The meta-learner's predictions are further improved using domain constraints and user feedback. We also developed a novel *XML Learner* that exploits the hierarchical structure in XML data to improve matching accuracy. Our experiments show that we can accurately match 71-92% of the tags in several domains.

In subsequent chapters we consider more complex matching scenarios, such as finding complex semantic mappings and matching ontologies. We extend the solution developed in this chapter to cover these scenarios.

Chapter 4

COMPLEX MATCHING

Virtually all current matching approaches (including the LSD approach described in the previous chapter) have focused on finding only 1-1 semantic mappings, such as location = area and comments = description. They do not consider *complex mappings* such as listed-price $=$ price $*(1-$ discount-rate$)$ and address $=$ *concat*(location, state). Since in practice complex mappings make up a significant portion of semantic mappings between data representations, the development of techniques to discover such mappings is essential to any practical mapping effort.

In this chapter we describe the COMAP approach which extends LSD to semi-automatically discover complex mappings. The next section defines the specific complex matching problem that we consider. Sections 4.2–4.4 describe our solution. Section 4.5 provides an empirical evaluation of the approach. Section 4.6 discusses the limitations and extensions of our solution, and Section 4.7 summarizes the chapter.

4.1 Complex Matching for Relational Schemas

For simplicity of exposition, in this chapter we consider complex matching for a relatively simple representation - relational schemas. However, the principles underlying our approach carry over to more complex data representation languages, such as XML DTDs and ontologies. Section 4.6 describes the changes that need to be made to adapt our solutions to such languages.

Recall from Chapter 2 that a *relational schema* consists of multiple *tables*, each with a set of *attributes*. Consider two relational databases S and T on house listings in Figure 4.1.a, which are managed by two different real-estate companies, respectively. Figure 4.1.a shows that the schema of database S has one table, LISTINGS, which list the houses for sale. This table has five columns which correspond to the five attributes area, listed-price, agent-name, agent-address, and agent-phone.

Each attribute has a domain from which its values are drawn. A table is populated with a set of tuples, where each tuple has one value for each attribute of the relation. For example, table LISTINGS has a tuple (“Atlanta, GA”,$370800,“Mike Brown”,“Athens, GA”, “(217) 354 1963”) which assigns value “Atlanta, GA” to attribute area, etc.

Now suppose that the two real-estate companies have decided to merge. To cut costs, they are consolidating the databases. Specifically, they have decided to eliminate database T by transferring all house listings from database T to database S. Such data transfer is not possible without knowing the semantic mappings between the relational schemas of the databases. The *mapping* of an attribute s in database S specifies how to create instances of s using the data of database T. If the instances of an attribute t of T can be transferred to be instances of s *without any modification*, we say that s has an *1-1 mapping* from T and denote that mapping as $s = t$. In Figure 4.1.a the mapping area $=$ location is a 1-1 mapping.

If the instances of s are obtained by transforming instances of t or by combining instances of multiple attributes in T, then we say s has a *complex mapping* from T. For example, attribute agent-

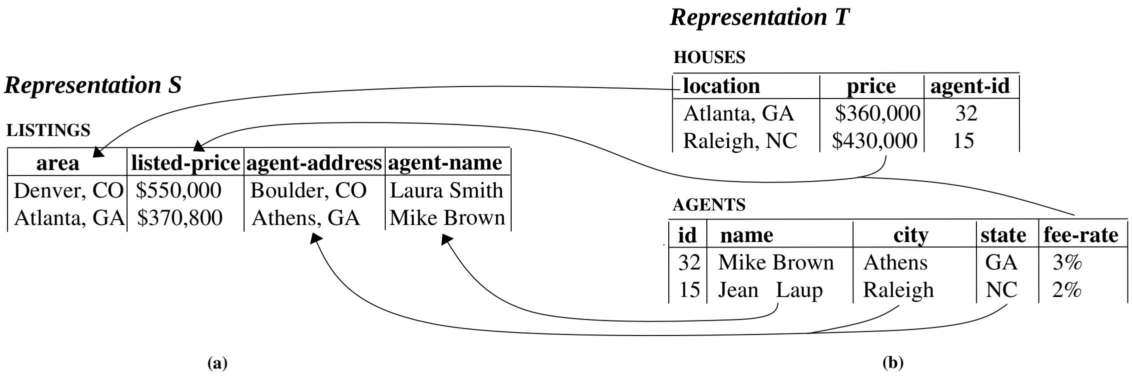


Figure 4.1: The schemas of two relational databases on house listing, and the semantic mappings between them.

address in Figure 4.1.a has a complex mapping because an instance of agent address is obtained by concatenating an instance of city and an instance of state.

In general, a complex mapping for relational data can be expressed with a SQL query [YMHF01] or a first-order logic statement that specifies the *attributes* involved, the *formula* used to combine the instances of the attributes, and the *relationship* among the instances. For example, consider the mapping agent-address = *concat*(city,state). Here, the attributes involved are city and state, the combination formula is *concat*(city,state), and the relationship is that the values of city and state that appear in the formula must belong to the *same tuple* of table AGENTS.

As another example, consider the complex mapping $\text{listed-price} = \text{price} * (1 + \text{fee-rate})$ (Figure 4.1). Here, the attributes involved are price and fee-rate, the combination formula is $\text{price} * (1 + \text{fee-rate})$, and the relationship is that price belongs to the house whose agent has fee-rate.

Thus, to obtain values for listed-price, we must *join* the two tables HOUSES and AGENTS based on the condition agent-id = id, then compute $\text{price} * (1 + \text{fee-rate})$ where price and fee-rate belong to the same tuple in the table created by the join operation. We refer to the condition agent-id = id as a *join path* for the above two tables.

Now we can state the specific complex matching problem that we consider as follows: *Given two relational schemas S and T, for each attribute s of S find the 1-1 or complex mapping that best transforms data instances of T into instances of s, utilizing all available schema information, data instances, and domain knowledge*.

Without loss of generality, we consider only the case where schema *S* has a single table. Furthermore, as the first step, we consider only certain types of relationships among the attributes that participate in a complex mapping. Specifically, we assume that if the attributes come from the same table, then they belong to the same tuple, and that if they come from multiple tables, then there exists only a single join path among these tables, via which the attributes relate.

To illustrate the join-path assumption, in the case of schema *T* in Figure 4.1.a, we can assume that agent-id = id is the only permissible join path between the two tables HOUSES and AGENTS, and that if attributes from these two tables participate in any complex-mapping formula, then they

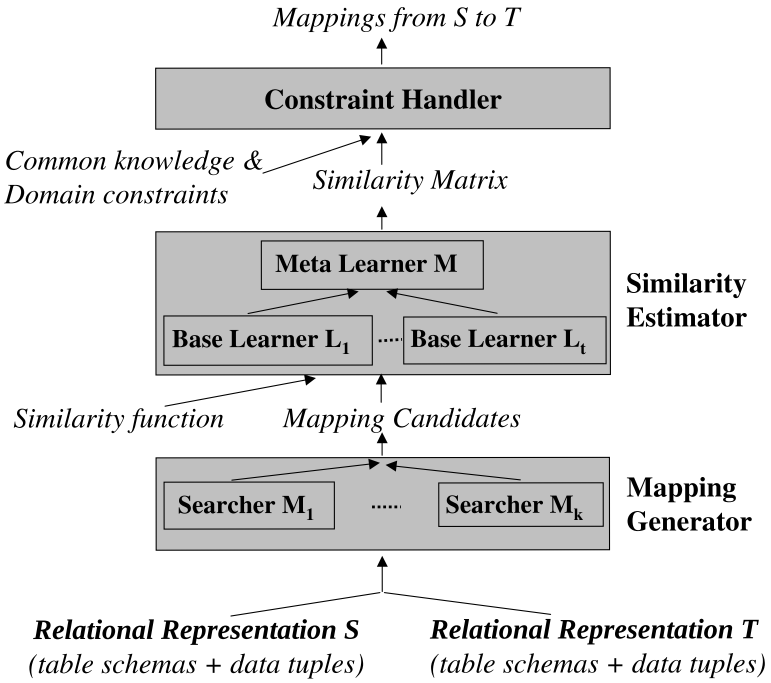


Figure 4.2: The COMAP architecture.

must relate to each other via that join path.

Therefore, with the above assumptions, the problem of finding the complex mapping for an attribute *s* of schema *S* reduces to finding the "right" attributes of *T* and the "right" formula that combines those attributes. We do not have to worry about finding the "right" relationship among the attributes. In the next section we discuss the COMAP approach to solving this problem. In Section 4.6, we discuss removing the join-path assumption and show how COMAP can be extended to solve the problem of finding the "right" relationship.

4.2 *The* COMAP *Approach*

The key idea underlying our COMAP approach is to reduce complex matching to an 1-1 matching problem. Specifically, for each attribute of *S* we search the space of all 1-1 and complex mappings to find a small set of best mapping candidates, then add these candidates to schema *T*, as additional "composite" attributes. Finally, we apply LSD to match *S* and the expanded schema *T*. If an attribute *s* in *S* is matched with a "composite" attribute in the expanded schema, then we say the best mapping for *s* is the complex mapping that corresponds to the "composite" attribute.

The above idea leads to the COMAP architecture shown in Figure 4.2, which consists of three main modules: *Mapping Generator, Similarity Estimator*, and *Constraint Handler*. The *Mapping Generator* takes as input two relational schemas *S* and *T*, together with their data instances. For each attribute *s* of *S*, it generates a set of *mapping candidates*. Next, it sends all candidates to the *Similarity Estimator*. For each candidate this module computes a similarity value between the candidate and each attribute of S. The output of this module is therefore a matrix that stores the similarity value of $\langle$mapping candidate, attribute of $S\rangle$ pairs. Finally, the *Constraint Handler* takes the similarity matrix, together with the domain constraints, and outputs the best semantic mappings for the attributes of schema S.

In the rest of this section we describe the *Mapping Generator*, which plays a key role in the COMAP system and constitutes a significant contribution of this chapter. Section 4.3 describes the *Similarity Estimator* and Section 4.4 describes the *Constraint Handler*.

We begin by describing how the *Mapping Generator* employs multiple modules called *searchers* to efficiently find promising complex-mapping candidates. Then we describe the implementation of searchers.

4.2.1 Employing Multiple Searchers

For each attribute of schema S, the *Mapping Generator* must search the space of all 1-1 and complex mappings to find a relatively small set of best mapping candidates. Here, the key challenge is that the search space tends to be extremely large or infinite, and hence we must develop efficient search methods. The *Mapping Generator* addresses this challenge by *breaking down* the search space. It employs *multiple searchers*. Each searcher exploits a certain type of information in the schema or data to efficiently conduct a specialized search. The set of best mapping candidates then is the *union* of the mapping candidates returned by the searchers. The following example illustrates the idea behind the searchers:

Example 5 Consider the *Text Searcher* and *Numeric Searcher* (which are described in detail later in this section). Given an attribute of schema S, the *Text Searcher* examines the space of all mappings that are *concatenations* of attributes in schema T, to find a small set of mappings that best match the attribute. The *Text Searcher* accomplishes this by analyzing the textual properties of the attributes of the two schemas. In the case of agent-address (Figure 4.1.a), this searcher may return the following mappings, in decreasing order of confidence: $concat(\mathsf{city}, \mathsf{state}), \mathsf{location}, concat(\mathsf{name}, \mathsf{state})$.

The *Numeric Searcher* exploits the values of numeric attributes to find mappings that are *arithmetic expressions* over the attributes of schema T. Given attribute listed-price of schema S in Figure 4.1.a, this searcher may return the following mappings: $\mathsf{price} * (1 + \mathsf{fee\text{-}rate}), \mathsf{price}, \mathsf{price} + \mathsf{agent\text{-}id}.\Box$

In general, a searcher is applicable to only certain types of attributes. For example, the *Text Searcher* examines the *concatenations* of attributes, and hence is applicable to only *textual* ones. This searcher employs a set of heuristics to decide if an attribute is textual. The heuristics examine the ratio between the number of numeric and non-numeric characters, as well as the average number of words per data value. As another example, the *Numeric Searcher* examines arithmetic expressions of attributes, and hence is applicable to only *numeric* attributes. Note that for each attribute of schema S, the *Mapping Generator* "fires" all applicable searchers, then unions their results to obtain the final set of best mapping candidates.

The key benefit of using multiple searchers is that the COMAP system is highly modular and easily extensible. For example, if later we have developed a specialized searcher that finds complex mappings for address attributes, then we can just plug the searcher into the system. We can also simply leverage mapping techniques that have been developed in other communities to develop specialized searchers (e.g., see the *Overlap Numeric Searcher* in Section 4.2.4).

4.2.2 Beam Search as the Default Search Technique

Searchers can be implemented using any suitable technique. We however provide a default implementation that uses *beam search*. In what follows we describe this default implementation and illustrate it using the *Text Searcher*.

The basic idea behind beam search is that at each stage of the search process, the searcher limits its attention to only k most promising candidates, where k is a pre-specified number. This way, the searcher can conduct a very efficient search in the typically vast space of states. The input of beam search consists of a set of initial *states*, a set of *operators* that can be applied to current states to construct new states, a *scoring function* that evaluates the quality of a state, a *stopping criterion* that can be used to decide if a state is a goal state, and a beam width k. The output of beam search is a goal state. The high-level algorithm of beam search is as follows:

1. Let C be the set of initial states.
2. Apply the given operators to states in C to construct a set of new states N.
3. Compute the scores of the states in N.
4. Select the k states from $C \cup N$ that have the highest scores. Set C to be this set.
5. If the stopping criterion is satisfied by a state in N, return the state; otherwise repeat Steps 1-4.

We adapt beam search to the complex matching context as follows. Consider the problem of finding the best mapping for attribute s of S. We take each candidate mapping for attribute s to be a state. The set of initial states is then the set of attributes of T, and the goal state is the best complex mapping for s.

Now that we have defined the states, two challenges arise. The first challenge is that we must find a score function that compute for each candidate mapping m a similarity value between m and attribute s. We use machine learning techniques to solve this problem. Specifically, we build a classifier for s using the data in schema S, then apply it to classify candidate mapping m. The classifier will return a confidence score that indicates the similarity between m and s.

The second challenge is that we do not have a criterion for deciding when to stop the search. We solve this as follows. In each iteration (Steps 1-4) of the beam search algorithm, we keep track of the highest score of any candidate mappings (that have been seen up to that point) in a variable v. Then if the difference in the values of v in *two consecutive iterations* is less than a pre-specified threshold δ, we stop the search and return the mapping with the highest score as the best mapping for s.

The following example illustrates the adaptation of beam search to our matching context:

Example 6 (Text Searcher) Consider the *Text Searcher* which finds mappings that are concatenations of attributes. Given a target attribute, such as agent-address, it begins by considering all 1-1 mappings, such as agent-address = location, agent-address = price, and so on (see Figure 4.1.a).

The *Text Searcher* then computes a score for each of the above mappings as follows. Consider the mapping agent-address = location. (1) The searcher assembles a set of training examples for attribute agent-address: a data instance in schema S is labeled "positive" if it belongs to agent-address and "negative" otherwise. (2) It trains a Naive Bayes text classifier on the training examples to learn a model for agent-address. The data instances are treated as text fragments in this training process. (See Section 3.4.3 in Chapter 3 for a description of the Naive Bayes text Classifier.) (3) It applies the trained Naive Bayes text classifier to each instance of location (in schema T) to obtain an estimate of the probability that that instance belongs to agent-address. (4) Finally, it returns the average of the instance probabilities as the desired score.

After computing the scores for the 1-1 mappings, the *Text Searcher* conducts a beam search. It starts by picking the k highest-scoring 1-1 mappings. Next, it generates new mappings by concatenating each of the k mappings with each attribute in T. For example, if agent-address = city is picked, then agent-address = *concat*(city, state) is generated as a new mapping. The searcher then computes the score of the new mappings as described above. In general, the score of mapping $s = f(t_1, \ldots, t_n)$ is computed by comparing the column corresponding to s and the "composite column" corresponding to $f(t_1, \ldots, t_n)$, and the comparison is carried out using the Naive Bayes text classifier. The searcher then picks the k best mappings among all mappings that it has seen so far, and the process repeats. The searcher stops when the difference in the scores of the best mappings in two consecutive iterations does not exceed a pre-specified threshold δ.□

In the next subsection we describe the currently implemented searchers, many of which employ variants of the above beam search technique. But first, we note that while the source and target representations typically do not share any data, there are still many practical mapping scenarios where they actually do (e.g., when two sources describe the same companies, or when two databases are views created from the same underlying database). We shall refer to the above scenarios as the *"disjoint"* and *"overlap"* cases, respectively. Clearly, in the "overlap" case the shared data can provide valuable information for the mapping process (e.g., as shown in [RHS01]). Hence, we have developed searchers for both cases.

4.2.3 Searchers for Disjoint Data Scenarios

The current COMAP implementation has four searchers for the disjoint data cases: *Text-, Numeric-, Category-*, and *Schema Mismatch Searcher*. The first three are general-purpose searchers that handle a large class of complex mapping, while the last searcher handles only a specific type of complex mapping. These searchers serve to illustrate the utility of our approach. In practice a COMAP-like system is likely to have tens of searchers, both general-purpose and domain-specific.

The *Text Searcher* was described in the previous subsection. We now describe the remaining three searchers.

Numeric Searcher: Given a numeric attribute s of schema S, this searcher examines the space of all 1-1 and complex mappings over the numeric attributes of schema T, in order to find the best mapping for s.

In building this searcher, we face two issues. The first issue is how to evaluate a complex mapping m. We observe that the column (of data values) of attribute s forms a value distribution, and that the "composite column" that is created by applying mapping m to data values in schema T forms another value distribution. Hence, we can compute the score of m to be a number that indicates the similarity between these two value distributions. The Kullback-Leibler divergence is a measure commonly used to compute distribution similarities [CT91, MS99]. Hence, we use this measure in the *Numeric Searcher*.

The second issue that we consider is the type of mappings the *Numeric Searcher* should examine. It is clear that the *Numeric Searcher* cannot consider an arbitrary space of mappings, because this will likely lead it to overfit the data and find an incorrect mapping. Hence, we limit the *Numeric Searcher* to consider only a *restricted* set of mappings over the numeric attributes of schema T. These mappings are supplied by the user based on his or her domain knowledge. For example, if schema T has attribute lot-area which uses the "square feet" measure unit, the user may want to supply the common conversion mapping "lot-area / 43560" (for the case where lot area is measured in acres in schema S). If schema T has attributes num-full-baths and num-half-baths, the user may want to supply the complex mapping num-full-baths + num-half-baths (for the case where schema S only lists the total number of bathrooms).

We note that, in a sense, having the user supplying likely complex mappings is the best we can do to discover arithmetic relationships, when the schemas involved do not share any data. Even after the user has supplied a set of complex mappings, it is still very difficult to discover likely relationships, due to the large number of numeric attributes and the difficulties of detecting similar value distributions. The *Numeric Searcher* substantially helps the user in this aspect. Furthermore, in the next subsection we show that when the schemas involved share some data, the *Numeric Searcher* can exploit the data to discover complex arithmetic relationships, without requiring input from the user.

Category Searcher: The *Category Searcher* finds "conversion" mappings between categorical attributes, such as $\texttt{waterfront} = f(\textsf{near-water})$, where $f(``yes") = 1$ and $f(``no") = 0$. Given a target-schema attribute A (e.g., waterfront), the searcher analyzes its data instances to estimate the number of *distinct values* of the attribute. If the number of distinct values n is below a threshold (currently set at 10), then the searcher considers the attribute to be a *category attribute* and considers each distinct value to be a *category value*. Otherwise, the searcher terminates, indicating that A is not a category attribute.

In the case A is a category attribute, the searcher attempts to find the corresponding category attribute in the source schema. It analyzes the data instances of source-schema attributes to locate likely category attributes. Suppose it finds m category attributes $B_1, B_2, \ldots, B_m$ of the source schema, such that each attribute B_i also has the same number of distinct values as A. Next, it uses the Kullback-Leibler divergence to compute the similarity between the value distribution of each of the B_i and A. It then prunes the B_i whose associated Kullback-Leibler similarity value falls below a pre-specified threshold.

Let the remaining category attributes be $B_1, \ldots, B_p, p \leq m$. Then for each $B_i, i \in [1, p]$, the searcher attempts to find a conversion function f_i that transforms the values of B_i to those of A. Currently, the function f_i that the searcher produces maps the value with the highest probability in the distribution of B_i with that in the distribution of A, then the value with the second highest probability in the distribution of B_i with that in the distribution of A, and so on.

The output of the searcher given the input attribute A is then the attributes $B_1, \ldots, B_p$ together with the conversion functions $f_1, \ldots, f_p$.

Schema Mismatch Searcher: This searcher finds mappings that relate the *data* of the source representation with the *schema* of the target representation. For example, it detects the case where if source attribute house-description contains the word "fireplace" in its value (e.g., "3-bedroom house with large fireplace"), then the target attribute fireplace should have the value 1.

This searcher applies only to category attributes. The searcher uses techniques similar to those employed by the *Category Searcher* in order to detect if a given target-schema attribute A is categorical with two distinct values. If it is, the searcher searches for the appearance of the name of A in the data instances of source-schema attributes. If the name of A appears at least t times in distinct data values of a source-schema element B (where t is pre-specified and is currently set at 5), then there is a possibility of schema mismatch between A and B.

The attribute B is then transformed into a category attribute B', such that each data instance of B is transformed into "1" if it contains the name of A, and 0 otherwise. Next, the *Schema Mismatch Searcher* applies techniques similar to those employed by the *Category Searcher* to create the conversion function f that transforms data values of B' into those of A.

This searcher therefore does not handle all possible cases of schema mismatch. However, it handles some of the very common cases, such as those between fireplace, sewer, and electricity and house-description.

4.2.4 Searchers for Overlapping Data Scenarios

We have described searchers for "disjoint" scenarios. In the "overlap" scenarios, it turns out that the shared data can provide valuable information for the mapping process; and indeed, there have been some works that rely on the overlap data to perform matching (e.g., [PE95, RHS01]). Hence, we consider the "overlap" case and adapt our searchers to exploit the shared data. In what follows we describe the adaptation that we have carried out:

Overlap Text Searcher: In the "overlap" case we can use this module to obtain improved mapping accuracy. The module applies the *Text Searcher* to obtain an initial set of mappings. It then uses the overlap data to re-evaluate the mappings: the new score of each mapping is the fraction of the overlap data entities for which the mapping is correct. For example, suppose we know that representations S and T in Figure 4.1 share a house listing ("Atlanta, GA,..."). Then, when re-evaluated, mapping agent-address $=$ location receives score 0 because it is not correct for the shared house listing, whereas mapping agent-address $= concat(\texttt{city},\texttt{state})$ receives score 1.

Overlap Numeric Searcher: For each numeric attribute s of schema S, this module finds the best mappings that are arithmetic expressions over numeric attributes of schema T. Suppose that the overlap data contains ten entities (e.g., house listing) and that the numeric attributes of T are t_1, t_2, t_3. Then for each entity the searcher assembles a *numeric tuple* that consists of the values of s and t_1, t_2, t_3 for that entity. Next, it applies an *equation discovery system* to the ten assembled numeric tuples in order to find the best arithmetic-expression mapping for attribute s. We use the recently developed LAGRAMGE equation discovery system [TD97] (the misspelling of the system name was intentional). This system uses a context-free grammar to define the search space of mappings. As a result, the *Numeric Searcher* can incorporate domain knowledge on numeric relationships in order to efficiently find the right numeric mapping. LAGRAMGE conducts a beam search in the space of arithmetic mappings. It uses the numeric tuples and the sum-of-squared-errors formula commonly used in equation discovery to compute mapping scores. For more details on LAGRAMGE, see [TD97].

Overlap Category- & Schema Mismatch Searchers: Similar to the *Overlap Text Searcher*, these searchers use their non-overlap counterparts (i.e., the *Category Searcher* and *Schema Mismatch*

Searcher) to find an initial set of mappings, then re-evaluate the mappings using the overlap data.

4.3 The Similarity Estimator

In the previous section we described the *Mapping Generator*. In this section we shall discuss the *Similarity Estimator* and in the next section the *Constraint Handler*. While the ideas underlying these two modules have been introduced in the LSD system (Chapter 3), here we show how they are extended and integrated with the *Mapping Generator* in order to build a comprehensive system that discovers both 1-1 and complex mappings.

After the *Mapping Generator* has suggested a set of mappings for a target attribute s, the *Similarity Estimator* examines each mapping m in detail and assigns to it a final score that measures the similarity between m and s. Notice that the searchers that suggested m have given it similarity scores. The *Text Searcher*, for example, assigns a score to each mapping it suggested using the Naive Bayes text classifier, as described earlier. However, for the sake of speed, such searcher-suggested scores were computed based on only a single type of information (e.g., the word frequencies, in the case of Naive Bayes), and therefore are not necessarily the most accurate.

The goal of the *Similarity Estimator* is then to exploit *all* available types of information to compute a more accurate score for each mapping. To this end, similar to the LSD system of Chapter 3, the *Similarity Estimator* uses a *multi-strategy learning* approach: to compute the score of a mapping, it applies multiple learners, each of which exploits a specific type of information to suggest a score, and then combines the suggested scores using a meta-learner.

The current implementation of *Similarity Estimator* uses an extra learner: the *Complex Name Learner*. Given a complex mapping m for attribute s, this learner computes the similarity between the names of m and s. The name of m is defined to be the concatenation of the names of all attributes that participate in m. The name of an attribute includes also the name of the table. For example, the name of attribute city of table AGENTS in Figure 4.1.a is "agents city". The implementation of *Complex Name Learner* is similar to that of the *Name Learner* in the LSD system (see Chapter 3).

For each attribute s of schema S, the *Similarity Estimator* combines learners' scores as follows. Suppose the *Mapping Generator* has applied the searchers $D_1, D_2, \ldots, D_n$ to s, to find a set M of candidate mappings. Consider mapping $m \in M$ and suppose that it is produced by searcher D_1, with a score c_1. The *Similarity Estimator* then applies the remaining searchers $D_2, \ldots, D_n$ to m, to generate scores $c_2, \ldots, c_n$. Next, the *Similarity Estimator* applies the *Complex Name Learner* to m, to generate score c_{NL}. Finally, the *Similarity Estimator* combines the scores $c_1, c_2, \ldots, c_n$ and c_{NL} using a meta-learner.

The meta-learner used by the *Similarity Estimator* is the same meta-learner described in Chapter 3 for the LSD system. The only difference is that there, in the data integration setting, the meta-learner is trained using data from several sources that have been manually mapped to the mediated schema. Here, the meta-learner is trained using the data associated with schema S.

4.4 The Constraint Handler

Once the *Similarity Estimator* has revised the score of the suggested mappings of *all* attributes of S, the best mapping combination is simply the one where each attribute of S is assigned the mapping with the highest score. However, this mapping combination may not be optimal, in the sense that it

Table 4.1: Real-world domains for experiments with COMAP.

Domains	Schema T		Schema S	# of 1-1 mappings	# of complex mappings				
	# tables	# attributes	# of attributes		Total	Text	Numeric	Category	Schema Mismatch
Real Estate I	2	16	6	2	4	4	0	0	0
Inventory	3	29	26	15	11	3	4	4	0
Real Estate II	2	31	19	6	13	5	4	3	1

may still violate certain domain constraints. For example, it may map *two* attributes of T to attribute listed-price of S, thus violating the domain heuristic that a house has only one price. Hence, the job of the *Constraint Handler* is to search for the best mapping combination that satisfies a given set of domain constraints.

This module is based on the constraint handler module employed by the LSD system in Chapter 3. There the handler deals only with 1-1 mappings, whereas here we have extended it to deal with both 1-1 and complex mappings. A particularly interesting extension that we have developed allows the handler to "clean up" complex mappings using domain constraints. For example, in our experiments the *Numeric Searcher* frequently suggested mappings such as $\text{lot-area} = (\text{lot-sq-feet}/43560) + 1.3\text{e-}15 * \text{baths}$. If the handler knows that source attribute baths maps to target attribute num-baths, and that lot area and the number of baths are semantically unrelated and hence typically do not appear in the same formula, then it can drop the terms involving baths (provided that the value of the term is very small), thus transforming the above mapping into the correct mapping. The same reasoning also applies to text mappings suggested by the *Text Searcher*.

4.5 Empirical Evaluation

We have evaluated COMAP on several real-world domains. Our goals were to evaluate the matching accuracy of COMAP and to measure the relative contributions of the different system components.

Domains and Data Sources: We report the evaluation of COMAP on three domains, whose characteristics are shown in Table 4.1. "Inventory" describes product inventories of a grocery business. Both "Real Estate I" and "Real Estate II" describe houses for sale, but the schemas of "Real Estate II" are much larger than those of "Real Estate I" (31-19 vs. 6-16 attributes). These real-estate domains are created from the real-estate domains described in Chapter 3 (for experiments with LSD), by removing, merging, and splitting certain schema elements. The objective was to create real-estate schemas that have a fair number of complex mappings, for the purpose of evaluating COMAP.

We began by obtaining a database for each domain. The "Inventory" database was selected from the sample databases of Microsoft Access 97. The "Real Estate" databases were selected from the set of real-estate databases we obtained from the Web for experiments with LSD. We focused on choosing complex databases with a mixture of attribute types (text, numeric, categorical, etc.). The numbers of tables and attributes in each database are shown under the headline "Source Representation" of Table 4.1. Next, for each database we asked a volunteer to examine and create complex query formulas that combine the attributes of the database. Examples of such queries are price = our-price * (1 - discount-rate), name = *concat*(first-name,last-name), and so on.

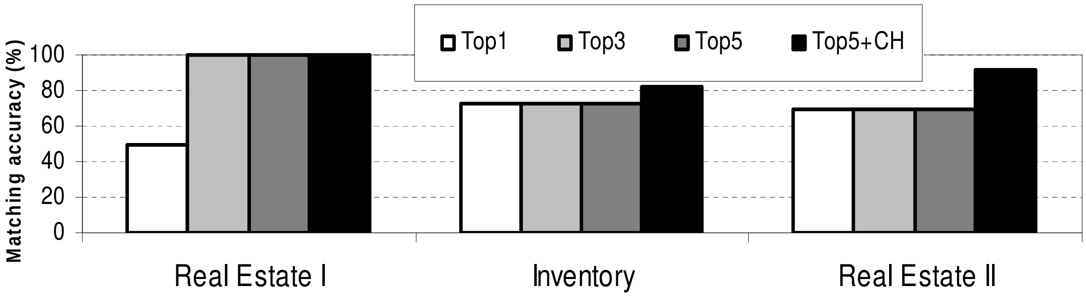
As discussed in Section 4.2, both the "overlap" and "disjoint" scenarios where the source and target representations do and do not share data occur frequently in practice. Hence we created both scenarios for experimental purposes. In the "overlap" scenario, for each database T that we described above, we apply the query formulas to T, then "glue" the query results together to create a new database S. The databases S and T therefore share a set of data entities. Our goal is then to find for each attribute s of S the best complex mapping over the attributes of T. In other words, for each attribute s of S we apply COMAP to re-discover the complex query that we use to create s in the first place.

In the "disjoint" scenario, for each database T, we partitioned T into disjoint databases T_1 and T_2 (by splitting all tables of T in half). Next, we apply the query formulas to database T_2 and "glue" the results together to form a new database S. The databases S and T_1 do not share any data entity. Our goal is to find complex mappings between S and T_1.

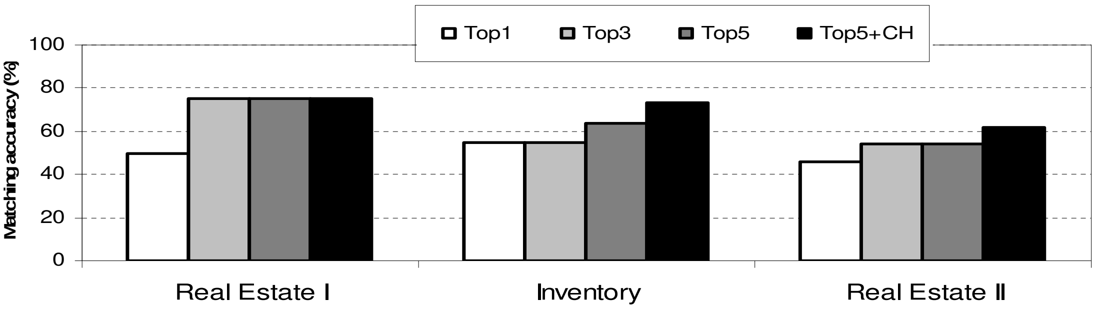
We note that in both scenarios the schemas of database S are the same, and are summarized in Table 4.1. The table shows the number of attributes in the schema, the number of 1-1 mappings (to the source schema) and then the number of complex mappings, broken down into mappings of different types.

Experiments: We had COMAP produce four types of output for each attribute of schema S: the top k mappings (i.e., those with the highest final score as computed by the *Similarity Estimator*) for $k = 1,3,5$, and the top 5 mappings together with the best mapping as predicted by the *Constraint Handler* for that attribute. Recall from our discussion on evaluating a solution output (Section 2.2.1 of Chapter 2) that if a solution output contains the correct mapping, the user can quickly locate it, and hence the output is counted as correct. The accuracy rate is then the fraction of attributes in S whose outputs are correct. We refer to the four accuracy rates as *top1, top3, top5*, and *top5+CH*.

Matching Accuracy: Figure 4.3.a-b shows the matching accuracy for different domains in the "overlap" and "disjoint" cases, respectively. For each domain, the bars from left to right represent the accuracy rates $top1, top3, top5$, and $top5+CH$. Figure 4.3.a shows that in the "overlap" case COMAP achieves high $top5+CH$ accuracy across all four domains, ranging from 82 to 100%. The $top1$ accuracy 50-73% is quite reasonable. Examining the top 3 or top 5 best mappings improves accuracy by as much as 50%. Examining also the best mapping returned by the *Constraint Handler* in addition to the top 5 mappings further improves accuracy by as much as 23%. In our experiments we found that most text, categorical and schema mismatch mappings were correct. This is not surprising given that the data overlap. However, most numerical mappings were incorrect because they contain extraneous terms (e.g., mappings for lot-area contain 1.3e-15 $*$ baths). The *Constraint Handler* cleaned up these mappings, as described in the previous section, thus yielding significant accuracy improvement. These results suggest that equation discovery systems achieve their greatest potential for schema matching *only* in conjunction with other searchers and with exploiting domain knowledge.



(a) accuracy for the "overlap" case



(b) accuracy for the "disjoint" case

Figure 4.3: Matching accuracies for complex mappings.

In the "disjoint" case (Figure 4.3.b), the $top5+CH$ accuracy rate is lower, in the range 62-75%, but still quite reasonable. The main reason for lower accuracy is that there is no overlap data to rely on. Hence accuracy for text mappings slightly decreases and many numeric mappings cannot be predicted. However, accuracy for categorical and schema mismatch mappings remains high.

4.6 Discussion

We now discuss limitations and extensions of the COMAP approach. First, we show how to extend COMAP to cover the removal of the single-join-path assumption made in Section 4.1. Second, COMAP has focused on creating complex mappings for attributes; we now study how to extend it to create mappings for an entire table. Finally, we discuss extending COMAP to more complex data representations such as XML DTDs and ontologies.

4.6.1 Removing the Single-Join-Path Assumption

In finding a complex mapping, we have searched only for the "right" attributes and combination formula, under the assumption that we know the "right" relationship between any set of attributes. Specifically, we assume that for any set of tables in schema T, *there exists only a single join path* that relates them, and that when we refer to attributes in these tables, the attributes relate to one another via that join path (see Section 4.1).

We can extend COMAP to cover the removal of this assumption as follows. First, for each set of tables in T, COMAP finds all possible join paths that can relate them. Note that the set of reasonable join paths for any set of tables is typically small. In practice, tables tend to relate via foreign key join paths. Such join paths can be discovered using a variety of techniques, including analyzing joins in queries that have been posed over the schemas and examining the data associated with the schemas [DJMS02]. The user can also suggest additional join paths for consideration.

Once COMAP has identified a small set of join paths per each group of tables, it modifies the search process to take the join paths into consideration. This modification is best explained via a simple example. Consider the *Text Searcher*. Suppose it is in the process of generating new candidate mappings. The current mapping is $s = concat(a,b)$, where a and b are attributes of table X in schema T. Now suppose the searcher is considering attribute c of table Y (also of T). Before, we assumed that tables X and Y relate via a single join path j_1. Hence, the searcher creates only a single new candidate mapping for the pair $concat(a,b)$ and c: namely, $concat(a,b,c)$ with a,b, and c relate via j_1.

Now suppose tables X and Y can also relate via an additional join path j_2. Then the *Text Searcher* would create two candidate mappings: $concat(a,b,c)$ with a,b,c relating via j_1, and $concat(a,b,c)$ with a,b,c relating via j_2. When materialized, each of the above two mappings will likely form a different column of values, due to using different join paths.

4.6.2 Creating a Complex Mapping for an Entire Table

Consider two relational schemas S and T. Suppose that S has a table C. So far we have focused only on creating complex mappings for each attribute c_i of C. Now we study how to extend COMAP to create a mapping for the entire table C. This mapping specifies how to obtain the tuples of C using the data of schema T.

Without loss of generalization, assume that table C has only two attributes c_1 and c_2, and that the mappings for them are $c_1 = concat(a,b)$, and $c_2 = d$, respectively, where attributes a and b come from table T_1 of schema T, and attribute d comes from table T_2 of T.

A mapping for table C will tell us how to generate a tuple of C. To generate a tuple, we must know how to *pair* a value of c_1 with a value of c_2. This means that we must know how to relate a,b, and d. Suppose we know that tables T_1 and T_2 relate only via join path j_1. Then the mapping for table C will be the tuples $(concat(a,b),d)$ where a,b, and d relate via join j_1.

If tables T_1 and T_2 relate via several join paths, we must choose the best join path to relate them. To do this, for each join path j_i, we must construct the table $(concat(a,b),d)$ using join path j_i, then compute the similarity between that table and table C. The best join path is then the one whose associated table is most similar to C.

We can use a variety of techniques to estimate the similarity between two tables. First, we can employ user feedback techniques as in [YMHF01] to select the best table among a set of possible tables. And second, we can employ classification techniques, just as we do for similarity between columns in Section 4.2. The key difference is that in the column case we build classifiers that deal with primitive data values (e.g., text fragments, numeric values, categorical values, etc.), whereas here we must build classifiers that can handle structured data in the form of tuples. We can adapt techniques on classifying structured data (e.g., the XML Learner in Chapter 3) for this purpose.

4.6.3 Complex Matching for More Expressive Representations

The techniques to find complex mappings for attributes and tables that we have discussed above can be generalized to more complex data representations.

For example, we can find complex mappings between two XML DTDs S and T as follows. First, for each leaf element of S, find its complex mapping (over the elements of T). Next, for each element that is composed of leaf elements, *assemble* its mapping from the mappings of its component leaf elements. For example, suppose that element contact-info is composed of the leaf elements name and address. Suppose further that we have already computed the mappings $\mathtt{name} = \mathit{concat}(\text{first-name}, \text{last-name})$ and $\mathtt{address} = \mathit{concat}(\mathtt{city}, \mathtt{state})$. Then to assemble the mapping for contact-info, we only need to know how first-name, last-name, city, and state relate. Constructing a mapping for element contact-info thus is similar to constructing a mapping for a table in relational schemas. We iterate the above process to eventually compose a mapping for the root element of schema S.

Creating complex mappings for ontologies can proceed in a similar manner. The key challenges facing complex matching for XML DTDs and ontologies are that (1) the number of relationships among the elements is substantially increased (compared to relational schemas), thus resulting in a significantly expanded search space, and (2) creating mappings for composite elements (as opposed to basic elements such as relational attributes or XML leaf elements) requires developing sophisticated methods for classifying structured data.

4.7 Summary

The vast majority of current works on representation matching focus on 1-1 mappings. In this chapter we have described a solution to the problem of finding complex mappings, which are widespread in practice. The key challenge with complex mappings is that the space of possible mappings greatly increases, as does the difficulty of evaluating the mappings. Our solution, as embodied in the COMAP system, is modular and extensible. It can easily accommodate new learning techniques as well as methods to exploit additional domain knowledge. The system is also extensible to new types of non 1-1 mappings, by adding appropriate search modules. Our experimental results demonstrate that COMAP already achieves accuracy between 62% and 100% on several matching problems with real-world data.

Chapter 5

ONTOLOGY MATCHING

In the previous two chapters we have studied the problems of finding 1-1 and complex semantic mappings between data representations. In this chapter we ask two questions. First, so far we have developed matching solutions only for relational and XML representations. Can we extend these solutions to the ontology context? As discussed in Chapter 1, ontologies are widely used as data representations for knowledge bases and marking up data on the emerging Semantic Web. Hence, techniques for matching ontologies should be an integral part of any practical and general solution to the representation matching problem.

Second, we have focused on developing a solution architecture that can efficiently incorporate multiple types of schema- and data information, as well as domain integrity constraints and user feedback. We have not considered the issue of the user's supplying a similarity measure between representation elements. Part of the reason for this is that in general the user cannot give a precise definition of the similarity measure that he or she wants (i.e., the similarity function $\mathcal{S}$ introduced in Section 2.3.1 of Chapter 2). However, there are still many cases where the user can articulate his or her notion of similarity measure, as we shall show in Section 5.1.2 of this chapter. Thus, the question is: can we extend our solutions to consider such cases?

We develop answers to the above two questions. In the next section we introduce the ontology-matching problem that we consider in the chapter. Sections 5.2–5.3 describe our solution, as embodied in the GLUE system. Section 5.4 presents an empirical evaluation. Section 5.5 discusses the limitations of the current solution and directions for future work, and Section 5.6 summarizes the chapter.

5.1 Introduction

In this section we define the ontology-matching problem, discuss its challenges, and outline our solution approach.

5.1.1 The Ontology-Matching Problem

We begin by introducing ontologies. An *ontology* specifies a conceptualization of a domain in terms of concepts, attributes, and relations [Fen01]. The *concepts* model entities of interest in the domain. They are typically organized into a *taxonomy tree* where each node represents a concept and each concept is a specialization of its parent. Figure 5.1 shows two sample taxonomies for the CS department domain (which are simplifications of real ones).

Each concept in a taxonomy is associated with a set of *instances*. For example, concept Associate-Professor has instances "Prof. Burn" and "Prof. Cook" as shown in Figure 5.1.a. By the taxonomy's definition, the instances of a concept are also instances of an ancestor concept. For example, instances of Assistant-Professor, Associate-Professor, and Professor in Figure 5.1.a are also instances

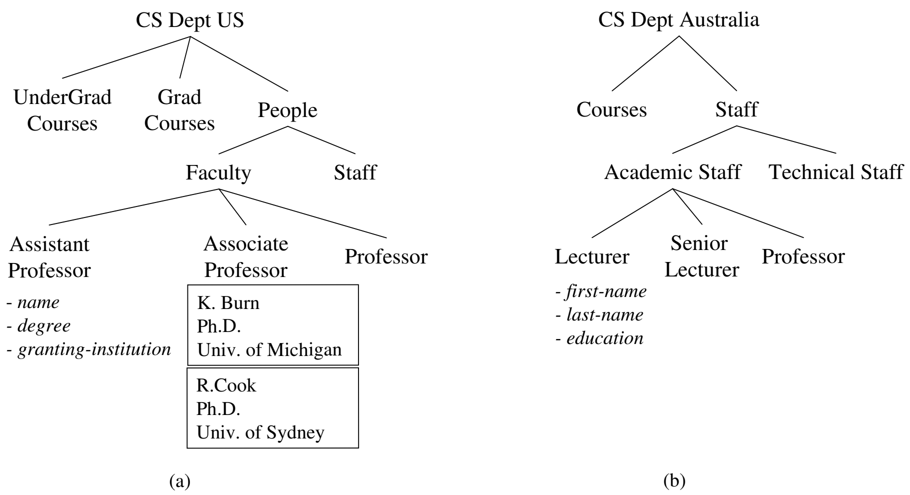


Figure 5.1: Computer Science Department Ontologies

of Faculty and People.

Each concept is also associated with a set of *attributes*. For example, the concept Associate-Professor in Figure 5.1.a has the attributes name, degree, and granting-institution. An *instance* that belongs to a concept has fixed attribute values. For example, the instance "Professor Cook" has values name = "R. Cook", degree = "Ph.D.", and so on. An ontology also defines a set of *relations* among its concepts. For example, a relation AdvisedBy(Student,Professor) (not shown in Figure 5.1) might list all instance pairs of Student and Professor such that the former is advised by the latter.

Many formal languages to specify ontologies have been proposed, such as OIL, DAML+OIL, SHOE, and RDF [BKD$^+$01, dam, HH01, BG00]. Though these languages differ in their terminologies and expressiveness, the ontologies that they model essentially share the same features we described above.

Given two ontologies, the *ontology-matching* problem is to find semantic mappings between them. The simplest type of mapping is *one-to-one (1-1)* mappings between the elements, such as "Associate-Professor maps to Senior-Lecturer", and "degree maps to education". Notice that mappings between different types of elements are possible, such as "the relation AdvisedBy(Student,Professor) maps to the attribute advisor of the concept Student". Examples of more complex types of mapping include "name maps to the concatenation of first-name and last-name", and "the union of Undergrad-Courses and Grad-Courses maps to Courses". In general, a mapping may be specified as a query that transforms instances in one ontology into instances in the other [CGL01].

In this chapter we focus on finding 1-1 mappings between the taxonomies. This is because the taxonomies are central components of ontologies, and successfully matching them would greatly aid in matching the rest of the ontologies. Extending matching to attributes and relations and considering more complex types of matching is the subject of ongoing research.

There are many ways to formulate a matching problem for taxonomies. The specific problem that we consider is as follows: *given two taxonomies and their associated data instances, for each node (i.e., concept) in one taxonomy, find the most similar node in the other taxonomy, for* *a pre-defined similarity measure, utilizing all available ontology information, data instances, and domain knowledge*. This is a very general problem setting that makes our approach applicable to a broad range of common ontology-related problems, such as ontology integration and data translation among ontologies.

5.1.2 Similarity Measures

We have decided to consider a similarity measure as a part of the input to the taxonomy matching problem. We now describe the similarity measures that we would like our approach to handle; but before doing that, we discuss the motivations leading to our choices.

First, we would like the similarity measures to be well-defined. A well-defined measure will facilitate the evaluation of our system. It also makes clear to the users what the system means by a match, and helps them figure out whether the system is applicable to a given matching scenario. Furthermore, a well-defined similarity notion may allow us to leverage special-purpose techniques for the matching process.

Second, we want the similarity measures to correspond to our intuitive notions of similarity. In particular, they should depend only on the semantic content of the concepts involved, and not on their syntactic specification.

Finally, it is clear that many reasonable similarity measures exist, each being appropriate to certain situations. Hence, to maximize our solution's applicability, we would like it to be able to handle a broad variety of similarity measures. The following examples illustrate the variety of possible definitions of similarity.

Example 7 A common task in ontology integration is to place a concept A into an appropriate place in a taxonomy T. One way to do this is to (a) use an "exact" similarity measure to find the concept B in T that is "most similar" to A, (b) use a "most-specific-parent" similarity measure to find the concept C in T that is the most specific superset concept of A, (c) use a "most-general-child" similarity measure to find the concept D in T that is the most general subset concept of A, then (d) decide on the placement of A, based on B, C, and D. □

Example 8 Certain applications may even use *different* similarity measures for different concepts. Suppose that a user instructs his online personal-assistant system to find houses in the range of $300-500K, located in Seattle. The user expects that the system will not return houses that do not satisfy the above criteria. Hence, the system should use exact mappings for price and address. But it may use approximate mappings for other concepts. If it maps house-description into neighborhood-info, that is still acceptable. □

Distribution-based Similarity Measures

We now give precise definitions of the similarity measures that we use, and show how our approach satisfies the motivating criteria. We begin by modeling each concept as a *set of instances*, taken from a *finite universe of instances*. In the CS domain, for example, the universe consists of all entities of interest in the world: professors, assistant professors, students, courses, and so on. The concept Professor is then the set of all instances in the universe that are professors. Given this model, the notion of the *joint probability distribution* between any two concepts A and B is well defined. This distribution consists of the four probabilities: $P(A,B), P(A,\overline{B}), P(\overline{A},B)$, and $P(\overline{A},\overline{B})$. A term such as $P(A,\overline{B})$ is the probability that a randomly chosen instance from the universe belongs to A but not to B, and is computed as the fraction of the universe that belongs to A but not to B.

A key observation that underlies our capability to handle similarity measures is that many practical similarity measures can be defined based solely on the joint distribution of the concepts involved. For instance, a possible definition for the "exact" similarity measure in Example 7 is

$$\begin{aligned} \textit{Jaccard-sim}(A,B) &= P(A \cap B)/P(A \cup B) \\ &= P(A,B)/[P(A,B) + P(A,\overline{B}) + P(\overline{A},B)]. \end{aligned} \tag{5.1}$$

This similarity measure is known as the *Jaccard* coefficient [vR79]. It takes the lowest value 0 when A and B are disjoint, and the highest value 1 when A and B are the same concept. Most of our experiments will use this similarity measure.

A definition for the "most-specific-parent" similarity measure in Example 7 is

$$MSP(A,B) = \begin{cases} P(A|B) & if\ P(B|A) = 1 \\ 0 & otherwise \end{cases} \tag{5.2}$$

where the probabilities $P(A|B)$ and $P(B|A)$ can be trivially expressed in terms of the four joint probabilities. This definition states that if B subsumes A, then the more specific B is, the higher $P(A|B)$, and thus the higher the similarity value $MSP(A,B)$ is. Thus it suits the intuition that the most specific parent of A in the taxonomy is the smallest set that subsumes A. An analogous definition can be formulated for the "most-general-child" similarity measure.

Instead of trying to estimate specific similarity values directly, GLUE focuses on computing the joint distributions. Then, it is possible to compute any of the above mentioned similarity measures as a function over the joint distributions. Hence, GLUE has the significant advantage of being able to work with a variety of similarity functions that have well-founded probabilistic interpretations.

5.1.3 Challenges and Outline of Solutions

As formulated, our taxonomy matching problem raises several significant challenges. The first challenge is how to compute the joint distribution of any two given concepts A and B. Under certain general assumptions (discussed in Section 5.2), a term such as $P(A,B)$ can be approximated as the fraction of instances that belong to both A and B (in the data associated with the taxonomies or, more generally, in the probability distribution that generated it). Hence, the problem reduces to deciding for each instance if it belongs to $A \cap B$. However, the input to our problem includes instances of A and instances of B in isolation. GLUE addresses this problem using machine learning techniques as follows: it uses the instances of A to learn a classifier for A, and then classifies instances of B according to that classifier, and vice-versa. Hence, we have a method for identifying instances of $A \cap B$.

Applying machine learning to our context raises the question of which learning algorithm to use and which types of information to use in the learning process. Many different types of information can contribute toward deciding the membership of an instance: its name, value format, the word frequencies in its value, and each of these is best utilized by a different learning algorithm. Hence, GLUE uses multi-strategy learning, as introduced in the LSD system (Chapter 3): it employs a set of learners, then combines their predictions using a meta-learner. In Chapters 3- 4 we have shown that multi-strategy learning is effective in the context of mapping between database schemas.

Finally, the taxonomy structure gives rise to domain constraints and general heuristics that have not been considered in the context of relational and XML data. Hence, GLUE attempts to exploit such constraints and heuristics in order to improve matching accuracy. An example heuristic is the observation that two nodes are likely to match if nodes in their neighborhood also match. An example of a domain constraint is "if node X matches Professor and node Y is an ancestor of X in the taxonomy, then it is unlikely that Y matches Assistant-Professor". Such constraints occur frequently in practice, and heuristics are very commonly used when manually mapping between ontologies. Previous works have exploited only one form or another of such knowledge and constraints, in restrictive settings [NM01, MZ98, MBR01, MMGR02]. Here, we develop a unifying approach to incorporate all such types of information. Our approach is based on *relaxation labeling*, a powerful technique used extensively in the vision and image processing community [HZ83], and successfully adapted to solve matching and classification problems in natural language processing [Pad98] and hypertext classification [CDI98]. We show that relaxation labeling can be adapted efficiently to our context, and that it can successfully handle a broad variety of heuristics and domain constraints.

In the rest of the chapter we describe the GLUE system and the experiments we conducted to validate it.

5.2 The GLUE Architecture

The basic architecture of GLUE is shown in Figure 5.2. It consists of three main modules: *Distribution Estimator*, *Similarity Estimator*, and *Relaxation Labeler*.

The *Distribution Estimator* takes as input two taxonomies O_1 and O_2, together with their data instances. Then it applies machine learning techniques to compute for every pair of concepts $\langle A \in O_1, B \in O_2 \rangle$ their joint probability distribution. Recall from Section 5.1.2 that this joint distribution consists of four numbers: $P(A,B), P(A,\overline{B}), P(\overline{A},B)$, and $P(\overline{A},\overline{B})$. Thus a total of $4|O_1||O_2|$ numbers will be computed, where $|O_i|$ is the number of nodes (i.e., concepts) in taxonomy O_i. The *Distribution Estimator* uses a set of base learners and a meta-learner. We describe the learners and the motivation behind them in Section 5.2.2.

Next, GLUE feeds the above numbers into the *Similarity Estimator*, which applies a user-supplied similarity function (such as the ones in Equations 5.1 or 5.2) to compute a similarity value for each pair of concepts $\langle A \in O_1, B \in O_2 \rangle$. The output from this module is a *similarity matrix* between the concepts in the two taxonomies.

The *Relaxation Labeler* module then takes the similarity matrix, together with the domain-specific constraints and the heuristic knowledge, and searches for the mapping configuration that best satisfies the domain constraints and the common knowledge, taking into account the observed similarities. This mapping configuration is the output of GLUE.

We now describe the *Distribution Estimator*. First, we discuss the general machine-learning technique used to estimate joint distributions from data, and then the use of multi-strategy learning in GLUE. Section 5.3 describes the *Relaxation Labeler*. The *Similarity Estimator* is trivial because it simply applies a user-defined function to compute the similarity of two concepts from their joint distribution, and hence is not discussed further.

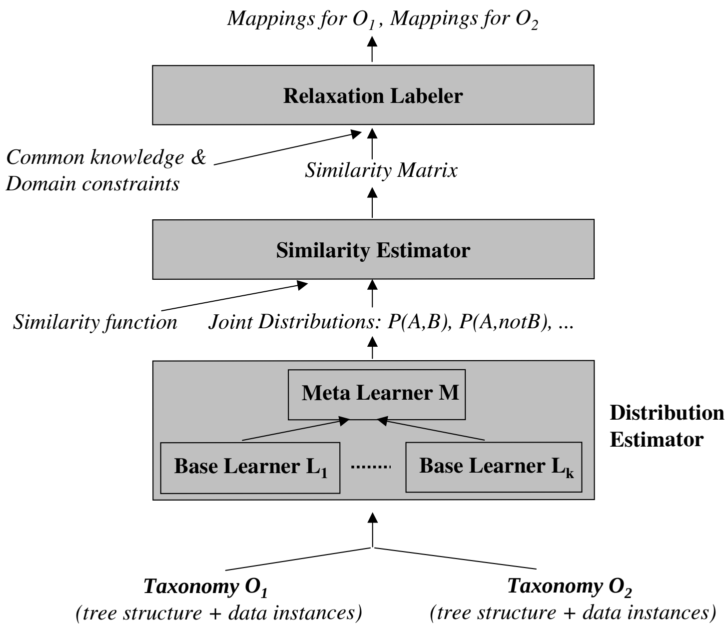


Figure 5.2: The GLUE Architecture.

5.2.1 The Distribution Estimator

Consider computing the value of $P(A,B)$. This joint probability can be computed as the fraction of the instance universe that belongs to both A and B. In general we cannot compute this fraction because we do not know every instance in the universe. Hence, we must estimate $P(A,B)$ based on the data we have, namely, the instances of the two input taxonomies. Note that the instances that we have for the taxonomies may be overlapping, but are not necessarily so.

To estimate $P(A,B)$, we make the general assumption that the set of instances of each input taxonomy is a *representative sample* of the instance universe covered by the taxonomy. This is a standard assumption in machine learning and statistics, and seems appropriate here, since there is no reason to suppose that the available instances were generated in some unusual way. We denote by U_i the set of instances given for taxonomy O_i, by $N(U_i)$ the size of U_i, and by $N(U_i^{A,B})$ the number of instances in U_i that belong to both A and B.

With the above assumption, $P(A,B)$ can be estimated by the following equation:[1]

$$P(A,B) = [N(U_1^{A,B}) + N(U_2^{A,B})] \ / \ [N(U_1) + N(U_2)], \tag{5.3}$$

Computing $P(A,B)$ then reduces to computing $N(U_1^{A,B})$ and $N(U_2^{A,B})$. Consider $N(U_2^{A,B})$. We can compute this quantity if we know for each instance s in U_2 whether it belongs to both A and B.

[1]Notice that $N(U_i^{A,B})/N(U_i)$ is also a reasonable approximation of $P(A,B)$, but it is estimated based only on the data of O_i. The estimation in (5.3) is likely to be more accurate because it is based on more data, namely, the data of both O_1 and O_2.

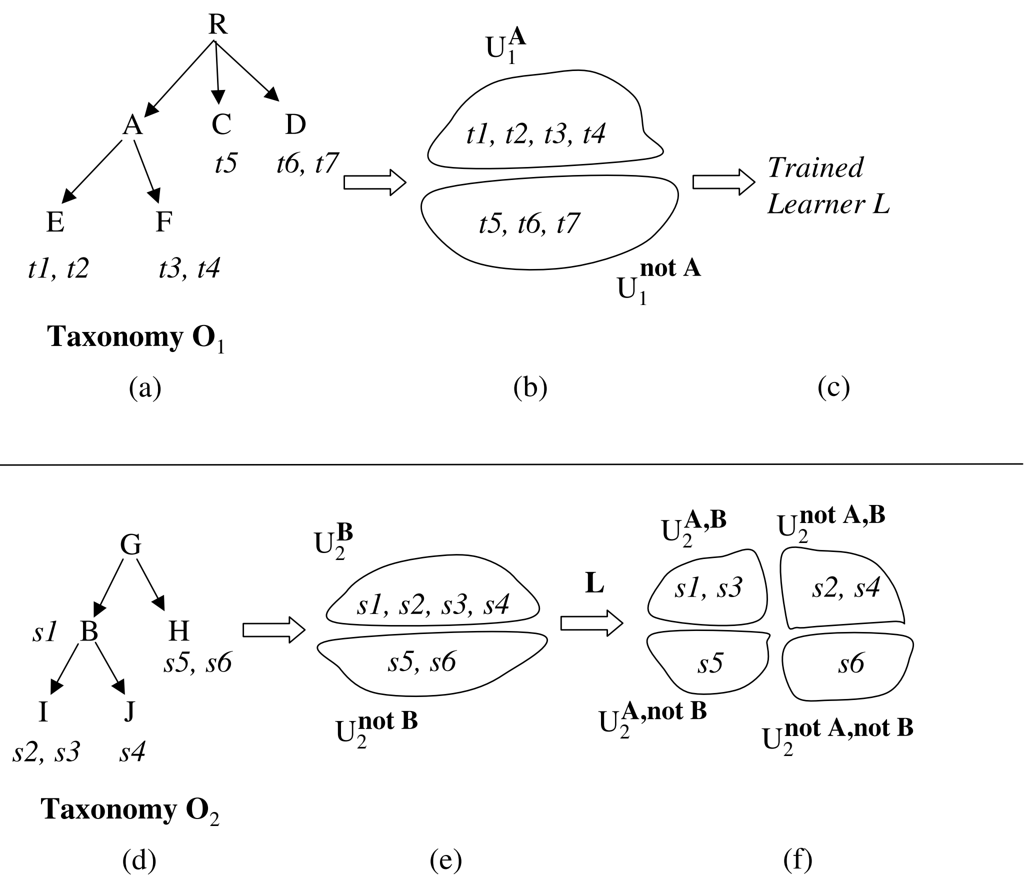


Figure 5.3: Estimating the joint distribution of concepts *A* and *B*.

One part is easy: we already know whether *s* belongs to *B* – if it is explicitly specified as an instance of *B* or of any descendant node of *B*. Hence, we only need to decide whether *s* belongs to *A*.

This is where we use machine learning techniques. Specifically, we partition U_1, the set of instances of ontology O_1, into the set of instances that belong to *A* and the set of instances that do not belong to *A*. Then, we use these two sets as positive and negative examples, respectively, to train a classifier for *A*. Finally, we use the classifier to predict whether instance *s* belongs to *A*.

In summary, we estimate the joint probability distribution of *A* and *B* as follows (the procedure is illustrated in Figure 5.3):

1. Partition U_1, into U_1^A and $U_1^{\overline{A}}$, the set of instances that do and do not belong to *A*, respectively (Figures 5.3.a-b).

2. Train a learner *L* for instances of *A*, using U_1^A and $U_1^{\overline{A}}$ as the sets of positive and negative training examples, respectively.

3. Partition U_2, the set of instances of taxonomy O_2, into U_2^B and $U_2^{\overline{B}}$, the set of instances that do

and do not belong to B, respectively (Figures 5.3.d-e).

4. Apply learner L to each instance in U_2^B (Figure 5.3.e). This partitions U_2^B into the two sets $U_2^{A,B}$ and $U_2^{\overline{A},B}$ shown in Figure 5.3.f. Similarly, applying L to $U_2^{\overline{B}}$ results in the two sets $U_2^{A,\overline{B}}$ and $U_2^{\overline{A},\overline{B}}$ (Figure 5.3.f).

5. Repeat Steps 1-4, but with the roles of taxonomies O_1 and O_2 reversed, to obtain the sets $U_1^{A,B}$, $U_1^{\overline{A},B}$, $U_1^{A,\overline{B}}$, and $U_1^{\overline{A},\overline{B}}$.

6. Finally, compute $P(A,B)$ using Formula 5.3. The remaining three joint probabilities are computed in a similar manner, using the sets $U_2^{\overline{A},B},\ldots,U_1^{\overline{A},\overline{B}}$ computed in Steps 4-5.

By applying the above procedure to all pairs of concepts $\langle A \in O_1, B \in O_2 \rangle$ we obtain all joint distributions of interest.

5.2.2 Multi-Strategy Learning

Given the diversity of machine learning methods, the next issue is deciding which one to use for the procedure we described above. A key observation in our approach is that there are *many* different types of information that a learner can glean from the training instances, in order to make predictions. It can exploit the *frequencies* of words in the text value of the instances, the instance *names*, the value *formats*, the *characteristics of value distributions*, and so on.

Since each learner is best at utilizing only certain types of information, GLUE follows the LSD system (Chapter 3) and takes a *multi-strategy learning* approach. In Step 2 of the above estimation procedure, instead of training a single learner L, we train a set of learners $L_1,\ldots,L_k$, called *base learners*. Each base learner exploits well a certain type of information from the training instances to build prediction hypotheses. Then, to classify an instance in Step 4, we apply the base learners to the instance and combine their predictions using a *meta-learner*. This way, we can achieve higher classification accuracy than with any single base learner alone, and obtain better approximations of the joint distributions.

The current implementation of GLUE has two base learners, *Text Learner* and *Name Learner*, and a meta-learner that is a linear combination of the base learners. We now describe these learners in detail.

The Text Learner: This learner exploits the frequencies of words in the *textual content* of an instance to make predictions. Recall that an instance typically has a *name* and a set of *attributes* together with their values. In the current version of GLUE, we do not handle attributes directly; rather, we treat them and their values as the *textual content* of the instance[2]. For example, the textual content of the instance "Professor Cook" is "R. Cook, Ph.D., University of Sydney, Australia". The textual content of the instance "CSE 342" is the text content of this course's homepage.

For each input instance d, the *Text Learner* employs the Naive Bayes learning technique [DH74, DP97, MN98] to analyze the textual content of d, to compute $P(A|d)$ and $P(\overline{A}|d)$. The learner

[2]However, more sophisticated learners can be developed that deal explicitly with the attributes, such as the *XML Learner* in Section 3.6 of Chapter 3.

employs Naive Bayes learning in a manner similar to that of the *Naive Bayes Learner* described in the LSD system (see Section 3.4.3 of Chapter 3 for more details). Then the *Text Learner* predicts that d belongs to A with probability $P(A|d)$, and belongs to $\overline{A}$ with the probability $P(\overline{A}|d)$.

The *Text Learner* works well on long textual elements, such as course descriptions, or elements with very distinct and descriptive values, such as color (red, blue, green, etc.). It is less effective with short, numeric elements such as course numbers or credits.

The Name Learner: This learner is similar to the *Text Learner*, but makes predictions using the *full name* of the input instance, instead of its *content*. The full name of an instance is the concatenation of names leading from the root of the taxonomy to that instance, expanded with synonyms. For example, the full name of the instance with the name s_4 in taxonomy O_2 (Figure 5.3.d) is "G B J s_4". This learner works best on specific and descriptive names. It does not well with names that are too vague or vacuous.

The Meta-Learner: The predictions of the base learners are combined using the meta-learner. The meta-learner assigns to each base learner a *learner weight* that indicates how much it *trusts* that learner's predictions. Then it combines the base learners' predictions via a weighted sum.

For example, suppose the weights of the *Text Learner* and the *Name Learner* are 0.6 and 0.4, respectively. Suppose further that for instance s_4 of taxonomy O_2 (Figure 5.3.d) the *Text Learner* predicts A with probability 0.8 and $\overline{A}$ with probability 0.2, and the *Name Learner* predicts A with probability 0.3 and $\overline{A}$ with probability 0.7. Then the meta-learner predicts A with probability $0.8 \cdot 0.6 + 0.3 \cdot 0.4 = 0.6$ and $\overline{A}$ with probability $0.2 \cdot 0.6 + 0.7 \cdot 0.4 = 0.4$.

In the current GLUE system, the learner weights are set manually, based on the characteristics of the base learners and the taxonomies. However, they can also be set automatically using stacking [Wol92, TW99], as we have shown with the LSD system in Chapter 3.

5.3 Relaxation Labeling

We now describe the *Relaxation Labeler*, which takes the similarity matrix from the *Similarity Estimator* and searches for the mapping configuration that best satisfies the given domain constraints and heuristic knowledge. We first describe relaxation labeling, then discuss the domain constraints and heuristic knowledge employed in our approach. Finally, we discuss an efficient implementation of relaxation labeling, as adapted to our matching context.

5.3.1 Relaxation Labeling

Relaxation labeling is an efficient technique to solve the problem of labeling the nodes of a graph, given a set of constraints. The key idea behind this approach is that the label of a node is typically influenced by the *features of the node's neighborhood* in the graph. Examples of such features are the labels of the neighboring nodes, the percentage of nodes in the neighborhood that satisfy a certain criteria, and the fact that a certain constraint is satisfied or not.

Relaxation labeling exploits this observation. The influence of a node's neighborhood on its label is quantified using a formula for the probability of each label as a function of the neighborhood features. Relaxation labeling assigns initial labels to nodes based solely on the intrinsic properties of the nodes. Then it performs *iterative local optimization*. In each iteration it uses the formula to change the probability of the label of a node based on the features of its neighborhood. This continues until the probabilities do not change from one iteration to the next, or some other convergence criterion is reached.

Relaxation labeling appears promising for our purposes because it has been applied successfully to similar matching problems in computer vision, natural language processing, and hypertext classification [HZ83, Pad98, CDI98]. It is relatively efficient, and can handle a broad range of constraints. Even though its convergence properties are not yet well understood (except in certain cases) and it is liable to converge to a local maximum, in practice it has been found to perform quite well [Pad98, CDI98].

We now explain how to apply relaxation labeling to the problem of mapping from taxonomy O_1 to taxonomy O_2. We regard nodes in O_2 as *labels*, and recast the problem as finding the best label assignment to nodes in O_1, given all knowledge we have about the domain and the two taxonomies.

Our goal is to derive a formula for updating the probability that a node takes a label based on the features of the neighborhood. Let X be a node in taxonomy O_1, and L be a label (i.e., a node in O_2). Let Δ_K represent all that we know about the domain, namely, the tree structures of the two taxonomies, the sets of instances, and the set of domain constraints. Then we have the following conditional probability

$$\begin{aligned} P(X=L|\Delta_K) &= \sum_{M_X} P(X=L, M_X|\Delta_K) \\ &= \sum_{M_X} P(X=L|M_X,\Delta_K)P(M_X|\Delta_K), \end{aligned} \tag{5.4}$$

where the sum is over all possible label assignments M_X to all nodes other than X in taxonomy O_1. Assuming that the nodes' label assignments are independent of each other given Δ_K, we have

$$P(M_X|\Delta_K) = \prod_{(X_i=L_i)\in M_X} P(X_i = L_i|\Delta_K). \tag{5.5}$$

Consider $P(X=L|M_X,\Delta_K)$. M_X and Δ_K constitutes all that we know about the neighborhood of X. Suppose now that the probability of X getting label L depends only on the values of n features of this neighborhood, where each feature is a function $f_i(M_X,\Delta_K,X,L)$. As we explain in the next section, each such feature corresponds to one of the heuristics or domain constraints that we wish to exploit. Then

$$P(X=L|M_X,\Delta_K) = P(X=L|f_1,\ldots,f_n). \tag{5.6}$$

If we have access to previously-computed mappings between taxonomies in the same domain, we can use them as the training data from which to estimate $P(X=L|f_1,\ldots,f_n)$ (see [CDI98] for an example of this in the context of hypertext classification). However, here we will assume that such mappings are not available. Hence we use alternative methods to quantify the influence of the features on the label assignment. In particular, we use the sigmoid or logistic function $\sigma(x) = 1/(1+e^{-x})$, where x is a linear combination of the features f_k, to estimate the above probability. This function is widely used to combine multiple sources of evidence [Agr90]. The general shape of the sigmoid is as shown in Figure 5.4. Thus:

$$P(X=L|f_1,\ldots,f_n) \propto \sigma(\alpha_1 \cdot f_1 + \cdots + \alpha_n \cdot f_n), \tag{5.7}$$

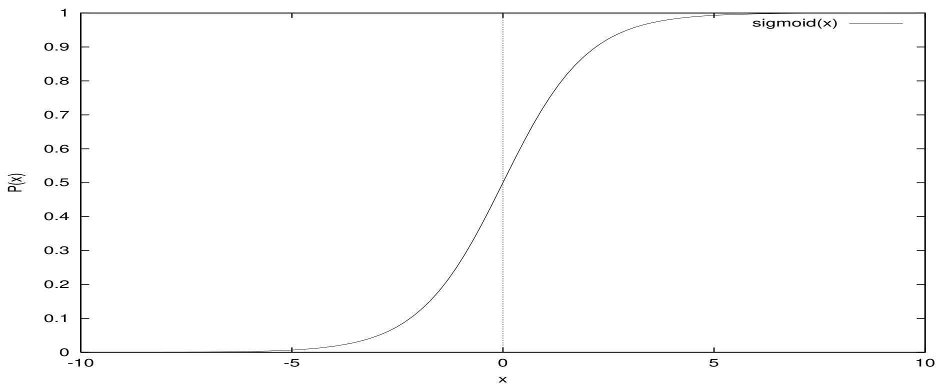


Figure 5.4: The sigmoid function

Table 5.1: Sample constraints that can be exploited to improve matching accuracy of GLUE.

Constraint Types		**Examples**
Domain-Independent	*Neighborhood*	Two nodes match if their children also match. Two nodes match if their parents match and at least x% of their children also match. Two nodes match if their parents match and some of their descendants also match.
	Union	If all children of node X match node Y, then X also matches Y.
Domain-Dependent	*Subsumption*	If node Y is a descendant of node X, and Y matches PROFESSOR, then it is unlikely that X matches ASSISTANT-PROFESSOR. If node Y is NOT a descendant of node X, and Y matches PROFESSOR, then it is unlikely that X matches FACULTY.
	Frequency	There can be at most one node that matches DEPARTMENT-CHAIR.
	Nearby	If a node in the neighborhood of node X matches ASSOCIATE-PROFESSOR, then the chance that X matches PROFESSOR is I increased.

where $\propto$ denotes "proportional to", and the weight α_k indicates the importance of feature f_k.

The sigmoid is essentially a smoothed threshold function, which makes it a good candidate for use in combining evidence from the different features. If the total evidence is below a certain value, it is unlikely that the nodes match; above this threshold, they probably do.

By substituting Equations 5.5-5.7 into Equation 5.4, we obtain

$$P(X=L|\Delta_K) \propto \sum_{M_X} \sigma\left(\sum_{k=1}^{n} \alpha_k f_k(M_X, \Delta_K, X, L)\right) \prod_{(X_i=L_i)\in M_X} P(X_i = L_i|\Delta_K). \tag{5.8}$$

The proportionality constant is found by renormalizing the probabilities of all the labels to sum to one. Notice that this equation expresses the probabilities $P(X=L|\Delta_K)$ for the various nodes in terms of each other. This is the iterative equation that we use for relaxation labeling.

5.3.2 Constraints

Table 5.1 shows examples of the constraints currently used in our approach and their characteristics. We distinguish two types of constraints: domain-independent and -dependent constraints. *Domain-independent constraints* (also called *heuristic knowledge*) convey our general knowledge about the interaction between related nodes. Perhaps the most widely used such constraint is the *Neighborhood Constraint*: "two nodes match if nodes in their neighborhood also match", where the neighborhood is defined to be the children, the parents, or both [NM01, MBR01, MZ98] (see Table 5.1). Another example is the *Union Constraint*: "if all children of a node A match node B, then A also matches B". This constraint is specific to the taxonomy context. It exploits the fact that A is the union of all its children. *Domain-dependent constraints* convey our knowledge about the interaction between specific nodes in the taxonomies. Table 5.1 shows examples of three types of domain-dependent constraints.

To incorporate the constraints into the relaxation labeling process, we model each constraint c_i as a feature f_i of the neighborhood of node X. For example, consider the constraint c_1: "two nodes are likely to match if their children match". To model this constraint, we introduce the feature $f_1(M_X, \Delta_K, X, L)$ that is the percentage of X's children that match a child of L, under the given M_X mapping. Thus f_1 is a numeric feature that takes values from 0 to 1. Next, we assign to f_i a *positive* weight α_i. This has the intuitive effect that, all other things being equal, the higher the value f_i (i.e., the percentage of matching children), the higher the probability of X matching L is.

As another example, consider the constraint c_2: "if node Y is a descendant of node X, and Y matches PROFESSOR, then it is unlikely that X matches ASSISTANT-PROFESSOR". The corresponding feature, $f_2(M_X, \Delta_K, X, L)$, is 1 if the condition "there exists a descendant of X that matches PROFESSOR" is satisfied, given the M_X mapping configuration, and 0 otherwise. Clearly, when this feature takes value 1, we want to substantially reduce the probability that X matches ASSISTANT-PROFESSOR. We model this effect by assigning to f_2 a *negative* weight α_2.

5.3.3 Efficient Implementation of Relaxation Labeling

In this section we discuss why previous implementations of relaxation labeling are not efficient enough for ontology matching, then describe an efficient implementation for our context.

Recall from Section 5.3.1 that our goal is to compute for each node X and label L the probability $P(X = L|\delta_K)$, using Equation 5.8. A naive implementation of this computation process would enumerate *all* labeling configurations M_X, then compute $f_k(M_X, \delta_K, X, L)$ for each of the configurations.

This naive implementation does not work in our context because of the vast number of configurations. This is a problem that has also arisen in the context of relaxation labeling being applied to hypertext classification ([CDI98]). The solution in [CDI98] is to consider only the top k configurations, that is, those with highest probability, based on the heuristic that the sum of the probabilities of the top k configurations is already sufficiently close to 1. This heuristic was true in the context of hypertext classification, due to a relatively small number of neighbors per node (in the range 0-30) and a relatively small number of labels (under 100).

Unfortunately the above heuristic is not true in our matching context. Here, a neighborhood of a node can be the entire graph, thereby comprising hundreds of nodes, and the number of labels can be hundreds or thousands (because this number is the same as the number of nodes in the other ontology to be matched). Thus, the number of configurations in our context is orders of magnitude more than that in the context of hypertext classification, and the probability of a configuration is computed by multiplying the probabilities of a very large number of nodes. As a consequence, even the highest probability of a configuration is very small, and a huge number of configurations have to be considered to achieve a significant total probability mass.

Hence we developed a novel and efficient implementation for relaxation labeling in our context. Our implementation relies on three key ideas. The first idea is that we divide the space of configurations into *partitions* $C_1, C_2, \ldots, C_m$, such that all configurations that belong to the same partition have the same values for the features $f_1, f_2, \ldots, f_n$. Then, to compute $P(X = L|\delta_K)$, we iterate over the (far fewer) partitions rather than over the huge space of configurations.

The one problem remaining is to compute the probability of a partition C_i. Suppose all configurations in C_i have feature values $f_1 = v_1, f_2 = v_2, \ldots, f_n = v_n$. Our second key idea is to approximate the probability of C_i with $\prod_{j=1}^{n} P(f_j = v_j)$, where $P(f_j = v_j)$ is the total probability of all configurations whose feature f_j takes on value v_j. Note that this approximation makes an independence assumption over the features, which is clearly not valid. However, the assumption greatly simplifies the computation process. In our experiments with GLUE, we have not observed any problem arising because of this assumption.

Now we focus on computing $P(f_j = v_j)$. We compute this probability using a variety of techniques that depend on the particular feature. For example, suppose f_j is the number of children of X that map to some child of L. Let X_j be the j^{th} child of X (ordered arbitrarily) and n_X be the number of children of the concept X. Let S_j^m be the probability that of the first j children, there are m that are mapped to some child of L. It is easy to see that S_j^m's are related as follows,

$$S_j^m = P(X_j = L')S_{j-1}^{m-1} + (1 - P(X_j = L'))S_{j-1}^m$$

where $P(X_j = L') = \sum_{l=1}^{n_L} P(X_j = L_l)$ is the probability that the child X_j is mapped to some child of L. This equation immediately suggests a dynamic programming approach to computing the values S_j^m and thus the number of children of X that map to some child of L. We use similar techniques to compute $P(f_j = v_j)$ for the other types of features that are described in Table 5.1.

5.4 Empirical Evaluation

We have evaluated GLUE on several real-world domains. Our goals were to evaluate the matching accuracy of GLUE, to measure the relative contribution of the different components of the system, and to verify that GLUE can work well with a variety of similarity measures.

Domains and Taxonomies: We evaluated GLUE on three domains, whose characteristics are shown in Table 5.2. The domains Course Catalog I and II describe courses at Cornell University and the University of Washington. The taxonomies of Course Catalog I have 34 - 39 nodes, and are fairly similar to each other. The taxonomies of Course Catalog II are much larger (166 - 176 nodes) and much less similar to each other. Courses are organized into schools and colleges, then into departments and centers within each college. The Company Profile domain uses ontologies from *Yahoo.com* and *TheStandard.com* and describes the current business status of the companies. Companies are organized into sectors, then into industries within each sector[3].

[3]Many ontologies are available from research resources (e.g., DAML.org, semanticweb.org, OntoBroker [ont], SHOE, OntoAgents). However, these ontologies currently have no or very few data instances.

Table 5.2: Domains and taxonomies for experiments with GLUE.

Taxonomies		# nodes	# non-leaf nodes	depth	# instances in taxonomy	max # instances at a leaf	max # children of a node	# manual mappings created
Course Catalog I	*Cornell*	34	6	4	1526	155	10	34
	Washington	39	8	4	1912	214	11	37
Course Catalog II	*Cornell*	176	27	4	4360	161	27	54
	Washington	166	25	4	6957	214	49	50
Company Profiles	*Standard.com*	333	30	3	13634	222	29	236
	Yahoo.com	115	13	3	9504	656	25	104

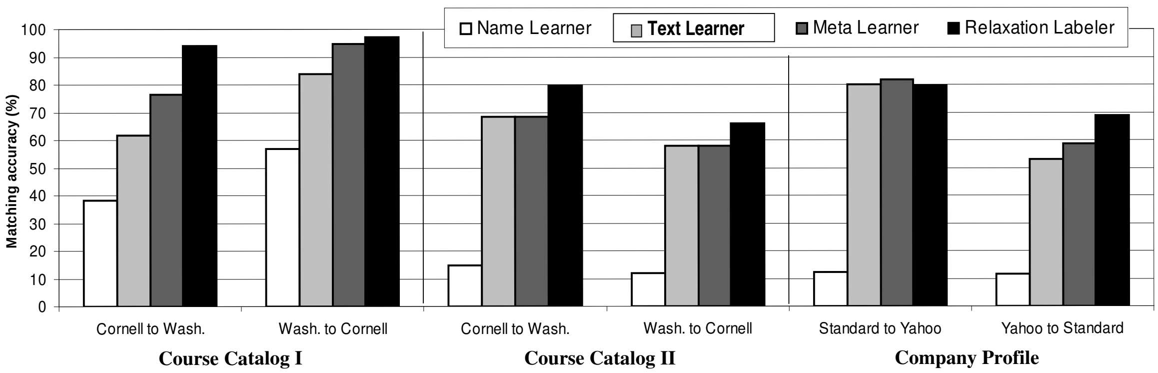


Figure 5.5: Matching accuracy of GLUE.

In each domain we downloaded two taxonomies. For each taxonomy, we downloaded the entire set of data instances, and performed some trivial data cleaning such as removing HTML tags and the phrase "course not offered" from the instances. We also removed instances of size less than 130 bytes, because they tend to be empty or vacuous, and thus do not contribute to the matching process. We then removed all nodes with fewer than 5 instances, because such nodes cannot be matched reliably due to lack of data.

Similarity Measure & Manual Mappings: We chose to evaluate GLUE using the *Jaccard* similarity measure (Section 5.1.2), because it corresponds well to our intuitive understanding of similarity. Given the similarity measure, we manually created the correct 1-1 mappings between the taxonomies in the same domain, for evaluation purposes. The rightmost column of Table 5.2 shows the number of manual mappings created for each taxonomy. For example, we created 236 one-to-one mappings from *Standard* to *Yahoo*, and 104 mappings in the reverse direction. Note that in some cases there were nodes in a taxonomy for which we could not find a 1-1 match. This was either because there was no equivalent node (e.g., School of Hotel Administration at Cornell has no equivalent counterpart at the University of Washington), or when it impossible to determine an accurate match without additional domain expertise.

Domain Constraints: We specified domain constraints for the relaxation labeler. For the taxonomies in Course Catalog I, we specified all applicable subsumption constraints (see Table 5.1). For the other two domains, because their sheer size makes specifying all constraints difficult, we specified only the most obvious subsumption constraints (about 10 constraints for each taxonomy). For the taxonomies in Company Profiles we also used several frequency constraints.

Experiments: For each domain, we performed two experiments. In each experiment, we applied GLUE to find the mappings from one taxonomy to the other. The *matching accuracy* of a taxonomy is then the percentage of the manual mappings (for that taxonomy) that GLUE predicted correctly.

5.4.1 Matching Accuracy

Figure 5.5 shows the matching accuracy for different domains and configurations of GLUE. In each domain, we show the matching accuracy of two scenarios: mapping from the first taxonomy to the second, and vice versa. The four bars in each scenario (from left to right) represent the accuracy produced by: (1) the *Name Learner* alone, (2) the *Text Learner* alone, (3) the meta-learner using the previous two learners, and (4) the relaxation labeler on top of the meta-learner (i.e., the complete GLUE system).

The results show that GLUE achieves high accuracy across all three domains, ranging from 66 to 97%. In contrast, the best matching results of the base learners, achieved by the *Text Learner*, are only 52 - 83%. It is interesting that the *Name Learner* achieves very low accuracy, 12 - 15% in four out of six scenarios. This is because all instances of a concept, say B, have very similar full names (see the description of the *Name Learner* in Section 5.2.2). Hence, when the *Name Learner* for a concept A is applied to B, it will classify *all* instances of B as A or all as $\overline{A}$, which is clearly often incorrect and leads to poor estimates of the joint distributions. The poor performance of the *Name Learner* underscores the importance of data instances in ontology matching.

The results clearly show the utility of the meta-learner and relaxation labeler. Even though in half of the cases the meta-learner only minimally improves the accuracy, in the other half it makes substantial gains, between 6 and 15%. And in all but one case, the relaxation labeler further improves accuracy by 3 - 18%, confirming that it is able to exploit the domain constraints and general heuristics. In one case (from Standard to Yahoo), the relaxation labeler decreased accuracy by 2%. The performance of the relaxation labeler is discussed in more detail below. In Section 5.5 we identify the reasons that prevent GLUE from identifying the remaining mappings.

In the current experiments, GLUE utilized on average only 30 to 90 data instances per leaf node (see Table 5.2). The high accuracy in these experiments suggests that GLUE can work well with only a modest amount of data.

5.4.2 Performance of the Relaxation Labeler

In our experiments, when the relaxation labeler was applied, the accuracy typically improved substantially in the first few iterations, then gradually dropped. This phenomenon has also been observed in many previous works on relaxation labeling [HZ83, Llo83, Pad98] (but no clear explanation has been found). Because of this, finding the right stopping criterion for relaxation labeling is of crucial importance. Many stopping criteria have been proposed, but no general effective criterion has been found.

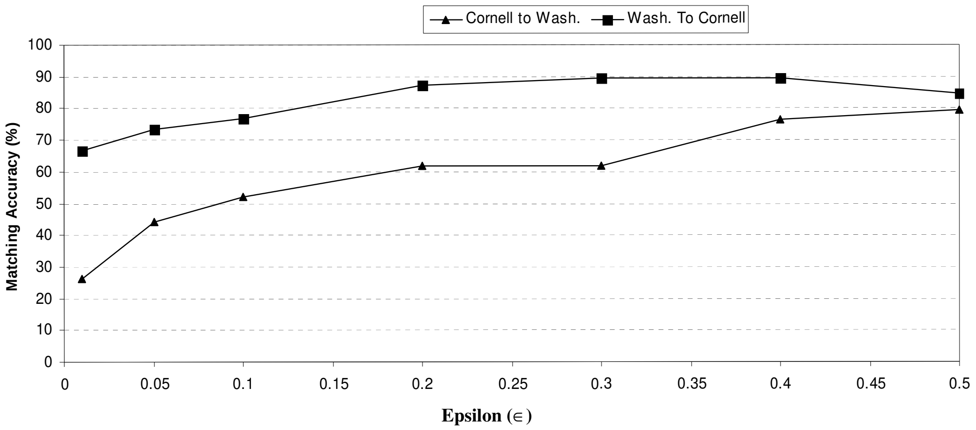


Figure 5.6: The accuracy of GLUE in the Course Catalog I domain, using the most-specific-parent similarity measure.

We considered three stopping criteria: (1) stopping when the mappings in two consecutive iterations do not change (the *mapping criterion*), (2) when the probabilities do not change, or (3) when a fixed number of iterations has been reached. (In the first criterion, by "the mapping of a node" we mean the most probable label of that node.)

We observed that when using the last two criteria the accuracy sometimes improved by as much as 10%, but most of the time it decreased. In contrast, when using the mapping criterion, in all but one of our experiments the accuracy substantially improved, by 3 - 18%, and hence, our results are reported using this criterion. We note that with the mapping criterion, we observed that relaxation labeling always stopped in the first few iterations.

In all of our experiments, relaxation labeling was also very fast. It took only a few seconds in Catalog I and under 20 seconds in the other two domains. This observation shows that relaxation labeling can be implemented efficiently in the ontology-matching context. It also suggests that we can efficiently incorporate user feedback into the relaxation labeling process in the form of additional domain constraints.

We also experimented with different values for the constraint weights (see Section 5.3), and found that the relaxation labeler was quite robust with respect to such parameter changes.

5.4.3 Most-Specific-Parent Similarity Measure

So far we have experimented only with the *Jaccard* similarity measure. We wanted to know whether GLUE can work well with other similarity measures. Hence we conducted an experiment in which we used GLUE to find mappings for taxonomies in the Course Catalog I domain, using the following similarity measure:

$$MSP(A,B) = \begin{cases} P(A|B) & if\ P(B|A) \geq 1-\varepsilon \\ 0 & otherwise \end{cases}$$

This measure is the same as the *most-specific-parent* similarity measure described in Section 5.1.2, except that we added an ε factor to account for the error in approximating $P(B|A)$.

Figure 5.6 shows the matching accuracy, plotted against ε. As can be seen, GLUE performed quite well on a broad range of ε. This illustrates how GLUE can be effective with more than one similarity measure.

5.5 Discussion

The accuracy of GLUE is quite impressive as is, but it is natural to ask what limits GLUE from obtaining even higher accuracy. There are several reasons that prevent GLUE from correctly matching the remaining nodes.

First, some nodes cannot be matched because of insufficient training data. For example, many course descriptions in Course Catalog II contain only vacuous phrases such as "3 credits". While there is clearly no general solution to this problem, in many cases it can be mitigated by adding base learners that can exploit domain characteristics to improve matching accuracy.

Second, the relaxation labeler performed local optimizations, and sometimes converged to only a local maximum, thereby not finding correct mappings for all nodes. Here, the challenge will be in developing search techniques that work better by taking a more "global perspective", but still retain the runtime efficiency of local optimization.

We note that some nodes cannot be matched automatically because they are simply ambiguous. For example, it is not clear whether "networking and communication devices" should match "communication equipment" or "computer networks". A solution to this problem is to incorporate user interaction into the matching process [NM00, DDH01, YMHF01].

Finally, GLUE currently tries to predict the best match for *every* node in the taxonomy. However, in some cases, such a match simply does not exist (e.g., unlike Cornell, the University of Washington does not have a School of Hotel Administration). Hence, an additional extension to GLUE is to make it be aware of such cases, and not predict an incorrect match when this occurs.

GLUE also makes heavy use of the fact that we have data instances associated with the ontologies we are matching. We note that many real-world ontologies already have associated data instances. Furthermore, the largest benefits of ontology matching come from matching the most heavily used ontologies; and the more heavily an ontology is used for marking up data (e.g, on the Semantic Web), the more data it has. Finally, we showed in our experiments that only a moderate number of data instances is necessary in order to obtain good matching accuracy.

5.6 Summary

With the proliferation of applications that employ ontologies to encode data, automated techniques for ontology matching must be an integral part of any generic solution to representation matching. We have described an approach that extends the LSD and COMAP solutions in Chapters 3–4 to match ontologies. Our approach exploits well-founded notions of semantic similarity, expressed in terms of the joint probability distribution of the concepts involved. We introduced relaxation labeling to the ontology-matching context, and showed that it can efficiently exploit a variety of heuristic knowledge and domain constraints to further improve matching accuracy. Our experiments showed that we can accurately match 66 - 97% of the nodes on several real-world domains.

Chapter 6

RELATED WORK

In this chapter we review works that relate to our representation-matching solution and discuss in detail how our solution advances the state of the art.

- First, we review formal semantics that have been developed for representation matching, as well as proposed notions of similarity.
- Second, we survey the vast body of matching solutions that have been developed in both the database and AI communities. We compare these solutions to ours from several perspectives, ans show how our solution provides a unifying framework for most current solutions.
- Third, our work has made contributions to several learning issues, such as multi-strategy learning, learning with structured data, and relaxation labeling. Hence, we also review works related to such learning scenarios.
- Finally, we discuss works in other knowledge-intensive domains (e.g., information extraction and solving crossword puzzles) which bear interesting resemblances to representation matching.

6.1 Formal Semantics and Notions of Similarity

Several works have addressed the issue of formal semantics for representation matching. In [BC86] the authors introduced the notion of *integration assertions* which relate the elements in two schemas (and therefore are essentially semantic mappings). Given two schemas S and T, an integration assertion has the form $e = f$, where e and f are expressions defined over the elements of S and T, respectively. The meaning of such an integration assertion is that there exist interpretations I_S and I_T (for S and T, respectively) that map e and f into the same concept in the universe.

In [MHDB02] the authors introduce more expressive forms of semantic mappings. In their framework a mapping is of the form $e \mathsf{op} f$, where e and f are defined as above, and the operator op is well defined with respect to the output types of e and f. For example, if both expressions have relations as output types, then op can be $=$ and $\subseteq$. If e outputs a constant and f outputs a unary relation, then op can be $\in$ [1].

In the above work the authors also show that some times one needs a helper representation to relate two expressions in S and T. For example if e and f refer to the students in Seattle and San Francisco, respectively, then they are disjoint sets and hence cannot be related directly to each other.

[1] In [MHDB02] the authors use the term *formula* to refer to a semantic mapping (as in our framework), and use *a mapping* to refer to the set of semantic mappings between the two given representations (and optionally a helper representation).

In this case, we need to relate both of them to the concept of students in a helper model. In the same work the authors also identify and study several important properties of mappings such as query answerability, mapping inference, and mapping composition.

Our formal semantics framework (described in Chapter 2) builds on previous works [MHDB02, BC86], but extends them in several important ways.

- First, we always use a helper representation (as introduced by [MHDB02]). This representation is the user domain representation $\mathcal{U}$ defined in Chapter 2. This simplifies the conceptual framework.

- Second, we introduce the notion of similarity distance between the elements (and expressions) in $\mathcal{U}$. We assume the user can define an *arbitrary* measure of similarity over concepts in the domain representation $\mathcal{U}$. This is in marked contrast to previous works, which either do not consider any similarity notion, or only very restricted forms of it (see the discussion on notions of similarity below). In our work, we contend that a similarity notion is a fundamental and integral part of the user's conceptualization of the domain, and hence must be given explicitly. The introduction of similarity notion provides a formal explanation for the working of representation matching algorithms: they attempt to approximate true similarity values using the syntactic clues (as discussed in Section 2.3.1 of Chapter 2).

- Finally, previous works define an expression, such as e, to be built from the elements of a representation, such as S, and a set of operators. The operators are *well defined over representation S*. This could be problematic if S and T use different representation languages. For example, suppose S is a relational representation and T is an XML one. Now consider a mapping that equates a nested XML element f in T with an expression e in S. Obviously, e must use some XML operators to construct an output type that is the same as the output type of f. However, it would be difficult to give well-defined semantics to such XML operators over the relational representation S. To avoid this problem, we describe all operators involved (in both S and T) as having semantics over the user domain representation $\mathcal{U}$ (see Section 2.3.2 of Chapter 2 for more details).

Notions of Similarity: Several works have considered the notion of similarity between concepts. The similarity measure in [RHS01] is based on the κ (Kappa) statistics, and can be thought of as being defined over the joint probability distribution of the concepts involved. In [Lin98] the authors propose an information-theoretic notion of similarity that is also based on the joint distribution. However, these works argue for a single best universal similarity measure, whereas we argue for the opposite. Furthermore, our solutions (e.g., GLUE) actually allow handling multiple application-dependent similarity measures. There have been many works on notions of similarity in machine learning, case-based reasoning, and cognitive psychology. For a survey of semantic similarity discussed in many such works, see Section 8.5 of [MS99].

6.2 Representation-Matching Algorithms

Matching solutions have been developed primarily in the database and AI communities. In this section we review and compare these solutions to ours from several perspectives.

6.2.1 Rule- versus Learner-based Approaches

Rule-based Solutions: The vast majority of current solutions employ hand-crafted rules to match representations. Works in this approach include [MZ98, PSU98, CA99, MWJ, MBR01, MMGR02] in databases and [Cha00, MFRW00, NM00, MWJ] in AI.

In general, hand-crafted rules exploit schema information such as element names, data types, structures, and number of subelements. A broad variety of rules have been considered. For example, the TranScm system [MZ98] employs rules such as "two elements match if they have the same name (allowing synonyms) and the same number of subelements". The DIKE system [PSU98, PSTU99, PTU00] computes the similarity between two representation elements based on the similarity of the characteristics of the elements and the similarity of related elements. The ARTEMIS and the related MOMIS [CA99, BCVB01] system compute the similarity of representation elements as a weighted sum of the similarities of name, data type, and substructure. The CUPID system [MBR01] employs rules that categorize elements based on names, data types, and domains. Rules therefore tend to be domain-independent, but can be tailored to fit a certain domain, and domain-specific rules can also be crafted.

Learner-based Solutions: Recently, several works have employed machine learning techniques to perform matching. Works in this direction include [LC00, CHR97, BM01, BM02, NHT$^+$02] in databases and [PE95, NM01, RHS01, LG01] in AI.

Current learner-based solutions have considered a variety of learning techniques. However, any specific solution typically employs only a single learning technique (e.g., neural networks or Naive Bayes). Learning techniques considered exploit both schema and data information. For example, the SemInt system [LC94, LCL00, LC00] uses a neural-network learning approach. It matches schema elements based on field specifications (e.g, data types, scale, the existence of constraints) and statistics of data content (e.g., maximum, minimum, average, and variance).

The DELTA system [CHR97] associates with each schema element a text string that consists of the element name and all other meta-data on the element, then matches elements based on the similarity of the text strings. DELTA uses information-retrieval similarity measures, like the *Name Learner* in LSD. The ILA system [PE95] matches the schemas of two sources by analyzing the description of objects that are found in both sources. The Autoplex and Automatch systems [BM01, BM02] use a Naive Bayes learning approach that exploits data instances to match elements. The HICAL system [RHS01] exploits the data instances in the overlap between the two taxonomies to infer mappings. The system described in [LG01] computes the similarity between two taxonomic nodes based on their signature TF/IDF vectors, which are computed from the data instances.

Rahm and Bernstein [RB01] provide the most recent survey on matching solutions, and describe some of the above works in detail. The survey in [BLN86] examines earlier works on matching which used mostly rule-based techniques. Both surveys consider works that have been developed in the database community.

Comparison of the Two Approaches: Each of the above two approaches – rule-based and learner-based – has its advantages and disadvantages. Rule-based techniques are relatively inexpensive. They do not require training as in learner-based techniques. Furthermore, they typically operate only on schemas (not on data instances), and hence are fairly fast. They can work very well in certain types of applications. For example, in ontology versioning a frequent task is to match two consecutive versions of an ontology [NM02]. The consecutive versions tend to differ little from each other, and hence are very amenable to rule-based techniques, as [NM02] shows. Finally, rules can provide a quick and concise method to capture valuable user knowledge about the domain. For example, the user can write regular expressions that encode times or phone numbers, or quickly compile a collection of county names or zip codes that help recognize those types of entities. As another example, in the course-listing domain, the user can write the following rule: "use regular expressions to recognize elements about times, then match the first time element with start-time and the second element with end-time". Notice that learning techniques would have difficulties being applied to these scenarios. They either cannot learn the above rules, or can do so only with abundant training data or with the right representations for training examples.

On the other hand, rule-based techniques also have major disadvantages. First, they cannot exploit data information effectively, even though the data can encode a wealth of information (e.g., value format, distribution, frequently occurring words, and so on) that would greatly aid the matching process. Second, they cannot exploit previous matching efforts, such as the initial mappings that the user manually created in the case of the LSD system (Chapter 3). Thus, in a sense, systems that rely solely on rule-based techniques have difficulties learning from the past, to improve over time. Finally, rule-based techniques have serious problems with schema elements for which no effective hand-crafted rules can be found. For example, it is not clear how one can hand craft rules that distinguish between movie description and user comments on the movies, both being long textual paragraphs.

In a sense, learner-based techniques are complementary to rule-based ones. They can exploit data information and past matching activities. They excel at matching elements for which hand-crafted rules are difficult to obtain. However, they can be more time-consuming than rule-based techniques, requiring an additional training phase, and taking more time processing data and schema information. They also have difficulties learning certain types of knowledge (e.g., times, zipcodes, county names, as mentioned above). Furthermore, current learner-based approaches employ only a single learner, and thus have limited accuracy and applicability. For example, the neural-network technique employed by SemInt does not handle textual elements very well, and the objects-in-the-overlap technique of ILA makes it unsuitable to the common case where sources do not share any object.

The Combination of Both Approaches in Our Solution: The complementary nature of rule- and learner-based techniques suggest that an effective matching solution should employ both – each whenever it is deemed effective. Our work in this dissertation offers a technique to do so. The multistrategy framework – introduced in LSD and subsequently extended in COMAP and GLUE – employs multiple base learners to make matching predictions, then combines their predictions using a meta-learner. While the majority of base learners that we have described employ learning techniques, it is clear that, in general, base learners can also employ hand-crafted rules. Our solution employs a meta-learning technique (stacking in Chapter 3) to automatically find out the effectiveness of each base learner in different situations. The multistategy framework therefore represents a significant step toward an effective and unifying matching solution.

6.2.2 Exploiting Multiple Types of Information

Many works in representation matching exploit multiple types of information, such as names, data types, integrity constraints, attribute cardinality, and so on. However, they employ a single strategy for this purpose. For example, the SemInt system [LC94, LCL00, LC00] employs neural networks, the Autoplex system [BM01] employs Naive Bayes classification techniques, and the DELTA system [CHR97] lumps all information about an element into a single long piece of text, then matches the pieces using information retrieval techniques.

Some works have considered several different matching strategies, based on the heuristic that the combination of multiple strategies may improve matching accuracy. The hybrid system described in [CHR97], for example, combines the predictions of the SemInt and DELTA system. However, these works combine strategies in a hardwired fashion, thus making it extremely difficult to add new strategies. Several recent works [CA99, BCVB01, DR02] solve the above problem by using schemes such as weighted sum to combine predictions coming from different matching strategies. The weights employed in such solutions must be hand-tuned, based on the specific application context.

This dissertation advances the state of the art on exploiting multiple types of information in several important aspects. First, we *bring this issue to the forefront* of representation matching, with our work on LSD. We clearly show that there are many different types of information available, and that a matching solution must exploit all of them to maximize matching accuracy.

Second, we *consider a much broader range* of information types than the previous works. Specifically, we advocate building a solution that can exploit both schema and data information, domain integrity constraints, heuristic knowledge, previous matching activities, user feedback, and other types of user knowledge about the matching application (e.g., similarity measure).

Third, we make the case that *there is no one-size-fit-all technique*: each type of information should be exploited using an appropriate strategy, be it Naive Bayes, neural network, decision tree, hand-crafted rule, or recognizer. This point has not been articulated in previous works on representation matching.

Fourth, we *introduce multistrategy learning* as a technique that can automatically select the weights that are used to combine multiple strategies. Thus, we provide a solution to the problem of manually tuning the weights (which is both tedious and inaccurate). However, multistrategy learning is not limited to just the use of weights. It also raises the possibility of employing more sophisticated techniques to combine strategies, such as decision trees or Bayesian networks.

Finally, we show for the first time that *the same multistrategy approach can also be carried over to complex matching* (Chapter 4).

6.2.3 Incorporating Domain Constraints and Heuristics

It was recognized early on that domain integrity constraints and heuristics provide valuable information for matching purposes. Hence, almost all the works we have mentioned exploit some forms of this type of knowledge.

In most works, integrity constraints have been used to match representation elements *locally*. For example, many works match two elements if they participate in similar constraints (among other things). The main problem with this scheme is that it cannot exploit "global" constraints and heuristics that relate the matching of *multiple* elements (e.g., "at most one element matches house-address”). To address this problem, in this dissertation we have advocated moving the handling of constraints to *after* the matchers. This way, the constraint handling framework can exploit “global” constraints and is highly extensible to new types of constraints.

While integrity constraints are *domain-specific* information (e.g., house-id is a key for house listings), heuristic knowledge makes *general* statements about how the matching of elements relate to each other. A well-known example of a heuristic is “two nodes match if their neighbors also match”, variations of which have been exploited in many systems (e.g., [MZ98, MBR01, MMGR02, NM01]). The common scheme is to *iteratively* change the mapping of a node based on those of its neighbors. The iteration is carried out one or twice, or all the way until some convergence criterion is reached.

Our GLUE work provides a solution to exploit a broad range of heuristic information, including those heuristics that have been commonly used in the matching literature. The solution builds on a well-founded probabilistic interpretation, and treats domain integrity constraints as well as heuristic knowledge in a uniform fashion.

6.2.4 Handling User Feedback

Most existing works have focused on developing automatic matching algorithms. They either ignore the issue of user interaction, or treat it as an afterthought. The typical assumption is that whenever a system cannot decide (e.g., between multiple matching alternatives), then it asks the user [MZ98].

The exceptions are several recent works in ontology matching [Cha00, MFRW00, NM00]. These works have powerful features that treat user feedback as an integral part of the matching process and allow for efficient user interaction. For example, the system in [NM00] frequently solicits user feedback on its matching decisions (e.g., confirm or reject the decisions), then makes subsequent decisions based on the feedback.

The Clio system [MHH00, YMHF01, PVH$^+$02] focuses on very fine-grained mappings, which are for example SQL or XQuery expressions that can be immediately executed to translate data from one representation to another. Clio makes two important contributions. First, it recognizes that creating such fine-grained mappings entails making decisions that require user input. Deciding if inner join or outer join should be used is an example of such decisions. Hence, like the previous works in ontology matching that we just described, it also brings the user to the center of the matching process. Second, it realizes that efficient interaction with the user is crucial to the success of matching. Hence, it develops techniques to minimize the amount of interaction required.

The key innovation we made regarding user feedback is that we treat such feedback as temporary domain constraints and heuristics. Thus, we allow users to specify as little or as much feedback as necessary. Our framework also allows users to iteratively interact with the matching system in an efficient manner (e.g., by rerunning the relaxation labeler as many times as necessary).

An important issue that Clio has touched on, and that we have not considered, is finding out how to minimize user interaction – asking them only what is absolutely necessary – and yet make the most out of such interaction. We shall return to this topic when we discuss future directions in the next chapter (Chapter 7).

6.2.5 1-1 and Complex Matching

The vast majority of current works focus only on finding 1-1 semantic mappings. Several works (e.g., [MZ98]) deal with complex matching in the sense that such matchings are hard-coded into rules. The rules are systematically tried on the elements of given representations, and when such a rule fires, the system returns the complex mapping encoded in the rule.

As mentioned earlier, the Clio system [MHH00, YMHF01, PVH$^+$02] creates complex mappings for relational and XML data. To create a complex mapping for a representation element, Clio assumes that the "right" attributes and formula have been given (either by the user, by data mining techniques, or by systems such as LSD). It then focuses on finding the "right" relationship between the attributes (see Chapter 4 for more detail on "right" attributes, formula, and relationships).

In a sense, our work (with the COMAP system) is complementary to Clio in that we find the "right" attributes and formula, assuming the "right" relationship is given. We show in Chapter 4 that our current framework can be extended to address the question of finding the "right" relationship. We believe that a complete and practical system to deal with complex mappings can be developed by combining the multi-searcher architecture and the learning/statistical techniques of COMAP with the powerful facilities for user interaction and for developing fine-grained mappings of Clio.

6.2.6 Generic vs. Application-Specific Solutions

A recent interesting trend covers both ends of the representation matching spectrum. At one end, there have been several works that focus on developing very specialized, application-specific matching solutions. The rationale for this is that representation matching is so difficult, that we should specialize our solution to exploit application-specific features. An example of such works is [NM02], which focuses on matching multiple versions of the same ontology. As mentioned, since consecutive versions tend to differ little from each other, solutions that utilize simple rules can be developed that achieve very high matching accuracy.

At the other end, several works have advocated building generic matching solutions (e.g., [RB01, DR02] and this dissertation), mostly because representation matching is a fundamental step in numerous data management applications. In the foreseeable future, it is likely that there will be a need for, and we shall continue to see, works in both directions.

6.2.7 Further Related Work

The works [Ber03, PB02] discuss model management and schema matching in that context. The work [RD00] discusses data cleaning and schema matching. Several recent works [RRSM01, RMR00, RS01, SRLS02] discuss the issue of building large-scale data integration systems in detail and the crucial role of schema matching in this process. The work [SR01] discusses the impact of XML on data sharing, in particular schema matching and object matching. The work [EJX01] discusses a schema matching approach that is similar to LSD, but using a different set of base learners and a simple averaging method to combine the base learners' predictions.

6.3 Related Work in Learning

We now briefly survey works that are related to learning issues in this dissertation.

Combining Multiple Learners: Multi-strategy learning has been researched extensively [MT94], and applied to several other domains (e.g., information extraction [Fre98], solving crossword puzzles [KSL+99], and identifying phrase structure in NLP [PR00]). In our context, our main innovations are the three-level architecture (base learners, meta-learner and prediction combiner) that allows learning from both schema and data information, and the use of integrity constraints to further refine the learner.

Learning with Structured Data: Yi and Sundaresan [YS00] describe a classifier for XML documents. However, their method applies only to documents that share the same DTD, which is not the case in our domain.

Relaxation Labeling for Learning to Label Interrelated Instances: This technique has been employed successfully to similar matching problems in computer vision, natural language processing, and hypertext classification [HZ83, Pad98, CDI98]. Our work on relaxation labeling is most similar to the work on hypertext classification of [CDI98]. The key difference is that we consider more expressive types of constraints and a broader notion of neighborhood. As a consequence, the optimization techniques of [CDI98] do not work efficiently for our context. To solve this problem, we develop new optimization techniques that are shown empirically to be accurate and extremely fast (see Section 5.3.3). These techniques are general and hence should also be useful for relaxation labeling in other contexts.

Exploiting Domain Constraints: Incorporating domain constraints into the learners has been considered in several works (e.g., [DR96]), but most works consider only certain types of learners and constraints. In contrast, our framework allows arbitrary constraints (as long as they can be verified using the schema and data), and works with any type of learner. This is made possible by using the constraints during the matching phase, to restrict the learner predictions, instead of the usual approach of using constraints during the training phase, to restrict the search space of learned hypotheses.

6.4 Related Work in Knowledge-Intensive Domains

Representation matching requires making *multiple interrelated inferences*, by combining a *broad variety* of relatively *shallow* knowledge types. In recent years, several other domains that fit the above description have also been studied. Notable domains are information extraction (e.g., [Fre98]), solving crossword puzzles [KSL+99], and identifying phrase structure in NLP [PR00]. What is remarkable about these studies is that they tend to develop similar solution architectures which combine the prediction of multiple independent modules and optionally handle domain constraints on top of the modules. These solution architectures have been shown empirically to work well. It will be interesting to see if such studies converge in a definitive blueprint architecture for making multiple inferences in knowledge-intensive domains.

Chapter 7

CONCLUSION

Representation matching is a critical step in numerous data management applications. Manual matching is very expensive. Hence, it is important to develop techniques to automate the matching process. Given the rapid proliferation and the growing size of applications today, automatic techniques for representation matching become ever more important.

This dissertation has contributed to both understanding the matching problem and developing matching tools. In this chapter, we recap the key contributions of the dissertation and discuss directions for future research.

7.1 Key Contributions

This dissertation makes two major contributions. The first contribution is a framework that formally defines a variety of representation-matching problems and explains the workings of subsequently developed matching algorithms.

The framework introduces a small set of notions: (1) a domain representation that serves as the user's conceptualization of the domain, (2) a mapping function that relates concepts in the representations to be matched to those in the domain representation, (3) a similarity function that the user employs to relate the similarity of concepts in the domain representation, (4) an assumption that relates the innate semantic similarity of concepts with their syntactic similarity, and (5) operators that are defined over concepts in the domain representation and that can be used to combine concepts to form complex mapping expressions.

We show that most types of input and output of representation matching problems (including output notions such as semantic mapping) can be explained in terms of the above five notions. An important consequence of this result is that it suggests a methodology to obtain input information about a matching problem by systematically checking what is known about each of the five notions. The more input information we have about a matching problem, the higher matching accuracy we can obtain.

The second major contribution of the dissertation is a solution to semi-automatically create semantic mappings. The key innovations that we made in developing this solution are:

- We brought the necessity of exploiting multiple types of information to the forefront of representation matching. Then we proposed a *multistrategy learning* solution, which applies multiple modules – each exploiting well a single type of information to make matching predictions – and then combines the modules' predictions. Employing *multiple independent matching modules* is a key idea underlying our solution, for both 1-1 and complex matching cases. This idea yields a solution that is highly modular and easily customized to any particular domain.

- We developed the A* and relaxation-labeling frameworks that exploit a broad range of integrity constraints and domain heuristics. These frameworks are made possible by our decision to layer constraint exploitation on top of the matching modules. (An alternative would have been to incorporate constraint handling directly into the modules.) Again, this two-layer architecture is modular and easily adapted to new domains, as we demonstrated by adapting our solution to data integration (Chapter 3), data translation (Chapter 4), and ontology matching (Chapter 5).

- We showed that explicit notions of similarity play an important part in practical matching scenarios. We then demonstrated that our solution can handle a broad variety of such notions (Chapter 5). This result is significant because virtually all previous works have not considered the notion of similarity explicitly.

- Finally, we showed that our solution can also naturally handle complex matchings, the types of matching that are common in practice but have not been addressed by most previous works. The first main idea here was to find a set of candidate complex mappings, then reduce the problem to an 1-1 matching problem. The second idea was to employ multiple search modules to examine the space of complex mappings, to find mapping candidates. The final main idea was to use machine learning and statistical techniques to evaluate mapping candidates.

7.2 Future Directions

We have made significant inroads into understanding and developing solutions for representation matching, but substantial work remains toward the goal of achieving a comprehensive matching solution. In what follows we discuss several directions for future work.

7.2.1 Efficient User Interaction

Matching solutions must interact with the user in order to arrive at final correct mappings. (Even if a solution is perfect, the user still has to verify the mappings.) We consider efficient user interaction *the* most important open problem for representation matching. Any practical matching tool must handle this problem, and anecdotal evidence abounds on deployed matching tools quickly being abandoned for irritating users with too many questions. Our experience with matching large schemas (e.g., while experimenting with the GLUE system) confirms that even just verifying a large number of created mappings is already extremely tedious.

The building and operating of future data sharing systems will further exacerbate this problem. Presumably many such systems will operate over hundreds or thousands of data sources. Even if a near perfect matching solution is employed, the system builder still has to verify the tens of thousands or millions of mappings that the solution created. Just the verification of mappings at such scales is already bordering on practical impossibility. Hence, efficient user interaction is crucial. The key is to discover how to minimize user interaction – asking only for absolutely necessary feedback, but maximizing the impact of the feedback.

7.2.2 Performance Evaluation

We have reported matching performance in terms of the predictive matching accuracy. Predictive accuracy is an important performance measure because (a) the higher the accuracy, the more reduction in human labor a matching system can achieve, and (b) the measure facilitates comparison and development of matching techniques. The next important task is to actually *quantify* the reduction in human labor that a matching system achieves. This problem is related to the problem of efficient user interaction that we mentioned above. It is known to be difficult, due to widely varying assumptions on how a matching tool is used, and has just recently been investigated [MMGR02, DMR02].

7.2.3 Unified Matching Framework

A third challenge is to develop a unified framework for representation matching that combines in a principled, seamless, and efficient way all the relevant information (e.g., user feedback, mappings from a different application) and techniques (e.g., machine learning, heuristics). The work on the GLUE system (Chapter 5) suggests that mappings can be given well-founded definitions based on probabilistic interpretations, and that a unified mapping framework can be developed by leveraging probabilistic representation and reasoning methods such as Bayesian networks.

7.2.4 Mapping Maintenance

In dynamic and autonomous environments (e.g., the Internet) sources often undergo changes in their schemas and data. Hence, the operators of a data sharing system must constantly monitor the component sources to detect and deal with changes in their semantic mappings. Clearly, manual monitoring is very expensive and not scalable. It is important therefore to develop techniques to automate the monitoring and repairing of semantic mappings. Despite the importance of this problem, it has not been addressed in the literature (though the related problem of wrapper maintenance has received some attention [Kus00b]).

7.2.5 Matching Other Types of Entities

Besides representation elements, the problems of matching other types of entities such as objects and Web services are also becoming increasingly crucial. The problem of deciding if two different objects in two sources (e.g., two house listings or two car descriptions) refer to the same real-world entity has received much attention in the database and data mining communities. This problem typically arises when multiple databases are merged and duplicate records must be purged (hence, it is also commonly known as the *merge/purge* problem). In the data integration context, the problem arises when we merge answers from multiple sources and must purge duplicate answers. As data integration becomes pervasive, this problem will become increasingly important.

The problem of deciding if two Web services share similar behaviors (in essence, matching the behaviors of services) will also become crucial as Web services proliferate and the need to mediate among them increases. It will be an interesting direction to examine how the techniques that have been developed for representation matching can be transferred to solving these new types of matching problems.

BIBLIOGRAPHY

[Agr90] A. Agresti. *Categorical Data Analysis*. Wiley, New York, NY, 1990.

[AK97] N. Ashish and C. Knoblock. Wrapper generation for semi-structured internet sources. *SIGMOD Record*, 26(4):8–15, 1997.

[BC86] J. Biskup and B. Convent. A formal view integration method. In *Proceedings of the ACM Conf. on Management of Data (SIGMOD)*, 1986.

[BCVB01] S. Bergamaschi, S. Castano, M. Vincini, and D. Beneventano. Semantic integration of heterogeneous information sources. *Data and Knowledge Engineering*, 36(3):215–249, 2001.

[Ber03] P. Bernstein. Applying model management to classical meta data problems. In *Proceedings of the Conf. on Innovative Database Research (CIDR)*, 2003.

[BG00] D. Brickley and R. Guha. Resource description framework schema specification 1.0, 2000.

[BHP00] P. Bernstein, A. Halevy, and R. Pottinger. A vision for management of complex models. *ACM SIGMOD Record*, 29(4):55–63, 200.

[BKD+01] J. Broekstra, M. Klein, S. Decker, D. Fensel, F. van Harmelen, and I. Horrocks. Enabling knowledge representation on the Web by extending RDF schema. In *Proceedings of the Tenth Int. World Wide Web Conference*, 2001.

[BLHL01] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, 279, 2001.

[BLN86] C. Batini, M. Lenzerini, and SB. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Computing Survey*, 18(4):323–364, 1986.

[BM01] J. Berlin and A. Motro. Autoplex: Automated discovery of content for virtual databases. In *Proceedings of the Conf. on Cooperative Information Systems (CoopIS)*, 2001.

[BM02] J. Berlin and A. Motro. Database schema matching using machine learning with feature selection. In *Proceedings of the Conf. on Advanced Information Systems Engineering (CAiSE)*, 2002.

[CA99] S. Castano and V. De Antonellis. A schema analysis and reconciliation tool environment. In *Proceedings of the Int. Database Engineering and Applications Symposium (IDEAS)*, 1999.

[CDI98] S. Chakrabarti, B. Dom, and P. Indyk. Enhanced hypertext categorization using hyperlinks. In *Proceedings of the ACM SIGMOD Conference*, 1998.

[CGL01] D. Calvanese, D. G. Giuseppe, and M. Lenzerini. Ontology of integration and integration of ontologies. In *Proceedings of the 2001 Description Logic Workshop (DL 2001)*, 2001.

[CH98] W. Cohen and H. Hirsh. Joins that generalize: Text classification using WHIRL. In *Proc. of the Fourth Int. Conf. on Knowledge Discovery and Data Mining (KDD)*, 1998.

[Cha00] H. Chalupsky. Ontomorph: A translation system for symbolic knowledge. In *Principles of Knowledge Representation and Reasoning*, 2000.

[CHR97] C. Clifton, E. Housman, and A. Rosenthal. Experience with a combined approach to attribute-matching across heterogeneous databases. In *Proc. of the IFIP Working Conference on Data Semantics (DS-7)*, 1997.

[CRF00] Donald D. Chamberlin, Jonathan Robie, and Daniela Florescu. Quilt: An XML query language for heterogeneous data sources. In *WebDB (Informal Proceedings) 2000*, pages 53–62, 2000.

[CT91] T. M. Cover and J. A. Thomas. *Elements of Information Theory*. Wiley, New York, NY, 1991.

[dam] www.daml.org.

[DDH01] A. Doan, P. Domingos, and A. Halevy. Reconciling schemas of disparate data sources: A machine learning approach. In *Proceedings of the ACM SIGMOD Conference*, 2001.

[DDH03] A. Doan, P. Domingos, and A. Halevy. Learning to match the database schemas: A multistrategy approach. *Machine Learning*, 2003. Special Issue on Multistrategy Learning. To Appear.

[DFF+99] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language for XML. In *Proceedings of the International Word Wide Web Conference, Toronto, CA*, 1999.

[DH74] R. O. Duda and P. E. Hart. *Pattern Classification and Scene Analysis*. John Wiley and Sons, New York, 1974.

[DJMS02] T. Dasu, T. Johnson, S. Muthukrishnan, and V. Shkapenyuk. Mining database structure; or, how to build a data quality browser. In *Proceedings of the ACM Conf. on Management of Data (SIGMOD)*, 2002.

[DMDH02] A. Doan, J. Madhavan, P. Domingos, and A. Halevy. Learning to map ontologies on the Semantic Web. In *Proceedings of the World-Wide Web Conference (WWW-02)*, 2002.

[DMR02] H. Do, S. Melnik, and E. Rahm. Comparison of schema matching evaluations. In *Proceedings of the 2nd Int. Workshop on Web Databases (German Informatics Society)*, 2002.

[DP97] P. Domingos and M. Pazzani. On the optimality of the simple bayesian classifier under zero-one loss. *Machine Learning*, 29:103–130, 1997.

[DR96] S. Donoho and L. Rendell. Constructive induction using fragmentary knowledge. In *Proc. of the 13th Int. Conf. on Machine Learning*, pages 113–121, 1996.

[DR02] H. Do and E. Rahm. Coma: A system for flexible combination of schema matching approaches. In *Proceedings of the 28th Conf. on Very Large Databases (VLDB)*, 2002.

[EJX01] D. Embley, D. Jackman, and L. Xu. Multifaceted exploitation of metadata for attribute match discovery in information integration. In *Proceedings of the WIIW Workshop*, 2001.

[EP90] AK. Elmagarmid and C. Pu. Guest editors' introduction to the special issue on heterogeneous databases. *ACM Computing Survey*, 22(3):175–178, 1990.

[Fen01] D. Fensel. *Ontologies: Silver Bullet for Knowledge Management and Electronic Commerce*. Springer-Verlag, 2001.

[Fre98] Dayne Freitag. Machine learning for information extraction in informal domains. *Ph.D. Thesis*, 1998. Dept. of Computer Science, Carnegie Mellon University.

[FW97] M. Friedman and D. Weld. Efficiently executing information-gathering plans. In *Proc. of the Int. Joint Conf. of AI (IJCAI)*, 1997.

[GMPQ+97] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, and J. Widom. The TSIMMIS project: Integration of heterogeneous information sources. *Journal of Intelligent Inf. Systems*, 8(2), 1997.

[HGMN+98] J. Hammer, H. Garcia-Molina, S. Nestorov, R. Yerneni, M. Breunig, and V. Vassalos. Template-based wrappers in the TSIMMIS system (system demonstration). In *ACM Sigmod Record*, Tucson, Arizona, 1998.

[HH01] J. Heflin and J. Hendler. A portrait of the Semantic Web in action. *IEEE Intelligent Systems*, 16(2), 2001.

[HNR72] P. Hart, N. Nilsson, and B. Raphael. Correction to "a formal basis for the heuristic determination of minimum cost paths". *SIGART Newsletter*, 37:28–29, 1972.

[HZ83] R.A. Hummel and S.W. Zucker. On the foundations of relaxation labeling processes. *PAMI*, 5(3):267–287, May 1983.

[iee01] *IEEE Intelligent Systems*, 16(2), 2001.

[IFF+99] Z. Ives, D. Florescu, M. Friedman, A. Levy, and D. Weld. An adaptive query execution system for data integration. In *Proc. of SIGMOD*, 1999.

[ILM+00] Z. Ives, A. Levy, J. Madhavan, R. Pottinger, S. Saroiu, I. Tatarinov, S. Betzler, Q. Chen, E. Jaslikowska, J. Su, and W. Yeung. Self-organizing data sharing communities with sagres. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, page 582, 2000.

[KMA+98] C. Knoblock, S. Minton, J. Ambite, N. Ashish, P. Modi, I. Muslea, A. Philpot, and S. Tejada. Modeling web sources for information integration. In *Proc. of the National Conference on Artificial Intelligence (AAAI)*, 1998.

[KSL+99] G. Keim, N. Shazeer, M. Littman, S. Agarwal, C. Cheves, J. Fitzgerald, J. Grosland, F. Jiang, S. Pollard, and K. Weinmeister. PROVERB: The probabilistic cruciverbalist. In *Proc. of the 6th National Conf. on Artificial Intelligence (AAAI-99)*, pages 710–717, 1999.

[Kus00a] N. Kushmerick. Wrapper induction: Efficiency and expressiveness. *Artificial Intelligence*, 118(1–2):15–68, 2000.

[Kus00b] N. Kushmerick. Wrapper verification. *World Wide Web Journal*, 3(2):79–94, 2000.

[LC94] W. Li and C. Clifton. Semantic integration in heterogeneous databases using neural networks. In *Proceedings of the Conf. on Very Large Databases (VLDB)*, 1994.

[LC00] W. Li and C. Clifton. SEMINT: A tool for identifying attribute correspondence in heterogeneous databases using neural networks. *Data and Knowledge Engineering*, 33:49–84, 2000.

[LCL00] W. Li, C. Clifton, and S. Liu. Database integration using neural network: implementation and experience. *Knowledge and Information Systems*, 2(1):73–96, 2000.

[LG01] M. Lacher and G. Groh. Facilitating the exchange of explixit knowledge through ontology mappings. In *Proceedings of the 14th Int. FLAIRS conference*, 2001.

[Lin98] D. Lin. An information-theoretic definition of similarity. In *Proceedings of the International Conference on Machine Learning (ICML)*, 1998.

[LKG99] E. Lambrecht, S. Kambhampati, and S. Gnanaprakasam. Optimizing recursive information gathering plans. In *Proc. of the Int. Joint Conf. on AI (IJCAI)*, 1999.

[Llo83] S. Lloyd. An optimization approach to relaxation labeling algorithms. *Image and Vision Computing*, 1(2), 1983.

[LRO96] A. Y. Levy, A. Rajaraman, and J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proc. of VLDB*, 1996.

[MBR01] J. Madhavan, P.A. Bernstein, and E. Rahm. Generic schema matching with Cupid. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 2001.

[MFRW00] D. McGuinness, R. Fikes, J. Rice, and S. Wilder. The Chimaera ontology environment. In *Proceedings of the 17th National Conference on Artificial Intelligence*, 2000.

[MHDB02] J. Madhavan, A. Halevy, P. Domingos, and P. Bernstein. Representing and reasoning about mappings between domain models. In *Proceedings of the National AI Conference (AAAI-02)*, 2002.

[MHH00] R. Miller, L. Haas, and M. Hernandez. Schema mapping as query discovery. In *Proc. of VLDB*, 2000.

[MHTH01] P. Mork, A. Halevy, and P. Tarczy-Hornoch. A model of data integration system of biomedical data applied to online genetic databases. In *Proceedings of the Symposium of the American Medical Informatics Association*, 2001.

[MMGR02] S. Melnik, H. Molina-Garcia, and E. Rahm. Similarity flooding: a versatile graph matching algorithm. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 2002.

[MN98] A. McCallum and K. Nigam. A comparison of event models for Naive Bayes text classification. In *Proceedings of the AAAI-98 Workshop on Learning for Text Categorization*, 1998.

[MS99] C. Manning and H. Schütze. *Foundations of Statistical Natural Language Processing*, pages 575–608. The MIT Press, Cambridge, US, 1999.

[MS01] A. Maedche and S. Saab. Ontology learning for the Semantic Web. *IEEE Intelligent Systems*, 16(2), 2001.

[MT94] R. Michalski and G. Tecuci, editors. *Machine Learning: A Multistrategy Approach*. Morgan Kaufmann, 1994.

[MWJ] P. Mitra, G. Wiederhold, and J. Jannink. Semi-automatic integration of knowledge sources. In *Proceedings of Fusion'99*.

[MZ98] T. Milo and S. Zohar. Using schema matching to simplify heterogeneous data translation. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 1998.

[NHT+02] F. Neumann, CT. Ho, X. Tian, L. Haas, and N. Meggido. Attribute classification using feature analysis. In *Proceedings of the Int. Conf. on Data Engineering (ICDE)*, 2002.

[NM00] N.F. Noy and M.A. Musen. PROMPT: Algorithm and tool for automated ontology merging and alignment. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 2000.

[NM01] N.F. Noy and M.A. Musen. Anchor-PROMPT: Using non-local context for semantic Matching. In *Proceedings of the Workshop on Ontologies and Information Sharing at the International Joint Conference on Artificial Intelligence (IJCAI)*, 2001.

[NM02] NF. Noy and MA. Musen. PromptDiff: A fixed-point algorithm for comparing ontology versions. In *Proceedings of the Nat. Conf. on Artificial Intelligence (AAAI)*, 2002.

[Ome01] B. Omelayenko. Learning of ontologies for the Web: the analysis of existent approaches. In *Proceedings of the International Workshop on Web Dynamics*, 2001.

[ont] http://ontobroker.semanticweb.org.

[Pad98] L. Padro. A hybrid environment for syntax-semantic tagging, 1998.

[PB02] R. Pottinger and P. Bernstein. Creating a mediated schema based on initial correspondences. *IEEE Data Engineering Bulletin*, 25(3), 2002.

[PE95] M. Perkowitz and O. Etzioni. Category translation: Learning to understand information on the Internet. In *Proc. of Int. Joint Conf. on AI (IJCAI)*, 1995.

[PR00] V. Punyakanok and D. Roth. The use of classifiers in sequential inference. In *Proceedings of the Conference on Neural Information Processing Systems (NIPS-00)*, 2000.

[PS98] C. Parent and S. Spaccapietra. Issues and approaches of database integration. *Communications of the ACM*, 41(5):166–178, 1998.

[PSTU99] L. Palopoli, D. Sacca, G. Terracina, and D. Ursino. A unififed graph-based framework for deriving nominal interscheme properties, type conflicts, and object cluster similarities. In *Proceedings of the Conf. on Cooperative Information Systems (CoopIS)*, 1999.

[PSU98] L. Palopoli, D. Sacca, and D. Ursino. Semi-automatic, semantic discovery of properties from database schemes. In *Proc. of the Int. Database Engineering and Applications Symposium (IDEAS-98)*, pages 244–253, 1998.

[PTU00] L. Palopoli, G. Terracina, and D. Ursino. The system DIKE: towards the semi-automatic synthesis of cooperative information systems and data warehouses. In *Proceedings of the ADBIS-DASFAA Conf.*, 2000.

[PVH$^+$02] L. Popa, Y. Velegrakis, M. Hernandez, R. J. Miller, and R. Fagin. Translating web data. In *Proceedings of the Int. Conf. on Very Large Databases (VLDB)*, 2002.

[RB01] E. Rahm and P.A. Bernstein. On matching schemas automatically. *VLDB Journal*, 10(4), 2001.

[RD00] E. Rahm and H. Do. Data cleaning: Problems and current approaches. *IEEE Data Engineering Bulletin*, 2000.

[RHS01] I. Ryutaro, T. Hideaki, and H. Shinichi. Rule induction for concept hierarchy alignment. In *Proceedings of the 2nd Workshop on Ontology Learning at the 17th Int. Joint Conf. on AI (IJCAI)*, 2001.

[RMR00] A. Rosenthal, F. Manola, and S. Renner. Getting data to applications: Why we fail, and how we can do better. In *Proceedings of the AFCEA Federal Database Conference*, 2000.

[RRSM01] A. Rosenthal, S. Renner, L. Seligman, and F. Manola. Data integration needs an industrial revolution. In *Proceedings of the Workshop on Foundations of Data Integration*, 2001.

[RS01] A. Rosenthal and L. Seligman. Scalability issues in data integration. In *Proceedings of the AFCEA Federal Database Conference*, 2001.

[SL90] AP. Seth and JA. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Survey*, 22(3):183–236, 1990.

[SR01] L. Seligman and A. Rosenthal. The impact of xml in databases and data sharing. *IEEE Computer*, 2001.

[SRLS02] L. Seligman, A. Rosenthal, P. Lehner, and A. Smith. Data integration: Where does the time go? *IEEE Data Engineering Bulletin*, 2002.

[TD97] L. Todorovski and S. Dzeroski. Declarative bias in equation discovery. In *Proceedings of the Int. Conf. on Machine Learning (ICML)*, 1997.

[TW99] K. M. Ting and I. H. Witten. Issues in stacked generalization. *Journal of Artificial Intelligence Research*, 10:271–289, 1999.

[UDB] UDB: The unified database for human genome computing. http://bioinformatics.weizmann.ac.il/udb.

[vR79] van Rijsbergen. *Information Retrieval*. London:Butterworths, 1979. Second Edition.

[Wol92] D. Wolpert. Stacked generalization. *Neural Networks*, 5:241–259, 1992.

[Wor] Wordnet: A lexical database for the English language. http://www.cogsci.princeton.edu/ wn.

[XML98] Extensible markup language (XML) 1.0. www.w3.org/TR/1998/REC-xml-19980210, 1998. W3C Recommendation.

[Xqu] XQuery: An XML query language. http://www.w3.org/TR/xquery.

[XSL99] XSL Transformations (XSLT), version 1.0. http://www.w3.org/TR/xslt, 13 August 1999. W3C Working Draft.

[YMHF01] L.L. Yan, R.J. Miller, L.M. Haas, and R. Fagin. Data driven understanding and refinement of schema mappings. In *Proceedings of the ACM SIGMOD*, 2001.

[YS00] J. Yi and N. Sundaresan. A classifier for semi-structured documents. In *Proc. of the 6th Int. Conf. on Knowledge Discovery and Data Mining (KDD-2000)*, 2000.

Appendix A

DATA PROCESSING FOR LSD EXPERIMENTS

In Chapter 3 we described experiments with the LSD system. To enable a thorough evaluation of the experiments, we now describe the process of creating experimental data in detail. Throughout the appendix, we illustrate this process with data from real-estate domains. The entire data set that we used for our experiments can be found in the Schema Matching Archive, at *http://anhai.cs.uiuc.edu/archive*.

A.1 Selecting the Domains

We selected three domains on the WWW: Real Estate, Time Schedule, and Faculty Listings. (Later we used real-estate data to create two domains: a smallish Real Estate I and a larger Real Estate II.) A domain was selected based primarily on four criteria. First, it should be about a topic that we were familiar with, so that we could create a good mediated DTD and evaluate the semantic mappings. Second, the domain must have at least 10 to 15 sources, because we needed five sources for creating the mediated DTD (see below) and five additional sources for experiments. Third, it should be relatively easy to extract data from the sources intended for the experiments. That is, the data at the sources should be either browsable or reachable via a *single* query interface (instead of via a *series* of complex query interfaces). Finally, the sources should have non-trivial structure, so as to not make the matching problem trivial.

Additional domains that we found satisfying the above criteria were auction listings (*ebay.com, auctions.yahoo.com*, etc.) and restaurant guides (*zagat.com, seattleinsider.com/restaurants*, etc.). These domains may serve as good testbeds for future research on schema matching.

A.2 Creating a Mediated DTD and Selecting Sources for Each Domain

A.2.1 Creating Mediated DTDs

For each domain, we created a mediated DTD. We began by selecting a source on the WWW that covers many relevant elements (i.e., attributes) of the domain. We created an XML DTD for the source, based on the source structure, then used the DTD as an initial mediated DTD. Next, we revised and expanded the initial mediated DTD, based on our knowledge about the domain.

Finally, we examined several additional sources in the domain, and revised the mediated DTD to take into account the elements of the new sources. This step was necessary to ensure that the final mediated DTD would be as comprehensive as possible.

The step turned out to be a slow and rather labor-intensive process. For each additional source that we examined, we had to create a DTD based on the source structure. Then for each DTD element we inspected its name and data to understand its meaning. Next, we decided if the element appears frequently enough (across the sources) to warrant being added to the mediated DTD. To

```
<!ELEMENT house_listing (house_address,house_description,price,bedrooms,bathrooms,
 lot_area,garage,school,mls_number,contact_info)>
<!ELEMENT house_address (#PCDATA)>
<!ELEMENT house_description (#PCDATA)>
<!ELEMENT price (#PCDATA)>
<!ELEMENT bedrooms (#PCDATA)>
<!ELEMENT bathrooms (#PCDATA)>
<!ELEMENT lot_area (#PCDATA)>
<!ELEMENT garage (#PCDATA)>
<!ELEMENT school (#PCDATA)>
<!ELEMENT mls_number (#PCDATA)>
<!ELEMENT contact_info (firm_info,agent_info)>
<!ELEMENT firm_info (firm_name,firm_address,firm_phone)>
<!ELEMENT firm_name (#PCDATA)>
<!ELEMENT firm_address (#PCDATA)>
<!ELEMENT firm_phone (#PCDATA)>
<!ELEMENT agent_info (agent_name,agent_address,agent_phone,agent_fax)>
<!ELEMENT agent_name (#PCDATA)>
<!ELEMENT agent_address (#PCDATA)>
<!ELEMENT agent_phone (#PCDATA)>
<!ELEMENT agent_fax (#PCDATA)>
```

Figure A.1: The mediated DTD of the Real Estate I domain.

estimate the element's frequency, we had to match it with other elements in other sources. Finally, if we were to add the element to the mediated DTD, we had to decide on *how* to add it.

For example, suppose that the mediated DTD has an element description, which consists of two subelements: basic-amenities and extra-amenities. Suppose also that we want to add the element house-desc, which consists of three subelements: interior-desc, exterior-desc, and lot-desc. How should we incorporate this element into the mediated DTD? In a sense, this incorporation process is schema integration, a task that is well-known to be difficult and labor intensive [BLN86].

Note that for the real-estate domain we actually created two mediated DTDs, one small (22 elements) and one large (66 elements), thus in effect creating two domains: Real Estate I and Real Estate II. Figure A.1 shows the mediated DTD of Real Estate I, and Figure A.2 shows that of Real Estate II.

A.2.2 Selecting Sources

For each domain, once the mediated DTD has been created, we selected five sources on the WWW (which are different from the sources used in the process of creating the mediated DTD). We focused on selecting sources that have fairly complex structure and enable realively easy data extraction. The selected sources were:

Real Estate: *homeseekers.com, nky.com, texasproperties.com, windermere.com*, and *http://list.realestate.yahoo.com*.

Time Schedule: The sources listed the time schedules of courses which were offered in Spring 2000 at Reed College (*reed.edu*), Rice University (*rice.edu*), the University of Washington at Seattle (*washington.edu*), the University of Wisconsin at Milwaukee (*uwm.edu*), and Washington State University (*wsu.edu*).

Faculty Listings: The sources listed the faculty homepages of five Computer Science Departments at the University of California at Berkeley (*cs.berkeley.edu*), Cornell University (*cs.cornell.edu*), the Univeristy of Michigan at Ann Arbor (*eecs.umich.edu*), the Universit of Texas at Austin (*cs.utexas.edu*), and the University of Washington at Seattle (*cs.washington.edu*).

A.3 Creating Data and Manual Semantic Mappings for Each Source in a Domain

A.3.1 Extracting HTML Data

All HTML data from the sources were extracted during a two-month period in Winter 1999 - Spring 2000. We carried out data extraction using a modified version of the WWW:Robot Perl module available at *cpan.org*.

We extracted HTML data from the Real Estate and Time Schedule domains. For the Faculty Listings domain, we decided to bypass the HTML stage and extracted data directly into XML format, for two reasons. First, the amount of data to be extracted was relatively small (at most 50 faculty homepages per source), so that manual extraction was possible. Second, most faculty homepages contain a large amount of free text. Hence, even if we were to extract the data into HTML format, we would still have ended up extracting data *manually* from free text into XML format.

A.3.2 Creating Source DTDs

Next, for each source we created a DTD. In doing so, we were careful to mirror the structure of the data in the source, and to use the terms from the source for the matching purposes. To do this, we created *three files* per source.

The first file contains the source DTD. Figure A.3 shows the DTD for Real Estate source *homeseekers.com*. The tag name of each element of this DTD is unique and descriptive, and serves as an internal ID for that element only.

The second file contains for each DTD element its "public name", that is, the name that appears (for that element) in the data of the source. Figure A.4 shows the public names for the DTD elements in Figure A.3. An empty public name means that the data corresponding to the DTD element would appear without any name in a house listing. For example, suppose address information in a source is formatted as follows:

```
City: Seattle
Washington, 98105
```

This means that cities always appear with the public name "city", while states and zipcodes appear with empty public names. Given this, we may create an XML element named address. The data elements of address would look as follows:

```
<address>
    <city> ... </city>
    <state-zip> ... </state-zip>
</address>
```

where address, city, and state-zip are internal IDs that refer to address, city, and state and zipcode, respectively. The portion of the "public name" file that corresponds to these DTD elements would look as follows:

```
<internal>address</internal>           <public></public>
<internal>city</internal>              <public>City</public>
<internal>state-zip</internal>         <public></public>
```

The third file contains for each DTD element X its "long public name": the concatenation of all public names of DTD elements on the path from the root to X. Figure A.5 shows the long public names (denoted by element path) of the DTD elements for source *homeseekers.com*. These long public names are used by the *Name Learner* (see for example Chapter 3) to match schema elements.

Figures A.6– A.9 show the DTDs that we created for the sources *nky.com, texasproperties.com, windermere.com*, and *realestate.yahoo.com*, respectively.

A.3.3 Converting HTML Data to XML Data

For each source in the Real Estate and Time Schedule domains, we converted the extracted HTML data into XML data that conforms to the source DTD. This step in essence built an HTML-to-XML wrapper for the source. Wrapper construction is well-known to be extremely tedious and labor-intensive. For each source-DTD element we had to examine several HTML listings to craft a rule that exploits HTML regularities to extract data for that element. The first rule that we crafted was often good but not perfect, in the sense that it did not cover all scenarios. Hence, typically we had to iterate several times to converge on an acceptable rule.
Here is a sample house listing in XML for source *homeseekers.com*, in the Real Estate I domain:

```
<?xml version='1.0' ?>
<house_listing>
<house_description>This 1 level home built in 1969 has 4 bedroom(s), 4 full bath(s)/1
half bath(s) and approximately 3654 square feet of living area. Rooms
include dining room, master bedroom. Features include air
conditioning.</house_description>
<contact_info>
   <agent_info>
      <agent_name>Gigi Winston</agent_name>
      <direct>202-333-4167</direct>
      <office>202-333-4167</office>
   </agent_info>
   <firm_info>
      <firm_name>Winston & Winston Real Estate</firm_name>
      <phone>202-333-4167</phone>
      <fax></fax>
   </firm_info>
</contact_info>
<list_price>$1,100,000</list_price>
<location>Washington, DC 20037</location>
<mls_#>DC2733876</mls_#>
<baths>4(full) 1(half)</baths>
<bedrooms>4</bedrooms>
</house_listing>
```

For sources in the Faculty Listings domain, we manually extracted data from HTML listings to create XML data. As mentioned earlier, this is because in this domain the HTML listings contain mostly *free text* descriptions, which provide too few regularities for us to craft extraction rules.

Note that we also performed some trivial data cleaning operations such as removing "unknown", "unk", and splitting "$70000" into "$" and "70000" on the created XML data.

A.3.4 Creating the Manual Semantic Mappings

After creating the XML data and obtaining a thorough understanding of the data, for each source we considered its DTD and created for each DTD element an 1-1 semantic mapping. The mapping pairs the source-DTD element with the semantically equivalent mediated-DTD element (or otherwise with the unique element OTHER).
Figure A.10 shows the semantic mappings we created for source *homeseekers.com*. The line

```
<source>agent_info</source> <mediated>agent_info</mediated>
```

means source-DTD element agent-info matches mediated-DTD element agent-info. Note that the tag names we use here are the *internal tagnames* that we created for data manipulation purposes (see Section A.3.2). We often created the same internal tagnames for semantically equivalent elements in the source- and mediated DTDs. Hence, the fact that matching elements often have the same internal tagnames (as seen in Figure A.10) should not be thought of as implying that, in general, schema elements with the same (public) tagnames would match.
In Figure A.10, a line such as

```
<source>fax</source>  <mediated></mediated>
```

means that source-DTD element fax matches element OTHER.

A.4 Creating Integrity Constraints for Each Domain

After creating the manual mappings for each source, we created the integrity constraints for each domain. Figure A.11 shows the constraints we created for the Real Estate I domain.

Frequency Constraints: The first set of constraints that we created was frequency constraints. These constraints are shown starting with the line

```
<name/>FREQUENCY.
```

A line such as

```
<value/>agent_address <= 1
```

specifies that at most one source-DTD element can match the mediated-DTD element agent_address.

Inclusion Constraints: Next, we created nesting constraints, also known as inclusion/exclusion constraints. These constraints are shown starting with the line

```
<name/>INCLUSION
```

Consider two source-DTD elements a and b. Assumes that b is a child of a (in the source-DTD tree). Then a constraint such as

```
<value/>agent_address INCLUDES ONLY
```

specifies that if a matches agent_address then b can only matches OTHER; b cannot match any other element of the mediated-DTD.
A line such as

```
<value/>agent_info INCLUDES ONLY
                        agent_name agent_address agent_phone agent_fax
```

specifies that if a matches agent_info then b can only matches agent_name, agent_address, agent_phone, agent_fax, or OTHER. b cannot match any other element of the mediated-DTD.

Contiguity Constraints: These constraints are shown starting with the line

```
<name/>CONTIGUITY.
```

A constraint such as

```
<value/>bathrooms bedrooms
```

specifies that if source-DTD elements a and b match mediate-DTD elements bathrooms and bedrooms, respectively, then a and b must be sibling nodes of the source-DTD tree.

Proximity Constraints: These constraints are shown starting with the line

```
<name/>PROXIMITY.
```

A constraint such as

```
<value/>firm_info firm_name firm_address firm_phone
```

specifies that if source-DTD elements a, b, c, and d match mediated-DTD elements firm_info, firm_name, firm_address, and firm_phone, respectively, then we prefer the source-DTD elements to be as close to one another as possible, all other things being equal.

The integrity constraints for other domains can be found in the Schema Matching Archive.

A.5 Pseudo Code of LSD

In Chapter 3 we have provided a *conceptual* view on the working of LSD. In this section we provide the pseudo code of the implemented LSD system. Figures A.12 and A.13 describe the training and matching phases, respectively.

```
<!ELEMENT house_listing (basic_info,additional_info)>

<!ELEMENT basic_info
          (house_location,house_price,contact_info,garage_info,
           size_info,rooms,schools,house_description)>

<!ELEMENT house_location
          (house_address,neighborhood,city,county,suburb,state)>
<!ELEMENT house_address (#PCDATA)>
<!ELEMENT neighborhood (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ELEMENT county (#PCDATA)>
<!ELEMENT suburb (#PCDATA)>
<!ELEMENT state (#PCDATA)>

<!ELEMENT house_price (#PCDATA)>

<!ELEMENT contact_info (agent_info,firm_info)>

<!ELEMENT agent_info
          (agent_name,agent_address,agent_phone,agent_fax,agent_pager,agent_email)>
<!ELEMENT agent_name (#PCDATA)>
<!ELEMENT agent_address (#PCDATA)>
<!ELEMENT agent_phone (#PCDATA)>
<!ELEMENT agent_fax (#PCDATA)>
<!ELEMENT agent_pager (#PCDATA)>
<!ELEMENT agent_email (#PCDATA)>

<!ELEMENT firm_info
          (firm_name,firm_address,firm_phone,firm_fax,firm_voice_mail,firm_email)>
<!ELEMENT firm_name (#PCDATA)>
<!ELEMENT firm_address (#PCDATA)>
<!ELEMENT firm_phone (#PCDATA)>
<!ELEMENT firm_fax (#PCDATA)>
<!ELEMENT firm_voice_mail (#PCDATA)>
<!ELEMENT firm_email (#PCDATA)>

<!ELEMENT garage_info (garage,carport)>
<!ELEMENT garage (#PCDATA)>
<!ELEMENT carport (#PCDATA)>

<!ELEMENT size_info
          (building_area,building_dimensions,lot_area,lot_dimensions)>
<!ELEMENT building_area (#PCDATA)>
<!ELEMENT building_dimensions (#PCDATA)>
<!ELEMENT lot_area (#PCDATA)>
<!ELEMENT lot_dimensions (#PCDATA)>

<!ELEMENT rooms (basement,bath_rooms,bed_rooms,dining_room,living_room)>
<!ELEMENT basement (#PCDATA)>
<!ELEMENT bath_rooms (#PCDATA)>
<!ELEMENT bed_rooms (#PCDATA)>
<!ELEMENT dining_room (#PCDATA)>
<!ELEMENT living_room (#PCDATA)>

<!ELEMENT schools (elementary_school,middle_school,high_school)>
<!ELEMENT elementary_school (#PCDATA)>
<!ELEMENT middle_school (#PCDATA)>
<!ELEMENT high_school (#PCDATA)>

<!ELEMENT house_description (#PCDATA)>

<!ELEMENT additional_info
          (utilities,amenities,mls_num,stories,type,
          architectural_style,date_built,age,availability)>

<!ELEMENT utilities (cooling,heating,gas,sewer,water,electricity)>
<!ELEMENT cooling (#PCDATA)>
<!ELEMENT heating (#PCDATA)>
<!ELEMENT gas (#PCDATA)>
<!ELEMENT sewer (#PCDATA)>
<!ELEMENT water (#PCDATA)>
<!ELEMENT electricity (#PCDATA)>

<!ELEMENT amenities
          (fireplace,patio,swimming_pool,spa,view,waterfront)>
<!ELEMENT fireplace (#PCDATA)>
<!ELEMENT patio (#PCDATA)>
<!ELEMENT swimming_pool (#PCDATA)>
<!ELEMENT spa (#PCDATA)>
<!ELEMENT view (#PCDATA)>
<!ELEMENT waterfront (#PCDATA)>

<!ELEMENT mls_num (#PCDATA)>
<!ELEMENT stories (#PCDATA)>
<!ELEMENT type (#PCDATA)>
<!ELEMENT architectural_style (#PCDATA)>
<!ELEMENT date_built (#PCDATA)>
<!ELEMENT age (#PCDATA)>
<!ELEMENT availability (#PCDATA)>
```

Figure A.2: The mediated DTD of the Real Estate II Domain.

```
<!ELEMENT house_listing (house_description,contact_info,list_price,location,
neighborhood,mls_#,baths,bedrooms,garage,lot_size)>

<!ELEMENT house_description (#PCDATA)>

<!ELEMENT contact_info (agent_info,firm_info)>

<!ELEMENT agent_info (agent_name,direct,office)>
<!ELEMENT agent_name  (#PCDATA)>
<!ELEMENT direct (#PCDATA)>
<!ELEMENT office (#PCDATA)>

<!ELEMENT firm_info (firm_name,phone,fax)>
<!ELEMENT firm_name (#PCDATA)>
<!ELEMENT phone (#PCDATA)>
<!ELEMENT fax (#PCDATA)>

<!ELEMENT list_price (#PCDATA)>
<!ELEMENT location (#PCDATA)>
<!ELEMENT neighborhood (#PCDATA)>
<!ELEMENT mls_# (#PCDATA)>
<!ELEMENT baths (#PCDATA)>
<!ELEMENT bedrooms (#PCDATA)>
<!ELEMENT garage (#PCDATA)>
<!ELEMENT lot_size (#PCDATA)>
```

Figure A.3: The DTD of source *homeseekers.com*.

```
<name_mappings>
<internal>agent_info</internal><public></public>
<internal>agent_name</internal><public></public>
<internal>baths</internal><public>baths</public>
<internal>bedrooms</internal><public>bedrooms</public>
<internal>contact_info</internal><public></public>
<internal>direct</internal><public>direct</public>
<internal>fax</internal><public>fax</public>
<internal>firm_info</internal><public></public>
<internal>firm_name</internal><public></public>
<internal>garage</internal><public>garage</public>
<internal>house_description</internal><public></public>
<internal>house_listing</internal><public></public>
<internal>list_price</internal><public>list_price</public>
<internal>location</internal><public>location</public>
<internal>lot_size</internal><public>lot_size</public>
<internal>mls_#</internal><public>mls_#</public>
<internal>neighborhood</internal><public>neighborhood</public>
<internal>office</internal><public>office</public>
<internal>phone</internal><public>phone</public>
</name_mappings>
```

Figure A.4: The “public names” of the elements of the DTD for source *homeseekers.com*.

```
<path_mappings>
<internal>agent_info</internal><path></path>
<internal>agent_name</internal><path></path>
<internal>baths</internal><path>baths</path>
<internal>bedrooms</internal><path>bedrooms</path>
<internal>contact_info</internal><path></path>
<internal>direct</internal><path>direct</path>
<internal>fax</internal><path>fax</path>
<internal>firm_info</internal><path></path>
<internal>firm_name</internal><path></path>
<internal>garage</internal><path>garage</path>
<internal>house_description</internal><path></path>
<internal>house_listing</internal><path></path>
<internal>list_price</internal><path>list_price</path>
<internal>location</internal><path>location</path>
<internal>lot_size</internal><path>lot_size</path>
<internal>mls_#</internal><path>mls_#</path>
<internal>neighborhood</internal><path>neighborhood</path>
<internal>office</internal><path>office</path>
<internal>phone</internal><path>phone</path>
</path_mappings>
```

Figure A.5: The “long public names” of the elements of the DTD for source *homeseekers.com*.

```
<!ELEMENT house_listing (house_location,price,bedrooms,baths,garage,suburb,school_dist,
 mls#,contact_info,house_description,lot_dimensions,directions)>

<!ELEMENT house_location (#PCDATA)>
<!ELEMENT price (#PCDATA)>
<!ELEMENT bedrooms (#PCDATA)>
<!ELEMENT baths (#PCDATA)>
<!ELEMENT garage (#PCDATA)>
<!ELEMENT suburb (#PCDATA)>
<!ELEMENT school_dist (#PCDATA)>
<!ELEMENT mls# (#PCDATA)>

<!ELEMENT contact_info (firm_info,agent_info)>
<!ELEMENT firm_info (firm,firm_phone)>
<!ELEMENT firm (#PCDATA)>
<!ELEMENT firm_phone (#PCDATA)>

<!ELEMENT agent_info (agent,agent_phone)>
<!ELEMENT agent (#PCDATA)>
<!ELEMENT agent_phone (#PCDATA)>

<!ELEMENT house_description (#PCDATA)>
<!ELEMENT lot_dimensions (#PCDATA)>
<!ELEMENT directions (#PCDATA)>
```

Figure A.6: The DTD of source *nky.com*.

```
<!ELEMENT house_listing (contact_info,mls#,house_location,price,bedrooms,full_baths,
                         half_baths,garage_spaces,house_description,approx_lot_size,
 school_district)>
<!ELEMENT contact_info (firm_information,agent_information)>
<!ELEMENT firm_information (firm_name,firm_location,firm_office,firm_fax)>
<!ELEMENT firm_name (#PCDATA)>
<!ELEMENT firm_location (#PCDATA)>
<!ELEMENT firm_office (#PCDATA)>
<!ELEMENT firm_fax (#PCDATA)>
<!ELEMENT agent_information (agent_name,agent_office,agent_fax)>
<!ELEMENT agent_name (#PCDATA)>
<!ELEMENT agent_office (#PCDATA)>
<!ELEMENT agent_fax (#PCDATA)>
<!ELEMENT mls# (#PCDATA)>
<!ELEMENT house_location (#PCDATA)>
<!ELEMENT price (#PCDATA)>
<!ELEMENT bedrooms (#PCDATA)>
<!ELEMENT full_baths (#PCDATA)>
<!ELEMENT half_baths (#PCDATA)>
<!ELEMENT garage_spaces (#PCDATA)>
<!ELEMENT house_description (#PCDATA)>
<!ELEMENT approx_lot_size (#PCDATA)>
<!ELEMENT school_district (#PCDATA)>
```

Figure A.7: The DTD of source *texasproperties.com*.

```
<!ELEMENT house_listing (price,mls_id_number,address,bathrooms,bedrooms,lot_size,garage,
schools,comments,for_more_information_contact)>

<!ELEMENT  price (#PCDATA)>
<!ELEMENT  mls_id_number (#PCDATA)>
<!ELEMENT  address (#PCDATA)>
<!ELEMENT  bathrooms (#PCDATA)>
<!ELEMENT  bedrooms (#PCDATA)>
<!ELEMENT  lot_size (#PCDATA)>
<!ELEMENT  garage (#PCDATA)>
<!ELEMENT  schools (elementary,middle_school,high_school)>
<!ELEMENT  elementary (#PCDATA)>
<!ELEMENT  middle_school (#PCDATA)>
<!ELEMENT  high_school (#PCDATA)>
<!ELEMENT  comments (#PCDATA)>
<!ELEMENT  for_more_information_contact (agent_name,firm_name,firm_location,
office_phone,cell_phone,fax)>
<!ELEMENT  agent_name (#PCDATA)>
<!ELEMENT  firm_name (#PCDATA)>
<!ELEMENT  firm_location (#PCDATA)>
<!ELEMENT  office_phone (#PCDATA)>
<!ELEMENT  cell_phone (#PCDATA)>
<!ELEMENT  fax (#PCDATA)>
```

Figure A.8: The DTD of source *windermere.com*.

```
<!ELEMENT house_listing (house_location,description,home_features,
date_posted,price,beds,baths,agency/brokerage,sq._footage,
lotsize,garage,school,mls,contact,location,fax,phone,ad_id)>
<!ELEMENT house_location (#PCDATA)>
<!ELEMENT description (#PCDATA)>
<!ELEMENT home_features (#PCDATA)>
<!ELEMENT date_posted (#PCDATA)>
<!ELEMENT price (#PCDATA)>
<!ELEMENT beds (#PCDATA)>
<!ELEMENT baths (#PCDATA)>
<!ELEMENT agency/brokerage (#PCDATA)>
<!ELEMENT sq._footage (#PCDATA)>
<!ELEMENT lotsize (#PCDATA)>
<!ELEMENT garage (#PCDATA)>
<!ELEMENT school (#PCDATA)>
<!ELEMENT mls (#PCDATA)>
<!ELEMENT contact (#PCDATA)>
<!ELEMENT location (#PCDATA)>
<!ELEMENT fax (#PCDATA)>
<!ELEMENT phone (#PCDATA)>
<!ELEMENT ad_id (#PCDATA)>
```

Figure A.9: The DTD of source *realestate.yahoo.com*.

```
<mappings>
<source>agent_info</source>          <mediated>agent_info</mediated>
<source>agent_name</source>          <mediated>agent_name</mediated>
<source>baths</source>               <mediated>bathrooms</mediated>
<source>bedrooms</source>            <mediated>bedrooms</mediated>
<source>contact_info</source>        <mediated>contact_info</mediated>
<source>direct</source>              <mediated>agent_phone</mediated>
<source>fax</source>                 <mediated></mediated>
<source>firm_info</source>           <mediated>firm_info</mediated>
<source>firm_name</source>           <mediated>firm_name</mediated>
<source>garage</source>              <mediated>garage</mediated>
<source>house_description</source>   <mediated>house_description</mediated>
<source>house_listing</source>       <mediated>house_listing</mediated>
<source>list_price</source>          <mediated>price</mediated>
<source>location</source>            <mediated>house_address</mediated>
<source>lot_size</source>            <mediated>lot_area</mediated>
<source>mls_#</source>               <mediated>mls_number</mediated>
<source>neighborhood</source>        <mediated></mediated>
<source>office</source>              <mediated>agent_phone</mediated>
<source>phone</source>                <mediated>firm_phone</mediated>
</mappings>
```

Figure A.10: Semantic mappings that we manually created for source *homeseekers.com*.

```
# contain constraints related to real-estate mediated schema elements
# in human readable format

##### for those that we don't specify in frequency constraint, they >= 0

<constraint>
   <name/>FREQUENCY
   <value/>agent_address <= 1
   <value/>agent_fax <= 1
   <value/>agent_info <= 1
   <value/>agent_name <= 1
   <value/>bathrooms <= 2 #### could have half bathrooms and full bathrooms
   <value/>bedrooms <= 1
   <value/>contact_info <= 1
   <value/>firm_address <= 1
   <value/>firm_info <= 1
   <value/>firm_name <= 1
   <value/>firm_phone <= 1
   <value/>garage <= 1
   <value/>house_address <= 1
   <value/>house_description <= 1
   <value/>house_listing <= 1
   <value/>lot_area <= 1
   <value/>mls_number <= 1
   <value/>price == 1
</constraint>

## INCLUDES ONLY also allows OTHERs
## a INCLUDES ONLY b c d means a can only include b c d or OTHER,
##                              but no other label in the mediated-schema
<constraint>
   <name/>INCLUSION
   <value/>agent_address INCLUDES ONLY
   <value/>agent_fax INCLUDES ONLY
   <value/>agent_info INCLUDES ONLY
                           agent_name agent_address agent_phone agent_fax
   <value/>agent_name INCLUDES ONLY
   <value/>agent_phone INCLUDES ONLY
   <value/>bathrooms INCLUDES ONLY
   <value/>bedrooms INCLUDES ONLY
   <value/>contact_info INCLUDES ONLY
                          agent_info agent_name agent_address agent_phone
                          agent_fax firm_info firm_name firm_address firm_phone
   <value/>firm_address INCLUDES ONLY
   <value/>firm_info INCLUDES ONLY firm_name firm_address firm_phone
   <value/>firm_name INCLUDES ONLY
   <value/>firm_phone INCLUDES ONLY
   <value/>garage INCLUDES ONLY
   <value/>house_address INCLUDES ONLY
   <value/>house_description INCLUDES ONLY
   <value/>house_listing INCLUDES ONLY
                           agent_address agent_fax agent_info agent_name agent_phone
                           bathrooms bedrooms contact_info firm_address firm_info firm_name firm_phone
                           garage house_address house_description lot_area mls_number price school
   <value/>lot_area INCLUDES ONLY
   <value/>mls_number INCLUDES ONLY
   <value/>price INCLUDES ONLY
   <value/>school INCLUDES ONLY
</constraint>

### allowing OTHERs in between
<constraint>
   <name/>CONTIGUITY
   <value/>contact_info firm_info agent_info firm_name firm_address firm_phone
           agent_name agent_address agent_phone agent_fax
   <value/>bathrooms bedrooms
</constraint>

<constraint>
   <name/>PROXIMITY
   <value/>firm_info firm_name firm_address firm_phone
   <value/>agent_info agent_name agent_address agent_phone agent_fax
   <value/>contact_info firm_info agent_info
</constraint>
```

Figure A.11: The integrity constraints that we created for the Real Estate I domain.

Algorithm LSD
Input: Mediated schema M.
Data sources $DS_1, \dots, DS_n$, with schemas $S_1, \dots, S_n$, respectively.

Output: Semantic mappings between M and the source schemas $S_1, \dots, S_n$.

Phase I – Training:

1. *(Manually Create Mappings)* Ask user to create semantic mappings between M and the first k source schemas: $S_1, \dots, S_k$.
2. *(Create Domain Constraints)* Ask user to examine M and specify a set of domain constraints C over M.
3. *(Extract Data Listings)* Extract from each training source $DS_i, i \in [1,k]$, a set of data listings D_i. Merge the sets D_i to obtain the set of all data listings T.
4. *(Extract Data Instances)* Extract from the set T all data instances (e.g., "⟨location⟩ Kent, WA⟨/location⟩" is an instance). Let the set of all data instances be $X = \{x_1, \dots, x_p\}$.
5. *(Train Meta-Learner)* The goal of this step is to obtain for each pair of learner L_i and label c_j a learner weight $W_{L_i}^{c_j}$, where label c_j is a mediated-schema tag.
   (a) For each base learner L_i:
      - Suppose learner L_i exploits only the i-th feature of each data instance x. Transform the set of all instances X into the set of instances for learner L_i by keeping only the i-th feature for each instance: $T(L_i) = \{f_i(x_1), \dots, f_i(x_p)\}$.
      - Divide the set of instances $T(L_i)$ into d equal parts: $T(L_i)_1, \dots, T(L_i)_d$.
      - For each part $T(L_i)_u$, train learner L_i on the remaining $(d-1)$ parts, then apply L_i to make predictions for each instance in $T(L_i)_u$. At the end of this step, L_i has made predictions for all instances in $T(L_i)$. Notice that a prediction made for instance t_q in $T(L_i)$ is actually a prediction made for instance x_q in X (t_q is simply $f_i(x_q)$, i.e., the i-th feature of x_q).

   (b) For each label c_j:
      - For each data instance x_q of X, each base learner L_i has issued a confidence score regarding x_q matching c_j (in Step 5.a). Use these confidence scores and the fact that x_q matches c_j or not (inferred from the manual mappings between mediated schema M and source schemas $S_1, \dots, S_k$) to assemble a training instance for the meta-learner. Let the set of these training instances be $T(ML, c_j)$.
      - Apply linear regression to $T(ML, c_j)$ to obtain a set of learner weights $\{W_{L_1}^{c_j}, W_{L_2}^{c_j}, \dots\}$, where each weight $W_{L_i}^{c_j}$ is the weight of learner L_i regarding predictions for label c_j.
6. *(Train Base Learners)* Train each base learner L_i on the set $T(L_i)$ (see Step 5.a on obtaining $T(L_i)$).

Figure A.12: The pseudo code for LSD: Phase I – Training.

Algorithm LSD **(Continued)**

Phase II – Matching: For each source $DS_i, i \in [k+1, n]$, the goal is to find semantic mappings between mediated schema M and source schema S_i.

1. *(Extract Source Data)* Extract a set of data listings from DS_i.
2. *(Classify Source-Schema Tags)* For each source-schema tag t_j:
   (a) Use the data listings to create for t_j a set of data instances Q.
   (b) For each data instance q_u in Q:
      - Apply each base learner to q_u.
      - Combine the base learners' predictions using the learner weights W that were obtained in the training phase.

   (c) Compute the average of the predictions of instances in Q. Let this average be the prediction for source-schema tag t_j.
3. *(Handle Domain Constraints & User Feedback)*
   REPEAT
   (a) Apply the constraint handler to the set of domain constraints C and the predictions of source-schema tags, to find the best mapping combination $m*$.
   (b) Ask user to give feedback on $m*$. Add the feedback to C.

   UNTIL user has verified that all mappings in $m*$ are correct.

Figure A.13: The pseudo code for LSD: Phase II – Matching.

Appendix B

DATA PROCESSING FOR COMAP EXPERIMENTS

In this appendix we describe the complex mappings created by the volunteers for the experiments with COMAP (Chapter 4).

Inventory Domain: Figure B.1 shows the mappings created for this domain. The source schema has a total of 26 elements. The left column of this figure shows the elements, and the right column shows the mappings for the elements. There are 15 one-to-one mappings and 11 complex mappings. The complex mappings involve operators such as _TIMES_, _CATCONVERT_, _CONCAT_, etc. The meaning of operators _TIMES_ and _CONCAT_ are obvious from their name.

Operator _CATCONVERT_ refers to "conversion" mappings between categorical attributes (see Section 4.2.3). For example, the source-schema element ship_via can take values 1, 2, or 3, whereas the target-schema element ship_via[1] can take values Federal_Shipping, Speedy_Express, and United_Package. Thus, the complex mapping

```
ship_via                        ##(_CATCONVERT_ ship_via)
```

says that source-schema element ship_via] can be obtained from target-schema element ship_via] using a "conversion" mapping. The specific conversion mapping specifies that Federal_Shipping, Speedy_Express, and United_Package map to 1, 2, and 3, respectively.

Real Estate I Domain: Figure B.2 shows the mappings created for this domain. The source schema has a total of 6 elements. The left column of this figure shows the elements, and the right column shows the mappings for the elements. There are 2 one-to-one mappings and 4 complex mappings. All complex mappings here are concatenation mappings.

Real Estate II Domain: Figure B.3 shows the mappings created for this domain. The source schema has a total of 19 elements. The left column of this figure shows the elements, and the right column shows the mappings for the elements. There are 6 one-to-one mappings and 13 complex mappings.

The only complex mapping that warrants further explanation here is

```
fireplace                ##(_APPEAR_ house_description).
```

This is a schema-mismatch mapping (see Section 4.2.3). It says that source-schema element fireplace can be obtained by examining the data of target-schema element house_description. Specifically, if house_description mentions or implies the existence of fireplaces, then the value of fireplace is 1 (meaning "yes"). Otherwise, it is 0 (meaning "no").

---

[1] Recall from Appendix A that these names are internal names, used for identification and data manipulation purposes. These two schema elements actually have different public names.

```
26
order_id                        ##order_id
unit_price                      ##unit_price
quantity                        ##quantity
discount                        ##discount
whole_cost                      ##(unit_price _TIMES_ quantity)
discount_cost                   ##((unit_price _TIMES_ quantity) _TIMES_ (1 - discount))
customer_id                     ##customer_id
order_date                      ##order_date
required_date                   ##required_date
shipped_date                    ##shipped_date
ship_via                        ##(_CATCONVERT_ ship_via)
freight                         ##freight
ship_name                       ##ship_name
employee_name                   ##(employee_first_name _CONCAT_ employee_last_name)
employee_phone                  ##(employee_area_code _CONCAT_ employee_phone_number)

ship_address                    ##(((ship_address _CONCAT_ ship_city)
                                  _CONCAT_ ship_postal_code) _CONCAT_ ship_country)

product_name                    ##(_CATCONVERT_ product_id)
supplier_id                     ##supplier_id
category_name                   ##(_CATCONVERT_ category_id)
quantity_per_unit               ##quantity_per_unit
units_in_stock                  ##units_in_stock
units_on_order                  ##units_on_order
reorder_level                   ##reorder_level
discontinued                    ##(_CATCONVERT_ discontinued)
product_in_stock_cost           ##(unit_price _TIMES_ units_in_stock)
product_order_cost              ##(unit_price _TIMES units_on_order)
```

Figure B.1: The complex mappings created for the Inventory domain.

```
6
house_address           ##((house_city _CONCAT_ house_state) _CONCAT_ house_zip_code)
price                   ##price
description             ##house_description
agent_name              ##(agent_first_name _CONCAT_ agent_last_name)
agent_phone             ##(agent_area_code _CONCAT_ agent_phone_number)
contact_address         ##(agent_city _CONCAT_ agent_state)
```

Figure B.2: The complex mappings created for the Real Estate I domain.

```
19
house_address          ##(((house_street _CONCAT_ house_city)
                          _CONCAT_ house_state) _CONCAT_ house_zip_code)
house_price            ##house_price
agent_name             ##(agent_first_name _CONCAT_ agent_last_name)
agent_phone            ##(agent_area_code _CONCAT_ agent_phone_number)
agent_email            ##agent_email
firm_name              ##firm_name
firm_address           ##(firm_city _CONCAT_ firm_state)
garage                 ##garage
building_area          ##(building_area _DIVIDE_ 43560)
lot_area               ##((lot_dimension1 _TIMES_ lot_dimension2) _DIVIDE_ 43560)
num_rooms              ##(((bath_rooms _PLUS_ bed_rooms)
                         _PLUS_ dining_rooms) _PLUS_ living_rooms)
school                 ##elementary_school
house_description      ##house_description
fireplace              ##(_APPEAR_ house_description)
sewer                  ##(_CATCONVERT_ sewer)
water                  ##(_CATCONVERT_ water)
electricity            ##(_CATCONVERT_ electricity)
utilities              ##(heating _CONCAT_ cooling)
type                   ##type
```

Figure B.3: The complex mappings created for the Real Estate II domain.

VITA

AnHai Doan grew up in a small fishing village in rural North-Central Vietnam. Even at an early age, he already showed an exceptional ability to move west. After high school in Vinh, Nghe An, he moved to Hungary, where he earned a B.S. degree from Kossuth Lajos University in 1993. Next, he landed in Wisconsin, where he drank beer, listened to (too) much jazz and blues, and earned a M.S. from the University of Wisconsin, Milwaukee in 1996. Then he moved again, this time going as far west as he could, finally ending up in Seattle. There, he switched to coffee (lots of it!), salmon, hiking, and partying. He fell in love, broke up, got married, had a baby, founded a global mailing list, had fun working on several research projects, irritated his advisors, published a few papers, and earned a Ph.D. from the University of Washington in 2002. All of his hard-earned degrees are in Computer Science.

P.S. In Fall 2002, he moved east to become a professor at the University of Illinois, Urbana-Champaign.