Mining Structures for Semantics

Xin Dong
University of Washington
lunadong@cs.washington.edu

Jayant Madhavan
University of Washington
jayant@cs.washington.edu

Alon Halevy
University of Washington
alon@cs.washington.edu

1. INTRODUCTION

Online data is available in two flavors: *unstructured* data that resides as free text in HTML pages, and *structured* data that resides in databases and knowledge bases. Unstructured data is easily accessed as human-readable text on a browser, while structured data is hidden behind web query interfaces (web forms), web services, and custom database APIs. Access to this data, popularly referred to as the *hidden web*, entails submitting correctly completed web forms or writing code to access web services using protocols such as SOAP.

The emergence of powerful search engines has greatly improved our ability to search for data on the web; however, such access is still primarily restricted to unstructured data. We can search for and access information available as HTML, but are not yet able to gain easy access to the hidden web. It is not easy to get to the correct web form, and even harder to find a suitable web service. When we do find the correct web form or web service, there is an additional step of understanding its schema, and reformulating the user's query to fit that schema. While humans do this regularly, *one form at a time*, it is difficult to automate the process of query reformulation, and therefore we cannot leverage the wealth of information residing behind web forms and services.

We observe that one reason for the success of today's search engines is their ability to apply statistics computed from large collections of HTML documents to rank their search results. For example, word-occurrence probabilities in the body and anchor text of HTML pages are used to identify words most relevant to individual pages. Such analysis of a corpus of documents has also been used successfully for other tasks such as classification and clustering of text documents. Given that we now have a vast collection of web forms and services, the natural question that arises is whether we can leverage a corpus of these in order to automate the process of query reformulation.

The techniques for exploiting corpora of documents do not apply directly to searching structured data. The main reason is that searching structured data requires understanding the underlying semantics of the data sources. This structure is mostly (but not completely) specified by the schema. However, in specifying these semantics, the actual words used and the information organization depend more on the developer's whim, and little variations may account for very different semantics.

We are pursuing a project whose goal is to show that large corpora of structures (i.e., web forms and services, database schemata) *can* be used to address the fundamental difficulty of bridging semantic heterogeneity. This paper briefly reviews two recent developments in this project, and compares between them. Specifically, we describe the following:

- *Searching for web services:* We describe *Woogle*[1] [9], an intelligent web service search engine. Among other things, Woogle exploits parameter naming statistics in a large collection of web service descriptions (WSDL files) to accurately search for the web service operations that best suit a user's requirements.
- *Schema Matching:* We describe *corpus-based schema matching* [14], a schema matching framework that uses collections of known schemas and mappings to better match (identify corresponding elements) new unseen schemas.

While both of these works exploit a corpus of structures, the problem settings are such that different techniques are required. Section 2 describes the Woogle web service search engine, and Section 3 describes corpus-based schema matching. Section 4 addresses the similarities and differences between the two approaches and describes related work and future directions in this project.

2. SEARCHING FOR WEB SERVICES

Web services are loosely coupled software components, published, located, and invoked across the web. A web service comprises of several operations (see examples in Figure 1). Each operation takes a SOAP package containing a list of input parameters, fulfills a certain task, and returns the result in an output SOAP package. Each web service has an associated WSDL file describing its functionality and interface. A web service is typically (though not necessarily) published by registering its WSDL file and a brief description in UDDI business registries.

The growing number of web services available within an organization and on the Web raises a new and challenging search problem: locating desired web services. In fact, to address this problem, several simple search engines have recently sprung up (e.g., [1]). Currently, these engines provide only simple keyword search: return services that contain the words in the web service descriptions (obtained from the WSDL file). However, the keyword search paradigm is

[1] http://www.cs.washington.edu/woogle

W_1: Web Service: GlobalWeather
Operation: GetTemperature
Input: Zip Output: Return
W_2: Web Service: WeatherFetcher
Operation: GetWeather
Input: PostCode
Output: TemperatureF, WindChill, Humidity
W_3: Web Service: GetLocalTime
Operation: LocalTimeByZipCode
Input: Zipcode
Output: LocalTimeByZipCodeResult
W_4: Web Service: PlaceLookup
Operation1: CityStateToZipCode
Input: City, State Output: ZipCode
Operation2: ZipCodeToCityState
Input: ZipCode Output: City, State

Figure 1: Several example web services (not including their textual descriptions). Note that each web service includes a set of operations, each with input and output parameters. For example, web services W_1 and W_2 provide weather information.

insufficient for two reasons. First, keywords do not capture the underlying semantics of web services. For example, when searching zipcode, the web services whose descriptions contain term zip or postal code but not zipcode will not be returned. Further, keyword search on web services does not suffice for accurately specifying users' information needs: users are typically looking for a specific operation with some specific input or output parameters.

To address the challenges involved in searching for web services, we built Woogle, a web-service search engine. In addition to simple keyword searches, Woogle supports similarity search for web services. Starting with a keyword search, a user can drill down to a particular web service operation. When not satisfied, a user can query for web-service operations similar to a given one, those that take similar inputs (or outputs), and those that compose with a given one. These search primitives greatly reduce the tedium involved in current web service search where a user might have to conduct multiple search sessions repeatedly modifying the search keywords until finding the most suitable web service operation. For example, the operation GetWeather in W_2 is similar to GetTemperature in W_1, and thus may serve as an alternative; while they provide weather information, LocalTimeByZipCode in W_3 provides other information about locations, and thereby may be of interest to the user; finally, composing CityStateToZipCode in W_4 with GetWeather in W_1 offers a solution for getting the weather when the zipcode is not known.

Overview of our approach: Similarity search for web services is challenging because neither the textual descriptions of web services and their operations nor the names of the input and output parameters completely convey the underlying semantics of the operation. Nevertheless, knowledge of the semantics is important in determining similarity between operations. Broadly speaking, our algorithm combines multiple sources of evidence to determine similarity. In particular, we consider similarity between the textual descriptions of the operations and of the entire web services, and similarity between the parameter names of the operations. The key ingredient of the algorithm is a novel technique that clusters parameter names found in the collection of web services into *semantically meaningful* concepts. By comparing the concepts to which input or output parameters belong, we are able to obtain much better search results. We thus demonstrate the ability to use a corpus to better search for web services on the Web.

In the rest of this section we outline our similarity search algorithm with particular emphasis on the clustering of parameter names. We also present results that demonstrate the effectiveness of our approach. For more details on this work, the reader is referred to [9].

2.1 Clustering Parameters into Concepts

To effectively compare the inputs and outputs of web-service operations, it is crucial to get at their underlying semantics. However, this is hard for two reasons. First, parameter naming is dependent on the developers' whim. Parameter names tend to be highly varied given the use of synonyms, hypernyms, and different naming rules. They might even not be composed of proper English words–there may be misspellings, abbreviations, etc. Therefore, lexical references, such as *Wordnet* [2], are hard to apply. Second, inputs/outputs typically have few parameters, and the associated WSDL files rarely provide rich descriptions for parameters. Traditional IR techniques, such as TF/IDF [19] and LSI [5], rely on word frequencies to capture the underlying semantics and thus do not apply well.

A parameter name is typically a sequence of concatenated words, with the first letter of every word capitalized (e.g. LocalTimeByZipCodeResult). Such words are referred to as *terms*. We exploit the co-occurrence of terms in web service inputs and outputs to cluster terms into meaningful concepts. As we shall see later, using these concepts, besides the original terms, greatly improves our ability to identify similar inputs/outputs and hence find similar web service operations.

Applying an off-the-shelf text clustering algorithm directly to our context does not perform well because the web service inputs/outputs are sparse. For example, whereas synonyms tend to occur in the same document in an IR application, they seldom occur in the same operation input/output; therefore, they will not get clustered. Our clustering algorithm is a refinement of *agglomerative* clustering [12]. We briefly outline the clustering algorithm below.

Clustering by Co-occurence: We base our clustering on the following heuristic: *parameters tend to express the same concept if they occur together often.* This heuristic is validated by our experimental results. We use this intuition to cluster parameters by exploiting their conditional probabilities of co-occurrence in inputs and outputs of web-service operations. We say that a term t_1 is *closely associated* with term t_2, if the association rule $t_1 \rightarrow t_2$ has a support and a confidence over the corresponding thresholds. Here the rules are a special case of the more general association rules [3].

As is typical, we are interested in clusters that have high *cohesion* and low *correlation*. The cohesion H_I of a cluster I is the percentage of closely associated term pairs over all term pairs in I. The correlation R_{IJ} between two clusters I and J is the percentage of closely associated cross-cluster term pairs. In order to balance the cohesion and correlation, we try to maximize the *average cohesion-correlation ratio* of the set of clusters $\mathcal{C}$.

$$S_{\mathcal{C}} = \frac{avg(cohesion)}{avg(correlation)} = \frac{(\parallel \mathcal{C} \parallel -1) \times \sum_{I \in \mathcal{C}} H_I}{2 \times \sum_{I,J \in \mathcal{C}, I \neq J} R_{IJ}}$$

Agglomerative Clustering: Our clustering algorithm is a series of refinements over the classical agglomerative clustering. We start with each term being in a cluster of its own. The algorithm proceeds in a greedy fashion. It sorts the association rules in descending order first by the confidence and then by the support. Infrequent rules with less than a minimum support t_s are discarded. At every step, the algorithm chooses the highest ranked rule that has not been considered previously. If the two terms in the rule belong to different clusters, the algorithm merges the clusters. We now motivate and outline two refinements to this basic algorithm.

Increasing cluster cohesion The basic agglomerative algorithm takes the single linkage method; that is, it links two clusters together when any two terms in the two clusters are closely associated. This merge condition is very loose and can easily result in low cohesion of clusters. To illustrate, suppose there is a concept for weather, containing temperature as a term, and a concept for address, containing zip as a term. If, when operations report temperature, they often report the area zipcode as well, then the confidence of rule temperature $\rightarrow$ zip is high. As a result, the basic algorithm will inappropriately combine the weather concept and the address concept.

The cohesion of a cluster is decided by the association of each pair of terms in the cluster. To ensure that we obtain clusters with high cohesion, we merge two clusters only if they satisfy a stricter condition: given a cluster C, a term is called a *kernel* term if it is closely associated with at least half[2] of the remaining terms in C; two clusters are merged only if *all the terms in the merged cluster are kernel terms.*

Splitting and Merging: A greedy algorithm pursues local optimal solutions at each step, but usually cannot obtain the global optimal solution. In parameter clustering, an inappropriate clustering decision at an early stage may prevent later appropriate clustering. Consider the case where there is a cluster for zipcode {zip, code}, formed because of the frequent occurrences of parameter ZipCode. Later we need to decide whether to merge this cluster with another cluster for address {state, city, street}. The term zip is closely associated with state, city and street, but code is not closely associated with them because it also occurs often in other parameters such as TeamCode and ProxyCode, which typically do not co-occur with state, city or street. Consequently, the two clusters cannot merge; the clustering result contrasts with the ideal one: {state, city, street, zip} and {code}

The solution to this problem is to split already-formed clusters so as to obtain a better set of clusters with a higher cohesion/correlation score. When analyzing a pair of candidate clusters, our algorithm considers two options: (a) split each cluster into a *ready-to-merge subset* that contains terms closely associated with terms in the union of the two clusters and a *left-alone subset* of the rest terms, and then merge the two ready-for-merge subsets; (b) leave the two clusters as they are. The option with the best cohesion-correlation ratio is selected.

Clustering Results: The term-level clustering algorithm outlined above still has two problems. First, the cohesion condition is too strict for large clusters, so may prevent closely associated large clusters to merge. Second, early inappropriate merging may prevent later appropriate merging. Although we do splitting, the terms taken off from the original clusters may have already missed the chances to merge with other closely associated terms. We solve the problems by running the clustering algorithm iteratively. After each pass, we replace each term with its corresponding concept, re-collect association rules, and then re-run the clustering algorithm. This process continues when no more clusters can be merged.

We now briefly outline the results of our clustering algorithm. Our dataset, which we will describe in detail in Section 2.3, contains 431 web services and 3148 inputs/outputs. There are a total of 1599 terms. The clustering algorithm converges after the seventh run. It clusters 943 terms into 182 concepts. The rest 656 terms, including 387 infrequent terms (each occurs in at most 3 inputs/outputs) and 54 frequent terms (each occurs in at least 30 of the inputs/outputs) are left unclustered. There are 59 *dense* clusters, each with at least 5 terms. Some of them correspond roughly to the concepts of address, contact, geology, maps, weather, finance, commerce, statistics, and baseball, etc. The overall cohesion is 0.96, correlation is 0.003, and average cohesion for the dense clusters is 0.76. This result clearly indicates a high cohesion within concepts and a low correlation between concepts.

2.2 Computing Operation Similarity

In this section we describe how we compute the similarity of web-service operations. We use the intuition that the similarity between two operations is related to the similarity of their descriptions, that of their input and output parameters, and that of their host web services. The similarity of a pair of inputs (or outputs) is related to the similarity of the parameter names, that of the concepts represented by the parameter names, and that of the operations to which they belong. Here the parameter name similarity compares inputs/outputs on a fine-grained level; while concept similarity compares inputs/outputs on a coarse-grained level.

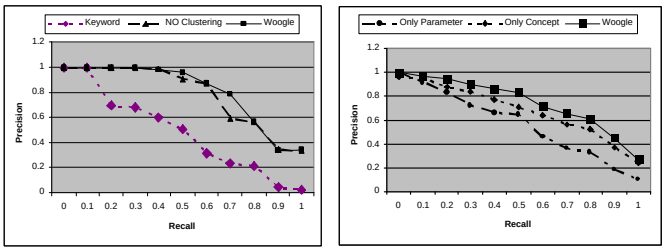
Parameter name similarity: We consider the terms in all parameter names in the input or output of an operation as a bag of words and use the TF/IDF measure [19] to compute the similarity of two such bags.

Input/output concept similarity: To compare the similarity of the concepts represented by the inputs/outputs, we replace each term in the bag of words described above with its corresponding concept, and then use the TF/IDF measure.

Input/output similarity: We compute the similarity of the inputs of two operations as a linear combination of the parameter name similarity, concept similarity and the operation similarity. As we see this similarity is recursively related to the similarity of the operations. We note that careful choice of the weights in the linear combination will ensure a closed-form solution for both the input/output similarity and the operation similarity.

Web-service operation similarity: We compute the similarity of two operations as a linear combination of the similarity of web-service descriptions, the similarity of the operation descriptions, the similarity of the inputs and the similarity of the outputs. The description similarities are computed again with TF/IDF with some simple pre-processing

[2]We tried different values for this fraction and found $\frac{1}{2}$ yielded the best results.



(a) Operation matching (b) Input/output matching

Figure 2: R-P curves for Woogle similarity search.

of the web-service description (obtained from the WSDL and the UDDI registry) and the operation description (obtained from the WSDL).

2.3 Experimental Results

We implemented a web-service search engine, called Woogle, that has access to 790 web services from the main authoritative UDDI repositories. We ran our experiments on the subset of web services whose associated WSDL files are accessible from the web; this set contains 431 web services and 1574 operations in total. Figure 2 plots the Recall-Precision curves for web-service operation matching and input/output matching. It clearly shows that considering parameter clusters can improve the performance for both matching tasks.

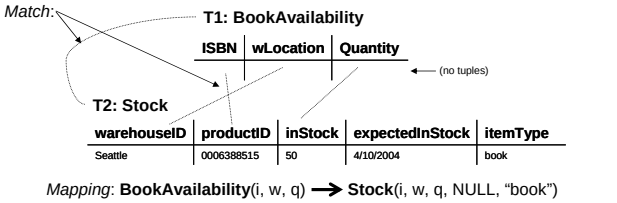
3. MATCHING SCHEMAS

Schema matching is the problem of determining a set of *correspondences* (a.k.a. *matches*) that identify similar elements in different schemas. Schema matching is inherently a difficult task to automate mostly because the exact semantics of the data are only completely understood by the designers of the schema, and not fully captured by the schema itself. In part, this is due to the limited expressive-power of the data model, and often is further hindered by poor database design and documentation. As a result, the process of producing semantic mappings requires a human in the loop and is typically labor-intensive, causing a significant bottleneck in building and maintaining data sharing applications.

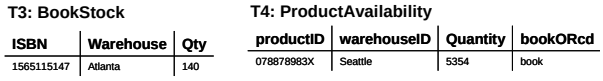
Schema matching has received steady attention in the database and AI communities over the years (see [17] for a recent survey and [16; 4; 6; 8; 10; 11; 15; 23; 21] for work since). A key conclusion from this body of research is that an effective schema matching tool requires a principled combination of several *base techniques*, such as linguistic matching of names of schema elements, detecting overlap in the choice of data types and representation of data values, considering patterns in relationships between elements, and using domain knowledge.

However, current solutions are often very brittle. In part, this is because they only exploit evidence that is present in the two schemas being matched. These schemas often lack sufficient evidence to be able to discover matches. For example, consider table definitions T1 and T2 in Figure 3(a). While both of these tables describe the availability of items, it is almost impossible to find a match by considering them in isolation.

This section describes *corpus-based matching*, an approach that leverages a corpus of schemas and mappings in a particular domain to improve the robustness of schema matching algorithms. A corpus offers a storehouse of *alternative representations* of concepts in the domain. We show how such alternate representations can be used to increase the evidence available in the matched schemas and thereby improve the ability to discover difficult matches. For further details, the reader is referred to [14].



(a) Matches and mappings between BookAvailability and Stock



(b) Other schemas about product availability

Figure 3: Knowledge of schemas T3 and T4 can be used to better match schemas T1 and T2.

Overview of our approach: To illustrate the intuition behind our techniques for exploiting a corpus, suppose that for any element e (e.g., table or attribute name) in a schema S, we are able to identify the set of elements, C_e, in the corpus that are similar to the element e. We can use C_e to *augment* the knowledge that we have about e. In our example, the table T1.BookAvailability is very similar to the table T3.BookStore in the corpus (their columns are also similar to each other). Similarly, T2.Stock is similar to T4.ProductAvailability. It is easy to see that combining the evidence in T3 with T1 and T4 with T2 better enables us to match T1 with T2: first, there is increased evidence for particular matching techniques, *e.g.*, alternative names for an element; and second, there is now evidence for matching techniques that lack evidences earlier, *e.g.* considering T3 with T1 includes data instances (tuples) where there were none initially. This is the intuition of augment that we will describe in Section 3.1.

Another method of exploiting the corpus is to estimate statistics about schemas and their elements. For example, given a corpus of inventory schemas, we can learn that Availability tables always have columns similar to ProductID and are likely to have a foreign key to a table about Warehouses. These statistics can be used to learn *domain constraints* about schemas (*e.g.*, a table is less likely to match Availability if it does not have a column that can match ProductID). We show how to learn such constraints, and how to use them to further improve schema matching. In particular these constraints are used in an A* search that selects matching element pairs from the element similarity values computed by augment. We note that previous work has shown that exploiting domain constraints is crucial to achieving high matching accuracy, but such constraints have always been specified manually. We outline the process of learning and application of constraints in Section 3.2.

3.1 The Augment Method

We now describe the augment method in detail. As noted earlier, the corpus is a collection of schemas and elements

within the schema. In order to find elements in the corpus that are similar to a given schema element s, we compute an *interpretation vector*, I_s, for s. I_s is a vector, $\langle \ldots, p_{e,s}, p_{f,s}, \ldots \rangle$, where $p_{e,s}$ is an estimate of how similar s is to element e in the corpus. We use machine learning to estimate these similarities. Specifically, for each element of the corpus, e, we *learn* a model; given an element s, the model of e predicts how similar e is to s.

Models for corpus elements: The model for each element is created via an ensemble of *base learners*, each of which exploits different evidences about the element. Some of the base learners that we use are as follows: a *name learner* that determines the word roots that are most characteristic of the name of an element (as compared to the names of other elements); a *data instance learner* that determines the words and special symbols (if any) that are most characteristic in instances of an element; and a *context learner* that determines the characteristics of elements that are related to an element. The predictions of the base learners are combined via a *meta-learner*.

Training each of these learners requires learner-specific positive and negative examples for the element on which it is being trained. For any element $s \in S$, the element is a positive example of itself, and all other elements in the schema are negative examples. If, in some mapping in the corpus, s is deemed similar to an element t in T, then the training examples for t can be added to the training examples for s. Mappings enable us to obtain more training data for elements and hence learn more general models. The meta-learner uses a linear combination to combine the predictions made by the base learners in the ensemble and is trained using a technique called *stacking* [20].

Note that all the base learners in the ensemble need not be classifiers. For example, we can also have a simple name comparator that uses string edit distance to estimate similarity of two names.

Augmenting and matching elements: The goal of the augment method is to enrich the models that we build for each element of the schemas being matched, thereby improving our ability to predict matches. Suppose we are deciding the match between an element s in schema S, and an element t in schema T. Given the interpretation vector for s, we pick C_s, a set of close elements from the corpus, using a simple criteria: pick the N elements that are most similar to e such that $p_{e,s} \geq \alpha$. The augmented models are constructed in a way similar to building models for each element in the corpus. Having determined C_s, an ensemble model is learned for s by putting together the training data for all the elements in C_s. In addition to C_s, other elements that are known to map to elements in C_s also contribute to the examples for s.

We note that since there are multiple schemas in the corpus, it is easier to find elements in the corpus that are similar to s than directly trying to match s with some element in T. Even if a few elements are incorrectly added to the augmented model, there are benefits as long as there are fewer of them than correctly added elements.

We use the learned augmented models for elements in schema S and T to compute the similarity between each $s \in S$ and $t \in T$. The similarities are computed as in [8], $sim(s,t)$ is average of the probabilities obtained by applying the augmented model for s on t and vice-versa.

In [13] we proposed an alternate method, pivot, for computing these similarities by simply computing the cosine measure between the interpretation vector. In practice, we found that augment performed better that pivot in all our domains.

The result of both the augment and the pivot methods is a similarity matrix: for each pair of elements $s \in S$ and $t \in T$, we have an estimate for their similarity (in the $[0, 1]$ range). Correspondence or matches can be selected using this matrix in a number of ways, as we describe next.

3.2 Constraints for Match Generation

The task of generating matches consists of picking the element-to-element correspondences between the two schemas being matched. As observed in previous work [15; 7], relying only on the similarity values does not suffice for two reasons. First, certain matching heuristics cannot be captured by similarity values (e.g., when two elements are related in a schema, then their close elements in the corpus should also be related). Second, knowledge of constraints plays a key role in pruning candidate matches.

Constraints can either be generic or dependent on a particular domain. As examples of the latter, if a table matches Books, then it must have a column similar to ISBN. If a column matches DiscountPrice, then there is likely another column that matches ListPrice. Most prior work has used only generic constraints, and when domain constraints have been used, they have been provided manually and only in the context where there is a single mediated schema for the domain [7].

In this section we briefly describe how domain constraints can be learned from a corpus of schemas, and also how we can estimate the relevance of the difference constraints to match generation. The latter is important because many of the constraints we employ are *soft* constraints; *i.e.*, they can be violated in some schemas.

Corpus Statistics: In order to learn constraints from the corpus, we must first estimate various statistics of its contents. For example, given a collection of Book schemas, we might find that all of them have a column for ISBN, 50% of them have author information in separate tables, and 75% have list and discount price information in the same table.

In order to estimate meaningful statistics about an element, we must have a *set* of examples for that element, e.g., we can make statements about tables of Books only when we have seen a few examples of similar tables. We hence group together elements in our corpus into clusters that intuitively correspond to *concepts*. As in the last section, we use agglomerative clustering to build clusters; however, here we are clustering elements in different schemas. We use a different set of refinements to the basic clustering algorithms; for example, we do not let two elements from the same schema ever be part of the same cluster. Given the clustering of corpus elements into concepts, here are some of the statistics that we estimate.

Tables and Columns: For relational schemas, compute for each table concept t_i and each column concept c_j, the conditional probability $P(t_i|c_j)$. This helps us identify the contexts in which columns occur. For example, the ISBN column most likely occurs in a Books table or an Availability table (as foreign key), but never in a Warehouse table.

Neighborhood: We compute for each concept the most likely other concepts they are related to. Briefly, we construct

itemsets from the relationship neighborhoods of each element, and learn association rules from these. For example, we learn that AvailableQuantity → WarehouseID, i.e., the attribute availability is typically specified w.r.t. a particular warehouse.

Ordering: If the elements in a schema have a natural ordering (*e.g.* the input fields in a web form, or the sub-elements of an XML element), then we can determine the likelihood of one concept preceding another.

Cost-based Match Generation: Given two schemas S and T, our goal is to select for each element in S the best corresponding element in T (and vice-versa). Specifically, for each element e in S we will assign either an element f in schema T, or *no match* (ϕ). Let M represent such a match. Thus, $M = \cup_i \{e_i \leftarrow f_i\}$, where $e_i \in S$ and $f_i \in T \cup \{\phi\}$.

We assign a cost to any such match M that is dependent on our estimated similarity values and constraints:

$$Cost(M) = -\sum log\ sim[e_i, f_i] + \sum w_j \times K_j(M) \qquad (1)$$

where $sim[e_i, f_i]$ is the estimated similarity of elements e_i and f_i, each K_j (≥ 0) is some penalty on the mapping M for violating the j^{th} constraint, and w_j is a weight that indicates the contribution of the j^{th} constraint. The first sum is an estimate of the total log likelihood of the mapping (if the similarities were interpreted as probabilities). The second sum is the penalty for violating various constraints. The task of generating the mapping from S to T is now reduced to the task of picking the mapping M with the minimal cost. We use A* search [18] to pick the best mapping M, which guarantees finding the match with the lowest cost. Our constraints are encoded as functions, K_j, that produce a value in the interval $[0, 1]$. We do not provide the details for each K_j, but note that the weight-learning algorithm described in the next section adapts w_j to the values K_j evaluates to.

Some of the constraints we use are the following. Among generic constraints, *uniqueness* states that each element must match with a distinct element in the target schema, and *mutual* states that e can match f only if e is one of the most similar elements of f and mutually f is one of the most similar elements of e. As domain constraints obtained from the corpus, we have the following: (1) *same-concept*: if two elements are to be matched, then they have to be similar to the same concept(s) in the corpus; (2) *likely-table*: if column e matches f, then e's table must be a likely table for f to be in; (3) *neighbors*: if element f is to be assigned to e, then elements that are likely related to f must be related to e; and (4) *ordering*: if element f is to be assigned to e, then the ordering corpus statistics should not be violated.

Learning Constraint Weights: Since many of the constraints we use in our matching algorithm are soft; *i.e.*, encode preferences rather than strict conditions, the choice of weight for each constraint is crucial. Prior work has always hard-coded these constraint weights. We now describe how these weights can actually be learned from known mappings. Consider a matching task between source schema S and target schema T. Consider a mapping M in which the correct matches are known for all elements in S except e. If f were the correct match for element e, then in order that e is also correctly matched with f, given the exact matches of other elements, the following condition must hold.

$$\forall f_i, f_i \neq f, Cost(M|e \leftarrow f) < Cost(M|e \leftarrow f_i) \qquad (2)$$

domain	type	# schemas	# elements (min-max, average, std.deviation)			# mappings	evidence
auto	webforms	30	3-28	11	6.7	74	text, variable names, select options
real estate	webforms	20	3-20	7	4.1	37	
invsmall	relational	26	11-33	18	4.9	39	names, examples, descriptions, context
inventory	relational	34	26-90	41	16.5	30	

Table 1: Characteristics of evaluation domains.

This can be re-written for each element f_i as below.

$$L(M, sim, e, f_i, \bar{w}) = \log\ sim[e, f] - \log\ sim[e, f_i] + \sum_j w_j \times [K_j(M|e \leftarrow f) - K_j(M|e \leftarrow f_i)] > 0 \qquad (3)$$

Element e is incorrectly matched if the above condition is violated for some f_i. In [14] we describe how we learn the values of the w_js (using *hill-climbing* search) by minimizing the number of incorrect violations of this above condition in a known set of mappings.

3.3 Experimental Results

We now present experimental results that demonstrate the performance of corpus-based matching. We show that corpus-based matching works well in a number of domains, and in general has better results than matching schemas directly. Furthermore, we show our techniques are especially effective on schema pairs that are *harder* to match directly.

Datasets: We used a variety of domains and Table 1 summarizes some of their basic characteristics. We note that the web form schemas (set of visible input fields) were extracted by a rather primitive text extractor and hence had a number of errors that made the matching difficult. The relational schemas were created independently by undergraduate students as part of a class project in their database class. These schemas were varied in their choice of tables (3-12), number of columns, and data types.

In each domain, we manually created mappings between randomly chosen schema pairs. The matches were *one-many*, i.e., an element can match any number of elements in the other schema. These manually-created mappings are used as training data and as a *gold standard* to compare the mapping performance of the different methods.

Experimental Methodology: We compared three methods: augment, direct, and pivot: augment is our complete corpus-based solution. direct uses the same base learners described in Section 3.1, but the training data for these learners is extracted *only* from the schemas being matched. direct is similar to the Glue system [8] and can be considered a fair representative of direct-matching methods. pivot, as described in Section 3.1, is the method that computes cosine distance of the interpretation vectors of two elements directly.

The result of each of our methods is a directional match: for each element in a schema, an element from the other schema is chosen such that the cost of entire mapping is minimized. If the gold standard has a match in which s matches a set of elements E, then a matcher is said to have predicted it correctly if s is predicted to match any one element in E, and every element in E is predicted to match s. As a result, any $1:m$ mapping is considered as $m+1$ separate matches. We report matching performance in terms of *F-Measure*. For a given schema pair, let c be the number of elements

in the two schemas for which a match is predicted and the predicted match is correct. If a match was predicted for n elements, and a match exists in the gold standard for m elements, the f-measure is the harmonic mean of the precision and recall.

$$\text{FMeasure} = \frac{2pr}{p+r}, \text{where } p = \frac{c}{n}, r = \frac{c}{m}$$

Optimizing for f-measure tries to balance the inverse relation between precision and recall.

Corpus improves F-Measure: Figure 4 compares the results of direct, augment, and pivot in each of the four domains. There are 22 (auto), 16(real estate), 19(invsmall), 16(inventory) schemas respectively and 6 mappings in the corpus. augment achieves a better f-measure than direct and pivot in all domains. There is a $0.03 - 0.11$ increase in f-measure as compared to direct, and a $0.04 - 0.06$ increase as compared to pivot. We note that augment has a better recall as compared to direct in all four domains, and better precision in three of the four domains (the precision is lower in the in the inventory domain due to the presence of many ambiguous matching columns, but there is a noticeable increase in recall). These results show that augmenting the evidence about schemas leads to the discovery of new matches, and in most cases fewer incorrect match predictions are made.

The results in this section consider only the single best match for each element. In an interactive schema-matching system, we typically offer the user the top few matches when there is a doubt. When we consider the top-3 candidate matches, then augment is able to identify the correct matches for 97.2% of the elements, as opposed to 91.1% by direct and 96.7% by pivot.

Difficult versus Easy matching tasks: Our central claim is that corpus-based matching offers benefits when there is insufficient *direct* evidence in the schemas. To validate this claim, we divided the manual mappings in each domain into two sets - easy and difficult: all the schema pairs in the test set were matched by direct and then sorted by direct's matching performance. The top 50% were identified as the easy pairs and the bottom 50% as the difficult pairs.

Figure 4b compares the average f-measure over the difficult matching tasks, showing that augment outperforms direct in these tasks. More importantly, the improvement in f-measure over direct is much more significant $(0.04 - 0.16)$ than in Figure 4a, e.g., there is an improvement of 0.16 in the invsmall domain, and 0.12 in the real estate domain as compared to the 0.11 and 0.07 increases when all tasks are considered.

Figure 4c shows the same comparison over the easy tasks. The performance of augment for these tasks, while still very good, is in fact slightly worse than direct in one domain and the improvements are much less in the other domains. This is quite intuitive in retrospect. When two schemas are rather similar (easy to match directly), including additional evidence can lead the matcher astray.

4. DISCUSSION AND RELATED WORK

We described two related projects that exploit corpora of structures. They correspond to the two steps in searching and accessing the hidden web: (1) locating the desired data source; (2) reformulating a query onto the schema of the source. While both steps rely on a good understanding of the structure information, they are different in the matching granularity: data-source location is essentially finding a similar schema, while schema matching looks for similar elements *within* two given schemas that are assumed to be related. In the former task, the exact details of the various elements and their relationships are less important: it suffices to know that a similar element exists, rather than knowing which element it is.

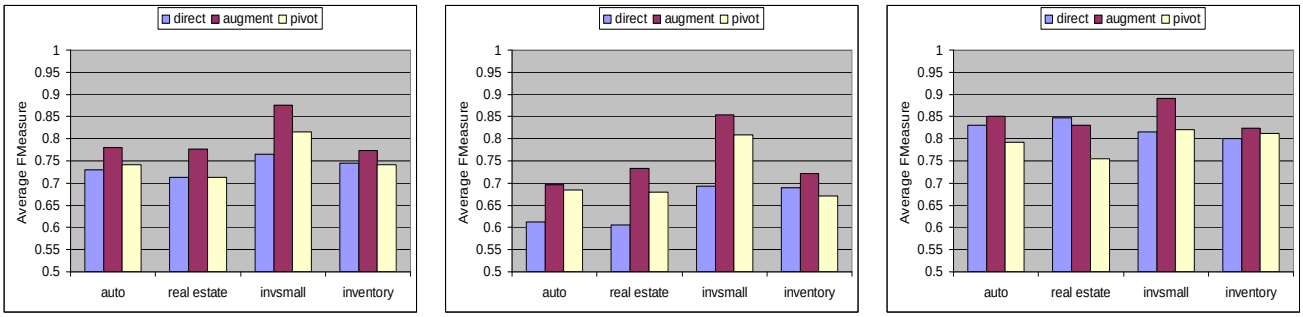
As we develop corpus-based techniques to searching structures with rich semantics, there is another dimension to keep in mind, namely, the cohesion of the underlying structure. In particular, a database schema includes a set of tightly-coupled tables and attributes that, together, are meant to model a set of objects. This raises the complexity of the matching, but meanwhile provides rich information. For example, the schema definition languages for databases are strong in defining constraints, such as type constraints, key and foreign key constraints, and thus present more opportunities for exploring constraints in schema matching. In contrast, the operations in web services are only loosely coupled, and each one in isolation has much less information. However, the descriptions in the WSDLs and the UDDI entries allow for applying the information retrieval techniques in matching. In addition, the input and output parameters in an operation are organized in a tree hierarchy rather than forming a flat vector and so imply the relationship between parameters. Web forms can be viewed as very simple database schemata, and are similar to web services in that they also take certain inputs. Although here the inputs do not have particular type information, other aspects of information can be explored to understand the inputs, such as text descriptions on the webpage, the layout of the input components on the page, and the values of the drop-down boxes. Furthermore, a field in a web form may already correspond to a selection or aggregation query over the underlying schema (e.g., price range or maximum price), rather than to the schema element itself.

4.1 Related Work

The use of previous schema and mapping knowledge has been proposed in the past, but in two very restricted settings. They either use previous mappings to map multiple data sources to a *single* known mediated schema [7], or compose known mappings to a common schema [6]. In our approach, we show that a corpus of schemas and mappings can be leveraged in many different ways to discover matches between two as *yet unseen* schemas. In [10], the authors construct a single mediated schema for a domain of web forms. They estimate the single most likely mediated schema that could generate all the web forms in a given collection. In [22] the authors collectively match a number of related web forms by clustering their fields. We use such clustering as a step in learning constraints.

4.2 Future Directions

There are several exciting future directions we are pursuing. First, we believe that corpus-based techniques should be helpful in *authoring* database schemata and *querying* unfamiliar schemas. Second, we plan to apply our techniques to software component matching, where the goal is to find methods that have the same signatures and behavior [24]. While some of our techniques for analyzing web service descriptions may apply, in the context of software components



(a) Overall F-Measure (b) Difficult Tasks (c) Easy Tasks

Figure 4: (a) shows that augment performs better overall than direct and pivot in all domains, but the improvement is more significant for difficult tasks (b) and more modest for easy tasks (c).

we may also leverage analysis of pre- and post-conditions.

5. REFERENCES

[1] Binding Point. http://www.bindingpoint.com.

[2] WordNet. http://www.cogsci.princeton.edu/∼wn/.

[3] R. Agarwal, T. Imielinski, and A. Swami. Mining Associations between Sets of Items in Massive Databases. In *SIGMOD*, 1993.

[4] J. Berlin and A. Motro. Database Schema Matching Using Machine Learning with Feature Selection. In *CAiSE*, 2002.

[5] S. C. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, and R. A. Harshman. Indexing by latent semantic analysis. *JASIS*, 41(6):391–407, 1990.

[6] H.-H. Do and E. Rahm. COMA - A System for Flexible Combination of Schema Matching Approaches. In *VLDB*, 2002.

[7] A. Doan, P. Domingos, and A. Y. Halevy. Reconciling Schemas of Disparate Data Sources: A Machine Learning Approach. In *SIGMOD*, 2001.

[8] A. Doan, J. Madhavan, P. Domingos, and A. Y. Halevy. Learning to Map between Ontologies on the Semantic Web. In *WWW*, 2002.

[9] X. Dong, A. Halevy, J. Madhavan, E. Nemes, and J. Zhang. Similarity Search for Web Services. In *VLDB*, 2004.

[10] B. He and K. C.-C. Chang. Statistical Schema Matching across Web Query Interfaces. In *SIGMOD*, 2003.

[11] J. Kang and J. Naughton. On schema matching with opaque column names and data values. In *SIGMOD*, 2003.

[12] L. Kaufman and P. J. Rousseeuw. *Finding Groups in Data: An Introduction to Cluster Analysis*. John Wiley & Sons, New York, 1990.

[13] J. Madhavan, P. Bernstein, K. Chen, A. Halevy, and P. Shenoy. Corpus-based Schema Matching. In *Information Integration Workshop at IJCAI*, 2003.

[14] J. Madhavan, P. A. Bernstein, A. Doan, and A. Halevy. Corpus-based Schema Matching. In *ICDE*, 2005.

[15] S. Melnik, H. Garcia-Molina, and E. Rahm. Similarity Flooding: A Versatile Graph Matching Algorithm. In *ICDE*, 2002.

[16] N. F. Noy and M. A. Musen. PROMPT: Algorithm and Tool for Automated Ontology Merging and Alignment. In *AAAI*, 2000.

[17] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *VLDB Journal*, 10(4), 2001.

[18] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. 2nd edition, 2003.

[19] G. Salton, editor. *The SMART Retrieval System—Experiments in Automatic Document Retrieval*, 1971.

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

[21] J. Wang, J.-R. Wen, F. Lochovsky, and W.-Y. Ma. Instance-based Schema Matching for Web Databases by Domain-specific Query Probing. In *VLDB*, 2004.

[22] W. Wu, C. Yu, A. Doan, and W. Meng. An Interactive Clustering-based Approach to Integrating Source Query interfaces on the Deep Web. In *SIGMOD*, 2004.

[23] L. Xu and D. Embley. Discovering Direct and Indirect Matches for Schema Elements. In *DASFAA*, 2003.

[24] A. M. Zaremski and J. M. Wing. Specification matching of software components. *TOSEM*, 6:333–369, 1997.