Machine learning techniques for annotating semantic web services

Andreas Heß   Eddie Johnston   Nicholas Kushmerick
Computer Science Department, University College Dublin
{andreas.hess, eddie.johnston, nick}@ucd.ie

Introduction

The vision of semantic Web Services is to provide the means for fully automated discovery, composition and invocation of loosely coupled software components. One of the key efforts to address this “semantic gap” is the well-known OWL-S ontology (The DAML Services Coalition 2003).

However, software engineers who are developing Web Services usually do not think in terms of ontologies, but rather in terms of their programming tools. Existing tools for both the Java and .NET environments support the automatic generation of WSDL. We believe that it would boost the semantic service web if similar tools existed to (semi-) automatically generate OWL-S or a similar form of semantic metadata.

In this paper we will present a tool called ASSAM—Automated Semantic Service Annotation with Machine Learning—that addresses these needs. ASSAM consists of two parts, a WSDL annotator application, and OATS, a data aggregation algorithm.

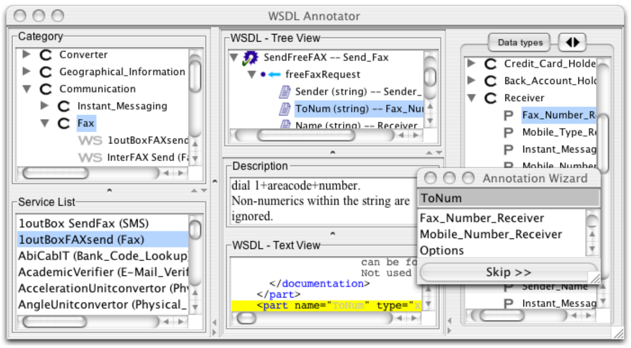
First, we describe the WSDL annotator application. This component of ASSAM uses machine learning to provide the user with suggestions on how to annotate the elements in the WSDL. In go on to describe the iterative relational classification algorithm that provides these suggestions. We evaluate our algorithms on a set of 164 Web Services.[1]

Second, we describe OATS, a novel schema mapping algorithm specifically designed for the Web Services context, and empirically demonstrate its effectiveness on 52 invokable Web Service operations. OATS addresses the problem of aggregating the heterogenous data from several Web Services.

ASSAM: A Tool for Web Service Annotation

One of the central parts of ASSAM is the WSDL annotator application. The WSDL annotator is a tool that enables the user to semantically annotate a Web Service using a point-and-click interface. The key feature of the WSDL annotator is the ability to suggest which ontological class to use to annotate each element in the WSDL.

Fig. 1 shows the ASSAM application. Note that our application’s key novelty—the suggested annotations created au-



dial 1+areacode+number.
Non-numerics within the string are
ignored.

```
                     can be fo
                     Not used
        </documentation>
      </part>
      <part name="ToNum" type="x
```

Figure 1: ASSAM uses learning techniques to semi-automatically annotate Web Services with semantic metadata.

tomatically by our machine learning algorithm—are shown in the small pop-up window.

Once the annotation is done it can be exported in OWL-S. The created OWL-S consists of a profile, a process model, a grounding and a concept file if complex types where present in the WSDL. Note that this also includes XSLT transformations as needed in the OWL-S grounding to map between the traditional XML Schema representation of the input and output data and the OWL representation.

Limitations. Because we do not handle composition and workflow in our machine learning approach, the generated process model consists only of one atomic process per operation. The generated profile is a subclass of the assigned category of the service as a whole – the category ontology services as profile hierarchy. The concept file contains a representation of the annotated XML schema types in OWL-S. Note that it is up to the ontology designer to take care that the datatype ontology makes sense and that it is consistent. No inference checks are done on the side of our tool. Finally, a grounding is generated that also contains the XSLT mappings from XML schema to OWL and vice versa.

For the OWL export, we do not use the annotations for the operations at the moment, as there is no direct correspondence in OWL-S for the domain of an operation. Atomic processes in OWL-S are characterized only through their in-

[1] All our our experimental data is available in the Repository of Semantic Web Services `smi.ucd.ie/RSWS`.

puts, outputs, preconditions and effects; and for the profile our tool uses the service category.

Related Work. (Paolucci *et al.* 2003) addressed the problem of creating semantic metadata (in the form of OWL-S) from WSDL. However, because WSDL contains no semantic information, this tool provides just a syntactic transformation. The key challenge is to map the XML data used by traditional Web Services to classes in an ontology.

Currently, (Patil *et al.* 2004) are also working on matching XML schemas to ontologies in the Web Services domain. They use a combination of lexical and structural similarity measures. They assume that the user's intention is not to annotate similar services with one common ontology, rather they also address the problem of choosing the right domain ontology among a set of ontologies.

(Sabou 2004) addresses the problem of creating suitable domain ontologies in the first place. She uses shallow natural language processing techniques to assist the user in creating an ontology based on natural language documentation of software APIs.

Iterative Relational Classification

For our learning approach, we cast the problem of classifying operations and datatypes in a Web Service as a text classification problem. Our tool learns from Web Services with existing semantic annotation. Given this training data, a machine learning algorithm can generalize and predict semantic labels for previously unseen Web Services.

In a mixed-initiative setting, these predictions do not have to be perfectly accurate to be helpful. In fact, the classification task is quite hard, because the domain ontologies can be very large. But for that reason it is already very helpful for a human annotator if he or she would have to choose only between a small number of ontological concepts rather than from the full domain ontology. In previous work (Heß & Kushmerick 2003) we have shown that the category of a services can be reliably predicted, if we stipulate merely that the correct concept be one of the top few (e.g., three) suggestions.

The basic idea behind our approach is to exploit the fact that there are dependencies between the category of a Web Service, the domains of its operations and the datatypes of its input and output parameters. Our algorithm is based on a set of features of the services, operations and parameters. Following Neville and Jensen (Neville & Jensen 2003), we distinguish between *intrinsic* and *extrinsic* features. The intrinsic features of a document part are simply its name and other text that is associated with it (e.g., text from the occasional `documentation` tags). Extrinsic features derive from the relationship between different parts of a document. We use the semantic classes of linked document parts as extrinsic features.

Initially, when no annotations for a service exist, the extrinsic features are unknown. After the first pass, where classifications are made based on the intrinsic features, the values of the extrinsic features are set based on the previous

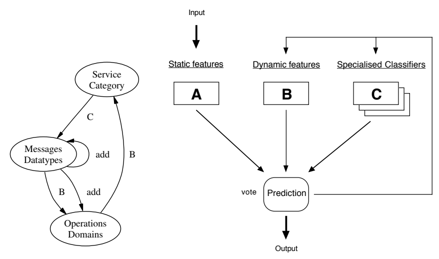


Figure 2: Feedback structure and algorithm.

classifications. Of course, these classifications may be partially incorrect. The classification process is repeated until a certain termination criterion (e.g. either convergence or a fixed number of iterations) is met. Fig. 2 shows an illustration of the classification phase of the algorithm.

For a more detailed discussion of our algorithm and the way it differs from other iterative algorithms the reader is referred to our paper (Heß & Kushmerick 2004) that describes the algorithm in greater detail from a machine learning point of view.

Evaluation. We evaluated our algorithm using a leave-one-out methodology. We compared it against a baseline classifier with the same setup for the static features, but without using the dynamic extrinsic features.

To determine the upper bound of improvement that can be achieved using the extrinsic features, we tested our algorithm with the correct class labels given as the extrinsic features. This tests the performance of predicting a class label for a document part when not only the intrinsic features but also the dynamic features, the labels for all other document parts, are known.

We also compared it against a non-ensemble setup, where the extrinsic features are not added using a separate classifier but rather are just appended to the static features. Classification is then done with a single classifier. This setup closely resembles the original algorithm proposed by Neville and Jensen. Again, the same set of static features was used.

In the evaluation we ignored all classes with one or two instances, such as occurred quite frequently on the datatype level. The distributions are still quite skewed and there is a large number of classes. There are 22 classes on the category level, 136 classes on the domain level and 312 classes on the datatype level.

Fig. 3 show the accuracy for categories, domains and datatypes. As mentioned earlier, in mixed-initiative scenario such as our semi-automated ASSAM tool, it is not necessary to be be perfectly accurate. Rather, we strive only to ensure that that the correct ontology class is in the top few suggestions. We therefore show how the accuracy increases when we allow a certain tolerance. For example, if the accuracy for tolerance 9 is 0.9, then the correct prediction is within the top 10 of the ranked predictions the algorithm made 90% of the time.

We could not achieve good results with the non-ensemble setup. This setup scored worse than the baseline. For the

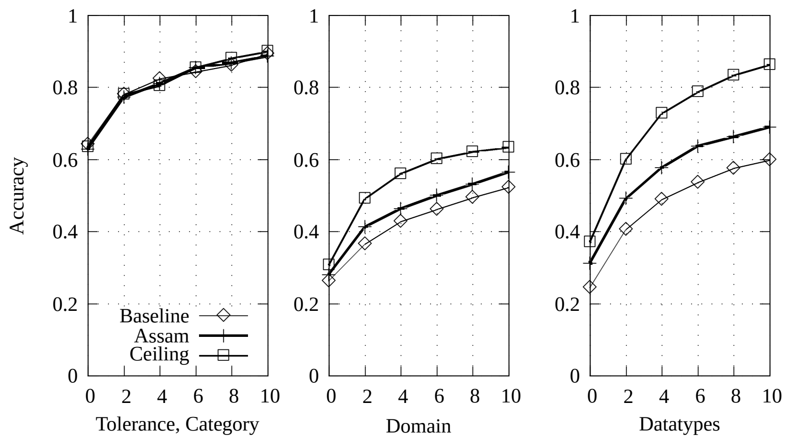


Figure 3: Accuracy of our algorithm on the three kinds of semantic metadata as a function of prediction tolerance.

datatypes, even the ceiling accuracy was below the baseline.

Related work. We already mentioned the algorithm by Neville and Jensen (Neville & Jensen 2000), but iterative classification algorithms were also used for link-based hypertext classification by Lu and Getoor (Lu & Getoor 2003). Relational learning for hypertext classification was also explored by Slattery et al., e.g. (Ghani, Slattery, & Yang 2001; Yang, Slattery, & Ghani 2002). A difference between their problem setting and ours is that the links in our dataset are only within one Web Services, where in the hypertext domain potentially all documents can link to each other.

Aggregating data from Web Services

ASSAM uses the machine learning technique just described to create semantic metadata that could assist (among other applications) a data integration system that must identify and invoke a set of Web Services operations that can answer some query. In order to automatically aggregate the resulting heterogeneous data into some coherent structure, we are currently developing OATS (Operation Aggregation Tool for Web Services), a schema matching algorithm that is specifically suited to aggregating data from Web Services.

While most schema matching algorithms don't consider instance data, those that do take as input whatever data happens to be available. In contrast, OATS actively probes Web Services with a small set of related queries, which results in contextually similar data instances and greatly simplifies the matching process. Another novelty of OATS is the use of ensembles of distance metrics for matching instance data to overcome the limitations of any one particular metric. Furthermore, OATS can exploit training data to discover which metrics are more accurate for each semantic category.

As an example, consider two very simple Web Service operations that return weather information. The first operation may return data such as

```
<weather><hi>87</hi><lo>56</lo>
              <gusts>NE, 11 mph</gusts></weather>
```

while the second operation may return data such as

```
<fcast><tmax>88</tmax><tmin>57</tmin>
           <wndspd>10 mph (N)</wndspd></fcast>
```

The goal of data aggregation is to consolidate this heterogeneous data into a single coherent structure.

The major difference between traditional schema matching and our Web Service aggregation task is that we can exert some control over the instance data. Our OATS algorithm probes each operation with arguments that correspond to the same real-world entity. For example, to aggregate operation O_1 that maps a ZIP code to its weather forecast, and operation O_2 that maps a latitude/longitude pair to its forecast, OATS could first select a specific location (e.g., Seattle), and then query O_1 with “98125” (a Seattle ZIP code), and query O_2 with “47.45N/122.30W” (Seattle's geocode). Probing each operation with the related arguments should ensure that the instance data of related elements will closely correspond, increasing the chances of identifying matching elements.

As in ILA (Perkowitz & Etzioni 1995), this probe-based approach is based on the assumption that the operations overlap—ie, there exists a set of real-world entities covered by all of the sources. For example, while two weather Web Service need not cover exactly the same locations in order to be aggregated, we do assume that there exists a set of locations covered by both.

The OATS algorithm. The input to the OATS algorithm is a set of Web Service operations $O = \{o_1, o_2, \ldots, o_n\}$, a set of probe objects $P = \{p_1, \ldots, p_m\}$, sufficient metadata about the operations so that each operation can be invoked on each probe ($V = \{v_1, \ldots, v_n\}$, where v_i is a mapping from a probe $p_k \in P$ to the input parameters that will invoke o_i on p_k), and a set of string distance metrics $D = \{d_1, d_2, \ldots\}$.

When invoked, an operation $o_i \in O$ generates data with elements $E_i = \{e^i_1, e^i_2, \ldots\}$. Let $E = \cup_i E_i$ be all the operations' elements. The output of the OATS algorithm is a partition of E.

One of the distinguishing features of our algorithm is the use of an ensemble of distance metrics for matching elements. For example, when comparing the `gusts` and `wndspd` instance data above, it makes sense to use a token based matcher such as TFIDF, but when comparing `hi` and `tmax`, an edit-distance based metric such as Levenshtein is more suitable. The OATS algorithm calculates similarities based on the average similarities of an ensemble of distance metrics. Later, we describe an extension to OATS which assigns weights to distance metrics according to how well they correlate with a set of training data.

The OATS algorithm proceeds as follows. Each of the n operations are invoked with the appropriate parameters for each of the m probe objects. The resulting nm XML documents are stored in a three-dimensional table T: $T[i,j,k]$ stores the value returned for element $e^i_j \in E_i$ by operation o_i for probe p_k.

Each element is then compared with every other element. The distance between an element pair $(e^i_j, e^{i'}_{j'}) \in E \times E$ is calculated for each string distance metric $d_\ell \in D$, and these values are merged to provide an ensemble distance value for these elements. The similarity between two elements $e^i_j \in E_i$ and $e^{i'}_{j'} \in E_{i'}$ is defined as $D(e^i_j, e^{i'}_{j'}) = \frac{1}{|D|}\sum_\ell(\bar{d}_\ell(e^i_j, e^{i'}_{j'}) - \mathrm{m}(\bar{d}_\ell))/R(\bar{d}_\ell)$, where $\bar{d}_\ell(e^i_j, e^{i'}_{j'}) = \frac{1}{m}\sum_k d_\ell(T[i,j,k], T[i',j',k])$, $\mathrm{M}(\bar{d}_\ell) = \max_{(e^i_j, e^{i'}_{j'})} \bar{d}_\ell(e^i_j, e^{i'}_{j'})$, $\mathrm{m}(\bar{d}_\ell) = \min_{(e^i_j, e^{i'}_{j'})} \bar{d}_\ell(e^i_j, e^{i'}_{j'})$, and $R(\bar{d}_\ell) = \mathrm{M}(\bar{d}_\ell) - \mathrm{m}(\bar{d}_\ell)$.

By computing the average distance $\bar{d}_\ell$ over m related sets of element pairs, we are minimizing the impact of any spurious instance data. Before merging the distance metrics, they are normalized relative to the most similar and least similar pairs, as different metrics produce results in different scales.

To get the ensemble similarity $D(e^i_j, e^{i'}_{j'})$ for any pair, we combine the normalized distances for each d_j. In the standard OATS algorithm, this combination is simply an unweighted average. We also show below how weights can be adaptively tuned for each element-metric pair.

Given the distances between each pair of elements, the final step of the OATS algorithm is to cluster the elements. This is done using the standard hierarchical agglomerative clustering (HAC) approach. Initially, each element is assigned to its own cluster. Next, the closest pair of clusters is found (using the single, complete, or average link methods) and these are merged. The previous step is repeated until some termination condition is satisfied. At some point in the clustering, all of the elements which are considered similar by our ensemble of distance metrics will be merged, and further iterations would only force together unrelated clusters. It is at this point that we should stop clustering. Our implementation relies on a user-specified termination threshold.

Weighted distance metrics. Instead of giving an equal weight to each distance metric for all elements, it would make sense to treat some metrics as more important than others, depending on the characteristics of the data being compared. We now show how we can exploit training data

address	city	state	fullstate	zip	areacode	lat	long	icao
110 135th Avenue	New York	NY	New York	11430	718	40.38	-74.75	KJFK
101 Harborside Drive	Boston	MA	Massachusetts	02128	781	42.21	-71.00	KBOS
18740 Pacific Highway South	Seattle	WA	Washington	98188	206	47.44	-122.27	KSEA
9515 New Airport Drive	Austin	TX	Texas	78719	512	30.19	-97.67	KAUS

Figure 4: The four probe objects for the zip and weather domains.

to automatically discover which distance metrics are most informative for which elements. The key idea is that a good distance metric will give a small value for pairs of semantically related instances, while giving a large value for unrelated pairs.

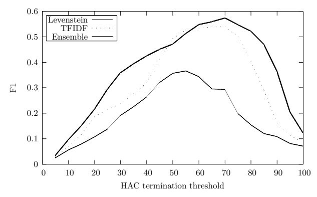
We assume access to a set of training data: a partition of some set of elements and their instance data. Based on such training data, the *goodness* of metric d_j for a non-singleton cluster C is defined as $G(d_j, C) = G'(d_j, C)/\frac{1}{c}\sum_{C'} G'(d_j, C')$, where c is the number of non-singleton clusters C' in the training data, $D_{\mathrm{intra}}(d_j, C)$ is the average *intra*-cluster distance—i.e., the average distance between pairs of elements within C, $D_{\mathrm{inter}}(d_j, C)$ is the average *inter*-cluster distance—i.e., the average distance between an element in C and an element outside C, and $G'(d_j, C) = D_{\mathrm{inter}}(d_j, C) - D_{\mathrm{intra}}(d_j, C)$. A distance metric d_j will have a score $G(d_j, C) > 1$ if it is "good" (better than average) at separating data from cluster C from data outside the cluster, while $G(d_j, C) < 1$ suggests that d_j is a bad metric for C.

Given these goodness values, we modify OATS in two ways. The first approach ("binary") gives a weight of 1 to metrics with $G > 1$, and ignores metrics with $G \leq 1$. The second approach ("proportional"), assigns weights that are proportional to the goodness values.

Evaluation. We evaluated our Web Service aggregation tool on three groups of semantically related Web Service operations: 31 operations providing information about geographical locations, 8 giving current weather information, and 13 giving current stock information. To enable an objective evaluation, a reference partition was first created by hand for each of the three groups. The partitions generated by OATS were compared to these reference partitions. In our evaluation, we used the definition of precision and recall proposed by (Heß & Kushmerick 2003) to measure the similarity between two partitions. The string distance metrics were selected from Cohen's SecondString library (Cohen, Ravikumar, & Fienberg 2003).

We ran a number of tests on each domain. We systematically vary the HAC termination threshold, from one extreme in which each element is placed in its own cluster, to the other extreme in which all elements are merged into one large cluster.

Each probe entity is represented as a set of attribute/value pairs. For example, Fig. 4 shows the four probes used for the weather and location information domains. We hand-crafted rules to match each of an operation's inputs to an attribute. To invoke an operation, the probe objects (ie, rows in Fig. 4) are searched for the required attributes.



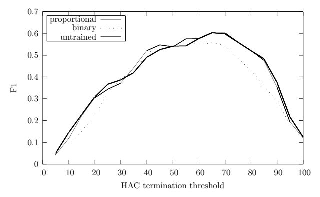


Figure 5: OATS ensemble vs. individual distance metrics (left); OATS with vs. without adaptive distance metric weighting (right).

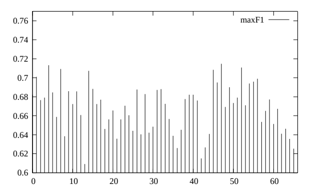
Results. First, we show that an ensemble of string metrics achieves better results than using the metrics separately. Fig. 5 (left) compares the ensemble approach to the Levenshtein and TFIDF metrics individually. We report the average performance over the three domains as F1 as a function of the HAC termination threshold. Note that, as expected, F1 peaks at an intermediate value of the HAC termination threshold. The average and maximum F1 is higher for the ensemble of metrics, meaning that it is much less sensitive to the tuning of the HAC termination threshold.

We now compare the performance of OATS with our two methods (binary and proportional) for using the learned string metric weights. These results are based on four probes. We used two-fold cross validation, where the set of operations was split into two equal-sized subsets, S_{train} and S_{test}. S_{train} was clustered according to the reference clusters, and weights for each distance metric were learned. Clustering was then performed on the entire set of elements. Note that we clustered the training data along with the test data in the learning phase, but we did not initialize the clustering process with reference clusters for the training data prior to testing. We measured the performance of the clustering by calculating precision and recall for just the elements of S_{test}. Fig. 5 (right) shows F1 as a function of the HAC termination threshold for the binary and proportional learners and the original algorithm. Although neither of the learning methods increase the maximum F1, they usually increase the average F1, suggesting that learning makes OATS somewhat less sensitive to the exact setting of the HAC termination threshold.

Active probe selection

Our experiments show that the accuracy of OATS improves with additional probes. Furthermore, some probes yield more informative output data than others—i.e. there can be a substantial difference in accuracy depending on the specific probes.

For example, Fig. 6 shows the variation in F1 when different combinations of probes (from a set of 12) were used to invoke the web services in one of our test domains. With 2 probes, there are $12!/2!(12-2!) = 66$ such choices, and F1 varies from 62% to 72%. With 6 probes, there are 924 choices and F1 ranges from 68% to 76%. These data demonstrate that performance generally increases with additional probing. More interestingly, they show that a small carefully chosen set of probes can be as effective as a much larger set



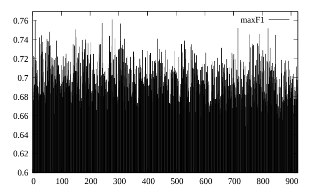


Figure 6: F1 variation for various combinations of 2 (left) and 6 (right) probes.

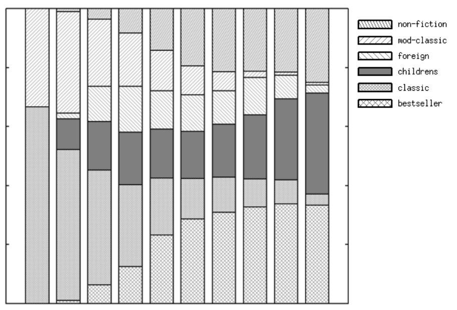


Figure 7: The fraction of books from each category, as a function of the average F1 resulting from probes selected with the given categories. The horizontal axis is F1, ranging from 33–43%; the vertical axis ranges from 0–100%.

chosen randomly. Given that each additional probe costs additional human effort as well as bandwith and processing, we are interested in exploring active approaches to Web Service aggegation that chose probes in order to maximize accuracy while minimizing cost.

The variation in performance of one set of probes compared to another could be due to a number of reasons, depending on the domain. For instance, Fig. 7 shows how the proportion of various genres of book probes changes as performance increases. In this case, invoking the Web Services with 'classic' probes (books such as *Oliver Twist*) results in poorer performance than is achieved if 'non-fiction' or 'bestseller' probes are used. Exactly why these probes are more effective is unclear. Perhaps the descriptions returned for bestsellers have fewer errors because they are deemed more important by the service providers?

Probes can interact. For example, perhaps probes p_1 and p_2 are both promising in isolation, but probing with both p_1 and p_2 offers no improvement, or even a decrease in accuracy. For example, in the weather domain, probing with multiple very close locations may yield non-discriminatory outputs which results in poorer results than would be returned using more discriminatory probes. Fig. 8 lists the data returned for 6 elements from 4 Web Services, for an initial probe with Fort Lauderdale, as well as the data return for three additional probes at varying distances.

The cost invested in the Miami probe is probably wasted, since each of the results is too similar to the initial result set and the same mistake would be made on each set. For example, `O2T2` (Relative Humidity) would be mistakenly

O1T1	O1T3	O2T2	O3T1	O4T2	O4T3
Ft Lauderdale	Florida	88	88	80.15	Florida
Miami	*Florida*	*88*	*87*	*80.19*	*Florida*
Jackson	*Florida*	*67*	*91*	*81.2*	*Florida*
Anchorage	*Alaska*	*-15*	*68*	*150.*	*Alaksa*

Figure 8: Data returned for probing four weather services for Fort Lauderdale, and 3 additional cities increasingly far from Fort Lauderdale.

matched with `O3T1` (Temperature). In this case, probing with only the intial probe would have returned the same result as with both probes but at half the cost. By examining the variation in performance resulting from various probe choices, it may be evident that additional probe objects should not be in Florida, should have a longitude that is substantially different from 80.1, etc.

There is of course a detailed explanation for these results: booksellers' pay more attention to the data for bestsellers than classics; nearby cities tend to have similar weather; etc. From our perspective, the ultimate explanation doesn't matter. Rather, our goal is to learn to exploit whatever regularities may exist. More concretly, can we devise an active learning algorithm that can automatically determine that, for example, in the books domain it is wise to avoid "classic" books, and that in weather domain it is best to avoid nearby cities? Armed with such knowledge, OATS could select its probes so as to reduce the total number of probes required, while maximizing accuracy.

A common approach to active learning (Cohn, Atlas, & Ladner 1994) is to select training data based on the learner's confidence: the learner is bootstrapped with some hand-classified instances from which it creates a classifier which is used to annotate each unlabelled instance. The instances with the lowest annotation certainty are then annotated by an expert and used to train the learner on the next iteration.

In our aggregation problem, we do not seek to annotate any instance data but wish to select the probe objects that will result in the highest quality data when used to invoke a set of web services. Instead of suggesting instance data that might potentially be misclassified, our aim would be to identify probes that will yiled data that can profitably be combined with data obtained from previous probes. We are currently exploring algorithms to address this problem.

Summary

We have presented ASSAM, a tool for annotating Semantic Web Services. We have presented the WSDL annotator application, which provides an easy-to-use interface for manual annotation, as well as machine learning assistance for semi-automatic annotation. Our application is capable of exporting the annotations as OWL-S.

We have also presented a new iterative relational classification algorithm that combines the idea of existing iterative algorithms with the strengths of ensemble learning. We have evaluated this algorithm on a set of real Web Services and have shown that it outperforms a simple classifier and that it is suitable for semi-automatic annotation.

Finally, we have described techniques for semantically aggregating the data returned from Web Services. Web Service aggregation is an instance of the schema matching problem in which instance data is particularly important. We have illustrated how actively probing Web Services with a small number of inputs can result in contextually related instance data which makes matching easier. Our experiments demonstrate how using an ensemble of distance metrics performs better than the application of individual metrics. We also proposed a method for adaptively combining distance metrics in relation to the characteristics of the data being compared. We have proposed to use active probe selection to choose highly-informative probes, yielding higher accuracy at lower cost. We are currently investigating adaptive algorithms that automatically discover domain-specific probe-selection strategies.

Acknowledgments. This research was supported by Science Foundation Ireland and the US Office of Naval Research.

References

Cohen, W. W.; Ravikumar, P.; and Fienberg, S. E. 2003. A comparison of string distance metrics for name-matching tasks. In *Int. Joint Conf. on AI, Workshop on Inf. Integr. on the Web*.

Cohn, D. A.; Atlas, L.; and Ladner, R. E. 1994. Improving generalization with active learning. *Machine Learning* 15(2):201–221.

Ghani, R.; Slattery, S.; and Yang, Y. 2001. Hypertext categorization using hyperlink patterns and meta data. In *18th Int. Conf. on Machine Learning*.

Heß, A., and Kushmerick, N. 2003. Learning to attach semantic metadata to web services. In *2nd Int. Sem. Web. Conf.*

Heß, A., and Kushmerick, N. 2004. Iterative ensemble classification for relational data: A case study of semantic web services. In *ECML*.

Lu, Q., and Getoor, L. 2003. Link-based classification. In *Int. Conf. on Machine Learning*.

Neville, J., and Jensen, D. 2000. Iterative classification in relational data. In *AAAI Workshop SRL*.

Neville, J., and Jensen, D. 2003. Statistical relational learning: Four claims and a survery. In *Workshop SRL, Int. Joint. Conf. on AI*.

Paolucci, M.; Srinivasan, N.; Sycara, K.; and Nishimura, T. 2003. Towards a semantic choreography of web services: From WSDL to DAML-S. In *ISWC*.

Patil, A.; Oundhakar, S.; Sheth, A.; and Verma, K. 2004. Meteor-s web service annotation framework. In *13th Int. WWW Conf.*

Perkowitz, M., and Etzioni, O. 1995. Category translation: Learning to understand information on the internet. In *Int. Joint Conf. on AI*.

Sabou, M. 2004. From software APIs to web service ontologies: a semi-automatic extraction method. In *ISWC*.

The DAML Services Coalition. 2003. OWL-S 1.0. White Paper.

Yang, Y.; Slattery, S.; and Ghani, R. 2002. A study of approaches to hypertext categorization. *Journal of Intelligent Information Systems* 18(2-3):219–241.