Semantic Labeling of Online Information Sources

December 8, 2006

Abstract

Software agents need to combine information from a variety of heterogeneous sources in the course of automating a task, such as planning a trip. In order to be able to use a source, an agent must have a model of it, i.e., understand the *semantics of the input and output data* it uses, as well as its *functionality.* Currently, source modeling is done by the user, but as large numbers of sources come online, it is impractical to expect the user to manually model them. To address this problem, it has been proposed that service providers use common ontologies. However, it appears to be equally impractical to expect service providers to conform to a standard, as there is very little incentive for them to do so. Instead, we propose to automatically learn the semantics of information sources by labeling the input and output parameters used by the source with semantic types of the user's domain model. We describe two machine learning techniques for semantic labeling: one that uses source's metadata, such as that contained in a Web Service Definition file, and one that uses the source's content to classify the semantic types it uses. We go beyond previous works by verifying the classifier's predictions by invoking the source with some sample data of the predicted type. We provide performance results of both classification methods and validate our approach on several live Web source — both Web services and HTML-form based sources. We also describe application of the semantic mapping technology within the CALO project.

1 Introduction

Software agents will soon assist users in the office environment by carrying out complex everyday tasks, such as planning a trip to a meeting. To complete their task successfully, these agents will need to combine information from a variety of heterogeneous sources. Take, for example, the task of arranging a trip. A user has to follow a number of steps in a sequence, including

- get the date and location of the meeting,
- find a convenient flight on those dates to that location
- get the weather forecast
- find all hotels in the city of the meeting with the required amenities (e.g., high speed Internet),
- keep hotels within 3 miles of the meeting site
- keep hotels with rates within government-allowed per diem
- book hotel from the list with the best reviews
- rent a car, etc.

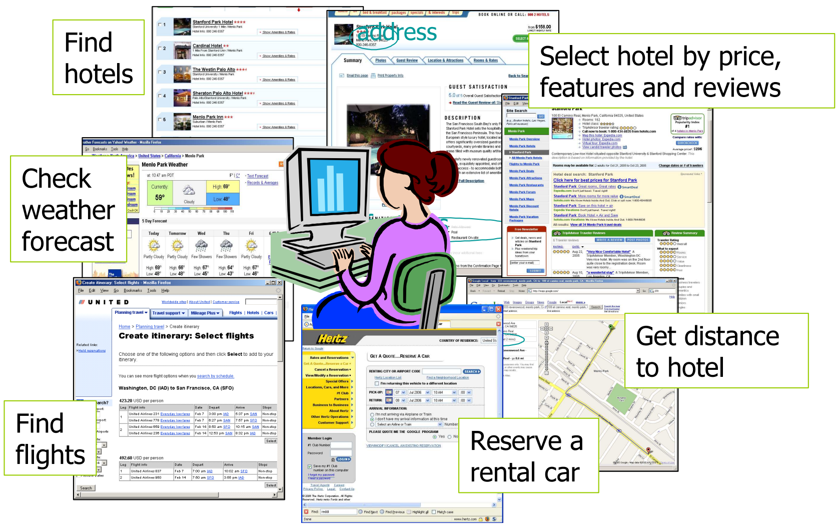


Figure 1: Information sources accessed by the user in the course of arranging a trip to a meeting

Some of the information sources a user interacts with in the course of carrying out this task are shown in Figure 1. In order to make use of these sources — whether they are Web services, HTML pages or databases — an intelligent agent needs to know the schema of the source. Currently, the user must manually model the source: specify the type of data it accepts and returns, and the functionality it provides. This is what is done by information integration systems [25] that, similarly to our envisioned intelligent office assistants, provide uniform access to heterogeneous online information sources in order to answer a specific user query or be composed with other sources as a new information service. However, as new sources come online or are discovered, it is infeasible to ask the user to explicitly model them. While various technologies, most notably the Semantic Web, have been proposed to enable programmatic access to new sources, they are slow to be adopted, and at best will offer only a partial solution, because information providers will not always agree on a common schema. Rather than rely on standards to which providers may or may not conform, we instead propose to automatically model a new source: i.e., *discover the semantics of its inputs and outputs, and the functionality it provides.*

We divide source modeling task into two subproblems: recognizing the semantics of the data it uses (*semantic labeling*), and inducing the logical definition of the source (*learning functionality*) [3]. This paper addresses the first problem, which is similar to the schema matching problem encountered by the database community, where the schema of one database has to be linked with the schema of another before the first database can use the data provided by the second. Unlike schema matching, where the content of both databases is available to the matching algorithm, we must first figure out how to invoke the source whose input and output parameters we are trying to label. We will show in the paper that we can use the source's metadata, if available, to classify the input parameters used by the source and assign them to predefined semantic types If the metadata is not easily available, but instead, a user has provided a few examples of the inputs by querying the source, we show that we can classify the data types used by the source using their content only. While others have used metadata to semantically classify the source's data types [8], we go further and verify a classifier's predictions by actively invoking the source. Successful querying both confirms the prediction and generates output data. We can then use the content-based classification algorithms to semantically label the output parameters. In case of unsuccessful invocation, two possible conclusions can be drawn: (1) the semantic type prediction is incorrect and (2) the prediction is correct but the source does not provide coverage over the data used as the input samples. Here, we simplify the problem by assuming that the source usually covers the sample data (returned by known sources contained in our domain model).

We use machine learning techniques to leverages existing knowledge in order to identify the semantics of new information sources. Our approach relies on background knowledge captured in the *domain model.* The domain model contains a hierarchy of semantic types, e.g., Temperature and Zipcode. These types correspond to objects in an ontology or their attributes. Our system knows how to populate the domain model with examples of each semantic type by querying sources with known definitions or schema. For example, queries to *weather.yahoo.com* return tuples of current weather conditions of the form (Zipcode, Temperature, Wind, Humidity). The domain model was created manually from sources with diverse schema and populated with the aid of wrappers. A Web wrapper is a tool that allows the user to define what items to extract from an information source — e.g, a Web page. The wrappers cited in this paper were created by different users for applications unrelated to the research problem described in this paper; hence, the domain model is far from being a formal ontology. Our goal is automatically relate the semantic types of the input and output parameters of new sources to our light weight domain model, and eventually, to automatically incorporate new sources into this domain model.

In a previous work [16] we described two approaches for automatically labeling the input and output parameters of Web services: the *metadata-based* and *content-based classification*. Web services are convenient to study because (i) they come with a syntactic description, contained in the Web Service Definition (WSDL) file, and (ii) output is encoded in XML, making it trivial to extract output data. We presented extensive evaluations of both classification approaches, showing dramatic improvement in metadata-based classification compared to previous works. We also validated our methods by semantically labeling several live Web services.

The present paper extends the approach to form-based Web sources for which the metadata may not readily available. Instead of metadata, we use example queries, which the user provided in the course of training a Web wrapper to extract data from the Web source, to predict the input parameters of this source. We present new experimental data evaluating the performance of our algorithm on a sources from a variety of information domains. This technology was deployed within the Defense Advanced Research Projects Agency's CALO project.[1] The Semantic Labeling tool, described in this paper, assists the user in mapping input and output parameters used by Web sources to a common schema understandable by all CALO components. Once a Web source has been mapped, it can be automatically assembled into invocable procedures or agents like the ones described above.

In the sections below we provide details of the metadata-based (Section 2.1) and the content-based classification algorithms (Section 2.2) for labeling the inputs and outputs of information sources and evaluate their performance on test datasets. In Section 4 we describe a novel application of this technology inside the CALO's cognitive desktop. We validate our approach by automatically modeling live Web services (Section 3) and form-based Web sources (Section 4.2).

2 Semantic Labeling as Classification

Given background knowledge contained in the domain model and a new source, our goal is to automatically assign classes (semantic types) to both input and output parameters of the source. First, and most importantly, we must learn how to query the source. If the source comes with metadata (as Web services do), we can use it to recognize inputs using metadata-based classification algorithm. Otherwise, we require the user to supply examples of queries and use content-based classification algorithm to semantically label the input parameters contained within queries. We can then invoke the source — Web services through SOAP requests, and HTML form-based sources through Web wrappers trained to extract data from them — to verify classification results and generate examples of output data. Outputs can then be semantically labeled using a combination of content-based and metadata-based classification algorithms described below.

[1]CALO stands for Cognitive Assistant that Learns and Organizes, and information about the project can be found at http://www.ai.sri.com/project/CALO.

```
– <xsd:complexType name=“Point”>
   – <xsd:sequence>
      <xsd:element name=“x” type=“xsd:double”/ >
      <xsd:element name=“y” type=“xsd:double”/ >
      <xsd:element name=“coordinateSystem”” nillable=“true” type=“ns11:CoordinateSystem”/ >
   </xsd:sequence>
< /xsd:complexType>
– <message name=“getAddress0In”>
   – <part name=“point” type=“ns11:Point”>
      <documentation>the x,y-coordinate.< /documentation>
   </part>
   – <part name=“addressFinderOptions” type=“ns13:AddressFinderOptions”>
      <documentation>options object.< /documentation>
   </part>
   – <part name=“token” type=“xsd:string”>
      <documentation>the authentication token.< /documentation>
   </part>
</message>
– <message name=“getAddress0Out”>
   – <part name=“Result” type=”ns13:Address”>
      <documentation>Address address object.< /documentation>
   </part>
</message>
– <operation name=“getAddress” parameterOrder=“point addressFinderOptions token”>
   <documentation>Returns an address from an x,y-coordinate.< /documentation>
   <input name=“getAddress0In” message=“tns:getAddress0In”/ >
   <output name=“getAddress0Out” message=“tns:getAddress0Out”/ >
</operation>
```

Figure 2: Portion of a Web Service Definition file from the ArcWeb AddressFinder service

2.1 Metadata-based Classification

Web services offer a natural domain for metadata-based classification techniques, because each service comes with a Web Service Definition (WSDL) file. The WSDL file specifies the protocol for invoking the service, lists supported operations, and specifies the data types of the inputs and outputs of the operations. Furthermore, an input may be constrained to a set of values, or facets, enumerated in the WSDL file. In most cases, however, operations, inputs and outputs carry no explicit semantic information — only descriptive names attached to them by the Web service developer. Figure 2 shows a portion of a WSDL file of an actual Web service that returns an address for a given location. The terms that appear in parameter names — "Address," "point," "coordinate system" — are quite useful for recognizing the functionality of the service.[2]

We exploit WSDL input/output parameter and operation names for classification. Our approach is based on the following heuristic: similar data types tend to be named by similar names and facets (if applicable), and/or belong to messages and operations that are similarly named [8, 7]. The desired classifier must be a soft classifier: it should not rigidly assign one class to a particular data type; instead, it should order all possible classes by likelihood scores, which will allow us to use examples from the next runner up class to invoke the service.

Hess and Kushmerick (2003) used a Naive Bayes classifier to assign semantic types to the input and output parameters using background knowledge collected from known services. They represented inputs and outputs by terms $\vec{t}$ extracted from the names of these parameters in the WSDL file. For example, in the WSDL file in Figure 2, the input parameter named "point" can be represented by the terms gathered from the message name and component subtypes: $\vec{t} = \{get, address, 0, in, point, x, y, coordinate, system, \}$. The classifier assigns an object represented by a feature vector $\vec{t}$ to class D (semantic type) that maximizes $P(D|\vec{t})$. In the example above, it should be assigned $D =$CoordinatePoint. $P(D|\vec{t})$ cannot be computed directly from data. Instead, one can estimate $P(\vec{t}|D)$, $P(\vec{t})$ and $P(D)$ from data, and then use them to compute $P(D|\vec{t})$ by using Bayes rule. Unfortunately, it is not feasible to compute $P(\vec{t}|D)$ (or $P(t_1, t_2, t_2, \ldots, t_n|D)$) directly, because the number of possible combinations of terms will be exponentially large. To solve this problem an independence assumption is introduced: terms are assumed to be conditionally independent given a particular class D. Thus, estimation of $P(\vec{t}|D)$ is reduced to the estimation of $\Pi_{i=1}^{n} P(t_i|D)$. Laplace smoothing is applied in this setting to prevent zero-value estimation in the case that some terms do not co-occur with a certain class.

2.1.1 Classification by a Discriminative Method

Potentially, because the independence assumption may not hold in this domain as we can perceive that Temperature and Fahrenheit terms in parameter named "TemperatureFahrenheit" are not independent from each other given a Temperature data type in weather domain. In addition, Naive Bayes does not directly estimate decision boundaries. Probably for these reasons, classification does not yield accurate enough results to enable the services to be successfully invoked. We thus ran another classification algorithm using Logistic Regression.[3] Without the term-class independence assumption, Logistic Regression directly estimates parameters, namely input weights, for computing $P(D|\vec{t})$ from the data [22, 21]. The formula for computing $P(D|\vec{t})$ is $P(D|\vec{t}) = logreg(\vec{w}\vec{t})$ where $\vec{t}$ is a feature vector (occurrences of terms) and $\vec{w}$ is a weight vector. In the training phase, the classifier adjusts the weight vector to maximize log data likelihood, $\Sigma_{j=1}^{m} ln(P(D_j|\vec{t}_i; \vec{w}))$, where D_j is the class label of the j-th sample. There are several methods to find such weight vector as mentioned in [19]. We applied conjugate gradient ascent optimizer to such a weight vector. We use a logistic regression optimizer package written by T. P. Minka [20] for this.

[2]This particular service also provides documentation strings which explain its functionality, although in general, WSDL files are not frequently documented.

[3]Logistic Regression and Naive Bayes classifiers are related: One can prove that Naive bayes can be represented in Logistic Regression form [21]. Also, if the independence assumption holds and there are infinite training data, both classifiers will give the same result [22].

Equation 1 gives the Logistic Regression equation for binary classification:

$$P(D = \pm 1 | \vec{t}, \vec{w}) = \frac{1}{1 + \exp(-D\vec{w}^T\vec{t})} \tag{1}$$

Here, $\vec{t}$ is a vector of features (terms), defined as $t_i = 1$ if feature t_i exists among the terms associated with the parameter we are classifying; otherwise $t_i = 0$. $D = 1$ if the parameter belongs to class D; otherwise $D = -1$, and $\vec{w}$ is a vector of weights to be learned. We used Equation 1 to learn logistic classifiers for each input/output parameter.

2.1.2 Evaluation

We evaluated metadata-based classification on a data set that consisted of 313 Web Service Definition files from web service portals (*Bindingpoint* and *Webservicex*) that aggregate services from several domains. We extracted names from several parts of the WSDL file — operations, messages, data types and facets. Subsequently, the names were decomposed into individual terms. Thus, "GetWeatherByZipRequest" was decomposed into five terms — get, by, request, weather, zip — where the first three words are stopwords and are ignored. Each extracted term was then stemmed with Porter Stemmer. Each input and output parameter was represented by a set of features: terms from its operation, message name and facets. We expanded the complex data types to their leaf nodes and used those as data types; meanwhile, terms from objects at higher levels were collected as "auxiliary features." In all, $12,493$ data types were extracted and labeled. Each was assigned to one of 80 classes: Latitude, City, Humidity, etc. These were user-defined classes which were collected from a large body of existing Web wrappers created by our group over the years. Other classes, e.g. Passenger, which never appeared in any of the wrappers, were treated as Unknown class.

Both Naive Bayes and Logistic Regression classifiers were tested using 10-fold cross validation. Both classifiers were trained using a one-against-all method, and predicted classes for each data type were ordered by their probabilities. Since data types from the same message generally tend to contain similar terms, to avoid introducing classification bias, data types from the same message were forced to be in the same fold. Alternatively, we ensure that all data types from a WSDL file are in the same fold, as it is likely the service developer used similar terminology for different data and operation names. Table 1 presents classification accuracy results for the two classifiers. Results are shown for increasing tolerance levels. Zero tolerance ($T = 0$) means that highest ranked prediction was the correct one; $T = 3$ means that correct class was among the top four guesses. The logistic regression classifier significantly outperformed Naive Bayes. Taking the top predictions to query a source with two input parameters, Naive Bayes method would result in correct queries only 42% of the time, while Logistic Regression would produce correct results 86% of the time.

2.2 Content-based Classification

The data returned by Web services usually have some structure to them — phone numbers, prices, dates, street addresses, names, *etc.* follow some format. Content-based classification method attempts to learn this structure and use it to recognize new examples of these semantic types. What makes the problem

Table 1: Classification accuracy for the Naive Bayes and Logistic Regression classifiers

	$T = 0$	$T = 1$	$T = 2$	$T = 3$
	(a) fold by wsdl			
Naive Bayes	0.64	0.80	0.84	0.86
Logistic Regr	0.87	0.94	0.95	0.96
	(b) fold by message			
Naive Bayes	0.65	0.84	0.88	0.90
Logistic Regr	0.93	0.98	0.99	0.99

of automatically recognizing these types difficult is a great multiplicity of formats for the same type and even inconsistent use of the format by the same source. We have developed a domain-independent pattern language [14] that allows us to represent the structure of data as a sequence of tokens or token types and, moreover, learn it from examples.

The token-level representation we employ balances descriptive power of character-level description — those given by regular expressions, for example — with computational efficiency of a coarser description, such as that given by word frequencies. Tokens are strings generated from an alphabet containing different character types: alphabetic, numeric, punctuation, *etc.* We use the token's character types to assign it to one or more syntactic categories: alphabetic, numeric, *etc.* These categories form a hierarchy shown in Figure 2.2. The hierarchical representation allows for multi-level generalization. Thus, the token "90210" can be represented by a specific token "90210," as well as general types 5Digit, Number, Alphanum. These general types have regular expression-like recognizers, which simply identify the syntactic category to which the token's characters belong. This representation is flexible and may be expanded to include domain specific information.

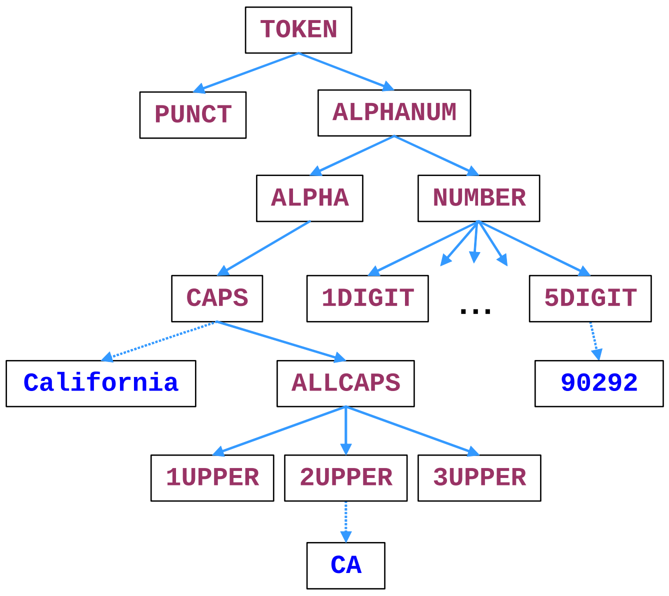


Figure 3: Hierarchy of syntactic token types

2.2.1 Learning Patterns from Data

In earlier work [15] we introduced an algorithm that learns patterns from positive examples of field. This algorithm finds statistically significant sequences of tokens, i.e., those that occur more frequently than would be expected if the tokens were generated randomly and independently of one another.

The algorithm estimates the baseline probability of a type's occurrence from the proportion of all types in the examples. Suppose we are learning a description of street addresses, and have already found a significant sequence — *e.g.*, the pattern consisting of the single token [Number] — and want to determine whether the more specific pattern, [Number Caps], is also significant. Knowing the probability of occurrence of Caps, we can compute how many times Caps can be expected to follow Number completely by chance. If we observe a considerably greater number of these sequences, we conclude that the longer pattern is also significant. One unintended consequence of this algorithms is that even a small number of occurrences of a rare token will be judged significant.

The learning algorithm is biased towards producing more specific patterns. For example, when learning

Table 2: Selection of patterns learned for some of the semantic types in the weather and geospatial domains. For reasons of compactness, we shorted the names of the general token types from Number, xDigit to Num and xDig respectively.

TemperatureF	**Sky**	**Windspeed**	**Visibility**
[32 F]	[Scattered Clouds]	[12 mph]	[7.00 mi]
[35.1Dig°F]	[Sunny and Windy]	[12 mph/19 kph]	[Unlimited]
[35 F]	[Cloudy]	[12 MPH]	[10.0 miles]
[36°F]	[Clear]	[9 mph]	[10.00 mi]
[36 F]	[Light Snow]	[9 mph/14 kph]	[10.00 Miles]
[39°F]	[Partly Cloudy]	[9 MPH]	
[39 F]	[A Few Clouds]	[22 MPH]	
[41 F]	[Overcast]	[2Dig mph]	
[70°F]	[Mostly Cloudy]	[2Dig mph/16 kph]	
[70 F]	[Fair]	[2Dig MPH]	
[2Dig°F]		[1Dig mph/0 kph]	
[2Dig F]			

Latitude	**Longitude**	**Latitude2**	**Longitude2**
[34.3Dig]	[-2Dig.0833]	[40 2Dig 00 N]	[40 2Dig 00 N]
[34.Num]	[-2Dig.9167]	[42 2Dig 00 N]	[42 2Dig 00 N]
[2Dig.617]	[-2Dig.0333]	[38 2Dig 00 N]	[38 2Dig 00 N]
[2Dig.067]	[-2Dig.2667]	[45 2Dig 00 N]	[45 2Dig 00 N]
[2Dig.6]	[-2Dig.4Dig]	[30 2Dig 00 N]	[30 2Dig 00 N]
[2Dig.6333]	[-2Dig.1Dig]	[37 2Dig 00 N]	[37 2Dig 00 N]
[2Dig.75]	[2Dig.183]	[2Dig 30 00 1Upp]	[2Dig 30 00 1Upp]
[2Dig.95]	[2Dig.324]	[2Dig 45 00 N]	[2Dig 45 00 N]
[2Dig.4Dig]	[2Dig.883]	[2Dig 24 00 N]	[2Dig 24 00 N]
[2Dig.3Dig]	[2Dig.983]	[2Dig 38 00 1Upp]	[2Dig 38 00 1Upp]
[2Dig.1Dig]	[2Dig.3Dig]	[2Dig 2Dig 00 S]	[2Dig 2Dig 00 S]
[2Dig.Num]	[2Dig.1Dig]	[2Dig 2Dig 00 N]	[2Dig 2Dig 00 N]

patterns for a set of addresses, where many are located on Main St and Elm St, the algorithm constructs (i) [Number Alphnum St], (ii) [Number Alpha St], (iii) [Number Caps St], (iv) [Number Main St] and (v) [Number Elm St]. It eliminates pattern (i), because all the examples that match the pattern (i) also match the more specific pattern (ii). It eliminates pattern (ii) for the same reason. If pattern (iii) explains significantly more examples than the specific patterns (iv) and (v), it is kept and (iv) and (v) are deleted; otherwise, (iv) and (v) kept and (iii) is deleted. This tends to produce content descriptions comprised of many patterns.

DataProG is not guaranteed to learn complete patterns, that is patterns that describe the examples in their entirety, only the starting patterns. If, for instance, a set of addresses also contained a few addresses with apartment or suite number given, DataProG may not have learned the longer patterns that describe these, if there was not sufficient regularity in the longer examples. Thus, in addition to the patterns, we also keep track of the mean and variance of the length (in tokens) of the training examples.

Table 2 and Table 3 show subset of the patterns learned for different types the weather, geospatial, cars and airlines domains. Here only a few of the specific patterns are shown for each semantic type, in addition to the general patterns. The pattern learning algorithm learned, on average, about 30 patterns for each of the 80 semantic types in the domain model. One of the future goals of our research is to create more compact representations that will reduce the size of the domain model without sacrificing accuracy.

2.2.2 Using Patterns to Recognize Data

We can use learned patterns to recognize semantic types of Web service's output parameters based on the content of data it returns. The basic premise is to check how well the set of patterns associated with each

Table 3: Selection of patterns learned for some of the semantic types in the cars and airlines domains

Bodycolor	**Price**	**Mileage**
[red]	[$30,988]	[42,3Dig]
[dkblue]	[$29,988]	[38,590]
[gold]	[$29,995]	[44,018]
[ltblue]	[$27,988]	[36,352]
[blue]	[$25,988]	[36,3Dig]
[white]	[$25,3Dig]	[41 k]
[desert silver]	[$24,3Dig]	[31,607]
[glacier white]	[$22,988]	[52 k]
[smoke silver]	[$19,988]	[2Dig,069]
[silver]	[$2Dig, 988]	[2Dig,352]
[black]	[$2Dig,900]	[2Dig , 3Dig]
[Alpha blue]	[$2Dig,995]	[2Dig k]
Airline	**Aircraft**	**Altitude**
[United]	[Boeing, 737-800]	[Num,700 feet]
[Alaska]	[Boeing 3Dig-200]	[Num,900 feet]
[American]	[Alpha 3Dig]	[Num,500 feet]
[American Eagle]	[Caps 1900]	[Num,000 feet]
[Caps Jet Aviation]	[Caps Caps 4Dig]	[Num,3Dig feet]
[Caps Wisconsin]	[Caps Caps 1Dig]	
[Caps Airways]	[Alpha-72]	**Flightnumber**
[Caps Aviation]	[Alpha SF, -, 340]	[884]
[Caps Air]	[Alpha/BAe RJ 85]	[3Dig]
[Caps Air Cargo]	[Alpha 328]	[4Dig]
[Caps Airlines]		

semantic type (from the domain model) describes the set of examples of each output parameter, and assign them to the best-scoring semantic type. To be considered a match, a pattern has to recognize a prefix of an example, not all the tokens in the example. General token types can recognize specific instances. Thus, both [Number Caps St] and [Number Main St] match the example "350 Main St." We developed heuristics to score how well each semantic type — or patterns associated with it — matches the examples. In a nutshell, the more specific the matched pattern, the higher the score. Factors that are considered in the score are:

- Number of patterns that match examples
- Pattern length — give longer patterns a higher score
- Pattern weight — give more patterns consisting of more specific tokens a higher score
- Penalty for unmatched tokens — scores are reduced based on the number of tokens left unmatched by the pattern

The output of the algorithm is top four guesses for the semantic type, sorted by score. The reason we display top four choices is the following: in a real world application, we don't expect the automatic algorithms to correctly guess the semantic type of data 100% of the time. We believe that allowing the user to select the correct semantic type among four guesses — with the correct type highly likely to be among these four — will still be very useful, as it will dramatically reduce the effort involved in semantically modeling a new information source.

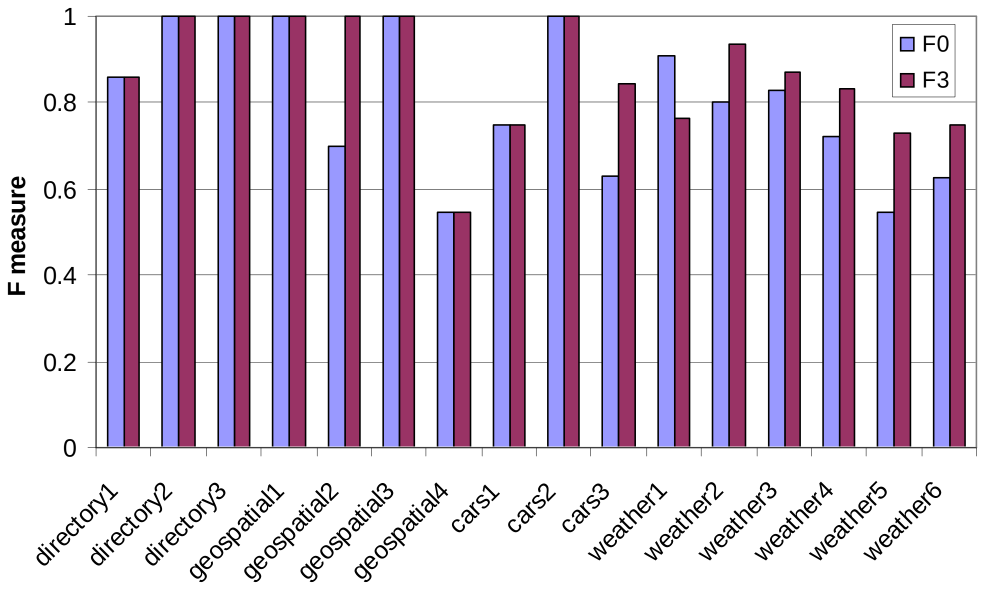


Figure 4: Performance of the content-based classification algorithm on data from a variety of domains.

2.2.3 Evaluation

We evaluated the performance of content-based classification on data sources from a variety of domains. We used existing wrappers created by different users to extract attributes from these sources. We reserved two or three sources from each domain for learning data patterns, and then used content-based classification to semantically label data from the remaining sources. The sources and extracted attributes were:

Airlines domain: Five sources related to flight status, with 44 attributes: airport, airline, time, date, flightstatus, . . .

Directory domain: Five directory services with 19 attributes extracted: fullname, streetaddress, city, state, zipcode, and phonenumber

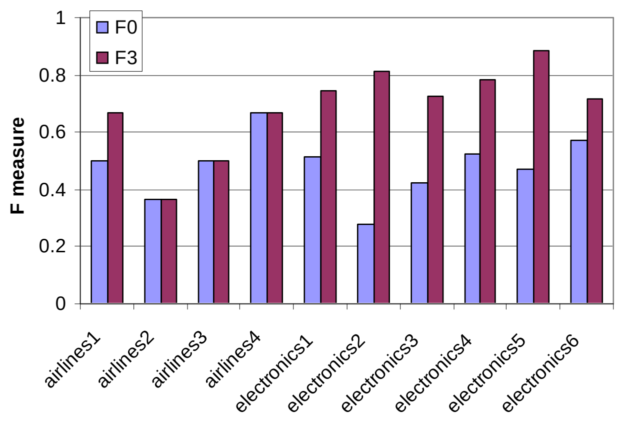
Geospatial domain: Eight sources, 21 test attributes extracted: latitude, distance, streetaddress, zipcode, . . .

Electronics domain: Twelve sources related to electronics equipment. 137 attributes extracted, such as modelname, manufacturer, modelnumber, hsize, vsize, brightness, power, colordepth, scanresolution, weight . . .

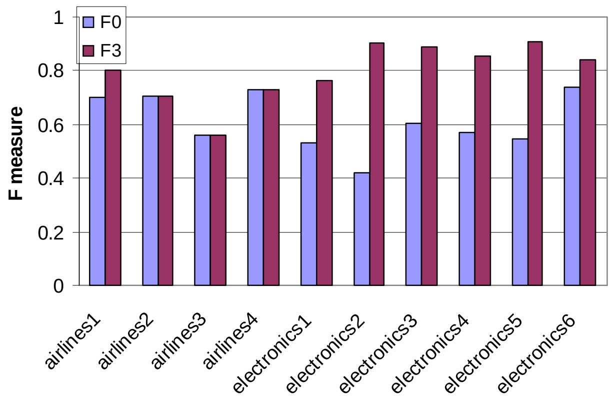
Cars domain: Five sources with 23 test attributes extracted: make, model, year, color, engine, price, mileage, bodystyle. Because most of the data was alphabetic, and each site had its own capitalization convention, we converted all data to lower case strings.

Weather domain: Ten sources with 87 test attributes: temperatureF, tempInF, sky, wind, pressure, humidity, . . .

Figure 4 shows performance of the content-based classification algorithm on data extracted from a variety of sources. The F-measure is a popular evaluation metric that combines recall and precision. Precision measures the fraction of the labeled data types that were correctly labeled, while recall measures the fraction of all data types that were correctly labeled. F0 refers to the case where the top-scored prediction of the classifier



(a)



(b)

Figure 5: Performance of content-based classifier on data from electronics and airlines domains when (a) considering all semantic types in classification, (b) when restricting semantic types to the domain of the source

was correct, and in results marked F3, the correct label was among the four top-scoring predictions. We believe that even if the algorithm cannot guess the semantic data type correctly, if it presents the user with a handful of choices, which contain the correct type, this will speed up the source modeling task.

Content-based classification attains very good performance labeling 150 fields from data sources shown in Figure 4, scoring at least $F = 0.80$ on 12 of the 16 sources. Performance on the remaining four sources suffered mainly due to variations in the data format: e.g., we learned capitalized city names, while the source represented them in all capitalized letters. Date and time were also difficult to recognize, because there is a great variety of formats, and chances were low that we learned date and time in the same format as on the new source.

Two other domains (representing 181 data fields) proved difficult to label, as shown in Figure 5(a). The electronics equipment domain was very hard. Most of the fields in the training data contained only numeric tokens (e.g., "1024" for horizontal display size). The learned patterns did not contain units information; therefore, they could not discriminate well between fields. In the airlines domain, precision suffered for the following reasons: non-standard capitalization of text fields, such as flightstatus, and incompatible date formats in training and test data. One way to improve precision and recall is by considering semantic types from a single domain only. The domain of a Web service can be identified in a number of ways, most directly from keywords in the WSDL file, or by analyzing semantic types assigned to data fields by the classifier and choosing the domain to which most of the semantic types belong. Figure 5(b) shows results of restricting the content-based classification algorithm to semantic types within one domain only (e.g., electronics equipment). As expected, performance of the algorithm improves.

3 Semantically Labeling Web Services

We validated our approach by semantically labeling live Web services in the Weather and Geospatial domains. Since quite a few services restrict access to registered users, we only tested those services for which we could obtain license keys. Furthermore, we filtered out services that only provide information about places outside of the United States, because these services included semantic types or data formats that were not in our domain model, or have different data coverage from the data types in our domain model. For example, even if we guess that "PostalCode" is equivalent to Zipcode, our domain model may not have any examples of British postal codes with which to query the Web service. This left eight invocable services. Several of the services supported more than one operation; therefore, we attempted to semantically model data types used by each of the 13 operations.[4]

3.1 Classifying Input Data

We classified the input parameters used by the services. Our only information about the service is the WSDL file; therefore, we employ metadata-based classification described in Section 2.1 to semantically label input parameters. We retrieved each service's WSDL file and processed it to extract features associated with each input and output. Next, we classified them using the Logistic Regression classifier (trained on all WSDL files from Bindingpoint and Webservicex directories). We then generated sample data for the input parameters. We took classifier's top guess as the semantic type of input parameter and chose a random sample of this type from data in the domain model.[5] We invoked the service with sample input data. If classifier's top guess was Unknown (the semantic type outside of our domain model), we left the corresponding input parameter blank. If we could not invoke the Web service successfully with the top guesses for the input parameters, we went back to classifier results and chose the next best guess. Clearly, if there are many input parameters, this combinatorial search will be expensive to execute. Fortunately, in the chosen domains, each Web service had one or two mandatory input parameters.

We classified 47 input parameters. The classifier chose Unknown as the most likely class for four inputs from two services. For one of them (Arcweb's AddressFinder), the Unknown parameters were optional (intersection, zipcode), and the service was successfully invoked. The other service (Terraworld) could not be invoked because a mandatory input parameter (streetaddress) was misclassified.

3.2 Invoking Services

Next, we attempted to invoke each service to verify the classifier predictions, as well as collect samples of output data. If a particular input parameter has predefined options listed in the WSDL file, we choose these values as inputs to query the source, regardless of types predicted by the classifier. If the input parameter has no predefined options, we take the most likely class predicted by the classifier and randomly choose a few samples from the examples of that semantic type in our domain model. If the class was Unknown, we leave that parameter empty. In many cases, a semantic type will have many different formats, with the service recognizing some formats but not others. For example, some service may request the State as a full name, and another as a two-character abbreviation. In such a case, we will have to try querying the service with all formats. Fortunately, for the services under investigation, this was not the case. Many of the operations required Zipcode or Address as input.

If service invocation is successful, it returns structured data according to the schema specified in the WSDL; otherwise, it returns an error. In that case, we go back to the classification results and try the next best result, and repeat the procedure. If there are several input parameters, it will obviously take many queries to find the correct combinations of parameter types. Fortunately, many of the services have only a few — mostly one or two — mandatory input parameters. In fact, we were able to successfully invoke seven of the eight services (or 12 of 13 operations) on the first try.

3.3 Classifying Output Data

We collected 3–6 samples for each output parameter and classified them using content-based classification algorithm described in Section 2.2. The classifier labeled 168 out of 213 output parameters. The classifier's top prediction was correct in 107 cases, leading to accuracy $A = 0.50$. These results are significantly worse than ones quoted in the previous section. This is due to an order of magnitude difference in sample sizes.

[4]Our services included commercial services, such as Arcweb and DEVPRIM geolocation services, DOTS and EJSE weather services, and a handful of free online services.

[5]These data come previous queries to known sources which have been stored in a local database.

Table 4: Input and output parameter classification results

classifier	total	correct	accuracy
	(a) intput parameters		
metadata-based	47	43	0.91
	(b) output parameters		
content-based	213	107	0.50
metadata-based	213	145	0.68
combined	213	171	0.80

We believe that performance will improve once we automate the data collection process and obtain more data samples from the services.

3.4 Combining Metadata and Content-based Classification

Sometimes the metadata-based classifier produced more specific semantic types: Sunrise rather than Time recognized by the content-based classifier, or Dewpoint rather than Temperature. At other times, especially when the service uses novel terms in parameter names, the content-based classifier outperforms the metadata-based classifier. Therefore, combining them may improve the accuracy of the output parameter labeling. We tested this hypothesis by using the metadata-based classifier to predict semantic types of the output parameters. The classifier's top prediction was correct in 145 cases ($A = 0.68$). Sixty one of the incorrect top-rated guesses were attributed to the Unknown class. This suggests a strategy of combining results of metadata and content-based classification to improve the performance of the semantic labeling algorithm. Namely, we take the most likely prediction produced by the metadata-based classifier as the semantic type of the output parameter, unless it happens to be Unknown. In that case, we use the top prediction of the content-based classifier. If content-based algorithm did not assign that output parameter to any semantic types, we look at the second-rated prediction of the metadata-based classifier. The combined approach correctly classifies 171 output parameters ($A = 0.80$), significantly outperforming individual classifiers (Table 4).

4 Semantic Modeling in CALO

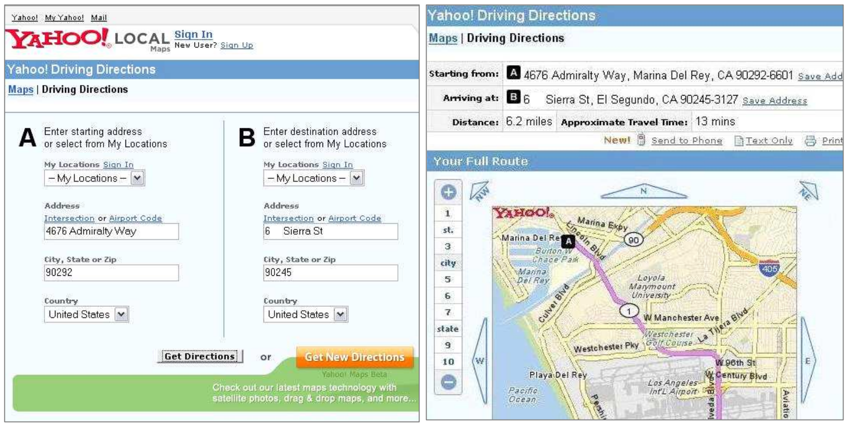
The goal of the Cognitive Assistant that Learns and Organizes (CALO) project is "to create cognitive software systems, that is, systems that can reason, learn from experience, be told what to do, explain what they are doing, reflect on their experience, and respond robustly to surprise."[6]

IRIS[7] is desktop application developed by the CALO project that, among other things, allows the system to learn procedures for complex office tasks, such as arranging a trip task described in the introduction. Machine learning technologies are integrated in many places within IRIS to improve the system's performance, capability, robustness and flexibility. Different research groups are providing learning technologies for CALO to create, or customize, new procedures by instruction, demonstration or observation. An equally important goal of the CALO project is to provide an easy-to-use, intuitive interface so that even users with little programming or technical expertise can effectively use the system. Here too machine learning is used to simplify user interactions with the system as well as hide some of the complexity.

[6]http://www.ai.sri.com/project/CALO
[7]http://www.openiris.org

4.1 Primitive Task Learning

We have created a Primitive Task Learning (**PrimTL**) plug-in for IRIS that allows the user to dynamically integrate new information sources into CALO as new primitive tasks. **PrimTL** applies the semantic labeling technology to ordinary Web sites rather than Web services. Since metadata for form-based sites is not readily available,[8] we rely instead on content-based classification. First, the user invokes EZBuilder, a tool created by Fetch Technologies, to wrap an information source. The user demonstrates to EzBuilder how to navigate the Web site to get to the required information. It the Web site provides access to data through Web forms, the user will need to provide several examples of how to query the Web site. Once the he gets to the results pages, user specifies what data he wants the wrapper to extract and trains the wrapper. EzBuilder produces a Web wrapper that can query a Web site and extract data from it. After the wrapper has been trained, **PrimTL** automatically links user-supplied inputs and extracted outputs to the semantic types in the CALO ontology. **PrimTL** also registers the newly created wrapper as a fully typed primitive information gathering task. The newly learned tasks can now be assembled into complex procedures by other CALO components.



Address: 4676 Admiralty Way
City, State or Zip: 90292
Country: United States

Address: 6 Sierra St
City, State or Zip: 90245
Country: United States

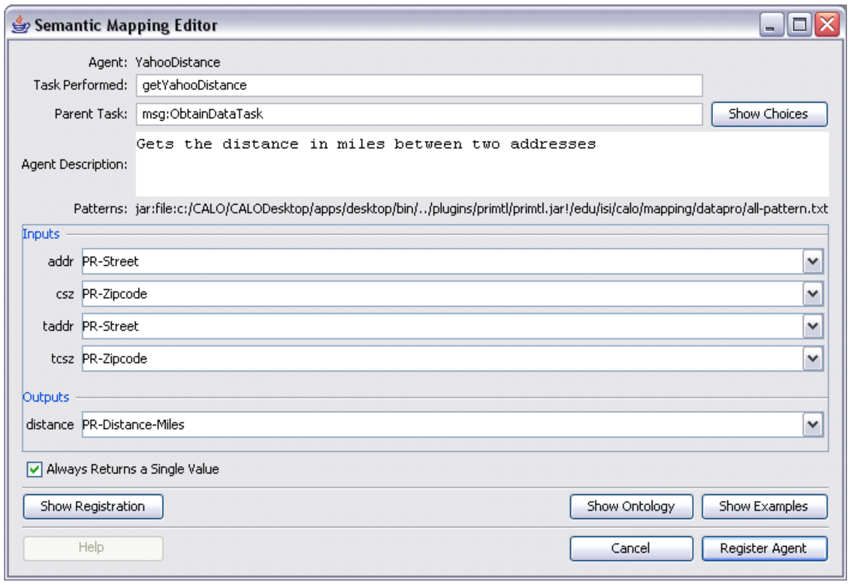
Starting from: A 4676 Admiralty Way, Marina Del Rey, CA 90292-6601 Save Add
Arriving at: B 6 Sierra St, El Segundo, CA 90245-3127 Save Address
Distance: 6.2 miles Approximate Travel Time: 13 mins

Figure 6: Yahoo directions source is a form-based Web site that returns driving directions (with distance) between two addresses

As an example, consider again the steps involved in arranging a trip. In order to find a hotel near the meeting site, IRIS needs to access a source that returns distance between two addresses. Yahoo driving directions source shown in Figure 4.1 is one of the sources that can provide this functionality. This source has a form-based interface, where the user can enter the two addresses (with zipcodes) and get the driving directions in the results page. The result also contains the total distance in miles between the addresses. Using EzBuilder, the user captures a few sample pages by filling out the form with different pairs of addresses. Next, the user trains the wrapper to extract distance in miles by showing it where this string can be found on the sample pages. Once the pages have been labeled, EzBuilder can learn extraction rules that will accurately extract data from all similar pages from the site.

After EzBuilder has learned extraction rules, IRIS launches Semantic Mapping Editor, shown in Figure 4.1, which allows the user to semantically annotate the source. Semantic Mapping Editor reads data collected by EzBuilder, which includes the input data the user typed into the form as well as data extracted from the result pages, and links them to types in the CALO ontology. The labels for the fields are extracted from the form (for inputs) or schema names defined by the user (for outputs). Semantic Mapping Editor presents top four choices for each input and output parameter to the user. For Yahoo driving directions, Semantic Mapping Editor correctly mapped "addr" and "taddr" to Street and strings "csz" and "tcsz" to Zipcode. The single output was mapped to DistanceInMiles. Had none of the top choices contained the correct type, the user could manually specify the correct semantic type for that element by launching the CALO ontology (through the "Show Ontology" button) and browsing it to find the correct type.

[8]One could extract labels from the input fields of the form, but this technology has not been developed.


Semantic Mapping Editor

Agent: YahooDistance

Task Performed: getYahooDistance

Parent Task: msg:ObtainDataTask

Agent Description: Gets the distance in miles between two addresses

Patterns: jar:file:c:/CALO/CALODesktop/apps/desktop/bin/../plugins/primtl/primtl.jar!/edu/isi/calo/mapping/datapro/all-pattern.txt

Inputs

addr	PR-Street
csz	PR-Zipcode
taddr	PR-Street
tcsz	PR-Zipcode

Outputs

distance	PR-Distance-Miles

[✓] Always Returns a Single Value

Figure 7: Semantic Mapping Editor frame showing semantically labeled Yahoo driving directions source. The semantic types of the input and output parameters were automatically computed from the data extracted from the source, while the user supplied the task annotations.

PrimTL with its Semantic Mapping technology has been integrated into CALO, where it is being used to semi-automatically generate procedures, such as those for arranging a trip [2]. Further enhancements to the system and integration with other components will be made in the near future.

4.2 Semantically Mapping Web Sources in CALO

We used the PrimTL plug-in to create wrappers for HTML-based information sources within the CALO project and to map the input and output parameters used by these sources to the CALO ontology. These sources were used to support a variety of procedures, such as, “Build a contact sheet listing contact information for attendees of a meeting,” or “Find hotels within 3 miles of the meeting site that have a pool and fast Internet connection.” In order to create a procedure that could complete one of the requests, the user had to create wrappers to extract data from relevant sources: directory or weather service, hotel or airline site, a service that returns distance between two points. The wrappers were then semantically labeled and manually assigned to a task in the Semantic Mapping Editor.

Figure 4.2 shows results of applying content-based classification algorithm to semantically label the input and output parameters of nine information sources. The correct semantic type was often the top prediction, and for more than half of the sources, the correct semantic type was among the top four predictions. The remaining cases achieved a decent $F \geq 0.8$, especially, when considering the top four predictions. These results show we can reliably use our approach to automate semantic modeling of information sources.

5 Related Work

The problem of identifying data types used by Web Services based on metadata is similar to problems in Named Entity Recognition, Information Extraction and Text Classification. Approaches that exploit surrounding information, such as adjacent tokens, local layout [13] and word distributions [1], have been used to assign a most likely class to the entity of interest. There are some aspects to the problem of

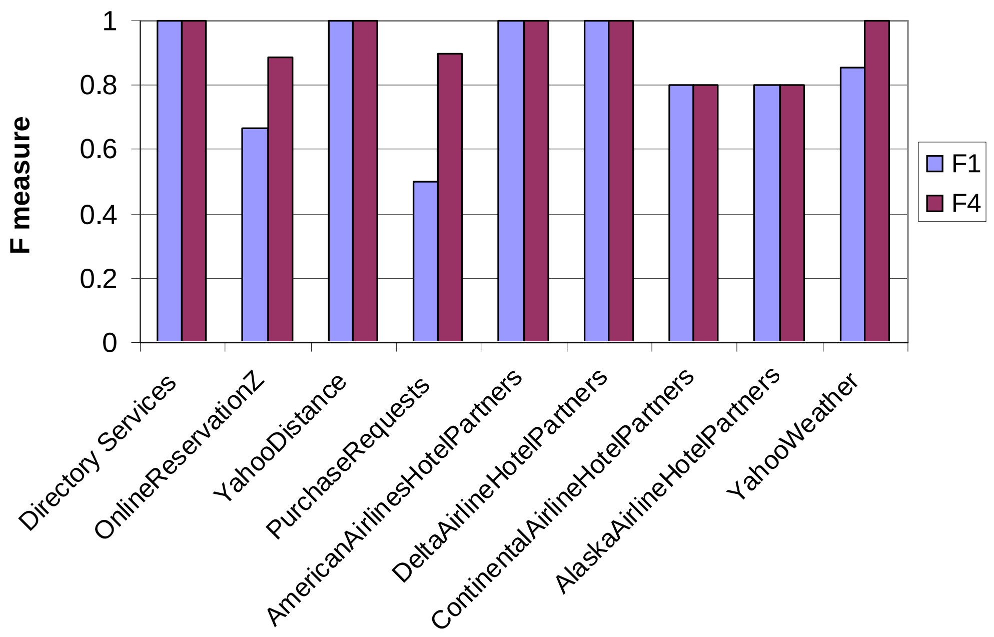


Figure 8: Results of semantically mapping input and output parameters of HTML-based information sources in the CALO project

classifying semantic data types of Web Services that make it distinct from those mentioned above. Usually, fewer tokens are used in naming a data type compared to those in documents. Even if one uses tokens from the corresponding Web Service messages and operations, the number is still small. We are, therefore, short on features useful for classifying data types. Secondly, texts in a WSDL file are generally ungrammatical, noisy and varied. Such situations cannot be tackled by previous solutions.

As information growth continues to yield abundant corpus samples, statistical machine learning approaches have been widely adopted. Various techniques have been developed to address the problems of data sparseness, inconsistency and noise. Several researchers have tried to categorize, or attach semantic information to, Web Service operations and data types. With respect to data type classification, rather than use metadata from WSDL files directly, [8] used those from HTML forms and metaphorically treated Web Form fields as the Web Service's parameters. Their assumption was based on a stochastic generative model. For a particular data type (or form field) D, Web Service developers use terms t drawn from a probability distribution $P(t|D)$ to name the semantic data type. Since it is infeasible to estimate parameters for $P(t|D)$ due to the terms' sparseness and huge vocabulary size, Hess and Kushmerick used the Naive Bayes' assumption to estimate $P(t|D)$: given a particular data type, terms are independent of each other. As shown in Section 2.1.2, the trained classifier using this methodology does not produce accurate enough predictions (probably due to the independence assumption and data sparseness) that can be used to automatically query the source. [7] proposed an unsupervised approach (Woogle) to cluster data types. The method is based on agglomerative clustering techniques which merge and split clusters using cohesion and correlation scores (distances), computed from the co-occurrence of terms within data types. This approach, however, has different a objective than ours: Woogle can roughly identify similarities or differences of data types; meanwhile, we need to still know the exact class in order to actively query a Web Service. In the Woogle approach, Zip, City and State data types might be in the same group, Address, since they tend to occur together often. Meanwhile, we need to know exactly whether the parameter is Zip or City in order invoke the service correctly.

Our work is similar to schema matching or integration [24, 5], where the objective is to produce a semantic mapping between instances of data from the two schemas. This is an important task in information integration, since queries expressed in some common language must be translated to the local schema of the database before they are submitted to the database. Past work on schema matching [17, 6, 5] included machine learning techniques that learn to classify new object instances based on features that include local schema names and content features. The content features used in these works are global in nature, such as word frequencies and format. Our approach, on the other hand, uses finer-grained descriptions enabled by the use of patterns to describe the structure of data. Another distinction is that unlike schema matching where the data is usually available, we need to actively retrieve the data by invoking services.

The schema matching community has used content- or instance-based matching [24] for aligning database columns of different database schemas. This technique is based on an observation that data instances of the same semantic class tend to have similar data syntax or pattern, and is thus similar to our content-based labeling approach. In those works, data patterns for particular fields are represented by numeric values, e.g. ratios of the number of numerical characters, statistics on data length, average value, variance, measurement units and so on, are used as features for discriminating semantic classes using supervised learning [18]. In unsupervised approach [11, 23], column matching is based on column-wise similarity scores such as those given by mutual information.

Johnston and Kushmerick [12] express the problem of aggregating data from Web services as a schema matching problem and introduce the OATS system that uses ensembles of distance metrics to match instance data. In other words, OATS chooses an appropriate distance metric based on whether the field is numeric or string, and can learn the appropriate distance metrics from the data. Hess et al. [9] integrated the OATS ensemble of distance metrics approach into ASSAM, a tool that assists a user in semantically labeling Web service parameters. This tool has a similar function to the Semantic Mapping Editor we described above. ASSAM uses an ensemble of classifiers to predict how various elements of the WSDL should be annotated. They then use the semantically annotated Web services for data aggregation. Our content-based classification algorithm is much more expressive for string-based fields, but does not handle the finer variations in numeric fields as well as OATS. Our goal for future work is to improve expressive power of the representation for numeric fields. Another difference from these works and ours is that they do not automatically learn how to invoke the service.

6 Conclusion

We presented the problem of automatically learning semantic types of data used by information sources. This is an integral part of learning a semantic model of an information source to enable it to be programatically invoked. We described metadata-based classification algorithm, that can be used to assign semantic types to the source's data parameters based on its metadata. This algorithm represents each parameter by a set of features — terms extracted from the service's WSDL file, for example, or HTML-based form. To test the classifier's predictions, we invoked the source with sample data of the predicted type. If the prediction was correct, the Web source returned output data. We then used content-based classifier to assign semantic types to the outputs. We evaluated performance of both classification algorithms on a variety of domains. Next, we validated our approach by semantically modeling several Web services in the Weather and Geospatial information domains. In addition to Web services, we applied our approach to semantically model the data used by HTML-based Web sources. The performance of the two classifiers was very good (especially when combined), showing that they can be used to accurately and automatically label data used by information.

In this study we classified each parameter independently of the others, specifically, regardless of semantic types of adjacent parameters within the message or operation. Collective classification methods, for example relaxation labeling [10, 4], could be applied to classify a whole set of input or output parameters of a certain message. We believe that the collective classification would improve both precision and recall scores and, thus, decrease times used in probing source whether a set of predicted classes is correct. Meta-analysis of content-based classification results can help improve them, by eliminating duplicates, for example.

As the number of known semantic types grows, we are faced with a performance problem, in both time and accuracy, of the content-based classifier. Take, for example, Date semantic type: it consists of Day, Month and Year. There are several ways to represent each atomic type (e.g., Month, have a few different formats: "6,"

”06,” “Jun,” and “June”), and several ways to combine them (“/”, “,”, etc). Other complications include reversal of Month and Day (European vs American standards) and optional parts, such as DayofWeek, and even Time. The pattern learning algorithm needs to have seen many examples of each format; otherwise, it may not be able to recognize a new example of a Date because it had simply not seen Date represented in that format before. In addition, the domain model will grow very large, because it needs to represent each format with at least several patterns, leading to performance deterioration. We can, however, efficiently represent and recognize the individual components of the field and deduce that the complex type, just because we have seen this combination used together in other sources. We will, therefore, develop a for representing and recognizing complex data types.

Acknowledgements

We are grateful to Tom Russ and Dipsy Kapoor for implementing Semantic Mapping Editor and integrating it within IRIS, and to Jim Blythe for incorporating the learned source models in the new procedures. This research is based by work supported in part by the NSF under Award Nos. IIS-0324955 and CNS-0509517, in part by the Air Force Office of Scientific Research under grant number FA9550-04-1-0105, in part by DARPA under Contract No. NBCHD030010. We would like to thank ArcWeb for providing us with access to their data.

The U.S.Government is authorized to reproduce and distribute reports for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of any of the above organizations or any person connected with them.

References

[1] D. L. Baker and K. A. McCallum. Distributional clustering of words for text classification. In *In Proc. of ACM SIG on Information Retrieval (SIGIR-1998)*. ACM Press, 1998.

[2] Jim Blythe. Building information integration procedures through instruction. In *submitted to Int. Joint Conference on Artificial Intelligence (IJCAI-06)*, Hyderabad, India, 2006.

[3] Mark Carman and Craig Knoblock. Learning semantic descriptions forweb information sources. In *submitted to Int. Joint Conference on Artificial Intelligence (IJCAI-07)*, Hyderabad, India, 2007.

[4] S. Chakrabarti, B. Dom, and P. Indyk. Enhanced hypertext categorization using hyperlinks. In *In Proc. of SIGMOD 98*. ACM Press, 1998.

[5] A. Doan, P. Domingos, and A. Halevy. Learning to match the schemas of databases: A multistrategy approach. *Machine Learning Journal*, 50:279–301, 2003.

[6] AnHai Doan, Pedro Domingos, and Alon Y. Halevy. Reconciling schemas of disparate data sources: A machine-learning approach. In *SIGMOD Conference*, pages 509–520, 2001.

[7] X. Dong, A. Halevy, J. Madhavan, E. Nemes, and J. Zhang. Similarity search for web services. In *In Proceedings of the International Conference on Very Large Databases (VLDB-2004)*. ACM Press, 2004.

[8] A. Hess and N. Kushmerick. Learning to attach semantic metadata to web services. In *Proceedings 2nd International Semantic Web Conference (ISWC2003)*, 2003.

[9] Andreas Hess, Eddie Johnston, and Nicholas Kushmerick. Assam: A tool for semi-automatically annotating semantic web services. In *Proc. Int. Semantic Web Conf.*, 2004.

[10] R. A. Hummel and S. W. Zucker. On the foundations of relaxation labeling processes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 5(3), 1983.

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

[12] Eddie Johnston and Nicholas Kushmerick. Aggregating web services with active invocation and ensembles of string distance metrics. In *Proc. Int. Conf. Knowledge Engineering and Knowledge Management*, 2004.

[13] Kristina Lerman, Lise Getoor, Steven Minton, and Craig A. Knoblock. Using the Structure of Web Sites for Automatic Segmentation of Tables. In *In* Proceedings of ACM SIG on Management of Data (SIGMOD-2004), jun 2004.

[14] Kristina Lerman, Steven Minton, and Craig Knoblock. Wrapper maintenance: A machine learning approach. *Journal of Artificial Intelligence Research*, 18:149–181, 2003.

[15] Kristina Lerman, Steven Minton, and Craig Knoblock. Wrapper maintenance: A machine learning approach. *Journal of Artificial Intelligence Research*, 18:149–181, 2003.

[16] Kristina Lerman, Anon Plangrasopchok, and Craig A. Knoblock. Automatically labeling the inputs and outputs of web services. In *In Proceedings of the National Conference on Artificial Intelligence (AAAI-2006)*, Menlo Park, CA, 2006. AAAI Press.

[17] 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.

[18] W.-S. Li and C. Clifton. Semantic integration in heterogeneous databases using neural networks. In *In Proceedings of the 20th VLDB Conference*, pages 49–84, 1994.

[19] T. P. Minka. A comparison of numerical optimizers for logistic regression. Technical report, Microsoft Corp., 2004. http://research.microsoft.com/ minka/papers/logreg/minka-logreg.pdf.

[20] T.P. Minka. A comparison of numerical optimizers for logistic regression, 2003. http://research.microsoft.com/minka/papers/logreg/.

[21] T. Mitchell. *Machine Learning*. 2nd edition (draft) edition, 2005. Chapter 1: Generative and Discriminative Classi- fiers: Nave Bayes and Logistic Regression.

[22] A. Y. Ng and M. I. Jordan. On discriminative vs. generative classifiers: A comparison of logistic regression and nave bayes. In *In Proc. of Neural Information Processing Systems (NIPS02)*. ACM Press, 2002.

[23] Patrick Pantel, Andrew Philpot, and Eduard Hovy. An information theoretic model for database alignment. In *In Proceedings of the SSBDM conference*, 2005.

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

[25] Snehal Thakkar, Jose Luis Ambite, and Craig A. Knoblock. Composing, optimizing, and executing plans for bioinformatics web services. *VLDB Journal, Special Issue on Data Management, Analysis and Mining for Life Sciences*, 14(3):330–353, 2005.