Automatic Data Extraction from Lists and Tables in Web Sources

Kristina Lerman[1], **Craig Knoblock**[1,2] **and Steven Minton**[2]
1. Information Sciences Institute
Univ. of Southern California
Marina del Rey, CA 90292-6695
2. Fetch Technologies
{lerman,knoblock,minton}@isi.edu

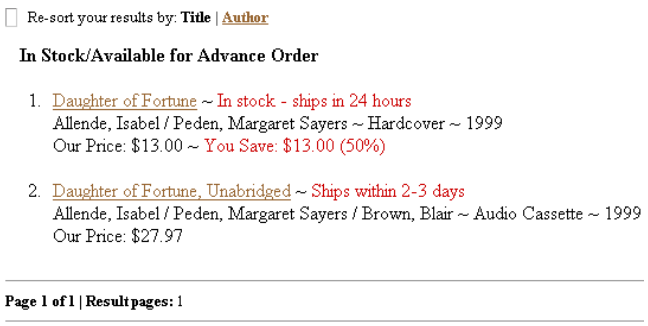
Abstract

We describe a technique for extracting data from lists and tables and grouping it by rows and columns. This is done completely automatically, using only some very general assumptions about the structure of the list. We have developed a suite of unsupervised learning algorithms that induce the structure of lists by exploiting the regularities both in the format of the pages and the data contained in them. Among the tools used are AutoClass for automatic classification of data and grammar induction of regular languages. The approach was tested on 14 Web sources providing diverse data types, and we found that for 10 of these sources we were able to correctly find lists and partition the data into columns and rows.

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, have been advanced to 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 wrapper is a piece of software that turns a Web source into a source that can be queried as if it were a database. The types of sources that this applies to are what are called semistructured sources. The pages that come from such 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 online catalogs, have a very regular structure that can be exploited to extract the data. Extraction rules, which the wrapper uses to identify the beginning and end of the data field to be extracted, form an important part of the wrapper. Quick and efficient generation of extraction rules, so-called wrapper induction, has been an active area of research in recent years [8; 9]. The most advanced of such wrapper induction systems use machine learning techniques to learn extraction rules by example. Using a graphical user interface a user marks up data to be extracted on several pages from an online source, and the system generates a set of extraction rules that accurately extract the required information. The wrapper induction system developed at USC [8; 11] is able to efficiently generate 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.


Re-sort your results by: **Title** | **Author**

In Stock/Available for Advance Order

1. Daughter of Fortune ~ In stock - ships in 24 hours
Allende, Isabel / Peden, Margaret Sayers ~ Hardcover ~ 1999
Our Price: $13.00 ~ You Save: $13.00 (50%)

2. Daughter of Fortune, Unabridged ~ Ships within 2-3 days
Allende, Isabel / Peden, Margaret Sayers / Brown, Blair ~ Audio Cassette ~ 1999
Our Price: $27.97

Page 1 of 1 | Result pages: 1

Figure 1: A fragment of a Web page containing a list. The page was downloaded from the Borders Books web site.

Wrapper induction for sources containing lists and tables presents a special challenge from the user interface point of view. Consider a Web source, e.g., Borders Books (Figure 1), which provides information about books for sale. The page in the example contains two listings, each consisting of the title, availability, author, format, year of publication, and price of the book. The first listing also contains information about the discount. In order to learn accurate extraction rules for data from this source, the wrapper induction system requires that the user label first, last and at least two consecutive listings. Since this has to be done for several pages, the labeling task quickly becomes tedious and time intensive. However, we have certain expectations about the structure of lists and ta-

bles that we can exploit for automatic extraction of data from such sources. For example, each column (e.g., author, availability) of a table usually contains the same type of data, and each row corresponds to a tuple — e.g., (title, availability, author, format, year, price, savings) tuple describing the book — that is repeated in different rows. Moreover, each list from the Borders book site starts after the heading "In Stock/Available for Advance Order" and ends before the string "Page 1 of x—Result pages: x."

In this paper we describe a technique for extracting data from lists and tables and grouping it by rows and columns. This is done completely automatically, using only some very general assumptions about the structure of a list. Lists and tables are alternate ways to present multiple sets of data, and we won't make a distinction between the two. Data on the page is not always laid out in a grid-like pattern; however, it is almost always arranged with a certain amount of regularity we can exploit. For example, in the Borders page containing book listings (see Fig. 1), the title of the book comes first, followed by author, availability, price, followed by the next listing, one listing or tuple per row. Moreover, the tuple elements are arranged in the same order for each listing, so if the author follows the title in one row, it usually follows the title in all other rows. Likewise, we never expect the author in a book listing to appear after the title of the next listing. While these principles may not apply universally, we have found them to be valid for most online sources we studied, including airport listings, online catalogs, hotel directories, *etc.* We tested our approach on 14 Web sources providing diverse types of data and found that for 10 of these sources we were able to correctly find lists and partition the data on the lists into columns and rows.

2 The Approach and Challenges

We have developed a suite of unsupervised learning algorithms that induce the structure of lists by exploiting the regularities both in the format of the pages and the data contained in them. The list below describes the approach at a high level, along with the learning techniques used at each step.

- Extract all data from lists
  - Compute the page template and identify the list on each page
  - Compute a set of features (separators and content) for each data extract
- Identify columns
  Classification of data
- Identify rows
  Grammar-induction on a sequence of class labels

We begin by tokenizing Web pages, that is, splitting the text of the Web pages into individual words or tokens. We analyze pages to find common structure. Many types of Web sources, especially those that return lists, generate the page from a template and fill it with results of a database query. For example, Borders source in the figure above puts book listings after the header "In Stock/Available for Advance Order." By comparing several pages, we are able to deduce the template used to generate them and identify the section of the template that contains the list. Next, we extract all data from the list. If the HTML table has been carefully formatted, this step would amount to extracting all visible text. However, in addition to HTML tags, punctuation characters, such as the tilde in the Borders example, are often used to separate data fields (columns); therefore, we define a column/row separator as a set of sequential HTML tags or any punctuation character excluding the set "(-)'.%" (the choice of the excluded set is discussed in Section 3.1). In the end, the extracted data are all sequences of tokens that occur between separators. We refer to these sequences of tokens as extracts.

As we mentioned above, we expect all data in the same column to be of the same type, book price for instance, and its content have the same or similar format. In addition to content, layout features, such as separators, may be useful hints for arranging extracts by columns. However, we cannot rely solely on separators — the table may have missing columns, separators before the first row and after the last one may be different from those separating rows within the list, *etc.* Likewise, we cannot rely solely on content — data has a lot of variability, and our representation scheme, like many others, may not be capable of fully capturing distinctions between data. Rather than using each type of evidence separately, we combine them by describing each extract by a set of features that include the separators as well as features that capture the content of data. We use AutoClass [6; 4] to cluster extracts. AutoClass is an unsupervised classification algorithm that finds the optimal number of classes and the best assignment of extracts to classes. In the resulting assignment, each data type ends up in a separate class, or column.

The final step of the analysis is to partition the list into rows. Ideally, it should be easy to identify rows from the class assignment, because a row corresponds to a tuple of data, which is repeated for every row. However, real lists and tables have missing columns, and AutoClass assignment may include errors; therefore, identifying the repeated row pattern is a non-trivial task. We use a grammar induction algorithm for this task. Each list, or rather the sequence of AutoClass-assigned column labels for the extracts in the list, can be thought of as a string generated by a regular language, which we try to learn from the examples of the language. The language captures the repeated structure in the sequences that corresponds to rows. We use this information to partition the list into tuples.

The end result of the application of the suite of algorithms is a complete assignment of data in the list to rows and columns. It is possible to do a meta-analysis of the assignment and fix any errors made along the way, but we have not done so at this point.

3 Algorithms for Automatic Data Extraction

In this section we present details of the algorithms for automatic data extraction. The input is a set of unlabeled Web pages containing lists.

3.1 Finding the page template

During the tokenization step, the text of each Web page is split into individual words, or more accurately tokens, and

each token is assigned one or more syntactic types [10], based on the characters appearing in it. Thus, a token can be an HTML token, an alphanumeric, or a punctuation token. If it's an alphanumeric token, it may also belong to one of two categories: numeric or alphabetic, which is further divided into capitalized or lowercased types, and so on. The syntactic token hierarchy is described in [10].

Many Web sources use templates to automatically generate pages and fill them with results of a database query. Given two or more example pages from the same source, we can induce the template used to generate them. Our 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 to build a sequence that's at least k tokens long,[1] and this sequence appears exactly once on each page, it becomes part of the page template. Figure 2 contains the details of the template finding algorithm.

If any of the lists contains more than two rows, the tags specifying the structure of the list will not be part of the page template, because they will appear more than once on that page. We can use this to our advantage. We look for sections of the page where these sequences of tokens appear more than once, because that's where we expect the list to be. The template finding algorithm has the following pitfall: it can happen that every list starts with exactly the same data; therefore, the beginning of the list will be improperly included in the page template. We can minimize this problem by including diverse pages in the set.

Once we identify the section of the page that contains the list, we extract all data from it. If the HTML table was carefully formatted, this step would amount to extracting all visible text. However, in addition to HTML tags, punctuation characters are often used to separate data fields (columns); therefore, we define a column/row separator as a set of sequential HTML tags or any punctuation character excluding the set ".(-)'%". The excluded set was chosen empirically. Sometimes a dash (-) is a good separator, but for many frequently encountered data types, such as phone numbers and zip codes, dash is part of data and not a separator. Likewise, comma (,) is sometimes a separator (*e.g.*, "123 Main St., Pasadena") and sometimes part of data (*e.g.*, in "1,000,000"), though we generally chose to treat it as a separator. In principle, there should be a less *ad hoc* method for choosing separators, which will be the subject of future research. In the end, we extract every sequence of text tokens between separators.

3.2 Identifying columns

We expect all data in the same column to be of the same type, e.g., book prices; therefore, we may be able to identify columns by grouping extracts by similarity of content. In addition to content, layout hints and separators, may be useful evidence for helping arrange extracts by column. However,

[1] In our experiments, we have found that $k = 3$ worked best as a minimum length for the page template element.

```
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) ≥ 3 )
            add(s, T)
          end if
          s = null
        end if
    end for
end
```

Figure 2: *Pseudocode of the template finding algorithm*

we cannot rely on either type of evidence by itself. Most methods for representing the content of data, including our own, would be hard pressed to distinguish restaurant names from cities. Likewise, examples of the same data type may contain lots of variability and may be erroneously separated into different columns. While we cannot, for the above reasons, rely on content information when making column assignment, neither can we rely solely on separators. If the table has missing columns, they will affect the separators surrounding visible data. In addition, separators before the first row of the list and after the last row may be different from the ones around the rows within the list. Rather than using either of the two types of evidence alone, we decided to combine them by describing each extract by a set of features that include both the separators and those that capture the content of data.

Each unique separator is assigned an integer. Every extract is described by a set of features, two of which are integers, one for the separator that precedes the extract, and one for the separator that immediately follows it. The content of data is captured by the data prototype, or patterns of specific tokens and syntactic token types that describe the common beginnings and ends of a set of examples of data [10]. For example, a set of street addresses may be well described by the starting pattern "NUMBER CAPS" and the ending pattern "Street", meaning that a significant fraction of addresses start with a number followed by a capitalized word and end in the word "Street." The algorithm (DataPro) that learns the patterns that describe data from positive examples of the field is presented in reference [10].

Our first approach to computing content features was to use all extracts as examples for DataPro. However, the algorithm tended to overgeneralize by producing patterns that describe more than one column, e.g., addresses and zip codes,

```
input:
    X = set of data extracts from the list
output:
    V = set of vectors describing extracts
begin
    for each x in X
        (V_x)_1 = leftseparator(x)
        (V_x)_2 = rightseparator(x)
    end for
    C = cluster(V)
    P = null
    for each c in C
        addpatterns(patterns(c), P)
    end for

    for each x in X
        for i=firstpattern(P) to lastpattern(P)
            if ( matches(x, patterns(i, P) )
                (V_x)_{i+2} = 1
            else
                (V_x)_{i+2} = 0
            end if
        end for
    end for
end
```

Figure 3: *Pseudocode of the algorithm*

which affected the subsequent performance of the classification algorithm. Instead, we adopted a two-step approach, as illustrated in Figure 3. First, we group the extracts by separators, so that the extracts that share at least one separator are in the same cluster. This already does a decent job of separating some of the extracts into columns, though many columns are split among different clusters, and not every extract ends up in a cluster. Extracts within a cluster belong to the same column, and we use the DataPro algorithm to learn the patterns that describe each cluster. Next, we evaluate every extract to see whether it is similar to any cluster. If any of the patterns associated with the nth cluster describe the extract, the value of the nth content feature is one; otherwise, it is zero. Thus, there are as many binary content features as there are clusters.

The two types of evidence are expressed in different units; therefore, standard clustering algorithms that use geometrically based similarity measure (*e.g.*, K-means) would not be appropriate for this purpose. We use AutoClass [6; 4] instead to cluster the extracts. This tool gives us the flexibility to combine different kinds of evidence: class instances may be described by continuous, discrete, and binary values at the same time. AutoClass is a mixture model-based unsupervised classification algorithm that finds both the optimal number of classes and the best (MAP-based) assignment of extracts to classes. It has been used for automatic discovery of classes in diverse data — from DNA to astronomical data sets [4]. In the resulting assignment, each data type ends up in a separate class, or column. Because it starts from a random initial assignment which serves as a starting point for the search for both the optimal number of classes and the model that explains the distribution of instance values in each class, AutoClass does not always converge on the same model of data. Thus, it is necessary to run AutoClass several times from different initial random assignments, and choose the outcome that corresponds to the greatest classification likelihood.

3.3 Identifying rows

Finally, we want to find associations between the columns of data by assigning data to tuples. If we anticipate using the Web source in the future, it is more efficient to build a wrapper for it, rather than analyze pages each time information from the source is required. In order to build a wrapper, we need to label the first, last and several consecutive elements of the list. The easiest way to guarantee that the required elements are labeled is to label every element of the list. It is for this reason that we must break the list into rows.

Ideally, it should be easy to identify rows from the column assignment, because each row corresponds to a tuple of data types, and the tuple is expected to be repeated in every row. However, real lists and tables have missing columns, additionally, AutoClass assignment may include errors; therefore, identifying rows is a non-trivial task. We use a grammar induction algorithm to find the repeated cycles of column assignments that correspond to rows. If the extracts are arranged sequentially as they appear in the list, the sequence of their AutoClass-assigned column labels forms a string in a language generated by a regular grammar. Our objective is to learn this grammar from the example strings and to use it to recognize the rows of the list.

Grammar induction, especially the identification of regular languages, has received a great amount of attention in the past three decades [7; 1; 5]. Carrasco and Oncina proposed an algorithm ALERGIA [2; 3] to learn grammars of stochastic regular languages using a state-merging method. The advantages of ALERGIA is that it learns from a set of positive examples of the languages alone; moreover, its performance is polynomial, and indeed has been shown to be linear [2], in the size of the example set. However, when there are few examples, ALERGIA tends to produce overly complicated grammars, because statistical significance judgments it uses to learn the grammar are less reliable for small data sets. We adapt a simplified version of ALERGIA to learn the regular grammars associated with lists (Figure 4). Like ALERGIA, we start by constructing a prefix-tree acceptor from the example strings and proceed by merging pairs of equivalent nodes until we arrive at the minimum finite state automaton (FSA) consistent with the language. We examine nodes in the same lexicographic order as ALERGIA; however, unlike ALERGIA, we merge two nodes, i and j, if their incoming arcs, $\delta_{k,i}(a)$ and $\delta_{l,j}(a)$, correspond to the same symbol a and at least one of the outgoing arcs, $\delta_{i,k}(b)$ and $\delta_{j,l}(b)$, from each node correspond to the same symbol. We add an another level of generalization by merging two nodes if they have an incoming arc with the same symbol, and one node is a parent of the other, thereby creating a loop. After a pair of states is merged, we determinize the FSA by making sure all descendents have at most one outgoing transition corresponding to a given symbol. The motivation for the state merging approach is the following: if a column B follows column A and pre-

cedes column C in one row, it is likely to follow the same pattern in other rows. Therefore, observing a sequence ABC more than once constitutes evidence that it forms a pattern for a row.

Finally, we extract all cycles from the merged FSA, where each cycle corresponds to a sequence of columns that could constitute a row. This step consists of finding all closed paths through the FSA, *i.e.*, paths that start and end at the same node. Loops, or states that accept a symbol, let's say A, but stay in the same state are represented as A* in the cycle. As in a regular language, this expression means that the symbol A may be repeated any number of times in the language.

To partition the list into rows, we begin at the first symbol of the sequence and find the longest cycle that matches the sequence starting with that symbol. After we find a match, we move to the first unmatched symbol and repeat the procedure for the remaining part of the sequence. Below are the tuples automatically extracted from the list shown in Figure 1 after applying the algorithm to several pages from the Borders book site.

```
(b)Daughter of Fortune
(a)In stock - ships in 24 hours
(c)Allende , Isabel
(c)Peden , Margaret Sayers
(f)Hardcover
(g)1999
(d)Our Price
(e)13 .  00
(h)You Save
(i)13 .  00 ( 50 % )

(b)Daughter of Fortune , Unabridged
(a)In stock - ships in 24 hours
(c)Allende , Isabel
(c)Peden , Margaret Sayers
(c)Brown , Blair
(f)Audio Cassette
(g)1999
(d)Our Price
(e)27 .  97
```

4 Results

We have validated our approach by applying it to extract data from 14 Web information sources containing a wide variety of data types. We randomly selected three or four pages from each source, with the only requirement being that the pages contain a list with at least two elements. We applied the extraction algorithm to each set of pages and manually checked whether the data from the list was partitioned correctly into tuples. The table in Fig. 5 summarizes the results. The approach worked for 10 of the 14 sources, though for one source, Yahoo stock quotes, the algorithm made a mistake with two of the 20 tuples. Conceivably, a meta-analysis of the final tuple assignments would be able to catch and correct any errors contained in the preceding steps.

Our approach failed in four cases for the following reasons. In one case (MapQuest) all three lists began with the same data; therefore, the page template finding algorithm did

```
input:
    S: set of example strings
output:
    minimum FSA
begin
    A = prefix tree acceptor from S
    for j = successor(firstnode(A)) to lastnode(A)
        for i = firstnode(A) to j
            if compatible(i, j)
              merge(A, i, j)
              determinize(A)
              exit(i-loop)
            end if
        end for
    end for
    return A
end main

COMPATIBLE(i, j)
input:
    i, j nodes
output:
    boolean
begin compatible
    δ_{k,i} = arc from node k to node i
    δ_{i,m} = arc from node i to node m
    if symbol(δ_{k,i}) = symbol(δ_{l,j}) for some k and l
      if symbol(δ_{i,m}) = symbol(δ_{j,n}) for some m and n
        return true
      end if
      if i is parent of j
        return true
      end if
    end if
end compatible

DETERMINIZE(A)
input:
    FSA A
begin determinize
    for i = firstnode(A) to lastnode(A)
        for j = firstsuccessor(i) to lastsuccessor(i)
            for ( k = nextsuccessor(j) to lastsuccessor(i)
                if symbol(δ_{i,j}) = symbol(δ_{i,k})
                  merge(j, k)
                  determinize(A)
                end if
            end for
        end for
    end for
end determinize
```

Figure 4: *Pseudocode of the grammar induction algorithm*

source	pages	extracts	columns	classes	result
airport	4	370	4	5	correct tuples
airport code, location					
Blockbuster	4	663		11	no tuples extracted
movies					
Borders	4	186	9	9	correct tuples
books					
Cuisinenet	3	535	17	15	no tuples extracted
restaurants					
RestaurantRow	4	273	14	14	correct tuples
Yahoo people	3	126	8	8	correct tuples
whitepages					
Yahoo quote	3	259	13	13	18/20 tuples correct
stocks					
Whitepages	3	73	9	5	correct tuples
MapQuest	3	83	5	5	tuples begin in the
driving directions					middle of the rows
hotel	4	163	6	6	correct tuples
CitySearch	4	204	4	6	correct tuples
restaurants					
car rental	4	161	8+	9	correct tuples
boston	4	174	4+	6	correct tuples
restaurants					
Arrow	3	366		10	no tuples extracted
electronic components					

Figure 5: Results of applying the automatic data extraction algorithm to different Web sources.

not locate the correct start of list. In the three other cases, the approach failed because the structure of the list and the resulting FSA for these sources was too complex to extract cycles. Our approach to breaking the list into rows by using grammar induction is clearly not sufficient, and another approach or a modification of the grammar induction algorithm is warranted.

5 Discussion

We have demonstrated that it is possible to accurately extract data from semistructured Web pages containing lists and tables by exploiting the regularities both in the format of the pages and the data contained in them. We have presented a suite of algorithms that extract data from lists and tables and automatically assign it to rows and columns. First, we assign extracts to columns using AutoClass, an unsupervised classification algorithm, then a grammar induction algorithm finds repeated patterns in the column assignments that correspond to rows. The grammar induction may also correct some of the mistakes made by the classification algorithm. It is possible to further analyze the assignment of extracts to tuples to correct mistakes not caught in the preceding steps; however, we have not done this step. We have been able to extract and label data from lists with high accuracy for 10 out of the 14 sources to which we have applied our algorithm. Therefore, we can easily create wrappers for these sources.

One limitation of our approach is that it requires several pages to be analyzed before data can be extracted from a single list. Often, we may have just a single page from a source. Conceivably, there is enough structure in a single list for us to exploit for the purposes of extraction. We are currently considering the extensions of our algorithm that will enable us to extract data from a single list.

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 and by the Air Force Office of Scientific Research under Grant Number F49620-01-1-0053.

References

[1] D. Angluin. Identifying languages from stochastic examples. Technical Report YALEU/ DCS/RR-614, Yale University, Dept. of Computer Science, New Haven, CT, 1988.

[2] 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 Colloqium on Grammatical Inference and Applications (ICGI94)*, volume 862 of *Lecture Notes on Artificial Intelligence*, pages 139–152, Berlin, September 1994. Springer Verlag.

[3] Rafael C. Carrasco and Jose Oncina. Learning deterministic regular grammars from stochastic samples in polynomial time. *RAIRO (Theoretical Informatics and Applications)*, 33(1):1–20, 1999.

[4] P. Cheeseman and J. Stutz. Bayesian classification (AUTOCLASS): Theory and results. *Advances in Knowledge Discovery and Data Mining*, 1996.

[5] E. A. Gold. Complexity of automaton identification from given data. *Information and Control*, 37:302–320, 1978.

[6] J. Hanson,R., Stutz and P. Cheeseman. Bayesian classification theory. Technical report, NASA Ames TR FIA-90-12-7-01, 1991.

[7] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, N. Reading, MA, 1980.

[8] 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.

[9] 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.

[10] Kristina Lerman and Steven Minton. Learning the common structure of data. In *Proceedings of the 15=7th National Conference on Artificial Intelligence (AAAI-2000) and of the 10th Conference on Innovative Applications of Artificial Intelligence (IAAI-98)*, Menlo Park, July 26–30 2000. AAAI Press.

[11] 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.