

Wrapper Maintenance: A Machine Learning Approach

Kristina Lerman¹, Craig Knoblock^{1,2}, Steven Minton²

1. Information Sciences Institute

Univ. of Southern California

4676 Admiralty Way

Marina del Rey, CA 90292-6695

2. Fetch Technologies

4676 Admiralty Way

Marina del Rey, CA 90292-6695

{lerman,knoblock}@isi.edu

minton@fetch.com

October 16, 2001

Abstract

The proliferation of online information sources has led to an increased use of Web wrappers — tools that can extract data from semi-structured Web sources. While most of the previous research has focused on quick and efficient generation of wrappers, the development of tools for wrapper maintenance, i.e., for automatic validation and recognition of data, has received less attention. This is an important research problem, because Web sources often change in ways that break the wrappers. We present an efficient algorithm that learns structural information about data from positive examples alone. We describe how this information can be used for wrapper maintenance applications: wrapper verification and reinduction. The wrapper verification task detects when a wrapper is not extracting correct data, usually because the Web source has changed its format. To validate our approach, we monitored 27 wrappers over a period of a year. The verification algorithm correctly discovered 35 of the 37 wrapper changes, and made 16 mistakes, resulting in precision of 73% and recall of 95%. The reinduction algorithm automatically recovers from changes in the Web source by identifying data on Web pages so that a new wrapper may be generated for this source. We validated the reinduction algorithm on ten Web sources returning a single tuple of data (rather than a list of tuples). We were able to successfully re-generate the wrappers' extraction rules using only the old training examples and new pages, obtaining precision and recall values of 88% and 92% respectively on the data extraction task. Our results indicate a feasible method to combine information about the structure of data and *a priori* expectations about page structure to automatically generate wrappers for previously unseen Web sources.

Contents

1	Introduction	3
2	Pattern Learning	4
2.1	Data Representation	4
2.2	Learning from Positive Examples	5
2.3	DataProG Algorithm	6
2.4	Previous Work on Pattern Learning	7
3	Applications of Pattern Learning	13
3.1	Wrapper Verification	14
3.1.1	Results	15
3.1.2	Discussion of Results	16
3.1.3	Previous Work on Wrapper Verification	17
3.2	Wrapper Reinduction	17
3.2.1	Results	19
3.2.2	Previous Work on Wrapper Reinduction	22
4	Conclusion	22
5	Acknowledgements	22

1 Introduction

There is a tremendous amount of information available online, but much of this information is formatted to be easily read by human users, not computer applications. Modern markup languages, like XML, simplify the exchange of information between applications, including software agents; however, XML is not yet in widespread use, it will not help with the legacy data sources not converted to the new standard, and even in the best case it will only address the problem within application domains where all interested parties can agree on the semantic schemas. Until XML becomes ubiquitous, most users will rely on the existing data extraction technologies, the most popular of which are Web wrappers.

A Web wrapper is a piece of software that enables a Web source to be queried as if it were a database. The types of sources that this applies to are what are called semistructured sources. These are sources have no explicit structure or schema, but have an implicit underlying structure. 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 an online catalog, has a very regular structure that can be exploited to extract all the data automatically.

Web wrappers rely on extraction rules to identify the beginning and end of the data field to be extracted. Semi-automatic creation of extraction rules, the so-called wrapper generation, has been an active area of research in recent years [Knoblock *et al.*, 2001b, Kushmerick *et al.*, 1997]. The most advanced of these wrapper generation systems use machine learning techniques to learn extraction rules by example. For example, the wrapper induction tool developed at USC [Knoblock *et al.*, 2001b, Muslea *et al.*, 1998] allows the user to mark up data to be extracted on several pages from an online source using a graphical user interface. The system then generates extraction rules for these data. The USC wrapper tool is able to efficiently create extraction rules from a small number of examples; moreover, it can extract data from pages that contain lists, nested structures, and other complicated formatting layouts.

In comparison to wrapper induction, wrapper maintenance has received less attention. This is an important problem, because even slight changes in the Web page layout can break a wrapper and prevent it from extracting data correctly. Wrapper verification applications monitor the validity of data returned by the wrapper. If the site changes, the wrapper may extract nothing at all or some data that is not correct. The verification application will detect data inconsistency and notify the operator or automatically launch a wrapper repair process. Wrapper reinduction repairs the extraction rules so that the wrapper works on changed pages.

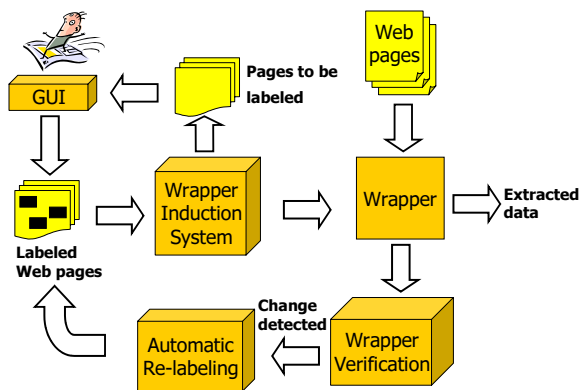


Figure 1: Life cycle of a wrapper

Figure 1 graphically illustrates the entire life cycle 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 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 structure of the extracted data. If a change is detected, the system can automatically repair

a wrapper by using this structural information to locate examples on the changed pages and re-running the wrapper induction system.

At the core of the wrapper maintenance applications is a machine learning algorithm that learns the structural information about common data fields. We introduced this algorithm, DataPro, in an earlier paper [Lerman and Minton, 2000], and briefly described its use in the wrapper maintenance applications. In this paper we expand the discussion of the two applications and describe modifications to the DataPro algorithm that significantly improves the performance of the verification task. Though we focus on web applications, the learning technique is not web-specific, and can be used for data validation in general.

2 Pattern Learning

Our objective is to learn the structure of data fields, such as addresses. Several examples of street addresses are given below.

4676 Admiralty Way
10924 Pico Boulevard
512 Oak Street
2431 Main Street
5257 Adams Boulevard

Figure 2: Examples of a street address field

2.1 Data Representation

In previous research, the structure of information extracted from Web pages was described by a sequence of characters [T. Goan and Etzioni, 1996] or a collection of global features, such as the number of words and the density of numeric characters [Kushmerick, 1999]. We employ an intermediate word-level representation that balances the descriptive power and specificity of the character-level representation with the compactness and computational efficiency of the global representation. Words, or more accurately tokens, are strings generated from an alphabet containing different types of characters: alphabetic, numeric, punctuation, etc. We use the token’s character types to assign it to one or more syntactic categories: alphabetic, numeric, etc. These categories form a hierarchy depicted in Figure 3, where the arrows point from more general to less general categories. A unique specific token type is created for every string that appears in at least k examples, as determined in a preprocessing step. The hierarchical representation allows for multi-level generalization. Thus, the token “California” belongs to the general token types ALPHANUM (alphanumeric strings), ALPHA (alphabetic strings), CAPS (capitalized words), as well as to the specific type representing the string “California”. This representation is flexible and may be expanded to include domain specific information. For example, the numeric type is divided into categories that include range information about the number: 1–, 2–, and 3–digit, and although we have not done so, information about the length of alphabetic strings may also be added to the hierarchy. Likewise, we may explicitly include knowledge about the type of information being parsed, e.g., some 5-digit numbers could be represented as ZIPCODE.

We claim that a sequence of specific and general token types is more useful for describing the content of information than the character-level finite state representations used in previous works [Carrasco and Oncina, 1994b, T. Goan and Etzioni, 1996]. The character-level description is far too fine grained to capture the regularities shared by data types commonly found on Web pages, which may be quite complex and have much variability. The coarse-grained token-level representation is more general, and thus it is more appropriate for most Web data types. In addition, the data representation schemes used in previous works attempt to describe the entire data field, while we use only the starting and ending sequences, or patterns, of tokens to capture the structure of the data fields. The reason for this is similar to the one above: using the starting and ending patterns allows us to generalize the structural information for many complex fields which have a lot of variability. Such fields, e.g., addresses, usually still have some regularity in how they start and end that we can exploit. We call the starting and ending patterns collectively a *data prototype*. As

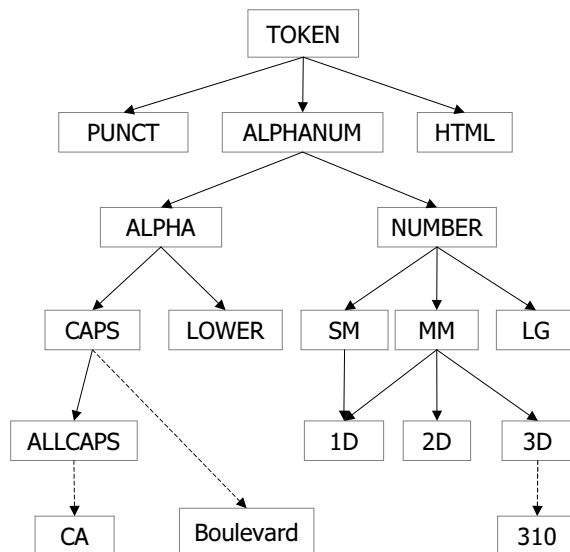


Figure 3: The token type syntactic hierarchy

an example, consider a set of street addresses in Fig. 2. All of the examples start with a pattern `<NUMBER CAPS>` and end with a specific type `<Boulevard>` or more generally `<CAPS>`. Note that these patterns are not regular expressions, since they don't allow loops or recursion. We believe that recursive expressions tend to over-generalize and are not useful representations of the types of data we are trying to learn.

2.2 Learning from Positive Examples

The problem of learning the data prototype from a set of examples that are labeled as belonging (or not) to a class, may be stated in one of two related ways: as a classification or as a conservation task. In the classification task, both positive and the negative instances of the class are used to learn a rule that will correctly classify new examples. Classification algorithms, like FOIL [Quinlan, 1990] use negative examples to guide the specialization of the rule. They construct discriminating descriptions - those that are satisfied by most of the positive examples and none of the negative examples. The conservation task, on the other hand, attempts to find a characteristic description [Dietterich and Michalski, 1981] or conserved patterns [Brazma *et al.*, 1995], in a set of positive examples of a class. Unlike the discriminating description, the characteristic description will often include redundant features. For example, when learning a description of street addresses, with city names serving as negative examples, a classification algorithm will learn that `<NUMBER>` is a good description, because all the street addresses start with it and none of the city names do. The capitalized word that follows the number in addresses is a redundant feature, because it does not add to the discriminating power of the learned description. However, an application using this description encounters a zipcode in the future, it will incorrectly classify it as a street address. This problem could have been avoided if `<NUMBER CAPS>` was learned as a description of street addresses. Therefore, when negative examples are not available to the learning algorithm, the description has to capture all the regularity of data, including the redundant features, in order to correctly identify new instances of the class and differentiate them from other classes. The characteristic description learned from positive examples is the same as the discriminating description learned by the classification algorithm in the presence of negative examples drawn from infinitely many classes. While most of the widely used machine learning algorithms (decision trees [Quinlan, 1993] and ILP [Muggleton, 1991]) solve the classification task, fewer algorithms that learn characteristic descriptions are in use [Dietterich and Michalski, 1981, Brazma *et al.*, 1995].

In our applications, an appropriate source of negative examples is problematic; therefore, we chose to frame the learning problem as a conservation task. In [Lerman and Minton, 2000] we introduced an algorithm

that learns data prototypes from positive examples of the data field alone. The algorithm finds statistically significant sequences of tokens. A sequence of token types is significant if it occurs more frequently than would be expected if the tokens were generated randomly and independently of one another, i.e., one that describes many of the positive examples of data and is highly unlikely to have been generated by chance. We begin by estimating the baseline probability of a token type’s occurrence from the proportion of all types in the examples of the data field that are of that type. Suppose we are learning a description of a set of street addresses in Fig. 2, and have already found a significant token sequence — e.g., the pattern consisting of the single token <NUMBER> — and want to determine whether the more specific pattern, <NUMBER CAPS>, is also a significant pattern. Knowing the probability of occurrence of the type CAPS, we can compute how many times CAPS can be expected to follow NUMBER completely by chance. If we observe a considerably greater number of these sequences, we conclude that the longer pattern is also significant.

Formally, we use hypothesis testing (Papoulis 1990) to decide whether a pattern is significant. The null hypothesis is that observed instances of this pattern were generated by chance, via the random, independent generation of the individual token types. Hypothesis testing decides, at a given confidence level, whether the data supports rejecting the null hypothesis. Suppose n identical sequences have been generated by a random source. The probability that a token type T (whose overall probability of occurrence is p) will be the next type in k of these sequences has a binomial distribution. For a large n , the binomial distribution approaches a normal distribution $P(x, \mu, \sigma)$ with $\mu = np$ and $\sigma^2 = np(1 - p)$. The cumulative probability is the probability of observing at least n_1 events:

$$P(k \geq n_1) = \int_{n_1}^{\infty} P(x, \mu, \sigma) dx \tag{1}$$

We use polynomial approximation formulas (Abramowitz & Stegun 1964) to compute the value of the integral.

The significance level of the test, α , is the probability that the null hypothesis is rejected even though it is true, and it is given by the cumulative probability above. Suppose we set $\alpha = 0.05$. This means that we expect to observe at least n_1 events 5% of the time under the null hypothesis. If the number of observed events is greater, we reject the null hypothesis (at the given significance level), i.e., decide that the observation is significant. Note that the hypothesis we test is derived from observation (data); therefore, we must subtract one from the number of observed events to take into account the reduction in the number of degrees of freedom resulting from an observation. This also prevents the anomalous case when a single occurrence of a rare event is judged to be significant.

2.3 DataProG Algorithm

We now describe DataProG, the algorithm that finds statistically significant patterns in a set of token sequences. DataProG is an adaptation of the DataPro algorithm [Lerman and Minton, 2000]. During the preprocessing step the text is tokenized, and the tokens are assigned one or more syntactic types (see Figure 3), as described previously. The training examples are encoded in a prefix tree, where each node corresponds to a token type. Like DataPro, DataProG relies on significance judgements to grow the tree and prune the nodes. Every path through the resulting tree starting at the root node corresponds to a significant pattern found by the algorithm. Here we focus the discussion on the version of the algorithm that learns starting patterns. The algorithm can be easily adapted to learn ending patterns.

The tree is grown incrementally. Adding a child to a node corresponds to extending the node’s pattern by a single token type. Thus, each child represents a different way to specialize the pattern. For example, when learning addresses from the examples in Fig. 2, we may have already found that a pattern <NUMBER CAPS> is significant. The final node corresponding to the type CAPS might have three children, corresponding to the patterns <NUMBER CAPS Boulevard>, <NUMBER CAPS Street>, and <NUMBER CAPS CAPS>.

As explained previously, a child node is judged to be significant with respect to its parent node if the number of occurrences of the pattern ending at the child node is sufficiently large, given the number of occurrences of the pattern ending at the parent node and the baseline probability of the token type used to extend the pattern. A pruning step insures that each child node is also significant given its more specific siblings. If the more general pattern was judged by the original DataPro not to be significant given the more specific patterns, it was pruned from the tree. If it was significant, then the pattern was kept, but all the

occurrences of the more specific patterns were deleted from it, so that each data example was counted only once in the tree. Thus, because DataPro always kept the specific patterns, it was strongly biased towards more specific patterns. During wrapper verification experiments, we found that the specific bias led to a high proportion of false positives. We have, therefore, modified the algorithm to reduce the specific bias. We still evaluate the patterns to see whether the more general pattern is significant given the more specific pattern. If it is, the general pattern is kept, and the specific pattern is deleted. Otherwise, the general pattern is deleted, while the specific one is kept. Let us illustrate the pruning step with an example. Suppose there are 10 examples of addresses that match the pattern <NUMBER CAPS Boulevard>, and 10 addresses that match <NUMBER CAPS Street>, and both of these are judged to be significant, then if the more general <NUMBER CAPS CAPS> matches significantly more than 20 addresses, it will be retained and the more specific patterns will be pruned from the tree. If the more general pattern does not occur significantly more than 20 times, it will be deleted from the tree and the more specific ones kept. Because every example appears in one node of the tree at each level, the pruning step ensures that the size of the tree does not become exponential. This keeps the runtime of the algorithm manageable.

Once the entire tree has been expanded, the algorithm includes a pattern extraction step that traverses the tree, checking whether the pattern <NUMBER CAPS> is still significant given the more specific (longer) patterns <NUMBER CAPS Boulevard>, <NUMBER CAPS Street> or <NUMBER CAPS CAPS>. In other words, DataProG decides whether the examples described by <NUMBER CAPS> but not by any of the longer sequences can be explained by the null hypothesis.

We present the pseudocode of the DataProG algorithm below and describe it in greater detail.

Like the original DataPro, DataProG grows the prefix tree by finding specializations of the significant patterns and pruning nodes. The tree is empty initially, and children are added to the root node. The children represent all tokens that occur in the first position in the training examples more often than expected by chance. The tree is extended incrementally at each node Q . A new child is added to Q for every significant specialization of the pattern ending at Q .

In the pruning step, the algorithm checks the children of Q and prunes the less significant of the general and specific patterns. In our notation, $A \subset B$ means that B is a generalization of A . The differences between the original algorithm and the new version lie in the pruning step. DataPro always keeps the specific patterns, and DataProG does not. Instead, it compares each pair of sibling nodes: if the more general pattern is found to be significant given the specific pattern, it is kept while the node corresponding to the specific pattern is deleted from the tree.¹ If the general pattern is not significant, it is pruned, and the node corresponding to the specific pattern is kept. This modification reduces the specific bias and results in more general patterns.

In the final step the algorithm extracts all significant patterns from the tree. Here, the significance judgement is made about whether the shorter (more general) pattern is significant given the longer specializations of it, which amounts to testing whether the excess number of examples that are explained by the shorter pattern, and not by the longer patterns, is significant. Any pattern that ends at a terminal node of the tree is significant. All significant patterns are added to the list. Note that the set of significant patterns may not cover all the examples in the data set, just a significant fraction of them. Figures 2.3–2.3 show examples of several data fields from a yellow pages-type source and a stock quote source, as well as the patterns learned for each field.

In order to find the common ending patterns shared by a set of data examples, we can reverse the order of tokens in every example and apply the algorithm above, or equivalently, start at the end of the examples and grow significant patterns to the left.

2.4 Previous Work on Pattern Learning

Grammar induction Several researchers have addressed the problem of learning the structure, or patterns, of text data. Grammar induction algorithms have been used in the past to learn the common structure of a set of strings. Carrasco and Oncina proposed ALERGIA [Carrasco and Oncina, 1994a], a stochastic grammar induction algorithm that learns a regular language from positive examples of the language. ALERGIA starts with a finite state automaton (FSA) that is initialized to be a prefix tree

¹We have experimented with different procedures and found it useful to not prune the children of the root node, because early pruning may appreciably reduce the number of examples available to the algorithm.

DATAPROG MAIN LOOP

```
Create root node of tree;
For next node Q of tree
    Create children of Q;
    Prune nodes;
Extract patterns from tree;
```

CREATE CHILDREN OF Q

```
For each token type T at next position in examples
Let C = NewNode;
Let C.token = T;
Let C.examples = Q.examples that are followed by T;
Let C.count = |C.examples|;
Let C.pattern = concat(Q.pattern, T);
If Significant(C.count, Q.count, T.probability,  $\alpha$ )
    AddChildToTree(C, Q);
End If
```

PRUNE NODES

```
For each child C of Q
    For each sibling S of C
        Let  $N = C.count - (S.count \cap S.pattern \subset C.pattern)$ 
        If Not(Significant( $N, Q.count, C.token.probability, \alpha$ ))
            Delete C;
            break;
        Else
            Delete S;
        End If
    End S loop
End C loop
```

EXTRACT PATTERNS FROM TREE

```
Create empty list;
For every node Q of tree
    For every child C of Q
        Let  $N = C.count - \sum_i (S_i.count | S_i \in \{Children(C)\})$ 
        If Significant( $N, Q.count, C.token.probability, \alpha$ )
            Add C.pattern to the list;
    End For
Return (list of patterns);
```

BUSINESS NAME	ADDRESS
Chado Tea House	8422 West 1st Street
Saladang	363 South Fair Oaks Avenue
Information Sciences Institute	4676 Admiralty Way
Chaya Venice	110 Navy Street
Acorda Therapeutics	330 West 58th Street
Cajun Kitchen	420 South Fairview Avenue
Advanced Medical Billing Services	9478 River Road
Vega 1 Electrical Corporation	1723 East 8th Street
21st Century Foundation	100 East 85th Street
TIS the Season Gift Shop	15 Lincoln Road
Hide Sushi Japanese Restaurant	2040 Sawtelle Boulevard
Afloat Sushi	87 East Colorado Boulevard
Prebica Coffee & Cafe	4325 Glencoe Avenue
L ' Orangerie	903 North La Cienega Boulevard
Emils Hardware	2525 South Robertson Boulevard
Natalee Thai Restaurant	998 South Robertson Boulevard
Casablanca	220 Lincoln Boulevard
Antica Pizzeria	13455 Maxella Avenue
NOBU Photographic Studio	236 West 27th Street
Lotus Eaters	182 5th Avenue
Essex On Coney	1359 Coney Island Avenue
National Restaurant	273 Brighton Beach Avenue
Siam Corner Cafe	10438 National Boulevard
Grand Casino French Bakery	3826 Main Street
Alejo ' s Presto Trattoria	4002 Lincoln Boulevard
Titos Tacos Mexican Restaurant Inc	11222 Washington Place
Killer Shrimp	523 Washington Boulevard
Manhattan Wonton CO	8475 Melrose Place
Starting patterns	
<ALPHA CAPS>	<NUMBER CAPS CAPS>
<ALPHA CAPS CAPS Restaurant>	<NUMBER CAPS CAPS Avenue>
<ALPHA '>	<NUMBER CAPS CAPS Boulevard>
Ending patterns	
<CAPS CAPS>	<CAPS CAPS>
<CAPS CAPS CAPS>	<LGNUM CAPS CAPS>
	<3DIGIT SOUTH CAPS CAPS>

Figure 4: Examples of the business name and address fields from the *bigbook* source, and the patterns learned from them

CITY	STATE	PHONE
Los Angeles	CA	(323) 655 - 2056
Pasadena	CA	(626) 793 - 8123
Marina Del Rey	CA	(310) 822 - 1511
Venice	CA	(310) 396 - 1179
New York	NY	(212) 376 - 7552
Goleta	CA	(805) 683 - 8864
Marcy	NY	(315) 793 - 1871
Brooklyn	NY	(718) 998 - 2550
New York	NY	(212) 249 - 3612
Buffalo	NY	(716) 839 - 5090
Los Angeles	CA	(310) 477 - 7242
Pasadena	CA	(626) 792 - 9779
Marina Del Rey	CA	(310) 823 - 4446
West Hollywood	CA	(310) 652 - 9770
Los Angeles	CA	(310) 839 - 8571
Los Angeles	CA	(310) 855 - 9380
Venice	CA	(310) 392 - 5751
Marina Del Rey	CA	(310) 577 - 8182
New York	NY	(212) 924 - 7840
New York	NY	(212) 929 - 4800
Brooklyn	NY	(718) 253 - 1002
Brooklyn	NY	(718) 646 - 1225
Los Angeles	CA	(310) 559 - 1357
Culver City	CA	(310) 202 - 6969
Marina Del Rey	CA	(310) 822 - 0095
Culver City	CA	(310) 391 - 5780
Marina Del Rey	CA	(310) 578 - 2293
West Hollywood	CA	(323) 655 - 6030
Starting patterns		
<CAPS CAPS>	<ALLCAPS>	<(3DIGIT) 3DIGIT - LGNUM>
<CAPS CAPS Rey>		
Ending patterns		
<CAPS CAPS>	<ALLCAPS>	<(3DIGIT) 3DIGIT - LGNUM>

Figure 5: Examples of the city, state and phone number fields from the *bigbook* source, and the patterns learned from them

PRICE CHANGE	TICKER	VOLUME	PRICE
+ 0 . 51	INTC	17 , 610 , 300	122 3 / 4
+ 1 . 51	IBM	4 , 922 , 400	109 5 / 16
+ 4 . 08	AOL	24 , 257 , 300	63 13 / 16
+ 0 . 83	T	8 , 504 , 000	53 1 / 16
+ 2 . 35	LU	9 , 789 , 300	68
	ATHM	5 , 646 , 400	29 7 / 8
- 10 . 84	COMS	15 , 388 , 200	57 11 / 32
- 1 . 24	CSCO	19 , 135 , 900	134 1 / 2
- 1 . 59	GTE	1 , 414 , 900	65 15 / 16
- 2 . 94	AAPL	2 , 291 , 800	117 3 / 4
+ 1 . 04	MOT	3 , 599 , 600	169 1 / 4
- 0 . 81	HWP	2 , 147 , 700	145 5 / 16
+ 4 . 45	DELL	40 , 292 , 100	57 3 / 16
+ 0 . 16	GM	1 , 398 , 100	77 15 / 16
- 3 . 48	CIEN	4 , 120 , 200	142
+ 0 . 49	EGRP	7 , 007 , 400	25 7 / 8
- 3 . 38	HLIT	543 , 400	128 13 / 16
+ 1 . 15	RIMM	307 , 500	132 1 / 4
	C	6 , 145 , 400	49 15 / 16
- 2 . 86	GPS	1 , 023 , 600	44 5 / 8
- 6 . 46	CFLO	157 , 700	103 1 / 4
- 0 . 82	DCLK	1 , 368 , 100	106
+ 2 . 00	NT	4 , 579 , 900	124 1 / 8
+ 0 . 13	BFRE	149 , 000	46 9 / 16
- 1 . 63	QCOM	7 , 928 , 900	128 1 / 16
Starting patterns			
<PUNCT 1DIGIT . 2DIGIT>	<ALLCAPS>	<NUMBER , 3DIGIT , 3DIGIT>	<MEDNUM 1DIGIT / NUMBER> <MEDNUM 15 / 16 >
Ending patterns			
<PUNCT 1DIGIT . 2DIGIT>	<ALLCAPS>	<1DIGIT , 3DIGIT , 3DIGIT> <2DIGIT , 3DIGIT , 3DIGIT>	<MEDNUM 1DIGIT / NUMBER> <2DIGIT 15 / NUMBER>

Figure 6: Data examples from the *yahoo quote* source, and the patterns learned from them

that represents all the strings of the language. ALERGIA uses a state-merging approach [Angluin, 1982, Stolcke and Omohundro, 1994] in which the FSA is generalized by merging pairs of statistically similar (at some significance level) subtrees. Similarity is based purely on the relative frequencies of substrings encoded in the subtrees. The end result is a minimum FSA that is consistent with the grammar.

Goan et al. [T. Goan and Etzioni, 1996] found that when applied to data domains commonly found on the Web, such as addresses, phone numbers, *etc.*, ALERGIA tended to merge too many states, resulting in an over-general grammar. They proposed modifications to ALERGIA, the algorithm WIL, aimed at reducing the number of faulty merges. The modifications were motivated by the observation that each symbol in a string belong to one of the following syntactic categories: NUMBER, LOWER, UPPER and DELIM. When viewed on the syntactic level, data strings contain additional structural information that can be effectively exploited to reduce the number of faulty merges. WIL merges two subtrees if they are similar (in the ALERGIA sense) and also if, at every level, they contain nodes that are of the same syntactic type. WIL also adds a wildcard generalization step in which the transitions corresponding to symbols of the same category that are approximately evenly distributed over the range of that syntactic type (*e.g.*, 0–9 for numerals) are replaced with a single transition corresponding to the type (*e.g.*, NUMBER). Goan *et al.* demonstrated that the grammars learned by WIL were more effective in recognizing new strings in several Web-relevant domains.

In order to compare DataProG to WIL on the same level of data, we used WIL to learn the grammar on the token level using data examples extracted by the wrappers, not on the character level as was done by Goan *et al.* Another difference was that, whereas Goan *et al.* needed on the order of 100 strings to arrive at a high accuracy rate, we have on the order of 20–30 examples to work with. Note that we can no longer apply the wildcard generalization step to the FSA, because we would need many more examples than we actually have to decide whether the token is approximately evenly distributed over that syntactic type. Instead, we compare DataProG against two versions of WIL: one without wildcard generalization (WIL1), and one in which every token in the initial FSA is replaced by its syntactic type (WIL2). In addition to the syntactic types used by Goan *et al.*, we also had to introduce another type ALNUM to be consistent with the examples. Neither version allows for multi-level generalization.

The algorithms were tested on the data extracted by wrappers from 26 Web sources on ten different occasions over a period of several months. Each time results of 20 – 30 queries were stored. For each wrapper, one data set was used as the training examples, and the data set extracted on the very next date was used as test examples. We used WIL1 and WIL2 to learn the grammar of each field of the training examples and then used the grammar to recognize the test examples. If the grammar recognized more than 80% of the test examples of a data field, we concluded that it recognized the entire data field; otherwise, we concluded that the grammar did not recognize the field, possibly because the data itself has changed. This is the same procedure we used in the wrapper verification experiments, and it is described in greater detail in Section 3.1.1. Over the period of time covered by the data, there were 21 occasions on which a Web site changed, thereby causing the data extracted by the wrapper to change as well. The precision and recall values for WIL1 (grammar induction on specific tokens) were $P = 0.20$, and $R = 0.81$; for WIL2 (grammar induction on wildcards representing tokens’ syntactic categories) the values were $P = 0.55$ and $R = 0.76$. WIL2 will result in over-general grammar. The recall and precision of DataProG tested on the same data set were $P = 0.73$ and $R = 1.0$.

FOIL As a sequence of n tokens, a pattern can also be viewed as a non-recursive n -ary predicate. Therefore, we can use a relation-learning algorithm like FOIL [Quinlan, 1990] to learn them. Given a set of positive and negative examples of a class, FOIL learns first order predicate logic clauses defining the class. Specifically, it finds a discriminating description that covers many positive and none of the negative examples.

We used foil.6 with no-negative-literals option to learn patterns describing several different data fields. In all cases the closed world assumption was used to construct negative examples from the known objects: thus, for the *bigbook* source, names and addresses were the negative examples for the phone number class. We used the following encoding to translate the training examples to allow foil.6 to learn logical relations. For each data field, FOIL learned clauses of the form

$$data_field(A) : \neg P(A) followed_by(A, B) P(B), \tag{2}$$

as a definition of the field, where A and B are tokens, and the terms on the right hand side are predicates. Predicate $followed_by(A, B)$ expresses the sequential relation between the tokens. The predicate $P(A)$ allows us to specify the token A as a specific token (e.g., $John(A)$) or a general type (e.g., $CAPS(A)$, $ALPHA(A)$), thus, allowing FOIL the same multi-level generalization capability as DataProG.

We ran foil.6 on the examples associated with the *bigbook* source (see Figs. 2.3–2.3) and *yahoo quote* source (see Fig. 2.3). For the *bigbook* examples, foil.6 learned the following definitions:

*** Warning: the following definition does not cover 23 tuples in the relation

```
NAME(A) :- followed_by(A,B), ALLCAPS(A)
NAME(A) :- CAPS(A), followed_by(A,B), NUMBER(B)
NAME(A) :- followed_by(A,B), Venice(B)
```

```
STREET(A) :- followed_by(A,B), LGNUM(A)
STREET(A) :- followed_by(A,B), MEDNUM(A), ALPHANUM(B)
```

** Warning: the following definition does not cover 9 tuples in the relation

```
CITY(A) :- Los(A)
CITY(A) :- Marina(A)
CITY(A) :- New(A)
CITY(A) :- Brooklyn(A)
CITY(A) :- West(A), followed_by(A,B), ALPHA(B)
```

```
STATE(A) :- CA(A)
STATE(A) :- NY(A)
```

```
PHONE(A) :- ((A))
```

For *yahoo quote* source foil.6 learned the following definitions:

```
PRICECHANGE(A) :- +(A)
PRICECHANGE(A) :- -(A)
```

```
SHAREPRICE(A) :- MEDNUM(A), followed_by(A,B), followed_by(B,C), ALLCAPS(C)
```

```
TICKER(A) :- ALLCAPS(A)
```

** Warning: the following definition does not cover 9 tuples in the relation

```
VOLUME(A) :- 1DIGIT(A), followed_by(A,B), followed_by(B,C), 3DIGIT(C)
```

In these cases and others we tried, there were many similarities between the definitions learned by FOIL and the patterns learned by DataProG; however, FOIL clauses tended to be overly general. For example, FOIL learned that ‘(’ was a good description of phone numbers for the *bigbook* source. The over-general description will result in poor performance on the reinduction task (Sec. 3.2), where the patterns are used to recognize instances of the data field. We attribute over-generalization to the incompleteness of the set of negative examples presented to FOIL. Another problem was when given examples of a class with little structure, such as names and book titles, FOIL tended to create clauses that covered single examples, or it failed to find any clauses.

3 Applications of Pattern Learning

As we explained in the introduction, because wrappers use information from the layout of HTML pages to create data extraction rules, they are vulnerable to changes in the layout, which occur frequently when the site is redesigned. In some cases the wrapper continues to extract, but the data is no longer correct. The output of the wrapper may also change because the format of the source data itself has changed: e.g., when

street number is dropped from the address field (“Main St” instead of “25 Main St”), or book availability changes from “Ships immediately” to “In Stock: ships immediately.” Because other applications, such as information aggregation [Knoblock *et al.*, 2001a] or agent-based applications [Chalupsky *et al.*, 2001], rely on the data extracted by the wrapper from a Web page, it is important to detect when the wrapper is no longer extracting correct data from the page. Wrapper maintenance, i.e., detecting when the wrapper stops working and automatically recovering from failure, is therefore an important research problem.

3.1 Wrapper Verification

The prototypes learned by DataProG 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 training examples. DataProG learns patterns that describe the common beginnings and endings of each field of the extracts. In the verification phase, the wrapper generates a set of test examples from pages retrieved using the same set of queries. Suppose that r_i training examples and t_i test examples match the i th pattern, as shown in Fig. 7. If the two distributions, \vec{k} and \vec{r} , are statistically the same (at some significance level), the wrapper is judged to be extracting correctly; otherwise, it is judged to have failed. The approach is not limited to patterns, but allows us to add other features to the two distributions describing the examples.

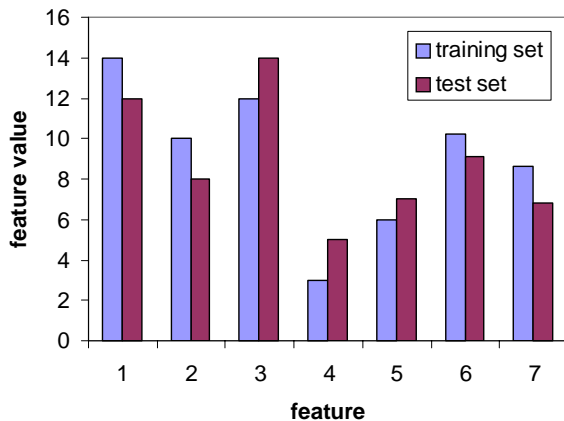


Figure 7: Feature distribution over the training and test examples. Features can be a pattern, whose value is the number of the examples in the set that satisfy the pattern, or a numeric feature, such as the average length of the examples.

Each distribution is described by a vector, the i th component of which is the value of the i th feature, such as the number of examples that match the pattern or another numeric feature. In addition to patterns, we use the following numeric features to describe the sets of training and test examples: the average number of tuples-per-page, mean number of tokens in the examples, mean token length, and the density of alphabetic, numeric, HTML-tag and punctuation types. We use goodness of fit method (Papoulis 1990) to decide whether the two distributions are the same. To use the goodness of fit method, we must first compute Pearson’s test statistic for the data. The Pearson’s test statistic is defined as:

$$q = \sum_{i=1}^m \frac{(t_i - e_i)^2}{e_i} \quad (3)$$

where t_i is observed value of the i th feature in the test data, and e_i is the expected value for that feature, and m is the number of features. For the patterns $e_i = nr_i/N$, where N is the number of examples in the training set and n is the number of examples in the test set, and for the numeric features it is simply the value of that feature for the training set. The test statistic q has a chi-squared distribution with $m - 1$ independent degrees of freedom. If $q < \chi^2(m - 1; \alpha)$, we conclude that at significance level α the two distributions are

Source	Type	Data Fields
<i>airport</i>	t//1	airport code, name
<i>altavista</i>	list	url, title
<i>amazon</i>	tuple	book author, title, price, availability, isbn
<i>arrowlist</i>	list	part number, manufacturer, price, status, description, url
<i>bigbook</i>	tuple	business name, address, city, state, phone
<i>barnes&noble</i>	tuple	book author, title, price, availability, isbn
<i>borders</i>	list	book author, title, price, availability
<i>cuisinenet</i>	list	restaurant name, cuisine, address, city, state, phone, link
<i>geocoder</i>	tuple	latitude, longitude, street, city, state
<i>hotel</i>	list	name, price, distance, url
<i>mapquest</i>	tuple	hours, minutes, distance, url
<i>northernlight</i>	list	url, title
<i>parking</i>	list	lotname, dailyrate
<i>quote</i>	tuple	stock ticker, price, pricechange, volume
<i>smartpages</i>	tuple	name, address, city, state, phone
<i>showtimes</i>	list	movie, showtimes
<i>theatre</i>	list	theater name, url, address
<i>washington post</i>	tuple	taxi price
<i>whitepages</i>	list	business name, address, city, state, phone
<i>yahoo people</i>	list	name, address, city, state, phone
<i>yahoo quote</i>	tuple	stock ticker, price, pricechange, volume
<i>yahoo weather</i>	tuple	temperature, forecast
<i>cia factbook</i>	tuple	country area, borders, population, etc.

Figure 8: List of sources used in the experiments and data fields extracted from them. Source type refers to how much data a source returns in response to a query — a single tuple or a list of tuples. For *airport* source, the type changed from a single tuple to a list over time.

the same; otherwise, we conclude that they are different. Values of χ^2 for different values of α and m can be looked up in a statistics table or calculated using an approximation formula.

In the series of verification experiments reported in [Lerman and Minton, 2000], we used the starting and ending patterns and the average number of tuples-per-page feature only when computing the value of q . However, we found that this method tended to overestimate the test statistic, because there was frequently substantial overlap between the starting and ending patterns, and therefore, the features were not independent as required by the goodness of fit method. In the experiments reported in this paper, we use only the starting patterns, and additional numeric features, as described above.

3.1.1 Results

We monitored 27 wrappers (representing 23 distinct Web sources) over a period of ten months, from May 1999 to March 2000. The sources are listed in Fig. 8. For each wrapper, the results of 15–30 queries were stored periodically. We used the same query set for each source, except for the *hotel* source, because it accepted dated queries, and we had to change the dates periodically to get valid results. Each set of new results (test examples) was compared with the last correct wrapper output (training examples).

The verification algorithm used DataProG to learn the starting patterns and numeric features for each field of the training examples and made a decision at a high significance level (corresponding to $\alpha = 0.001$) about whether the test set was statistically similar to the training set. If none of the starting patterns matched the test examples or if the data was found to have changed significantly for any data field, we concluded that the wrapper failed to extract correctly from the source; otherwise, if all the data fields

returned statistically similar data, we concluded that the wrapper was working correctly.

Manual check of results identified 37 wrapper changes out of the total 438 comparisons. Of these 30 were attributed to changes in the source layout and data format, and the rest to modifications internal to the wrapper (e.g., wrapper was changed to extract “\$22.00” instead of “22.00” for the price field). The verification algorithm correctly discovered 35 of these changes and made 15 mistakes. Of these, 13 were *false positives*, which means that the verification program decided that the wrapper failed when in reality it was working correctly. Only two of the errors were the more important *false negatives*, meaning that the algorithm did not detect a change in the data source. The numbers above result in the following precision, recall and accuracy values:

$$\begin{aligned}
 P &= \frac{\text{true positives}}{\text{true positives} + \text{false positives}} = 0.73, \\
 R &= \frac{\text{true positives}}{\text{true positives} + \text{false negatives}} = 0.95, \\
 A &= \frac{\text{true positives} + \text{true negatives}}{\text{all results}} = 0.97.
 \end{aligned}$$

These results are an improvement over those reported in Ref. [Lerman and Minton, 2000], which produced $P = 0.47$, $R = 0.95$, $A = 0.91$. The poor precision value reported in that work was due to 40 false positives obtained on the same data set.

3.1.2 Discussion of Results

Though we have succeeded in significantly reducing the number of false positives, we have not managed to eliminate them altogether. There are a number of reasons for their presence, some of which point to limitations in our approach. Below we list the sources of false positives:

hotel URL field (2 cases): very long field containing alphanumeric keys. Some keys may have embedded punctuation symbols, which the tokenizer splits into different tokens:

from `http://...&Stamp=Q4aaiEGSp68*itn/hot%3da11204,itn/agencies/newitn...`
to `http://...&Stamp=8 bEgGEQrCo*itn/hot%3da11204,itn/agencies/newitn...`

mapquest URL field (3 cases): very long fields containing alphanumeric keys, with embedded punctuation symbols, which our tokenizer splits into many tokens. The key or its format changed on two occasions:

from `http://...FDT7w%7clw72dz%24.9uy5qw2:%26%40%24:%26%408:qu7:9fy...`
to `http://...FDT7w%7clw72uw%24.9uy5gz7%3a%26%40%24%3a%26%40z...`

One time, the server name inside the URL changed:

from `http://enterprise.mapquest.com/mqmapgend?MQMapGenRequest=...`
to `http://sitemap.mapquest.com/mqmapgend?MQMapGenRequest=...`

showtimes showtimes field (2 cases): very long examples, many longer (more specific) patterns. Distribution of data satisfied by the patterns changes:

`<(NUMBER : 2DIGIT ALLCAPS) , (SMNUM : 2DIGIT) , 7 : 2DIGIT , NUMBER : 2DIGIT >`
`<(NUMBER : 2DIGIT ALLCAPS) , (SMNUM : 2DIGIT) , (SMNUM : 2DIGIT) , (4 : 2DIGIT) , 6 : 2DIGIT , 7 : 2DIGIT , 9 : 2DIGIT , 10 : 2DIGIT >`

arrowlist part number and description fields (3 cases): very complex fields with lots of variation. Many specific patterns created; results sensitive to the distribution of data:

`ALLCAPS PUNCT ALLCAPS HDR SLD TAIL R / A NO EAR 2DIGIT`
`ALLCAPS NUMBER / SMNUM FOR WDU 2DIGIT , SMNUM POLE`
`ALLCAPS NUMBER . 5 / 2X2AN / D PUNCT`

altavista URL field (1 case): internal database changed — results returned in response to the same queries are different.

quote price change field (1 case): data changed — more positive than negative price movements in the test examples

factbook irrigated land field (1 case): data format change:

from <NUMBER km2 >
to <NUMBER sq km >

The URL field accounted for a significant fraction of the false positives, in large part due to the design of our tokenizer, which splits text strings on punctuation marks. If the URL contains embedded punctuation (as part of the alphanumeric key associated with the user or session id, as in the sources above), it will be split into a varying number of tokens. This makes comparison between the training and test examples difficult. In addition, our algorithm sometimes failed for very long, complex fields (arrowlist, showtimes) when there was variation between the training and test data. It is worth pointing out, however, that it performed correctly in over two dozen comparisons for these sources. In some cases (arrowlist, altavista, quote), the results returned by the source changed significantly, because the underlying database has changed. This was especially striking for the stock quotes source: the day the training data was collected, there were more down movements in the stock price than up, and the opposite was true on the day the test data was collected. As a result, the price change fields for those two days were dissimilar. This problem can be solved by including a larger, more diverse set of examples in the training data. Finally, because DataProG learns the format of data, it fails when the format changes. This is the case for the factbook source, where the units of area changed from “km2” to “sq km”.

3.1.3 Previous Work on Wrapper Verification

Kushmerick [Kushmerick, 1999] addressed the problem of wrapper verification by proposing an algorithm RAPTURE to verify that a wrapper correctly extracts data from a Web page. In that work, each data field was described by a collection of global numeric features, such as word count, average word length, HTML density, alphabetic character density, etc. The type density feature was defined as the proportion of the characters in the examples that are of that type. Given a set of queries for which the wrapper output is known, RAPTURE checked that the wrapper generated a new result with the expected combination of feature values for each of the queries. Kushmerick found that the HTML density alone could correctly identify almost all of the changes in the sources he monitored. Addition of other features to the HTML density in the probability calculation about the wrapper’s correctness significantly reduced the algorithm’s performance. In our experiments, RAPTURE would have missed 17 wrapper changes (false negatives) if it were relying solely on the HTML density feature.

3.2 Wrapper Reinduction

If the wrapper stops extracting correctly, the next challenge is to rebuild it automatically [Cohen, 1999]. The extraction rules for our Web wrappers [Muslea *et al.*, 2001a], as well as many others (cf. [Kushmerick *et al.*, 1997],[Hsu and Dung, 1998]), are generated by machine learning algorithms, which take as input several pages from the same source and examples of data to extract from each page. Therefore, if we identify examples of data on pages for which the wrapper fails, we can use these examples as input to the induction algorithm to generate new extraction rules. In this paper we will only discuss wrapper reinduction for information sources that return a single tuple of data per page. In order to create data extraction rules for sources that return lists of tuples, the STALKER wrapper induction algorithm requires user to specify the first and last elements of the list, as well as at least two consecutive elements. Therefore, we need to be able to identify these data elements with high degree of certainty. Although the present wrapper reinduction algorithm cannot guarantee that all required elements will be extracted from new pages, we have already initiated work in this direction. Preliminary results are reported in [Kristina Lerman and Minton, 2001].

Below we describe a method that takes a set of training examples, extracted from the source when the wrapper was known to be working correctly, and a set of pages from the same source, and uses a mixture of supervised and unsupervised learning techniques to identify examples of the data field on new pages. We assume that the format of data did not change. Below is the outline of the reinduction algorithm:

- *Learn patterns* from the training examples
- *Identify extraction candidates* using patterns

- *Group candidates* according to positional features)
- *Score groups* according to similarity to the training examples
- *Induce the wrapper* using the examples in the highest scoring group

First, DataProG learns the patterns that describe the start and end of each data field in 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. These segments, which we call candidates, include examples of the field we want to extract from the new pages. The candidates are then clustered to identify subsets that share common features. The features used to describe each candidate are its location on the page and the context in which it appears, and whether it is visible to the user. Each cluster is then given a score based on how similar it is to the training examples. We expect the highest ranked cluster to contain the correct examples of the data field on the new pages. Next, we provide the details of each step of the algorithm.

Identify extract candidates DataProG learns the patterns that describe the training examples. These patterns are used to identify candidates for the examples of the same data fields on the new pages. 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. In addition to patterns, we also calculate the parameters (mean and variance) of the number-of-tokens distribution of the training examples. Text segments that contain significantly more or fewer tokens than expected based on this feature are eliminated from the candidate set. Among the possibly hundreds of spurious text segments on each page are the correct examples of the data field we want to extract.

Group candidates We exploit some simple a priori assumptions about the structure of Web pages to help us separate interesting extracts from noise. We expect examples of the same data field to appear roughly in the same position and in the same context on each page. For example, Fig. 9 shows fragments of two Web pages from the same source displaying restaurant information. On both pages the relevant information about the restaurant appears after the heading “ALL LISTINGS” and before the phrase “Appears in the Category:”. Also, restaurant address always follows the restaurant name (in bold) and precedes the city and zip code, i.e., it appears in the same context on every page. The same data field is usually visible to the user on every page, or it is part of an HTML tag on every page. To use this information, we describe each candidate extract by a feature vector, which includes positional information, context and visibility. The context is captured by the adjacent tokens, and it is represented by two numbers, one for the token immediately preceding the candidate extract and one for the token immediately following it. The number is a unique integer corresponding to each distinct token. The final binary feature is one if the token is visible to the user and zero if it is part of an HTML tag.

Calculation of the positional information requires more explanation. Many Web sources use templates, or page skeletons, to automatically generate pages and fill them with results of a database query. This is evident in the example in Fig. 9. The template consists of the heading “RESULTS”, followed by the number of results that match the query, the phrase “Click links associated with businesses for more information”, then the heading “ALL LISTINGS”, followed by the anchors “map”, “driving directions” and “add to My Directory”, and the bolded phrase “Appears in the Category.” Obviously, data is not part of the template — rather, it appears in the *slots* between template elements. Candidate extracts that are part of the page template are, therefore, eliminated from consideration. We expect the same field, e.g., address, to appear in the same slot on each page. The value of the position feature is the index of the slot in which the candidate extract appears.

Given two or more example pages from the same source, we can induce the template used to generate them (see Figure 10). The template finding algorithm looks for all sequences of tokens — both HTML tags and text — that appear exactly once on each page. The algorithm works in the following way: we pick the smallest page in the set as the template seed. Starting with the first token on this page, we grow a sequence by appending tokens to it, subject to the condition that the sequence appears on every page. If we managed

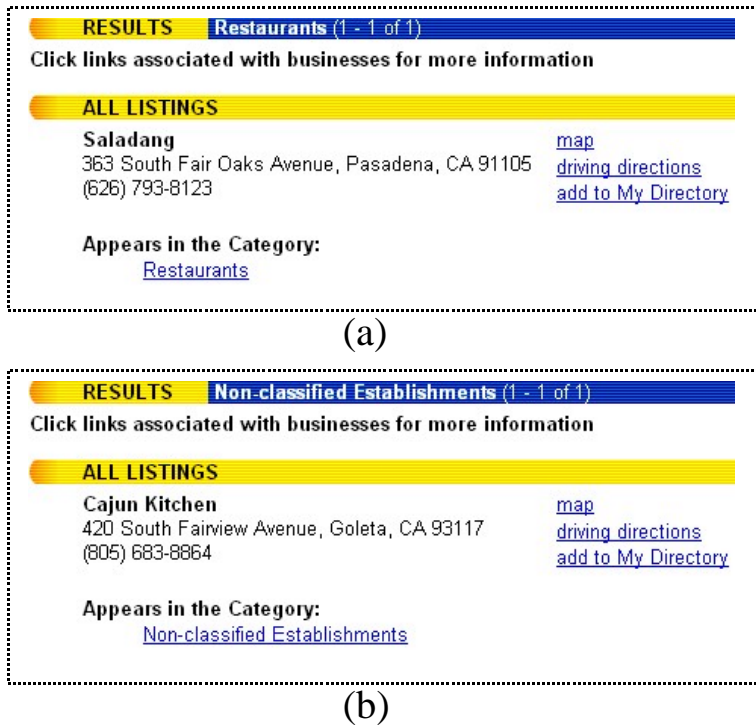


Figure 9: Fragments of two Web pages from the same source displaying restaurant information.

to build a sequence that’s at least three² tokens long, and this sequence appears exactly once on each page, it becomes part of the page template.

Score groups Once the candidates have been assigned feature vectors, we split them into groups. Within each group, the candidate extracts are described by the same feature vector. The next step is to score clusters. We expect the highest scoring cluster to contain the correct examples of the data field. The clusters are scored based on how many extracts they have in common with the training examples. This technique generally works well, because at least some of the data usually remains the same when the Web page layout changes. Of course, this assumption does not apply to data that changes frequently, such as weather information, flight arrival times, stock quotes, etc. However, we have found that even in these sources, there is enough overlap in the data that our approach works. If the scoring algorithm fails to assign a score to any cluster, a second scoring algorithm is invoked. This scoring method follows the wrapper verification procedure and finds the cluster that is most similar to the training examples based on the patterns learned from the training examples.

Induce the wrapper The final step of the wrapper reinduction process is to provide the extracts in the top ranking cluster to the STALKER wrapper induction algorithm [Muslea *et al.*, 2001b] along with the new pages. STALKER learns data extraction rules for the new changed pages.

3.2.1 Results

To evaluate the reinduction algorithm we used only the ten sources (listed in Fig. 8) that returned a single tuple of data, except for the *geocoder* and *cia factbook*. The *geocoder* wrapper accessed the source through another application; therefore, the pages were not available for analysis. The reason for excluding the *factbook* is that it is a formatted text source, while our methods apply to Web pages. Note also that in the

²The best value for the minimum length for the page template element was determined empirically to be three.

```

input:
    P = set of N Web pages
output:
    T = page template
begin
    p = minimum(P)
    T = null
    s = null
    for t = firsttoken(p) to lasttoken(p)
        s' = concat(s, t)
        if ( s' appears on every page in P )
            s = s'
            continue
        else
            n =  $\sum_{page=1}^N$  count(s, page)
            if ( n = N AND length(s)  $\geq$  3 )
                add(s, T)
            end if
            s = null
        end if
    end for
end

```

Figure 10: *Pseudocode of the template finding algorithm*

verification experiments, we had two wrappers for the *mapquest* source, each extracting different data. In the experiments described below, we used one of them that have more data during the time period listed.

The method of data collection was described in Sec. 3.1.1. In the reinduction experiments, we used a subset of the data, collected between October 1999 and March 2000. Over this period of time, there were eight format changes in the ten sources we studied. We evaluated the algorithm by using it to extract data from Web pages for which correct output is known. Specifically, we took ten tuples from a data set collected on one date, and used this information to extract data from ten pages (not necessarily those returned by the same queries) collected on a later date, regardless of whether the source had actually changed or not, for a total of ten such comparisons for each wrapper. We reserved the rest of the pages in the set for testing the learned STALKER rules.

The output of the reinduction algorithm is a list of tuples from the set of ten pages, as well as extraction rules generated by STALKER for these pages. Though in most cases we were not able to extract every data element in every tuple, we can still learn good extraction rules, because most of the time STALKER requires a small number of labeled examples for learning. We evaluated the reinduction algorithm in two stages: first, we checked how many data fields for each source were identified successfully; second, we checked the quality of the learned STALKER rules by using them to extract data from test pages.

Automatic data extraction We judged a data field to be successfully extracted if the reinduction algorithm was able to identify it correctly on at least two of the ten pages. In practice, such a low success rate only occurred for one field each in two of the sources: *quote* and *yahoo quote*. For all other wrapper, if a field was successfully extracted, it was correctly identified in at least three, and in most cases almost all, of the pages in the set. A false positive occurred when the reinduction algorithm incorrectly identified some text on a page as an example of a data field. In many cases, false positives contained partial examples of the field, e.g., “Cloudy” rather than “Mostly Cloudy” for the *yahoo weather* source. If the reinduction algorithm did not identify any text as an example of a data field for the wrapper, we counted this as a false negative. We ran the reinduction experiment on ten different sets of pages for each of the ten wrappers listed in Fig. 11, attempting to extract a total of 338 fields. We were able to correctly identify 277 of the fields, and made 61

Source	num fields	avg fields	extract %	extract %
<i>airport</i>	2	1.9	100	
<i>amazon</i>	5	4.2	90	75
<i>bigbook</i>	5	3.3	54	
<i>barnes&noble</i>	5	4.4	83	86
<i>mapquest</i>	2	2.0	98	
<i>quote</i>	4	2.1	63	50
<i>smartpages</i>	5	4.1	77	40
<i>washington post</i>	1	1.0	100	
<i>yahoo quote</i>	4	3.4	82	75
<i>yahoo weather</i>	2	1.3	58	

Figure 11: List of sources used in reinduction experiments and results. The second and third columns list the number of data fields for which we have training examples, and the average number of fields successfully extracted by the reinduction algorithm. The fourth column lists the percentage of fields for which STALKER rules could be learned that were successfully extracted from test pages. The last column lists the percentage of all the data fields that were correctly extracted from the test pages for the sources that changed.

mistakes, of which 31 were attributable to false positives and 30 to the false negatives, resulting in precision $P = 0.90$ and recall $R = 0.90$.

The table in Fig. 11 lists some of the reasons the reinduction algorithm failed to extract data from pages correctly. For the yellow pages-type sources *bigbook* and *smartpages*, we can significantly improve results by using more data for the training examples: by using 20 or 30 examples for each field rather than ten, we can achieve 100% success rate. In two cases, the errors were attributable to changes in the data format, which resulted in patterns failing to capture the structure of data correctly: the *airport* source changed airport names from capitalized words to allcaps, and in the *quote* source in which the patterns were not able to identify negative price changes because they were learned for a data set in which most of the stocks had a positive price change. In two sources the reinduction algorithm could not distinguish between correct examples of the field and other examples of the same data type: for the *quote* source, in some cases it extracted opening price or high price for the stock price field, while for the *yahoo weather* source, it extracted high or low temperature, rather than the current temperature. In these cases, user intervention may be necessary to improve results of automatic data extraction.

Extraction with STALKER The final validation experiment consisted of using the automatically generated STALKER rules to extract data from test pages. Column four in Table 11 lists the percentage of the data fields that were correctly extracted. We only considered those fields for which STALKER rules could be created. As noted above, the automatic data extraction step did not always generate enough examples for some data fields for the wrapper induction system to learn correct extraction rules. Overall, we obtained $P = 0.88$ and $R = 0.92$ for precision and recall.

Within the data set we studied, five sources, listed in bold in Table 11, experienced a total of seven changes. In addition to these sources, the *airport* source changed the format of the data it returned, but since it simultaneously changed the presentation of data from a single tuple to a list of tuples, we could not use this data to learn STALKER rules. The last column in Table 11 list the percentage of *all* data fields on the test pages that were correctly extracted by the automatically learned STALKER rules, including those fields for which an insufficient number of examples was identified during the data extraction step. The recall and precision values for the changed sources were $P = 0.90$ and $R = 0.75$. These results can be substantially improved if we include more training examples in the set used by DataProG to learn the patterns to describe the data (in the automatic data extraction step). Using 15 rather than ten examples leads to the following percentage of the total number of data fields correctly extracted from the test pages: *amazon* = 90%, *barnes&noble* = 88%, *quote* = 50%, *smartpages* = 60%, *yahoo quote* = 100%, with $P = 0.94$, $R = 0.86$.

Lists We have also applied the reinduction algorithm to extract data from pages containing lists of tuples, and, in many cases, have successfully extracted at least several examples of each field from several pages.

However, in order to learn the correct extraction rules for sources returning lists of data, STALKER requires that the first, last and at least two consecutive list elements be correctly specified. The methods presented here cannot guarantee that the required list elements are extracted, unless all the list elements are extracted. We are currently working on new approaches to data extraction from lists [Kristina Lerman and Minton, 2001] that will enable us to use STALKER to learn the correct data extraction rules.

3.2.2 Previous Work on Wrapper Reinduction

There has been relatively little prior work on the wrapper reinduction problem. Cohen [Cohen, 1999] adapted WHIRL, a “soft” logic that incorporates a notion of statistical text similarity, to recognize page structure of a narrow class of pages: those containing simple lists and simple hotlists (defined as anchor-URL pairs). Previously extracted data, combined with page structure recognition heuristics, was used to reconstruct the wrapper once the page structure changed. Cohen conducted wrapper maintenance experiments using original data and corrupted data as examples for WHIRL. However, his procedure for corrupting data was not realistic or representative of how data on the Web changes. Although we cannot at present guarantee good performance of our algorithm on the wrapper reinduction for sources containing lists, we handle the realistic data changes of Web sources containing single tuple of data.

4 Conclusion

In this paper we have described DataProG, a modification of the algorithm we introduced in an earlier work [Lerman and Minton, 2000] that learns structural information about a data field from a set of examples of the field. Unlike the original algorithm, DataProG is biased to learn more general patterns. We use these patterns in Web wrapper maintenance applications: (i) **verification** — detecting when a wrapper stops extracting data correctly from a Web source, and (ii) **reinduction** — identifying new examples of the data field in order to rebuild the wrapper if it stops working. We have validated the pattern learning algorithm in these two application domains. We found that the use of more general patterns significantly reduces the number of false positives in the wrapper verification task. In the reinduction task, the patterns were used to identify a large number of data fields on Web pages, which were in turn used to automatically learn STALKER rules for these Web sources. The new extraction rules were validated by using them to successfully extract data from sets of test pages.

There remains work to be done on wrapper maintenance. Our current algorithms are not sufficient to automatically create STALKER rules for sources that return lists of tuples. However, preliminary results indicate [Kristina Lerman and Minton, 2001] that it is feasible to combine information about the structure of data with *a priori* expectations about the structure of Web pages containing lists to automatically extract data from lists and assign it to rows and columns. This is a first step towards automatic Web wrapper creation. Another exciting direction for future work is using the DataProG algorithm to automatically create wrappers for sources that were not previously seen. For example, we can learn the author, title and price fields for the Amazon source, and use them to extract the same fields on the Barnes&Noble source. Preliminary results show that this is feasible. Yet another direction is to use the DataProG approach to learn characteristic patterns within data fields, as opposed to just the starting and ending patterns. The algorithm is efficient enough to be used this way, and there are many applications where this would be useful.

5 Acknowledgements

We would like to thank Priyanka Pushkarna for carrying out the wrapper verification experiments. 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 and by the Air Force Office of Scientific Research under Grant Number F49620-01-1-0053 and by the National Science Foundation under award number DMI-0090978. 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

- [Angluin, 1982] D. Angluin. Inference of reversible languages. *Journal of the ACM*, 29(3):741–765, 1982.
- [Brazma *et al.*, 1995] A. Brazma, I. Jonassen, I. Eidhammer, and D. Gilbert. Approaches to the automatic discovery of patterns in biosequences. Technical report, Department of Informatics, University of Bergen, 1995.
- [Carrasco and Oncina, 1994a] R. C. Carrasco and J. Oncina. Learning stochastic regular grammars by means of a state merging method. *Lecture Notes in Computer Science*, 862:139–??, 1994.
- [Carrasco and Oncina, 1994b] Rafael C. Carrasco and Jose Oncina. Learning stochastic regular grammars by means of a state merging method. In Rafael C. Carrasco and Jose Oncina, editors, *Proceedings of the Second International Colloquium on Grammatical Inference and Applications (ICGI94)*, volume 862 of *Lecture Notes on Artificial Intelligence*, pages 139–152, Berlin, September 1994. Springer Verlag.
- [Chalupsky *et al.*, 2001] Hans Chalupsky, Yolanda Gil, Craig A. Knoblock, Kristina Lerman, Jean Oh, David V. Pynadath, Thomas A. Russ, and Milind Tambe. Electric elves: Applying agent technology to support human organizations. In *Proceedings of IAAI-2001, Seattle, WA*, 2001.
- [Cohen, 1999] William W. Cohen. Recognizing structure in web pages using similarity queries. In *Proc. of the 16th National Conference on Artificial Intelligence (AAAI-99)*,, pages 59–66, 1999.
- [Dietterich and Michalski, 1981] T. Dietterich and R. Michalski. Inductive learning of structural descriptions. *Artificial Intelligence*, 16:257–294, 1981.
- [Hsu and Dung, 1998] C. Hsu and M. Dung. Generating finite-state transducers for semi-structured data extraction from the web. *Journal of Information Systems*, 23:521–538, 1998.
- [Knoblock *et al.*, 2001a] C. A. Knoblock, S. Minton, J. L. Ambite, N. Ashish, I. Muslea, A. G. Philpot, and S. Tejada. The ariadne approach to web-based information integration. *Intl. Journal on Cooperative Information Systems (IJCIS)*, 10(1/2):145–169, 2001.
- [Knoblock *et al.*, 2001b] Craig A. Knoblock, Kristina Lerman, Steven Minton, and Ion Muslea. Accurately and reliably extracting data from the web: A machine learning approach. *Data Engineering Bulletin*, 2001.
- [Kristina Lerman and Minton, 2001] Craig Knoblock Kristina Lerman and Steven Minton. Automatic data extraction from lists and tables in web sources. In *Proceedings of the workshop on Advances in Text Extraction and Mining (IJCAI-2001)*, Menlo Park, 2001. AAAI Press.
- [Kushmerick *et al.*, 1997] N. Kushmerick, D. S. Weld, and R. B. Doorenbos. Wrapper induction for information extraction. In *Intl. Joint Conference on Artificial Intelligence (IJCAI)*, pages 729–737, 1997.
- [Kushmerick, 1999] N. Kushmerick. Regression testing for wrapper maintenance. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI-1999)*, page ??, 1999.
- [Lerman and Minton, 2000] Kristina Lerman and Steven Minton. Learning the common structure of data. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-2000)*, Menlo Park, July 26–30 2000. AAAI Press.
- [Muggleton, 1991] S. Muggleton. Inductive logic programming. *New Generation Computing*, 8:295–318, 1991.
- [Muslea *et al.*, 1998] I. Muslea, S. Minton, and C. Knoblock. Wrapper induction for semistructured web-based information sources. In *Proceedings of the Conference on Automated Learning and Discovery (CONALD)*., 1998.

- [Muslea *et al.*, 2001a] Ion Muslea, Craig Knoblock, and Steven Minton. Hierarchical wrapper induction for semistructured information sources. *Journal of Autonomous Agents and Multi-Agent Systems*, 4:93–114, 2001.
- [Muslea *et al.*, 2001b] Ion Muslea, Steven Minton, and Craig A. Knoblock. Hierarchical wrapper induction for semistructured information sources. *Autonomous Agents and Multi-Agent Systems*, 4:93–114, 2001.
- [Quinlan, 1990] J. R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5(3):239–266, ? 1990.
- [Quinlan, 1993] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Mateo, CA, 1993.
- [Stolcke and Omohundro, 1994] A. Stolcke and S. Omohundro. Inference of finite-state probabilistic grammars. In *Proc. 2nd Int. Colloquium on Grammar Induction, (ICGI-94)*, pages 106–118, 1994.
- [T. Goan and Etzioni, 1996] N. Benson T. Goan and O. Etzioni. A grammar inference algorithm for the world wide web. In *Proceedings of AAAI Spring Symposium on Machine Learning in Information Access*, 1996.