Accurately and Reliably Extracting Data from the Web: A Machine Learning Approach

Craig A. Knoblock
University of Southern California
and Fetch Technologies

Kristina Lerman
University of Southern California

Steven Minton
Fetch Technologies

Ion Muslea
University of Southern California

Abstract

A critical problem in developing information agents for the Web is accessing data that is formatted for human use. We have developed a set of tools for extracting data from web sites and transforming it into a structured data format, such as XML. The resulting data can then be used to build new applications without having to deal with unstructured data. The advantages of our wrapping technology over previous work are the the ability to learn highly accurate extraction rules, to verify the wrapper to ensure that the correct data continues to be extracted, and to automatically adapt to changes in the sites from which the data is being extracted.

1 Introduction

There is a tremendous amount of information available on the Web, but much of this information is not in a form that can be easily used by other applications. There are hopes that XML will solve this problem, but XML is not yet in widespread use and even in the best case it will only address the problem within application domains where the interested parties can agree on the XML schema definitions. Previous work on wrapper generation in both academic research [4, 6, 8] and commercial products (such as OnDisplay's eContent) have primarily focused on the ability to rapidly create wrappers. The previous work makes no attempt to ensure the accuracy of the wrappers over the entire set of pages of a site and provides no capability to detect failures and repair the wrappers when the underlying sources change.

We have developed the technology for rapidly building wrappers for accurately and reliably extracting data from semistructured sources. Figure 1 graphically illustrates the entire lifecycle of a wrapper. As shown in the Figure, the wrapper induction system takes a set of web pages labeled with examples of the data to be extracted. The user provides the initial set of labeled examples and the system can suggest additional pages for the user to label in order to build wrappers that are very accurate. The output of the wrapper induction system is a set of extraction rules that describe how to locate the desired information on a Web page. After the system creates a wrapper, the wrapper verification system uses the functioning wrapper to learn patterns that describe the data being extracted. If a change is detected, the system can automatically repair a wrapper by using the same patterns

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

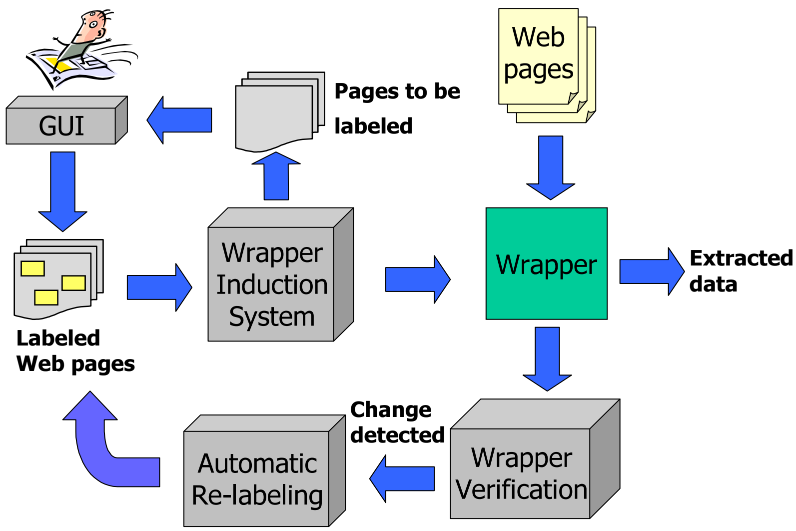

Figure 1: **The Lifecycle of a Wrapper**

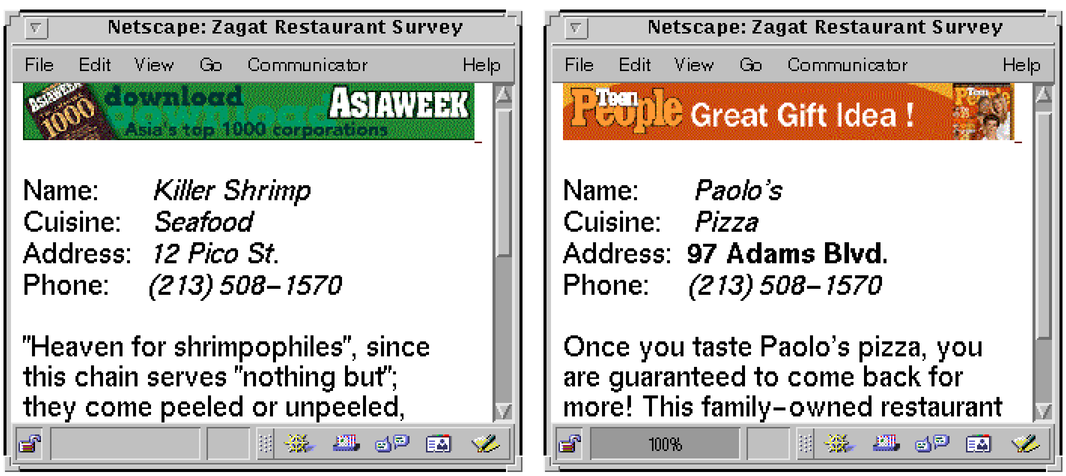

Figure 2: **Two Sample Restaurant Documents From the Zagat Guide.**

to locate examples on the changed pages and re-running the wrapper induction system. The details of this entire process are described in the remainder of this paper.

2 Learning Extraction Rules

A wrapper is a piece of software that enables a semistructured Web source to be queried as if it were a database. These are sources where there is no explicit structure or schema, but there is an implicit underlying structure (for example, consider the two documents in Figure 2). Even text sources, such as email messages, have some structure in the heading that can be exploited to extract the date, sender, addressee, title, and body of the messages. Other sources, such as online catalogs, have a very regular structure that can be exploited to extract the data automatically.

One of the critical problems in building a wrapper is defining a set of extraction rules that precisely define how to locate the information on the page. For any given item to be extracted from a page, one needs an extraction rule to locate both the beginning and end of that item. Since, in our framework, each document consists of a sequence of tokens (e.g., words, numbers, HTML tags, etc), this is equivalent to finding the first and last tokens of an item. The hard part of this problem is constructing a set of extraction rules that work for *all* of the pages in the source.

A key idea underlying our work is that the extraction rules are based on "landmarks" (i.e., groups of consecutive tokens) that enable a wrapper to locate the start and end of the item within the page. For example, let us consider the three restaurant descriptions E1, E2, and E3 presented in Figure 3. In order to identify the beginning of the address, we can use the rule

```
E1:   ...Cuisine:<i>Seafood</i><p>Address:<i>  12 Pico St. </i><p>Phone:<i>...
E2:   ...Cuisine:<i>Thai   </i><p>Address:<i> 512 Oak Blvd.</i><p>Phone:<i>...
E3:   ...Cuisine:<i>Burgers</i><p>Address:<i> 416 Main St. </i><p>Phone:<i>...
E4:   ...Cuisine:<i>Pizza</i><p>Address:<b> 97 Adams Blvd. </b><p>Phone:<i>...
```

Figure 3: **Four sample restaurant documents.**

R1 = $SkipTo$(Address) $SkipTo$(<i>)

which has the following meaning: start from the beginning of the document and skip every token until you find a landmark consisting of the word Address, and then, again, ignore everything until you find the landmark <i>. **R1** is called a *start rule* because it identifies the beginning of the address. One can write a similar *end rule* that finds the end of the address; for sake of simplicity, we restrict our discussion here to start rules.

Note that **R1** is by no means the only way to identfy the beginning of the address. For instance, the rules

R2 = $SkipTo$(Address : <i>)
R3 = $SkipTo$(Cuisine : <i>) $SkipTo$(Address : <i>)
R4 = $SkipTo$(Cuisine : <i> *_Capitalized_* </i> <p> Address : <i>)

can be also used as start rules. **R2** uses the 3-token landmark that immediately precedes the beginning of the address in examples E1, E2, and E3, while **R3** relies on two 3-token landmarks. Finally, **R4** is defined based on a 9-token landmark that uses the wildcard *_Capitalized_*, which is a placeholder for any capitalized alphabetic string (other examples of useful wildcards are *_Number_*, *_AllCaps_*, *_HtmlTag_*, etc).

To deal with variations in the format of the documents, our extraction rules allow the use of *disjunctions*. For example, let us assume that the addresses that are within one mile from your location appear in bold (see example E4 in Figure 3), while the other ones are displayed as italic (e.g., E1, E2, and E3). We can extract all the names based on the disjunctive start rule

either $SkipTo$(Address : <b>)
or $SkipTo$(Address) $SkipTo$(<i>)

Disjunctive rules are *ordered lists* of individual disjuncts (i.e., decision lists). Applying a disjunctive rule is a straightforward process: the wrapper successively applies each disjunct in the list until it finds the first one that matches. Even though in our simplified examples one could have used the nondisjunctive rule

$SkipTo$(Address : *_HtmlTag_*),

there are many real world sources that cannot be wrapped without using disjuncts.

We have developed STALKER [11], a *hierarchical* wrapper induction algorithm that learns extraction rules based on examples labeled by the user. We have a graphical user interface that allows a user to mark up several pages on a site, and the system then generates a set of extraction rules that accurately extract the required information. Our approach uses a greedy-covering inductive learning algorithm, which incrementally builds the extraction rules from the examples.

In contrast to other approaches [4, 6, 8], a key feature of STALKER is that it is able to efficiently generate extraction rules from a small number of examples: it rarely requires more than 10 examples, and in many cases two examples are sufficient. The ability to generalize from such a small number of examples has a two-fold explanation. First, in most of the cases, the pages in a source are generated based on a fixed template that may have only a few variations. As STALKER tries to learn landmarks that are part of this template, it follows that for templates with little or no variations a handful of examples usually will be sufficient to induce reliable landmarks.

Second, STALKER exploits the *hierarchical* structure of the source to constrain the learning problem. More precisely, based on the schema of the data to be extracted, we *automatically* decompose one difficult problem (i.e., extract all items of interest) into a series of simpler ones. For instance, instead of using one complex rule that extracts all restaurant names, addresses and phone numbers from a page, we take a hierarchical approach. First we apply a rule that extracts the whole list of restaurants; then we use another rule to break the list into tuples that correspond to individual restaurants; finally, from each such tuple we extract the name, address, and phone number of the corresponding restaurant. Our hierarchical approach also has the advantage of being able to extract data from pages that contain complicated formatting layouts (e.g., lists embedded in other lists) that previous approaches could not handle (see [11] for details).

STALKER is a sequential covering algorithm that, given the training examples E, tries to learn a minimal number of *perfect disjuncts* that cover *all* examples in E. By definition, a perfect disjunct is a rule that covers at least one training example and on any example the rule matches it produces the correct result. STALKER first creates an initial set of candidate-rules C and then repeatedly applies the following three steps until it generates a perfect disjunct:

- select most promising candidate from C
- refine that candidate
- add the resulting refinements to C

Once STALKER obtains a perfect disjunct P, it removes from E all examples on which P is correct, and the whole process is repeated until there are no more training examples in E. STALKER uses two types of refinements: *landmark refinements* and *topology refinements*. The former makes the rule more specific by adding a token to one of the existing landmarks, while the latter adds a new 1-token landmark to the rule.

For instance, let us assume that based on the four examples in Figure 3, we want to learn a start rule for the address. STALKER proceeds as follows. First, it selects an example, say E4, to guide the search. Second, it generates a set of *initial candidates*, which are rules that consist of a single 1-token landmark; these landmarks are chosen so that they match the token that *immediately precedes* the beginning of the address in the guiding example. In our case we have two initial candidates:

R5 = $SkipTo$(<b>)

R6 = $SkipTo$(*_HtmlTag_*)

As the <b> token appears only in E4, **R5** does not match within the other three examples. On the other hand, **R6** matches in all four examples, even though it matches *too early* (**R6** stops as soon as it encounters an HTML tag, which happens in all four examples *before* the beginning of the address). Because **R6** has a better generalization potential, STALKER selects **R6** for further refinements.

While refining **R6**, STALKER creates, among others, the new candidates **R7**, **R8**, **R9**, and **R10** shown below. The first two are obtained via landmark refinements (i.e., a token is added to the landmark in **R6**), while the other two rules are created by topology refinements (i.e., a new landmark is added to **R6**). As **R10** works correctly on all four examples, STALKER stops the learning process and returns **R10**.

R7 = $SkipTo$(: *_HtmlTag_*)

R8 = $SkipTo$(*_Punctuation_ _HtmlTag_*)

R9 = $SkipTo$(**:**) $SkipTo$(*_HtmlTag_*)

R10 = $SkipTo$(`Address`) $SkipTo$(*_HtmlTag_*)

By using STALKER, we were able to successfully wrap information sources that could not be wrapped with existing approaches (see [11] for details). In an empirical evaluation on 28 sources proposed in [8], STALKER had to learn 206 extraction rules. We learned 182 *perfect* rules (100% accurate), and another 18 rules that had an accuracy of at least 90%. In other words, only 3% of the learned rules were less that 90% accurate .

3 Identifying Highly Informative Examples

STALKER can do significantly better on the hard tasks (i.e., the ones for which it failed to learn perfect rules) if instead of $random$ examples, the system is provided with carefully selected examples. Specifically, the most informative examples illustrate exceptional cases. However, it is unrealistic to assume that a user is willing and has the skills to browse a large number of documents in order to identify a sufficient set of examples to learn perfect extraction rules. This is a general problem that none of the existing tools address, regardless of whether they use machine learning.

To solve this problem we have developed an *active learning* approach that analyzes the set of unlabeled examples to automatically select examples for the user to label. Our approach, called *co-testing* [10], exploits the fact that there are often multiple ways of extracting the same information [1]. In the case of wrapper learning, the system can learn two different types of rules: *forward* and *backward* rules. All the rules presented above are *forward* rules: they start at the beginning of the document and go towards the end. By contrast, a *backward* rule starts at the end of the page and goes towards its beginning. For example, one can find the beginning of the addresses in Figure 3 by using one of the following backward rules:

R11 = $BackTo$(`Phone`) $BackTo$(*_Number_*)

R12 = $BackTo$(`Phone : <i>`) $BackTo$(*_Number_*)

The main idea behind co-testing is straightforward: after the user labels one or two examples, the system learns *both* a forward and a backward rule. Then it runs *both* rules on a given set of unlabeled pages. Whenever the rules disagree on an example, the system considers that as an example for the user to label next. The intuition behind our approach is the following: if both rules are 100% accurate, on *every* page they must identify *the same* token as the beginning of the address. Furthermore, as the two rules are learned based on different sets of tokens (i.e., the sequences of tokens that precede and follow the beginning of the address, respectively), they are highly unlikely to make the exact same mistakes. Whenever the two rules disagree, at least one of them must be wrong, and by asking the user to label that particular example, we obtain a highly informative training example. Co-testing makes it possible to generate accurate extraction rules with a very small number of labeled examples.

To illustrate how co-testing works, consider again the examples in Figure 3. Since most of the restaurants in a city are *not* located within a 1-mile radius of one's location, it follows that most of the documents in the source will be similar to `E1`, `E2`, and `E3` (i.e., addresses shown in italic), while just a few examples will be similar to `E4` (i.e., addresses shown in bold). Consequently, it is unlikely that an address in bold will be present in a small, randomly chosen, initial training set. Let us now assume that the initial training set consists of `E1` and `E2`, while `E3` and `E4` are *not labeled*. Based on these examples, we learn the rules

Fwd-R1 = $SkipTo$(`Address`) $SkipTo$(`<i>`)

Bwd-R1 = $BackTo$(`Phone`) $BackTo$(*_Number_*)

Both rules correctly identify the beginning of the address for all restaurants that are more than one mile away, and, consequently, they will agree on all of them (e.g., `E3`). On the other hand, **Fwd-R1** works *incorrectly* for examples like `E4`, where it stops at the beginning of the phone number. As **Bwd-R1** is correct on `E4`, the two rules disagree on this example, and the user is asked to label it.

To our knowledge, there is no other wrapper induction algorithm that has the capability of identifying the most informative examples. In the related field of information extraction, where one wants to extract data from free text documents, researchers proposed such algorithms [13, 12], but they cannot be applied in a straightforward manner to existing wrapper induction algorithms.

We applied co-testing on the 24 tasks on which STALKER fails to learn perfect rules based on 10 random examples. To keep the comparison fair, co-testing started with one random example and made up to 9 queries. The results were excellent: the average accuracy over all tasks improved from 85.7% to 94.2% (error rate reduced by 59.5%). Furthermore, 10 of the learned rules were 100% accurate, while another 11 rules were at least 90% accurate. In these experiments as well as in other related tests [10] applying co-testing leads to a significant improvement in accuracy without having to label more training data.

4 Verifying the Extracted Data

Another problem that has been largely ignored in past work on extracting data from web sites is that sites change and they change often. Kushmerick [7] addressed the wrapper verification problem by monitoring a set of generic features, such as the density of numeric characters within a field, but this approach only detects certain types of changes. In contrast, we address this problem by applying machine learning techniques to learn a set of patterns that describe the information that is being extracted from each of the relevant fields. Since the information for even a single field can vary considerably, the system learns the statistical distribution of the patterns for each field. Wrappers can be verified by comparing the patterns of data returned to the learned statistical distribution. When a significant difference is found, an operator can be notified or we can automatically launch the wrapper repair process, which is described in the next section.

The learned patterns represent the structure, or format, of the field as a sequence of words and wildcards [9]. Wildcards represent syntactic categories to which words belong — alphabetic, numeric, capitalized, etc. — and allow for multi-level generalization. For complex fields, and for purposes of information extraction, it is sufficient to use only the starting and ending patterns as the description of the data field. For example, a set of street addresses — **12 Pico St.**, **512 Oak Blvd.**, **416 Main St.** and **97 Adams Blvd.** — all start with a pattern **(*_Number_ _Capitalized_*)** and end with **(Blvd.)** or **(St.)**. We refer to the starting and ending patterns together as the *data prototype* of the field.

The problem of learning a description of a field (class) from a set of labeled examples can be posed in two related ways: as a classification or a conservation task. If negative examples are present, the classification algorithm learns a *discriminating* description. When only positive examples are available, the conservation task learns a *characteristic* description, *i.e.* one that describes many of the positive examples but is highly unlikely to describe a randomly generated example. Because an appropriate set of negative examples not available in our case, we chose to frame the learning problem as a conservation task. The algorithm we developed, called DataPro [9], finds all statistically significant starting and ending patterns in a set of positive examples of the field. A pattern is said to be significant if it occurs more frequently than would be expected by chance if the tokens were generated randomly and independently of one another. Our approach is similar to work on grammar induction [2, 5], but our pattern language is better suited to capturing the regularities in small data fields (as opposed to languages).

The algorithm operates by build building a prefix tree, where each node corresponds to a token whose position in the sequence is given by the node's depth in the tree. Every path through the tree starting at the root node is a significant pattern found by the algorithm.

The algorithm grows the tree incrementally. Adding a child to a node corresponds to extending the node's pattern by a single token. Thus, each child represents a different way to specialize the pattern. For example, when learning city names, the node corresponding to "New" might have three children, corresponding to the patterns "New Haven", "New York" and "New *_Capitalized_*". A child node is judged to be significant with respect to its parent node if the number of occurrences of the child pattern is sufficiently large, given the number of occurrences of the parent pattern and the baseline probability of the token used to extend the parent pattern. A pruning step insures that each child node is also significant given its more specific siblings. For example, if there are 10 occurrence of "New Haven", and 12 occurrences of "New York", and both of these are judged to be significant, then "New *_Capitalized_*" will be retained only if there are significantly more than 22 examples that match "New *_Capitalized_*". Similarly, once the entire tree has been expanded, the algorithm includes a pattern extraction step that traverses the tree, checking whether the pattern "New" is still significant given the more specific patterns "New York", "New Haven" and "New *_Capitalized_*". In other words, DataPro decides whether the examples described by "New" but not by any of the longer sequences can be explained by the null hypothesis.

The patterns learned by DataPro lend themselves to the data validation task and, specifically, to wrapper verification. A set of queries is used to retrieve HTML pages from which the wrapper extracts some training examples. The learning algorithm finds the patterns that describe the common beginnings and endings of each field of the training examples. In the verification phase, the wrapper generates a test set of examples from pages retrieved using the same or similar set of queries. If the patterns describe statistically the same (at a given significance level) proportion of the test examples as the training examples, the wrapper is judged to be extracting correctly; otherwise, it is judged to have failed.

Once we have learned the patterns, which represent our expectations about the format of data, we can then configure the wrappers to verify the extracted data before the data is sent, immediately after the results are sent, or on some regular frequency. The most appropriate verification method depends on the particular application and the tradeoff between response time and data accuracy.

We monitored 27 wrappers (representing 23 distinct Web sources) over a period of several months. For each wrapper, the results of 15-30 queries were stored periodically. All new results were compared with the last correct wrapper output (training examples). A manual check of the results revealed 37 wrapper changes out of the total 443 comparisons. The verification algorithm correctly discovered 35 of these changes. The algorithm incorrectly decided that the wrapper has changed in 40 cases. We are currently working to reduce the high rate of false positives.

5 Automatically Repairing Wrappers

Most changes to Web sites are largely syntactic and are often minor formatting changes or slight reorganizations of a page. Since the content of the fields tend to remain the same, we exploit the patterns learned for verifying the extracted results to locate correct examples of the data field on new pages. Once the required information has been located, the pages are automatically re-labeled and the labeled examples are re-run through the inductive learning process to produce the correct rules for this site.

Specifically, the wrapper reinduction algorithm takes a set of training examples and a set of pages from the same source and uses machine learning techniques to identify examples of the data field on new pages. First, it learns the starting and ending patterns that describe each field of the training examples. Next, each new page is scanned to identify all text segments that begin with one of the starting patterns and end with one of the ending patterns. Those segments, which we call candidates, that are significantly longer or shorter than the training examples are eliminated from the set, often still leaving hundreds of candidates per page. The candidates are then clustered to identify subgroups that share common features. The features used in clustering are the candidate's relative position on the page, adjacent landmarks, and whether it is visible to the user. Each group is then given a score based on how similar it is to the training examples. We expect the highest ranked group to contain the correct examples of the data field.

Figure 4 shows a actual example of a change to Amazon's site and how our system automatically adapts to the change. The top frame shows an example of the original site, the extraction rule for a book title, and the extracted results from the example page. The middle frame shows the source and the incorrectly extracted result after the title's font and color were changed. And the bottom frame shows the result of the automatic reinduction process with the corrected rule for extracting the title.

We evaluated the algorithm by using it to extract data from Web pages for which correct output is known.

Figure 4: **An Example of the Reinduction Process**

The output of the extraction algorithm is a ranked list of clusters for every data field being extracted. Each cluster is checked manually, and it is judged to be correct if it contains only the complete instances of the field, which appear in the correct context on the page. For example, if extracting a city of an address, we only want to extract those city names that are part of an address.

We ran the extraction algorithm for 21 distinct Web sources, attempting to extract 77 data fields from all the sources. In 62 cases the top ranked cluster contained correct complete instances of the data field. In eight cases the correct cluster was ranker lower, while in six cases no candidates were identified on the pages. The most closely related work is that of Cohen [3], which uses an information retrieval approach to relocating the information on a page. The approach was not evaluated on actual changes to Web pages, so it is difficult to assess whether this approach would work in practice.

6 Discussion

Our work addresses the complete wrapper creation problem, which includes:

- building wrappers by example,
- ensuring that the wrappers accurately extract data across an entire collection of pages,
- verifying a wrapper to avoid failures when a site changes,

- and automatically repair wrappers in response to changes in layout or format.

Our main technical contribution is in the use of machine learning to accomplish all of these tasks. Essentially, our approach takes advantage of the fact that web pages have a high degree of regular structure. By analyzing the regular structure of example pages, our wrapper induction process can detect landmarks that enable us to extract desired fields. After we developed an initial wrapper induction process we realized that the accuracy of the induction method can be improved by simultaneously learning "forward" and "backward" extraction rules to identify exception cases. Again, what makes this possible is the regularities on a page that enable us to identify landmarks both before a field and after a field.

Our approach to automatically detecting wrapper breakages and repairing them capitalizes on the regular structure of the extracted fields themselves. Once a wrapper has been initially created, we can use it to obtain numerous examples of the fields. This enables us to profile the information in the fields and obtain structural descriptions that we can use during monitoring and repair. Essentially this is a bootstrapping approach. Given the initial examples provided by the user, we first learn a wrapper. Then we use this wrapper to obtain many more examples which we then analyze in much greater depth. Thus by leveraging a few human-provided examples, we end up with a highly scalable system for wrapper creation and maintenance.

Acknowledgements

The research reported here was supported in part by the Rome Laboratory of the Air Force Systems Command and the Defense Advanced Research Projects Agency (DARPA) under contract number F30602-98-2-0109, in part by the United States Air Force under contract number F49620-98-1-0046, and in part by the Integrated Media Systems Center, a National Science Foundation Engineering Research Center, under Cooperative Agreement No. EEC-9529152. 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] A. Blum and T. Mitchell. Combining labeled and unlabeled data with co-training. In *Proc. of the 1988 Conference on Computational Learning Theory*, pages 92–100, 1998.

[2] R. Carrasco and J. Oncina. Learning stochastic regular grammars by means of a state merging method. In *Lecture Notes In Computer Science*, page 862, 1994.

[3] W. Cohen. Recognizing structure in web pages using similarity queries. In *Proc. of the 16th National Conference on Artificial Intelligence AAAI-1999*, pages 59–66, 1999.

[4] D. Freitag and N. Kushmerick. Boosted wrapper induction. In *Proc. of the 17th National Conference on Artificial Intelligence AAAI-2000*, pages 577–583, 2000.

[5] T. Goan, N. Benson, and O. Etzioni. A grammar inference algorithm for the world wide web. In *Proc. of the AAAI Spring Symposium on Machine Learning in Information Access*, 1996.

[6] C. Hsu and M. Dung. Generating finite-state transducers for semi-structured data extraction from the web. *Journal of Information Systems*, 23(8):521–538, 1998.

[7] N. Kushmerick. Regression testing for wrapper maintenance. In *Proc. of the 16th National Conference on Artificial Intelligence AAAI-1999*, pages 74–79, 1999.

[8] N. Kushmerick. Wrapper induction: efficiency and expressiveness. *Artificial Intelligence Journal*, 118(1-2):15–68, 2000.

[9] K. Lerman and S. Minton. Learning the common structure of data. In *Proc. of the 17th National Conference on Artificial Intelligence AAAI-2000*, pages 609–614, 2000.

[10] I. Muslea, S. Minton, and C. Knoblock. Co-testing: Selective sampling with redundant views. In *Proc. of the 17th National Conference on Artificial Intelligence AAAI-2000*, pages 621–626, 2000.

[11] I. Muslea, S. Minton, and C. Knoblock. Hierarchical wrapper induction for semistructured information sources. *Journal of Autonomous Agents and Multi-Agent Systems*, 2001. (to appear).

[12] S. Soderland. Learning extraction rules for semi-structured and free text. *Machine Learning*, 34:233–272, 1999.

[13] C. Thompson, M. Califf, and R. Mooney. Active learning for natural language parsing and information extraction. In *Proc. of the 16th International Conference on Machine Learning ICML-99*, pages 406–414, 1999.