Semi-Automatically Annotating Semantic Web Services (Extended Abstract)

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

Overview

The semantic Web Services vision requires that each service be annotated with semantic metadata. Various metadata languages (such as OWL-S (DAML-S Coalition 2003)) have been proposed to fill this "semantic gap". However, manually creating such metadata is tedious and error-prone. Software engineers, accustomed to tools that automatically generate WSDL, might not want to invest the required effort.

This extended abstract describes ASSAM, a tool that assists a user in creating semantic metadata for Web Services. ASSAM's capabilities to automatically create semantic metadata are supported by two machine learning algorithms. First, we have developed an iterative relational classification algorithm for semantically classifying Web Services, their operations, and input and output messages. Second, to aggregate the data returned by multiple semantically related Web Services, we have developed a schema mapping algorithm based ensembles of string distance metrics.

The remainder of this paper is organized as follows: (1) 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. (2) We describe and evaluate the iterative relational classification algorithm that provides these suggestions. (3) We describe and evaluate OATS, a schema mapping algorithm specifically designed for the Web Services context.

This extended abstract merely summarizes our efforts; for details and a discussion of related work, see (Heß & Kushmerick 2004; Johnston & Kushmerick 2004; Heß, Johnston, & Kushmerick 2004).

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.

ASSAM is designed primarily for users who want to annotate many similar services. These users could be end-users wanting to integrate several similor Web Services into his or her business processes, or the administrators of a centralized semantic Web Service registry. Our tool could also be useful for programmers who are only interested in annotating a single Web Service they have created. In order to make his or her service compatible with existing services, a developer might want to annotate it with the same ontology that has already been used for some other Web Services. The developer could import the existing Web Services in ASSAM and use them as training data in order to obtain recommendations on how to annotate his or her own service.

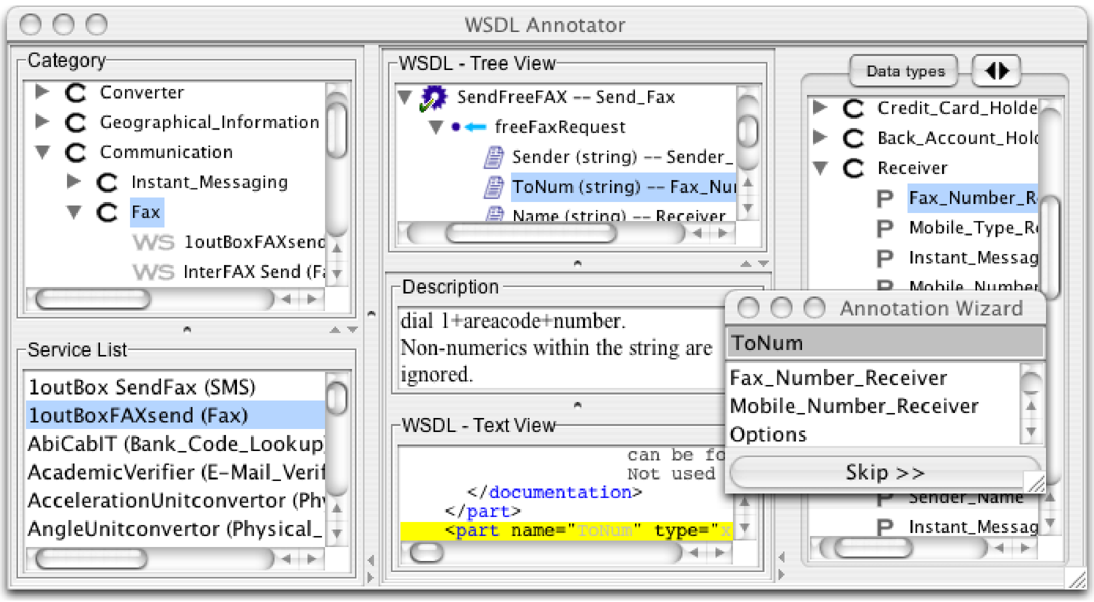
Fig. 1 shows the ASSAM application. Note that our application's key novelty—the suggested annotations created automatically by our machine learning algorithm—is shown in the small pop-up window.

The left column in the main window contains a list of Web Services and the category ontology. Web Services can be associated with a category by clicking on a service in a list and then on a node in the category tree. When the user has selected a service and wants to focus on annotating it this part of the window can be hidden.

The middle of the window contains a tree view of the WSDL. Port types, operations, messages and complex XML schema types are parsed from the WSDL and shown in a tree structure. The original WSDL file is also shown as well as plain text descriptions from the occasional documentation tags within the WSDL or a plain text description of the service as a whole, such as often offered by a UDDI registry or a Web Service indexing web site.

When the user clicks on an element in the WSDL tree view, the corresponding ontology is shown in the right column and the user can select an appropriate class by clicking on an element in the ontology view. Currently, different ontologies for datatypes and operations are used. At present we allow annotation for operations, message parts and XML schema types and their elements. Port types or messages cannot be annotated, because there is no real semantic meaning associated with the port type or the message itself that is not covered by the annotation of the operations or the message parts.

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 from 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



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 machine learning to annotate Web Services with semantic metadata.

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.

Iterative Relational Classification

We now describe the machine learning algorithms behind ASSAM's annotation wizard. 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.

Terminology. Before describing our approach in detail, we begin with some terminology. By introducing this terminology we do not advocate a new standard. Instead we believe that our approach is generic and independent of the actual format used for the semantic Web Service description.

We use the term *category* to denote the semantic meaning of the service as a whole. The category ontology corresponds to a profile hierarchy in OWL-S. The term *domain* denotes the semantic meaning of a single operation. An operation in WSDL usually maps to an atomic process in OWL-S, but there is no direct relation of the domain of an operation to OWL-S, as atomic processes are only characterized through their inputs, outputs, preconditions and effects. Finally, the term *datatype* denotes the semantic type of a single input/output variable. This usage is intended to map on to, for example, a property in an ontology, and should not be confused with low-level syntactic datatypes such as "integer" or "string".

For information retrieval or classification tasks the objects that are classified or searched are usually referred to as *documents*. When we use the word *document*, we mean the Web Service's WSDL representation. We use *document part* to denote an object within the Web Service that we want to classify: operations, input and output messages, and XML schema types.

Iterative Classification Ensemble. 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. In previous work (Heß & Kushmerick 2003), we exploited these dependencies in a Bayesian setting and evaluated it on Web forms. In this paper, we present an generalization to the iterative classification algorithm proposed by (Neville & Jensen 2003).

Like any classification system, our algorithm is based on a set of features of the services, operations and parameters. Following (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 classifications. Of course, these classifications may be partially incorrect. The classification process is repeated until a certain termination criterion (e.g. convergence) is met.

Our iterative algorithm differs in several ways from Neville and Jensen's algorithm. In their approach, one single classifier is trained on all (intrinsic and extrinsic) features. In a variety of tasks, ensembles of several classifiers have been shown to be more effective (e.g., (Dietterich 2000)). For this reason, we train two separate classifiers, one on the intrinsic features ("A") and one on the extrinsic features ("B"), and vote together their predictions. Another advantage of combining the evidence in that way is that the classifier cannot be mislead by missing features in the beginning when the extrinsic features are yet unknown, because the classifier trained on the extrinsic features is simply not used for the first pass.

We also introduce a second mode for incorporating the extrinsic features: We train a set of classifiers on the intrinsic features of the datatypes, but each of them is only on the subset of the instances that belong to one specific category.

More precisely, once we have classified the category of a service, we use the classifier for the datatypes that has been trained on instances from that category[1]. To avoid biasing the algorithm too strongly, we still combine the results of the A_{spec} classifier with the A classifier in each iteration. For each level we use either B or the A_{spec} classifiers, but not both. We chose the A_{spec} method for the datatypes and the B method for the category and the domain.

We did not exploit every possible dynamic extrinsic feature. We used *static* extrinsic features on the domain and datatype level by incorporating text from children nodes: Text associated with messages was added to the text used by the operations classifier, and text associated with elements of complex types were added to the text used by the datatype classifier classifying the complex type itself. The features we used and the feedback structures for the dynamic features are based on preliminary tests. We used a fixed number of 5 iterations.

In the evaluation section, we report results for this setup. For a more detailed discussion of the parameters of our algorithm and their effects 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.

[1]To avoid over-specialization, these classifiers are actually not trained on instances from a single category, but rather on instances from a complete top-level branch of the hierarchically organized category ontology. Note that this is the only place where we make use of the fact that the class labels are organized as an ontology, and we do not do any further inference.

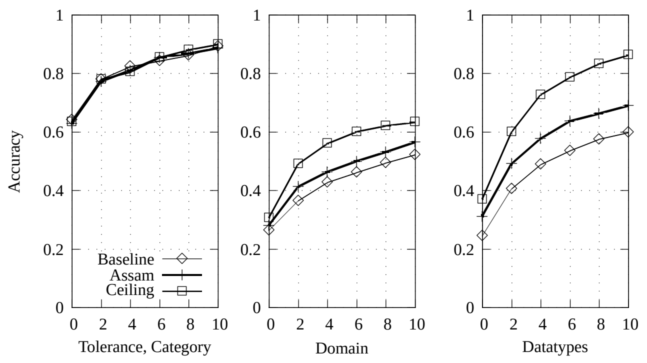


Figure 2: Accuracy of our algorithm on the three kinds of semantic metadata as a function of prediction tolerance.

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. 2 show the accuracy for categories, domains and datatypes. 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 90% of the time, the correct answer is in the top 10 predictions.

We could not achieve good results with the non-ensemble setup. This setup scored worse than the baseline. For the datatypes, even the ceiling accuracy was below the baseline.

Note that on the category level incorporating the additional evidence from the extrinsic features does not help. In fact, for some tolerance values the ceiling accuracy is even worse than the baseline.

On the datatype level, our algorithm achieves 31.2% accuracy, where as the baseline scores only at 24.5%. Thus, our algorithm improves performance by almost one third. The overall performance might be considered quite low, but due to the high number of classes it is a very hard classification problem. Given that in two of three cases the user has to choose only between 10 class labels rather than between all 312 labels in the datatype ontology we are still convinced that this could save a considerable amount of workload. On the domain level, our approach increases the accuracy for exact matches from 26.3% to 28%.

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 distance 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, i.e, there exists a set of real-world entities that are covered by all of the sources to be aggregated. For example, while two weather Web Service need not over 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)[2], 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.

[2]In our experiments, probes are encoded as a table of attribute/value pairs, and v_i is the set of attributes needed by o_i.

Learning distance metric weights. 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 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.

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.

The ensemble of distance metrics was selected from Cohen's SecondString library (Cohen, Ravikumar, & Fienberg 2003). We chose eight representative metrics, consisting of a variety of character-based, token-based and hybrid metrics: TFIDF, SlimTFIDF, Jaro, CharJaccard, Levenstein [sic], SmithWaterman, Level2Jaro and Level2JaroWinkler.

Each probe entity is represented as a set of attribute/value pairs. For example, Fig. 3 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. 3) are searched for the required attributes.

address	city	state	fullstate	zip	acode	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 3: The four probe objects for the zip and weather domains.

Results. First, we show that by using an ensemble of string metrics, we achieve better results than using the metrics separately. Fig. 4 (top) compares the ensemble approach to the Levenshtein and TFIDF metrics individually. We report the average performance over the three domains in two ways: F1 as a function of the HAC termination threshold, and a precision/recall curve. 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}. We used just two folds due to the relatively small number of operations. 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. 4 (bottom) shows F1 as a function of the HAC termination threshold, and the precision/recall curves 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.

Finally, we have found that accuracy improves with additional probe queries, but that performance is beginning to level off after just a few probes. Recall that we used up to four probe queries. We note that accuracy improved more between one and two queries, than between three and four. We anticipate that the performance increase will rapidly slow beyond a relatively small number of probes. Note that we did not carefully select the probe objects in order to maximize performance. Indeed, some of the operations returned missing elements for some probes. Our experiments suggest that the active invocation approach makes OATS robust to "bad" probes.

Discussion

In this paper, we have presented ASSAM, a tool for annotating Semantic Web Services. We have presented the WSDL

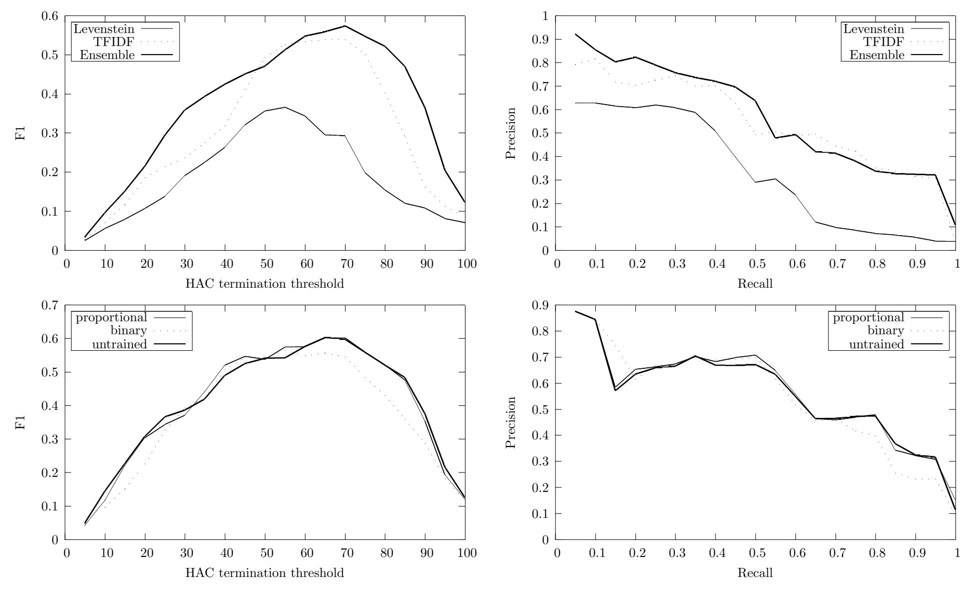


Figure 4: OATS ensemble vs. individual distance metrics (top); OATS with vs. without adaptive distance metric weighting (bottom); F1 as a function of the HAC termination threshold (left); precision/recall curve (right).

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 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.

We have also shown how semantically annotated Web Services could be used to enhance data aggregation systems, and how Web Service aggregation can be viewed as 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, which although not always successful, usually increased the average F1. We plan to examine the use of more sophisticated techniques for aggregating the ensemble matchers.

One of the constraints of our aggregation system is that there must be an overlap between the sources, i.e. all of the sources must “know about” the entity being queried. Ultimately, we would like our system to learn new objects from some information sources that could be used to probe other partially overlapping sources. We envisage a tool that, given a set of seed Web Services and probe queries, could find sets of related Web Services and learn new probe objects to query them with. Such visions would require the automated discovery, composition and invocation of Web Services, but this interoperability requires Services to be semantically annotated. We believe that the methods we have presented here are a reasonable first step towards the realization of these goals.

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*.

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

Dietterich, T. G. 2000. Ensemble methods in machine learning. In *Lecture Notes in Computer Science*, volume 1857.

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 *European Conf. Machine Learning*.

Heß, A.; Johnston, E.; and Kushmerick, N. 2004. ASSAM: A tool for semi-automatically annotating semantic Web Services. In *Int. Semantic Web Conf.*

Johnston, E., and Kushmerick, N. 2004. Aggregating web services with active invocation and ensembles of string distance metrics. In *Int. Conf. Knowledge Engineering and Knowledge Management*.

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

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