Reconciling Schemas of Disparate Data Sources: A Machine-Learning Approach

AnHai Doan, Pedro Domingos, Alon Halevy
{anhai, pedrod, alon}@cs.washington.edu
Department of Computer Science and Engineering
University of Washington, Seattle, WA 98195

ABSTRACT

A data-integration system provides access to a multitude of data sources through a single mediated schema. A key bottleneck in building such systems has been the laborious manual construction of semantic mappings between the source schemas and the mediated schema. We describe LSD, a system that employs and extends current machine-learning techniques to semi-automatically find such mappings. LSD first asks the user to provide the semantic mappings for a small set of data sources, then uses these mappings together with the sources to train a set of learners. Each learner exploits a different type of information either in the source schemas or in their data. Once the learners have been trained, LSD finds semantic mappings for a new data source by applying the learners, then combining their predictions using a meta-learner. To further improve matching accuracy, we extend machine learning techniques so that LSD can incorporate domain constraints as an additional source of knowledge, and develop a novel learner that utilizes the structural information in XML documents. Our approach thus is distinguished in that it incorporates multiple types of knowledge. Importantly, its architecture is extensible to additional learners that may exploit new kinds of information. We describe a set of experiments on several real-world domains, and show that LSD proposes semantic mappings with a high degree of accuracy.

1. INTRODUCTION

The increasing need of enterprises to uniformly access multiple sources of data and the rapid growth of structured data available on the WWW have spurred considerable interest in building data-integration systems (e.g., [8, 9, 24, 15, 11, 13]). Such a system provides users with a uniform interface to a multitude of data sources, thus freeing them from the details of the schemas of the sources and the particular mode of interaction with each source. 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 data-integration system uses a set of *semantic mappings* between the mediated schema and the local schemas of the data sources. The system uses the mappings to reformulate a user query into a set of queries on the data sources. Wrapper programs, attached to each data source, handle the data formatting transformations between the local data model and the data model in the integration system. Figure 1 illustrates a data-integration system that helps users find houses on the real-estate market.

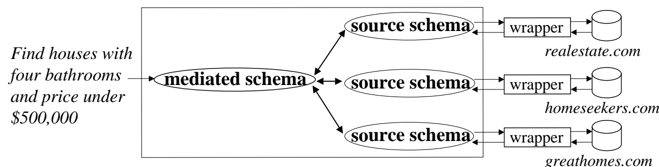


Figure 1: A data-integration system in the real-estate domain.

A key bottleneck in building data-integration systems is the acquisition of semantic mappings. Today these mappings are provided manually by the builders of the system, resulting in a laborious and error-prone process. The emergence of XML as a standard *syntax* for sharing data among sources further fuels data sharing applications, and hence underscores the need to develop methods for acquiring semantic mappings. Clearly, while the task of finding semantic mappings cannot be fully automated, the development of tools for assisting the process is crucial to truly achieve large-scale data integration.

In this paper we describe the LSD (Learning Source Descriptions) system that uses and extends machine learning techniques to semi-automatically create semantic mappings. Throughout the discussion, we shall assume that the sources present their data in XML, and that the mediated- and source schemas are represented with DTDs. Then, the schema-matching problem is to find correspondences among the elements of the mediated schema and the source DTDs.

The key idea underlying our approach is that after a small set of data sources have been manually mapped to the mediated schema, LSD should be able to glean significant information from these mappings to successfully propose mappings for subsequent data sources.

Example 1. Consider the data-integration system in Figure 1. To apply LSD, first we select a source, say *realestate.com*. Next, we manually specify the mappings between the schema of this source and the mediated schema. In particular, suppose the mappings specify that source-schema elements listed-price, phone, and comments match mediated-schema elements PRICE, AGENT-PHONE, and DESCRIPTION, respectively (see the dotted arrows in Figure 2.a).

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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ACM SIGMOD 2001 May 21-24, Santa Barbara, California, USA
Copyright 2001 ACM 1-58113-332-4/01/05 ...$5.00.

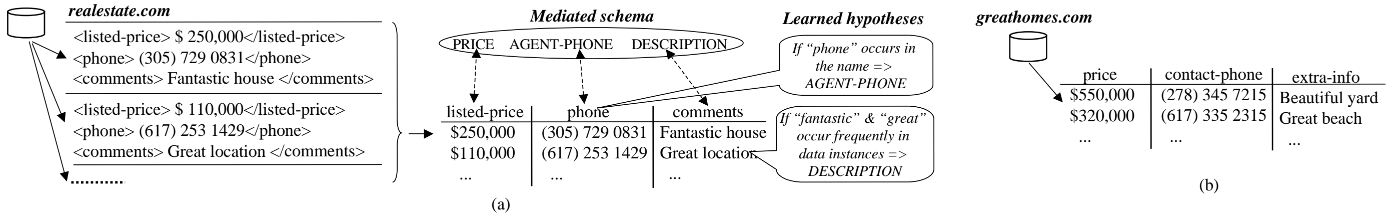


Figure 2: Once LSD has trained a set of learners on source *realestate.com* in (a), it can apply the learners to find semantic mappings for source *greathomes.com* in (b).

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 *greathomes.com*. First, LSD extracts data from this source to populate a table where each column consists of data instances for a single element of the source schema (Figure 2.b). Next, LSD applies the learners to each column. A learner examines the name and the data instances of the column, and applies its learned hypothesis to predict the matching mediated-schema element. For example, when applied to column extra-info, a word-frequency learner will recognize that the data instances in the column 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. First, we must 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 [17]: we employ a multitude of learners, called *base learners*, then combine the learners' predictions using a *meta-learner*. The meta-learner uses the training data to learn for each base learner a set of weights that indicate the relative importance of that learner. The weights can be different for each mediated-schema element, reflecting that different learners may be most appropriate in different cases. 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 arose from the hierarchical nature of XML files and DTDs. In many cases, a decision on how to match an XML element depends crucially on its structure: the number and types of elements it contains, the positions of these elements, the distribution of text within these elements, and so on. However, the problem of considering hierarchical structure has received relatively little attention in the machine learning literature. We develop a novel learner, called the *XML learner*, that handles hierarchical structure and further improves the accuracy of our mappings.

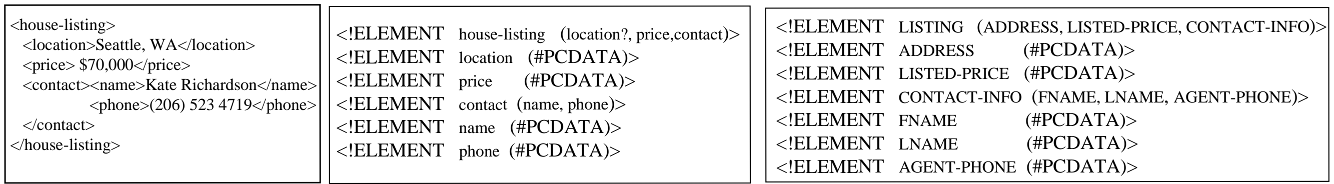
In the rest of the paper we describe the LSD system and the experiments we conducted to validate it. Specifically, the paper makes the following contributions:

- We describe the use of multi-strategy learning for finding semantic mappings. The LSD system, embodying this approach, is able to learn from both schema- and data-related features. The system is easily extensible to additional learners, and requires little initial effort from the user – as the user gets to know the domain better, new learners can be added as needed.
- We extend machine learning techniques to incorporate integrity constraints and user feedback, in order to improve the accuracy of the mappings.
- We develop the XML learner, a novel learning algorithm that classifies structured elements.
- We describe a set of experiments on several data integration domains to validate the effectiveness of LSD. 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 adding new learners and how the system profits from learning from schema- and data-related features, and user feedback.

The paper is organized as follows. The next section defines the schema-matching problem. Sections 3–5 describe the LSD sytem. Section 6 presents our experiments. Section 7 discusses the limitations of the current system. Section 8 reviews related work. Section 9 discusses future work and concludes.

2. PROBLEM DEFINITION

The goal of schema matching is to produce semantic mappings that enable transforming data instances from one schema



(a) An XML house listing in a source S (b) The schema of source S (c) The mediated schema

Figure 3: Examples of XML listing, source schema (DTD), and mediated schema.

to instances of the other. In many common cases, the mappings are *one-to-one (1-1)* between elements in the two schemas (e.g., "location is mapped to address"), while in others, the mappings may be more complex (e.g., "num-baths maps to half-baths + full-baths"). In general, a mapping may be specified as a query that transforms instances of one schema into instances of the other [18, 22]. The focus of our work is to compute 1-1 mappings between the elements of the two schemas. Computing more complex mappings is the subject of ongoing research.

It is important to emphasize that the input to the schema matching problem is already *structured* data. In many data-integration applications, it is necessary to precede schema matching by a *data extraction* phase. In this phase, unstructured data (e.g., text, HTML) is mapped to some structured form (e.g., tuples, XML). Data extraction has been the focus of intensive research on wrappers, information extraction, and segmentation (e.g., [14, 7]), often also benefiting from machine learning techniques.

2.1 Schema Matching for XML DTDs

XML [26] is increasingly being used as a protocol for the dissemination and exchange of information from data sources. Hence, we decided to consider the problem of discovering semantic mappings in the context of XML data. Recall that in addition to encoding relational data, XML can encode object-oriented, hierarchical, and semi-structured data. 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. A document contains a unique root element, in which all others are nested. Figure 3.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 paper we treat its attributes and sub-elements in the same fashion.

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. We assume that the mediated schema is a DTD over which users pose queries, and that each data source 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. Figures 3.b-c show sample source- and mediated-DTDs in the real-estate domain. Throughout the paper, we shall use this font to denote source-schema elements, and THIS FONT for mediated-schema elements.

Given a mediated DTD and a source DTD, the *schema-matching problem* is to find semantic mappings between the two DTDs. In this paper we start by considering the restricted case of finding *one-to-one (1-1) mappings* between tag names of the source DTD and those of the mediated DTD. For example, in Figures 3.b-c, tag location matches ADDRESS, and contact matches CONTACT-INFO. Intuitively, two tags (i.e., schema elements) match if they refer to semantically equivalent concepts. The notion of semantic equivalence is subjective and heavily dependent on the particular domain and context for which data integration is performed. However, as long as this notion is interpreted by the user *consistently* across all data sources in the domain, the equivalence relations LSD learns from the training sources provided by the user should still apply to subsequent, unseen sources.

2.2 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* $\{(x_1, c_{i1}), \ldots, (x_m, c_{im})\}$, where each x_j is an object and c_{ij} is the observed label of that object. During the *training phase*, the learner inspects the training examples and builds an *internal classification model* (e.g., the hypotheses in Figure 2.a) on how to classify objects.

In the *matching phase*, given an object x the learner L uses its internal classification model to predict label for x. In this paper we assume the prediction is of the form $\langle s(c_1|x,L), s(c_2|x,L), \ldots, s(c_n|x,L)\rangle$, where $\sum_{j=1}^{n} s(c_j|x,L) = 1$, and $s(c_j|x,L)$ is learner L's *confidence score* that x matches label c_j. 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.)

For example, consider the *name matcher* which assigns label to an XML element based on its *name* (see Section 3.3 for more details). Given an XML element, such as "⟨phone⟩ (235) 143 2726⟨/phone⟩", the name matcher inspects the name, which is "phone", and may issue a prediction such as ⟨ADDRESS:0.1,DESCRIPTION:0.2, AGENT-PHONE:0.7⟩.

3. THE LSD APPROACH

We now describe LSD in detail. The system consists of four major components: *base learners*, *meta-learner*, *prediction converter*, and *constraint handler*. It operates in two phases: training and matching (Figure 4). 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. Finally, it trains the meta-learner. The output of the training phase

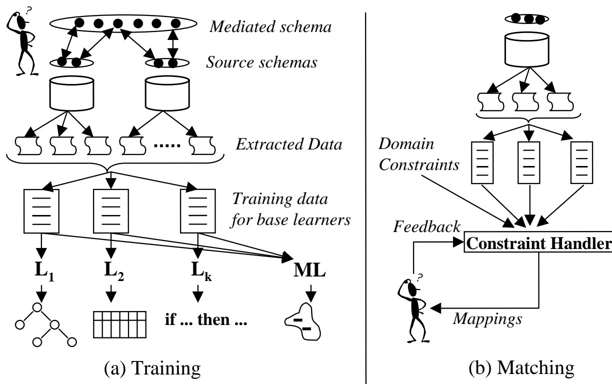


Figure 4: The two phases of LSD.

is the internal classification models of the base-learner and the meta-learner.

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 elements that belong to it. Second, it applies the base learners to the XML elements in the column, then combines the learners' predictions using the meta-learner and the prediction converter. 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 base learners, the meta-learner, and the prediction converter. The next section (Section 4) describes the constraint handler. Then Section 5 describes the *XML learner*, a novel base learner we developed to handle nested DTD elements.

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 5.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 5.b, which says that location matches ADDRESS, comments matches DESCRIPTION, and so on.

The specification should be a relatively easy task, because it involves labeling only the *schemas*, not the *data instances* of the sources. It is done only once, at the beginning of the training phase; thus 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 5.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, 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 matcher and the Naive Bayes learner (both are described in detail in Section 3.3). The name matcher matches an XML element based on its *tag name*. Therefore, for each of the 12 extracted XML elements (Figure 5.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 5.d lists the 12 training examples for the name matcher. Some training examples are duplicates, but that is fine because most learners, including the name matcher, 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 5.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 4.a.

5. Train the Meta-Learner: Finally, LSD trains the meta-learner. This learner uses a technique called *stacking* [25, 23] to combine the predictions of the base learners. Training the meta-learner proceeds as follows [23]. 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 judgement, it then assigns to each combination of label c_i and base learner L_j a weight $W^{c_i}_{L_j}$ 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 3.2 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

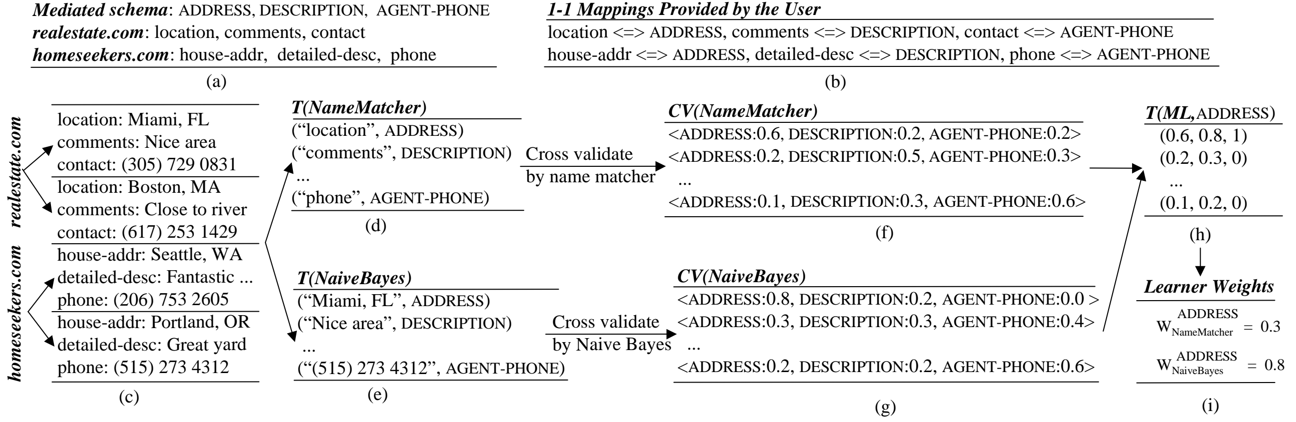


Figure 5: An example of creating training data for the base learners and the meta-learner.

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 5.f shows the set CV for the name matcher. Here, the first line is the prediction made by the name matcher for the first training example in $T(NameMatcher)$, which is ("location",ADDRESS) in Figure 5.d. The second line is the prediction for the second training example, and so on. Figure 5.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 5.c: "location: Miami, FL". The name matcher predicts that it matches ADDRESS with score 0.6 (see the first tuple of Figure 5.f). The Naive Bayes learner predicts that it matches ADDRESS with score 0.8 (see the first tuple of Figure 5.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 5.h.

(c) Perform Regression to Compute Learner Weights: Finally, for each label c_i, the meta-learner computes the learner weights $W_{L_j}^{c_i}$, $j \in [1, k]$, 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 $\sum_j \left(l(c_i, x_j) - \sum_t s(c_i|x_j, L_t) * W_{L_t}^{c_i}\right)^2$, 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 probability that an instance matches c_i when it does and a low probability 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 $W_{NameMatcher}^{ADDRESS} = 0.3$ and $W_{NaiveBayes}^{ADDRESS} = 0.8$ (Figure 5.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 matcher in predicting label ADDRESS.

3.2 The Matching Phase

Once the learners have been trained, LSD is ready to predict semantic mappings for new sources. Figure 6 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 6.a). Next, for each source-DTD tag, LSD collects all the instances of elements with that tag from the listings. Figures 6.b and 6.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 6.b). To match this instance, LSD applies the base learners, then combine their predictions using the meta-learner.

The name matcher will take the instance *name*, which is "area", and issue the prediction:
⟨ADDRESS:0.5, DESCRIPTION:0.3, AGENT-PHONE:0.2⟩

The Naive Bayes learner will take the instance *content*, which is "Orlando, FL", and issue another prediction:
⟨ADDRESS:0.7, DESCRIPTION:0.3, AGENT-PHONE:0.0⟩

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 $W_{NameMatcher}^{ADDRESS} = 0.3$ and $W_{NaiveBayes}^{ADDRESS} = 0.8$, 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,

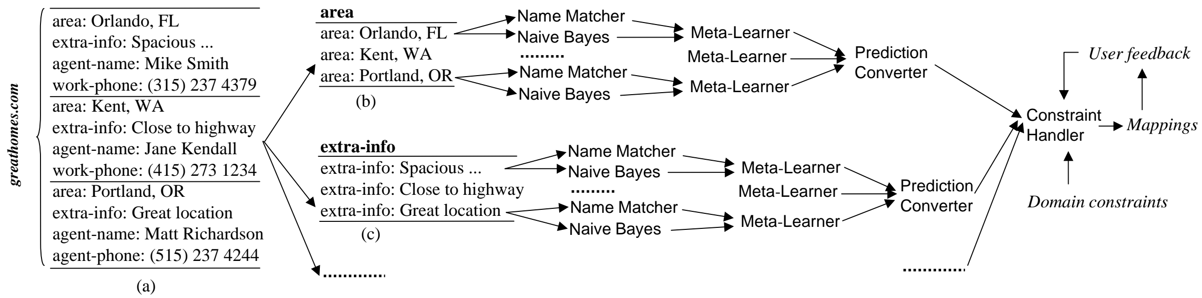


Figure 6: Matching the schema of source *greathomes.com*.

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 matcher 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 converter then combines the three predictions of the three data instances into a single prediction for area. Currently, the prediction converter 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 converter 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 4 describes the constraint handler, together with domain constraints and user feedback.

3.3 The Base Learners

LSD uses the following base learners (Section 5 describes the XML base learner).

The Name Matcher matches an XML element using its tag name (expanded with synonyms and all tag names leading to this element from the root element). It uses Whirl, the nearest-neighbor classification model developed by Cohen and Hirsh [4]. The name matcher stores all training examples of the form (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 distance from t. The similarity distance between any two examples is the TF/IDF distance (commonly employed in information retrieval) between the *tag names* of the examples. See [4] for more details.

This 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).

The Content Matcher also uses Whirl. However, this learner matches an XML element using its *data content*, instead of its *tag name* as with the name matcher. 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.

The Naive Bayes Learner is one of the most popular and effective text classifiers [5]. 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. Let $d = \{w_1, \ldots, w_k\}$ be an input instance, where the w_j are tokens. Then the Naive Bayes learner assigns d to the class c_i that maximizes $P(c_i|d)$. This is equivalent to finding c_i that maximizes $P(d|c_i)P(c_i)$. $P(c_i)$ is approximated as the portion of training instances with label c_i. Assuming that the tokens w_j appear in d *independently* of each other given c_i, we can compute $P(d|c_i)$ as $P(w_1|c_i)P(w_2|c_i)\cdots P(w_k|c_i)$, where $P(w_j|c_i)$ is estimated as $n(w_j, c_i)/n(c_i)$. $n(c_i)$ 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 (for an explanation, see [5]).

The Naive Bayes learner works best when there are tokens that are strongly indicative of the correct label, by virtue of their frequencies (e.g., "beautiful" and "great" in house descriptions). 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 and zip code.

The County-Name Recognizer 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.

4. EXPLOITING DOMAIN CONSTRAINTS

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.

4.1 Domain 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

Constraint Types		Examples	Can Be Verified With
Hard	*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**, **b**, and **c** match CITY, FIRM-NAME, and FIRM-ADDRESS, resp., then **a** & **b** functionally determine **c**.	Schema + data from target source
Soft	*Binary*	Number of elements that match DESCRIPTION is not more than 3.	”
	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

Table 1: Types of domain constraints. Variables a, b and c refer to source-schema elements.

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 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 for any source S in the domain, given a *candidate mapping* 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 are those that the user think absolutely cannot be violated. Let $T_{hard} = \{t_1, \ldots, t_u\}$ be the set of hard constraints. Then we define $cost(m, T_{hard})$ to be 0 if m satisfies $t_1 \wedge t_2 \wedge \cdots \wedge t_u$, and ∞ otherwise.

Table 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 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 are those 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 1 shows examples of binary and numeric soft constraints.

4.2 The Constraint Handler

The constraint handler takes the domain constraints, together with the predictions produced by the prediction converter, and outputs the 1-1 mappings. Conceptually, it searches through the space of possible candidate mappings, to find the one with the lowest cost, where cost is defined based on the likelihood of the mapping and the degree to which the mapping satisfies the domain constraints. LSD uses the A* algorithm to search this space [10]. Our A* implementation uses a domain-independent heuristic (see Section 6.3) to direct the search and find the best candidate mapping.

Specifically, let $e_1, \ldots, e_q$ be the DTD tags of the source schema, and $c_1, \ldots, c_n$ be the class labels. We denote a *candidate mapping* 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 m is defined as $cost(m) = \sum_{i=1}^{v} \alpha_i * cost(m, T_i) - \beta * log\ prob(m)$, where $cost(m, T_i)$ represents the degree to which m satisfies domain constraints of type T_i, and $\alpha_1, \ldots, \alpha_v, \beta$ are the scaling coefficients that represent the trade-offs among the cost components. The term $prob(m)$ denotes the probability of candidate mapping m, and is approximated as $\prod_{j=1}^{q} s(c_{ij}|e_j, PC)$, where $s(c_{ij}|e_j, PC)$ is the confidence score that source-DTD element e_j matches label c_{ij}, returned by the prediction converter PC. The formula for $prob(m)$ 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.

The definition of $cost(m)$ implies that we prefer the candidate mapping with the highest probability, all other things being equal.

4.3 User Feedback

User feedback can further improve matching accuracy, and is necessary in order to match ambiguous schema elements. 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 greatly aid the system by manually matching a few hard-to-match schema elements, as we show empirically in Section 6.3.

5. 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

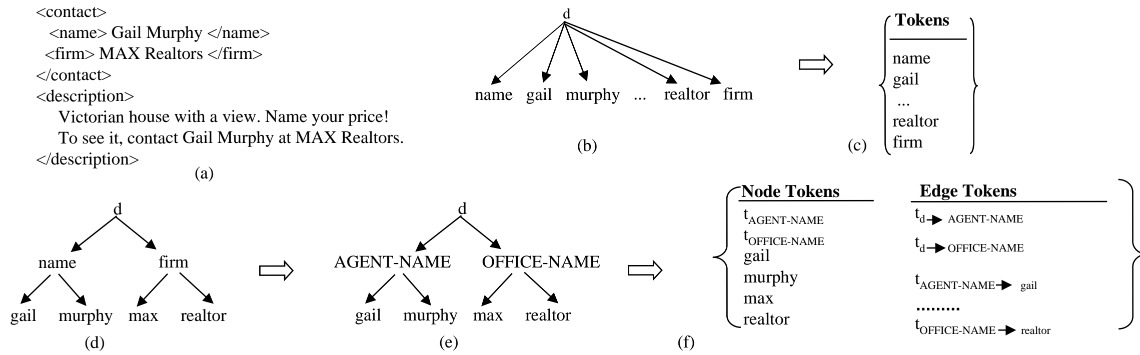


Figure 7: (a)-(c) The working of the Naive Bayes learner on the XML element contact in (a); and (d)-(f) the working of the XML learner on the same element.

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 (see Section 3.3).

Table 2: The XML learner algorithm.

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 7.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 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 2 for pseudo code). Consider the XML element contact in Figure 7.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 7.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 7.c. Finally, it multiplies the token probabilities to find the best label for the element (as discussed in Section 3.3).

In contrast, the XML learner first creates the tree in Figure 7.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 7.e shows the modified tree. Then the XML classifier generates the set of tokens shown in Figure 7.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 $t_{l_1 \rightarrow l_2}$. Finally, the classifier 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 7.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").

6. 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. 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 Listing 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

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: Domains and data sources for our experiments.

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 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. Currently we specified only hard constraints. For each mediated-schema tag, we specified all non-trivial column and frequency constraints that we can find. For each pair of mediated-schema tags, we specified all applicable nesting constraint. Finally, we specified all contiguity and exclusivity constraints that we think should apply to the vast majority of sources. See Table 1 for examples of hard constraints of different types. In general, most constraints we used are frequency, nesting, or column constraints. We specified very few contiguity and exclusivity constraints.

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.

6.1 Matching Accuracy

Figure 8.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 high accuracy across all four domains, ranging from 71 to 92%. In contrast, the best matching results of the base learners (achieved by either the Naive Bayes or the Name Matcher, depending on the domain) are only 42 - 72%.

As expected, adding the meta-learner improves accuracy substantially, by 5 - 22%. Adding the domain-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 7 we identify the reasons that prevents LSD from correctly matching the remaining 10 - 30% of the tags.

Performance Sensitivity: Figures 8.b-c 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.

6.2 Lesion Studies

Figure 9.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 8.a). The fifth bar represents the accuracy of the complete LSD system, for comparison purpose. The results show that each component contributes to the overall performance, and there appears to be no clearly

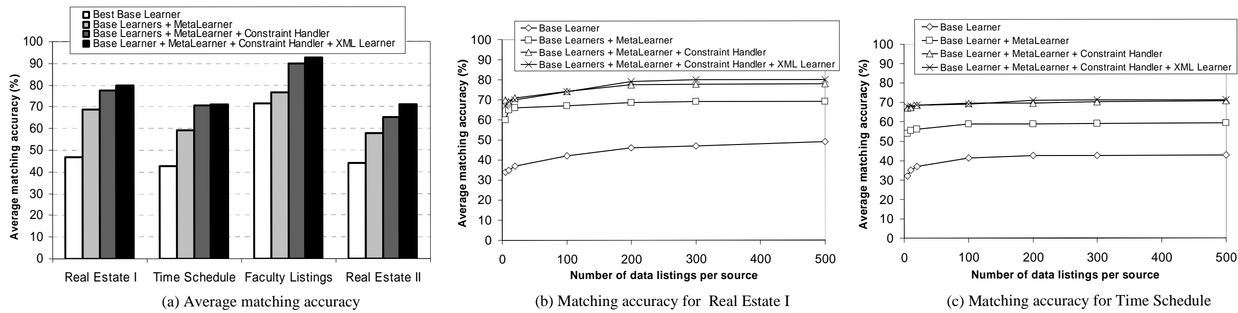


Figure 8: (a) Average matching accuracy; experiments were run with 300 data listings from each source; for sources from which fewer than 300 listings were extracted, all listings were used. (b)-(c) The average domain accuracy as a function of the amount of data available per source.

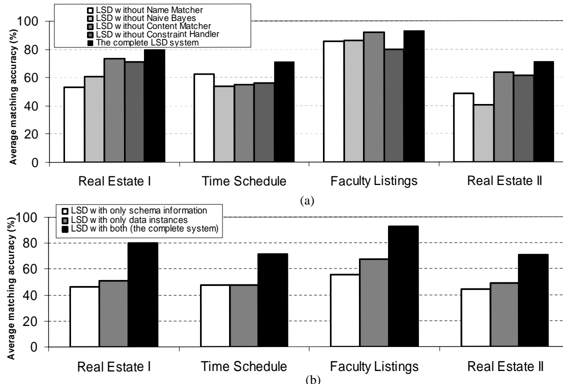


Figure 9: The average matching accuracy of LSD versions (a) with each component being left out versus that of the complete LSD system, (b) with only schema information or data instances versus that of the LSD version with both.

dominant component.

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 9.b, the first bar of each domain shows the average accuracy of the LSD version that consists of the Name Matcher 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.

6.3 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 associate with each tag in the testing source a score that measures the extent to which it 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 testing source in decreasing order of their scores[1]. 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 judiciously provided by the user in order to achieve perfect or near-perfect matching.

7. DISCUSSION

We now address the limitations of the current LSD system. 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

[1]This is the same order used by our A* implementation to refine states, that is, to direct its search through the space of matching combinations.

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 descendent 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 on 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.

8. RELATED WORK

We describe work related to LSD from several perspectives.

Schema Matching: Work on schema matching can be classified into rule- and learner-based approaches. (For a comprehensive survey on schema matching, see [22].) Rule-based approach includes [19, 20, 2]. The Transcm system [19] performs matching based on the *name* and *structure* of schema elements. The Artemis system [2] uses names, structures, as well as domain types of schema elements to match schemas. In general, rule-based systems utilize only schema information in a hard-coded fashion, whereas our approach exploits both schema and data information, and does so automatically, in an extensible fashion.

In the learner-based approach, the Semint system [16] uses a neural-network learner. It matches schema elements using properties such as field specifications (e.g., data types and scale) and statistics of data content (e.g., maximum, minimum, and average). Unlike LSD, it does not exploit other types of data information such as word frequencies and field formats. The ILA system [21] matches schemas of two sources based on comparing objects that it knows to be the same in both sources. Both Semint and ILA employ a single type of learner, and therefore have limited applicability. For example, the neural net of Semint does not deal well with textual information, and in many domains it is not possible to find overlaps in the sources, making ILA's method inapplicable.

Clifton et al. [3] describe DELTA, which associates with each attribute a text string that consists of all meta-data on the attribute, then matches attributes based on the similarity of the text strings. The authors also describe a case study using DELTA and Semint and note the complimentary nature of the two methods. With LSD, both Semint and DELTA could be plugged in as new base learners, and their predictions would be combined by the meta-learner.

The Clio system [18] introduces *value correspondences*, which specify functional relationships among related elements (e.g., hotel-tax = room-rate $*$ state-tax-rate). Given a set of such correspondences, Clio produces the SQL queries that translate data from one source to the other. A key challenge in creating the queries is to find semantically meaningful ways to relate the data elements in a source, in order to produce data for the other source. For example, given the above value correspondence, Clio must discover that room-rate belongs to a hotel that is located in the state where the tax rate is state-tax-rate. To this end, Clio explores joining elements along foreign-key relationship paths. There can be many foreign keys, thus many possible ways to join. Hence, the problem boils down to searching for most likely join path. Clio uses several heuristics and user feedback to arrive at the best join path, and thus the best query candidate. Clio is therefore complimentary to the current work of ours. It can take the mappings produced by LSD as part of its input.

Combining Multiple Learners: Multi-strategy learning has been researched extensively [17], and applied to several other domains (e.g., information extraction [7], solving crossword puzzles [12]). 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, the use of integrity constraints to further refine the learner, and the XML learner that exploits the structure of XML documents. Yi and Sundaresan [27] 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.

Exploiting Domain Constraints: Incorporating domain constraints into the learners has been considered in several works (e.g., [6]), 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.

9. CONCLUSIONS & FUTURE WORK

We have described an approach to schema matching that employs and extends machine learning techniques. Our approach utilizes both schema and data from the sources. To match a source-schema element, 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 hierar-

chical structure in XML data to improve matching accuracy. Our experiments show that we can accurately match 71-92% of the tags on several domains.

More broadly, we believe that our approach contributes an important aspect to the development of schema-matching solutions. Given that schema matching is a fundamental step in numerous data management applications [22], it is desirable to develop a generic solution that is robust and applicable across domains. In order to be robust, such a solution must have several important properties. First, it must *improve over time* – knowledge gleaned from previous instances of the schema-matching problem should contribute to solving subsequent instances. Second, it must allow knowledge to be incorporated in an *incremental* fashion, so that as the user gets to know the domain better, knowledge can be easily modified or added. Third, it must allow *multiple types* of knowledge to be used to maximize the matching accuracy. Machine learning techniques, and in particular, multi-strategy learning, provide a basis for supporting these properties. Hence, while we do not argue that our techniques provide a complete solution to the schema-matching problem, we do believe that we provide an indispensable component of any robust solution to the problem.

We are extending our current work in two ways. First, we are addressing the limitations of the current system, as outlined in Section 7. And second, we are extending our approach to non 1-1 mappings, and to find translation queries, as discussed in Section 2.

The most up-to-date information on LSD can be found on its website [1]. The site also stores a public repository of data intended to be used as benchmarks in evaluating schema matching algorithms.

Acknowledgment: We thank Phil Bernstein, Oren Etzioni, David Hsu, Geoff Hulten, Zack Ives, Jayant Madhavan, Erhard Rahm, Igor Tatarinov, and the reviewers for invaluable comments. This work is supported by NSF Grants 9523649, 9983932, IIS-9978567, and IIS-9985114. The second author is also supported by an IBM Faculty Partnership Award, and the third author is supported by a Sloan Fellowship and gifts from Microsoft Research, NEC and NTT.

10. REFERENCES

[1] LSD's website: cs.washington.edu/homes/anhai/lsd.html.
[2] S. Castano and V. D. Antonellis. A schema analysis and reconciliation tool environment for heterogeneous databases. In *Proc. of the Int. Database Engineering and Applications Symposium (IDEAS-99)*, pages 53–62.
[3] 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.
[4] 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.
[5] P. Domingos and M. Pazzani. On the optimality of the simple Bayesian classifier under zero-one loss. *Machine Learning*, 29:103–130, 1997.
[6] S. Donoho and L. Rendell. Constructive induction using fragmentary knowledge. In *Proc. of the 13th Int. Conf. on Machine Learning*, pages 113–121, 1996.
[7] D. Freitag. Machine learning for information extraction in informal domains. *Ph.D. Thesis*, 1998. Dept. of Computer Science, Carnegie Mellon University.
[8] 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 Information Systems*, 8(2):117–132, 1997.
[9] L. Haas, D. Kossmann, E. Wimmers, and J. Yang. Optimizing queries across diverse data sources. In *Proc. of VLDB*, 1997.
[10] 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.
[11] Z. Ives, D. Florescu, M. Friedman, A. Levy, and D. Weld. An adaptive query execution system for data integration. In *Proc. of SIGMOD*, 1999.
[12] 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.
[13] 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.
[14] N. Kushmerick. Wrapper induction: Efficiency and expressiveness. *Artificial Intelligence*, 118(1–2):15–68, 2000.
[15] A. Y. Levy, A. Rajaraman, and J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proc. of VLDB*, 1996.
[16] 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.
[17] R. Michalski and G. Tecuci, editors. *Machine Learning: A Multistrategy Approach.* Morgan Kaufmann, 1994.
[18] R. Miller, L. Haas, and M. Hernandez. Schema mapping as query discovery. In *Proc. of VLDB*, 2000.
[19] T. Milo and S. Zohar. Using schema matching to simplify heterogeneous data translation. In *Proc. of VLDB*, 1998.
[20] 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.
[21] M. Perkowitz and O. Etzioni. Category translation: Learning to understand information on the Internet. In *Proc. of Int. Joint Conf. on AI (IJCAI)*, 1995.
[22] E. Rahm and P. Bernstein. On matching schemas automatically. *Tech. report MSR-TR-2001-17*, 2001. Microsoft Research, Redmon, WA.
[23] K. M. Ting and I. H. Witten. Issues in stacked generalization. *Journal of Artificial Intelligence Research*, 10:271–289, 1999.
[24] A. Tomasic, L. Raschid, and P. Valduriez. Scaling access to distributed heterogeneous data sources with Disco. *IEEE Transactions On Knowledge and Data Engineering*, 1998.
[25] D. Wolpert. Stacked generalization. *Neural Networks*, 5:241–259, 1992.
[26] Extensible markup language (XML) 1.0. www.w3.org/TR/1998/REC-xml-19980210. W3C Recommendation.
[27] 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.