A Hierarchical Approach to Wrapper Induction

Ion Muslea, Steve Minton, and Craig Knoblock
University of Southern California
4676 Admiralty Way
Marina del Rey, CA 90292-6695
{muslea, minton, knoblock}@isi.edu

Abstract

With the tremendous amount of information that becomes available on the Web on a daily basis, the ability to quickly develop information agents has become a crucial problem. A vital component of any Web-based information agent is a set of wrappers that can extract the relevant data from semistructured information sources. Our novel approach to wrapper induction is based on the idea of hierarchical information extraction, which turns the hard problem of extracting data from an arbitrarily complex document into a series of easier extraction tasks. We introduce an inductive algorithm, STALKER, that generates high accuracy extraction rules based on user-labeled training examples. Labeling the training data represents the major bottleneck in using wrapper induction techniques, and our experimental results show that STALKER does significantly better then other approaches; on one hand, STALKER requires up to two orders of magnitude fewer examples than other algorithms, while on the other hand it can handle information sources that could not be wrapped by existing techniques.

1 Introduction

With the expansion of the Web, computer users have gained access to a large variety of comprehensive information repositories. However, the Web is based on a browsing paradigm that makes it difficult to retrieve and integrate data from multiple sources. The most recent generation of *information agents* (e.g., WHIRL [7], Ariadne [11], and Information Manifold [10]) address this problem by enabling information from pre-specified sets of Web sites to be accessed via database-like queries. For instance, consider the query "What seafood restaurants in L.A. have prices below $20 and accept the Visa credit-card?". Assume that we have two information sources that provide information about LA restaurants: the Zagat Guide and LA Weekly (see Figure 1). To answer this query, an agent could use Zagat's to identify **seafood** restaurants **under $20** and then use LA Weekly to check which of these accept **Visa**.

Information agents generally rely on *wrappers* to extract information from *semistructured* Web pages (a page is semistructured if the desired information can be located using a concise, formal grammar). Each wrapper consists of a set of extraction rules and the code required to apply those rules. Some systems, such as TSIMMIS [5] and ARANEUS [3] depend on humans to write the necessary grammar rules. However, there are several reasons why this is undesirable. Writing extraction rules is tedious, time consuming and requires a high level of expertise. These difficulties are multiplied when an application domain involves a large number of existing sources or the format of the source documents changes over time.

In this paper, we introduce a new machine learning method for wrapper construction that enables unsophisticated users to painlessly turn Web pages into relational information sources. The next section presents a formalism describing semistructured Web documents, and then Sections 3 and 4 present a domain-independent information extractor that we use as a skeleton for all our wrappers. Section 5 describes STALKER, a supervised learning algorithm for inducing extraction rules, and Section 6 presents a detailed example. The final sections describe our experimental results, related work and conclusions.

2 Describing the Content of a Page

Because Web pages are intended to be human readable, there are some common conventions for structuring HTML pages. For instance, the information on a page often exhibits some hierarchical structure; furthermore, semistructured information is often presented in the form of lists of tuples, with explicit separators used to distinguish the different elements. With these observations in mind, we developed the *embedded catalog* ($\mathcal{EC}$) formalism, which can describe the structure of a wide-range of semistructured documents.

The $\mathcal{EC}$ description of a page is a tree-like structure in which the leaves are the items of interest for the user (i.e., they represent the relevant data). The internal nodes of the $\mathcal{EC}$ tree represent *lists* of k-tuples (e.g., lists of restaurant descriptions), where each item in the k-tuple can be either a leaf l or another list L (in which case L is called an embedded list). For instance, Figure 2 displays the $\mathcal{EC}$ descriptions of the LA-Weekly and Zagat pages. At the top level, an LA-Weekly page is a *list* of 5-tuples that contain the `name`, `address`, `phone`, `review`, and an *embedded list* of `credit cards`. Similarly, a Zagat document can be seen as a 7-tuple that includes a list of addresses, where each

Figure 1: **LA-Weekly and Zagat's Restaurant Descriptions**

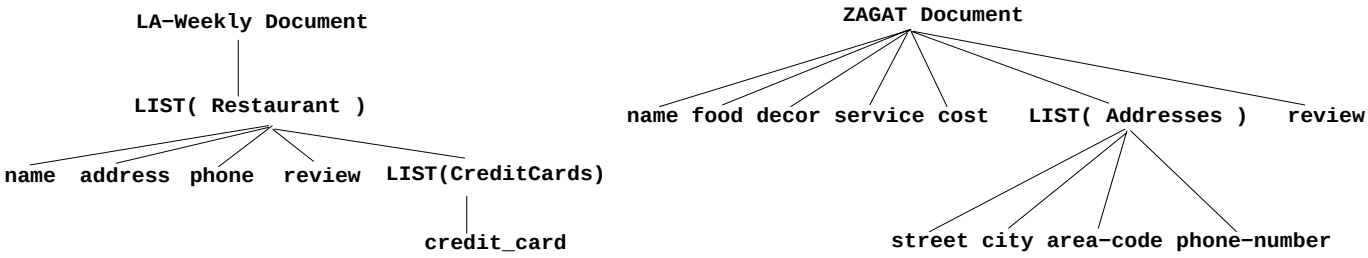


Figure 2: $\mathcal{EC}$ **description of LA-Weekly and ZAGAT pages.**

individual address is a 4-tuple `street, city, area-code,` and `phone-number`.

3 Extracting Data from a Document

Given the $\mathcal{EC}$ description of a document together with an *extraction rule* attached to each edge and a *list iteration rule* associated with each list node, a wrapper can extract any item of interest (i.e., any leaf) by simply determining the path P from the root to the corresponding leaf and by successively extracting each node $x \in P$ from its parent p. In order to extract x from p, the wrapper applies the extraction rule r that is attached to the $edge(p, x)$; if p is a list node, the wrapper has to apply first the iteration rule that decomposes p into individual tuples, and then it applies r to each extracted tuple.

In our framework a document is a sequence of tokens ST (e.g., words, numbers, HTML tags, etc). It follows that the content of the root node in the $\mathcal{EC}$ tree is the whole sequence ST, while the content of each of its children is a subsequence of ST. More generally, the content of an arbitrary node x represents a subsequence of the content of its parent p. A key idea underlying our work is that the extraction rules can be based on "landmarks" (i.e., groups of consecutive tokens) that enable a wrapper agent to locate the content of x within the content of p.

For instance, let us consider the restaurant descriptions presented in Figure 3. In order to identify the beginning of the restaurant name, we can use the rule

$$\mathbf{R1} = SkipTo(\texttt{})$$

which has the following meaning: start from the beginning of the document and skip everything until you find the `<b>` landmark. More formally, the rule **R1** is applied to the content of the node's parent, which in this particular case is the whole document; the effect of applying **R1** consists of consuming the *prefix of the parent*, which ends at the beginning of the restaurant name. Similarly, one can identify the end of a node's content by applying a rule that consumes the corresponding suffix of the parent. For instance, in order to find the end of the restaurant name, one can apply the rule

$$\mathbf{R2} = SkipTo(\texttt{})$$

from the end of the document towards its beginning.

The rules **R1** and **R2** are called *start* and *end rules*, and, in most of the cases, they are not unique. For instance, we can replace **R1** either by

$$\mathbf{R3} = SkipTo(\texttt{Name})\ SkipTo(\texttt{<b>})$$

or by

$$\mathbf{R4} = SkipTo(\ \texttt{Name}\ Symbol\ HtmlTag\ )$$

R3 has the meaning "ignore everything until you find a `Name` landmark, and then, again, ignore everything until you find `<b>`", while **R4** is interpreted as "ignore all tokens until you find a 3-token landmark that consists of the token `Name`, immediately followed by a punctuation sign and an HTML tag". As the rules above successfully identify the start of the restaurant name, we say that they *match correctly*. By contrast, the start rules $SkipTo(\texttt{:})$ and $SkipTo(\texttt{<i>})$ are said to

```
1: <p> Name: <b> Yala </b><p> Cuisine: Thai <p><i>
2: 4000 Colfax, Phoenix, AZ 85258 (602) 508-1570
3: </i> <br> <i>
4: 523 Vernon, Las Vegas, NV 89104 (702) 578-2293
5: </i> <br> <i>
6: 403 Pico, LA, CA 90007 (213) 798-0008
7: </i>
```

Figure 3: **A simplified version of a Zagat document.**

match incorrectly because they consume too few or too many tokens, respectively. Finally, a rule like $SkipTo(\texttt{<table>})$ *fails* because the landmark `<table>` does not exist in the document.

In order to deal with variations in the format of the documents, our extraction rules allow the use of disjunctions. For example, if the names of the recommended restaurants appear in bold, while the other ones are displayed as italic, one can extract all the names based on the disjunctive start and end rules

either $SkipTo(\texttt{})$ **or** $SkipTo(\texttt{<i>})$ and

either $SkipTo(\texttt{})$ **or** $SkipTo(\texttt{Cuisine})SkipTo(\texttt{</i>})$

A disjunctive rule matches if at least one of its disjuncts matches. In case several disjuncts match, we nondeterministically choose one of the matching disjuncts.

To illustrate how the extraction process works for list members, lets consider the case where the wrapper agent has to extract all the area codes from a sample document. In this case, the agent starts by extracting the entire list of addresses, which can be done based on the start rule $SkipTo(\texttt{<p><i>})$ and the end rule $SkipTo(\texttt{</i>})$. Then the wrapper has to iterate through the content of `LIST(Addresses)` (lines 2-6 in Figure 3) and to break it into individual tuples. In order to find the start of each individual address, the agent repeatedly applies $SkipTo(\texttt{<i>})$ to the content of the list (each successive rule-matching starts at the point where the previous one ended). Similarly, the agent determines the end of each tuple by repeatedly applying the end rule $SkipTo(\texttt{</i>})$. In our example, the list iteration process leads to the creation of three individual addresses that have the contents showed on the lines 2, 4, and 6, respectively. Then the wrapper applies to each address the `area-code` start and end rule (e.g., $SkipTo($ '(' $)$ and $SkipTo($ ')' $)$, respectively).

Now let us assume that instead of the area codes, the wrapper has to extract the ZIP codes. The list extraction and the list iteration remain unchanged, but the ZIP code extraction is more difficult because there is no landmark that separates the state from the ZIP code. Even though in such situations the $SkipTo()$ rules are not sufficiently expressive, they can be easily extended to a more powerful extraction language. For instance, we can use either

$$\mathbf{R5} = SkipTo(,)\ SkipUntil(Number)$$

or

$$\mathbf{R6} = SkipTo(AllCaps)\ NextLandmark(Number)$$

to extract the ZIP code from the entire address. The arguments of $SkipUntil()$ and $NextLandmark()$ describe *a prefix of the content of the item to be extracted*, and they are not consumed when the rule is applied (i.e., the rules stops *immediately before* their occurrence). The rule **R5** means "ignore all tokens until you find the landmark ',', and then ignore everything until you find, but do not consume, a number". Similarly, **R6** is interpreted as "ignore all tokens until you encounter an *AllCaps* word, and make sure that the next landmark is a number". In other words, the only difference between $SkipTo(l_1)\ SkipUntil(l_2)$ and $SkipTo(l_1)\ NextLandmark(l_2)$ consists of the way l_2 is matched: the former allows any number of tokens between l_1 and l_2, while the later requires that the landmark l_2 immediately follows l_1. Rules like **R5** and **R6** can be extremely useful in practice, and they represent only variations of the $SkipTo()$ rules in which the last landmark has a special meaning. In order to keep the presentation simple, the rest of the paper focuses mainly on $SkipTo()$ rules. When necessary, we will explain the way in which our approach can be extended to also handle $SkipUntil()$ and $NextLandmark()$.

The extraction rules presented in this section have two main advantages. First of all, the *hierarchical* extraction based on the $\mathcal{EC}$ tree allows our agent to wrap information sources that have arbitrary many levels of embedded data. Second, as each node is extracted independently of its siblings, our approach does not rely on there being a fixed ordering of the items, and we can easily handle extraction tasks from documents that may have missing items or items that appear in various orders. Consequently, in the context of using an inductive algorithm that generates the extraction rules, our approach turns an extremely hard problem into several simpler ones: rather then finding a single extraction rule that takes into account all possible item orderings and becomes more complex as the depth of the $\mathcal{EC}$ tree increases, we create several simpler rules that deal with the easier task of extracting each item from its $\mathcal{EC}$ tree parent.

4 Extraction Rules as Finite Automata

We now introduce two key concepts that can be used to define extraction rules: *landmarks* and *landmark automata*. In the rules described in the previous section, each argument of a $SkipTo()$ function is a *landmark*, while a group of $SkipTo()$s that must be applied in a pre-established order represents a landmark automaton. In other words, any extraction rule from the previous section is a landmark automaton.

In this paper, we focus on a particular type of landmark: the *linear landmark*. A linear landmark is described by a sequence of tokens and wildcards (a wildcard represents a class of tokens, as illustrated in the previous section, where we used the wildcards $Number$, $Sign$, and $HtmlTag$). Linear landmarks are interesting for two reasons: on one hand, they are sufficiently expressive to allow efficient navigation within the $\mathcal{EC}$ structure of the documents, and, on the other hand, as we will see in the next section, there is a simple way to generate and refine linear landmarks during learning.

Landmark automata ($\mathcal{LA}$s) are nondeterministic finite automata in which each transition $S_i \rightsquigarrow S_j$ $(i \neq j)$ is labeled by a landmark $l_{i,j}$; that is, the transition

$$S_i \overset{l_{i,j}}{\rightsquigarrow} S_j$$

takes place if and only if in the state S_i the input is a string s that is accepted by the landmark $l_{i,j}$. *Simple Landmark Grammars* ($\mathcal{SLG}$s) are the class of $\mathcal{LA}$s that correspond to the disjunctive rules introduced in the previous section. Any $\mathcal{SLG}$ has the following properties:

- the initial state S_0 has a branching-factor of k;

E1: 513 Pico, Venice, Phone: 1-800-555-1515

E2: 90 Colfax, Palms , Phone: (818) 508-1570

E3: 523 1st St., LA , Phone: 1-888-578-2293

E4: 403 Vernon, Watts , Phone: (310) 798-0008

Figure 4: **Four examples of restaurant addresses.**

- it has exactly k accepting states (one per branch);
- all k branches that leave the S_0 are *sequential* $\mathcal{LA}$s (i.e., from each non-accepting state S_i, there are exactly two possible transitions: a loop to itself, and a transition to the next state;
- *linear landmarks* label each non-looping transition;
- all looping transitions have the meaning "consume all tokens until you encounter the linear landmark that leads to the next state".

In the next section we present the STALKER inductive algorithm that generates $\mathcal{SLG}$ rules that identify the start and end of an item x within its parent p. Note that finding a *start rule* that consumes the prefix of p with respect to x (for short $Prefix_x(p)$) is similar to finding an *end rule* that consumes the suffix of p with respect to x (i.e., $Suffix_x(p)$); in fact, the only difference between the two types of rules consists of *how* they are actually applied: the former starts by consuming the first token in p and goes towards the last one, while the later starts at the last token in p and goes towards the first one. Consequently, without any loss of generality, in the rest of this paper we discuss only the way in which STALKER generates $\mathcal{SLG}$ start rules.

5 Learning Extraction Rules

The input to STALKER is a set of sequences of tokens that represent the prefixes that must be consumed by the induced rule. In order to create such training examples, the user has to select a few sample pages and to use a graphical user interface (GUI) to mark up the relevant data (i.e., the leaves of the $\mathcal{EC}$ tree); once a page is marked up, the GUI generates the sequences of tokens that represent the content of the parent p, together with the index of the token that represents the start of x and uniquely identifies the prefix to be consumed by the induced $\mathcal{SLG}$.

For instance, let us assume that the user marked the four area codes from Figure 4 and invokes STALKER on the corresponding four training examples (that is, the prefixes of the addresses $E1$, $E2$, $E3$, and $E4$ that end immediately before the area code). STALKER, which is a sequential covering algorithm, begins by generating a *linear* $\mathcal{LA}$ that covers as many as possible of the four positive examples. Then it tries to create another *linear* $\mathcal{LA}$ for the remaining examples, and so on. Once STALKER covers all examples, it returns the disjunction of all the induced $\mathcal{LA}$s. In our example, the algorithm generates first the rule **R1**$::=$ *SkipTo*(()), which has two important properties:

- it accepts the positive examples in $E2$ and $E4$;
- it rejects both $E1$ and $E3$ because **R1** can not be matched on them.

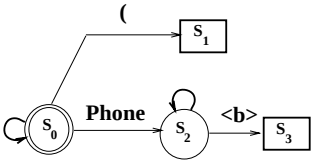


Figure 5: **An $\mathcal{SLG}$ for the start of the area code.**

During a second iteration, the algorithm considers only the uncovered examples $E1$ and $E3$, and it generates the rule **R2**$::=$ *SkipTo* (Phone) *SkipTo*(). In Figure 5, we show the $\mathcal{SLG}$ that is equivalent to the *disjunctive rule* **either R1 or R2**.

In order to generate a rule R that extracts x from its parent p, STALKER (see Figure 6) has to induce the number of disjuncts in R together with the number and the actual values of the landmarks that form each disjunct. STALKER's input consists of pairs (T_i, Idx_i), where each sequence of tokens T_i is the content of an instance of p, and $T_i[Idx_i]$ is the token that represents the start of x within p. Any sequence $S ::= T_i[1], T_i[2], \ldots, T_i[Idx_i - 1]$ (i.e., any instance of $Prefix_x(p)$) represents a positive example, while any other sub-sequence or super-sequence of S represents a negative example. STALKER tries to generate an $\mathcal{SLG}$ that accepts all positive examples and rejects all negative ones.

STALKER is a typical *sequential covering* algorithm: as long as there are some uncovered positive examples, it tries to learn a *perfect disjunct* (i.e., a sequential $\mathcal{LA}$ that accepts only true positives). When all the positive examples are covered, STALKER returns the solution, which consists of an $\mathcal{SLG}$ in which each branch corresponds to a learned disjunct.

The function $LearnDisjunct()$ is a *brute force* algorithm for learning *perfect disjuncts*: it generates an initial set of *candidates* and repeatedly selects and refines the *best candidate* until either it finds a *perfect disjunct*, or it runs out of candidates. To find the best disjunct in *Candidates*, STALKER looks for a disjunct D that accepts the largest number of positive examples. In case of a tie, the best disjunct is the one that accepts fewer false positives.

Each initial candidate is a 2-state landmark automaton in which the transition $S_0 \rightsquigarrow S_1$ is labeled by a landmark that is either a token t that ends one $Prefix_x(p)$, or a wildcard that "matches" such a t. The rationale behind the choice of the initial candidates is straightforward: as disjuncts have to completely consume each positive example, it follows that any disjunct that consumes a t-ended prefix must end with a landmark that consumes the trailing t.

Intuitively, the $Refine()$ function tries to obtain (potentially) better disjuncts either by making its landmarks more specific (*landmark refinements*), or by adding new states in the automaton (*topology refinements*). In order to perform a refinement, STALKER uses a *refining terminal*, which can be either a token or a wildcard (besides the seven predefined wildcards *Numeric, AlphaNumeric, Alphabetic, Capitalized, AllCaps, HtmlTag,* and *Symbol*, STALKER can also use domain specific wildcards that are defined by the user). The refining tokens are computed by $GetTokens()$, which returns all tokens that appear at least once in each training example.

Given a disjunct D, a landmark l from D, and a refining terminal t, a *landmark refinement* makes l more specific by concatenating t either at the beginning or at the end of l. By contrast, a *topology refinement* adds a new state S and leaves the existing landmarks unchanged. For instance, if D has a transition $A \rightsquigarrow^l B$ (i.e., the transition from A to B is

STALKER(*Examples*)
- let $RetVal$ be an empty $\mathcal{SLG}$
- WHILE $Examples \neq \emptyset$
  - $aDisjunct$ =**LearnDisjunct**($Examples$)
  - remove all examples covered by $aDisjunct$
  - add $aDisjunct$ to $RetVal$
- return $RetVal$

LearnDisjunct(*Examples*)
- $Terminals = Wildcards \cup$ **GetTokens**($Examples$)
- $Candidates$ = **GetInitialCandidates**(*Examples*)
- WHILE $Candidates \neq \emptyset$ DO
  - let D = **BestDisjunct**($Candidates$)
  - IF D is a perfect disjunct THEN return D
  - FOR EACH $t \in Terminals$ DO
  $Candidates = Candidates \cup$ **Refine**(D, t)
  - remove D from $Candidates$
- return best disjunct

Figure 6: **The** STALKER **algorithm.**

labeled by the landmark l), then given a refining terminal t, a *topology refinement* creates two new disjuncts in which the transition above is replaced either by $A \rightsquigarrow^{l} S \rightsquigarrow^{t} B$ or by $A \rightsquigarrow^{t} S \rightsquigarrow^{l} B$.

Finally, STALKER can be easily extended such that it also uses the $SkipUntil()$ and $NextLandmark()$ features. The rule refining process remains unchanged (after all, the two new features change only the meaning of the last landmark in a disjunct), and the only modification involves $GenerateInitialCandidates()$. More precisely, for each initial candidate $SkipTo(t)$ and each wildcard w that matches the first token in an instance of x, STALKER must also generate the initial candidates $SkipTo(t)SkipUntil(w)$ and $SkipTo(t)NextLandmark(w)$.

6 Example of Rule Induction

Let us consider again the restaurant addresses from Figure 4. In order to generate an extraction rule for the `area-code`, we invoke STALKER with the training examples $\{E1, E2, E3, E4\}$. During the first iteration, $LearnDisjunct()$ creates the four initial candidates shown in Figure 8; that is, one for each token that ends a $Prefix_x(p)$ (i.e., **R1** and **R2**), and one for each wildcard that matches such a token (i.e., **R3** and **R4**). As **R1** is a *perfect disjunct*[1], $LearnDisjunct()$ returns **R1** and the first iteration ends.

During the second iteration, $LearnDisjunct()$ is invoked with the uncovered training examples $\{E1, E2\}$ and computes the set of refining terminals

{ `Phone <b> </b> : , .` *HtmlTag Word Symbol* }

Then $LearnDisjunct()$ generates the initial candidate rules **R5** and **R6** (see Figure 7). As both candidates accept the same false positives (i.e., the prefix of each example that ends before the city name), $LearnDisjunct()$ randomly selects the rule to be refined first - say **R5**. By refining **R5**, STALKER creates the topological refinements **R7**, **R8**, ..., **R16** (Figure 7 shows only the first four of them), together with the landmark refinements **R17** and **R18**.

As **R7** is a perfect disjunct (i.e., it covers both $E1$ and $E3$), there is no need for additional iterations. Finally, STALKER completes its execution by returning the disjunctive rule **either R1 or R7**.

[1] Remember that a *perfect disjunct* correctly matches several examples (e.g., $E2$ and $E4$) and rejects all other examples.

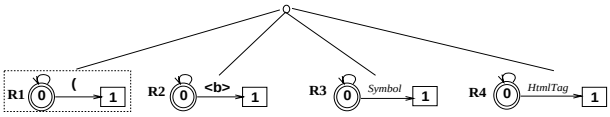


Figure 8: **Initial candidates (first iteration).**

SRC	Name	Nmb. of Leaves	Nmb. Docs.	Available Examples
S1	OKRA	4	252	3335
S2	BigBook	6	235	4299
S3	Address Finder	6	10	57
S4	Quote Server	18	10	22

Table 1: **Illustrative information sources.**

7 Experimental Results

In Table 1, we present four illustrative information sources that were selected from the larger set of sources on which Kushmerick's WIEN [12] system was tested.[2] **S1** and **S2** are the hardest sources that WIEN could wrap (i.e., they required the largest number of training examples), while **S3** and **S4** were beyond WIEN's capabilities because they have missing items and/or items that appear in various orders. For each source, Table 1 provides the following information: the name of the source, the number of leaves in the $\mathcal{EC}$ tree, the number of documents that were used by Kushmerick to generate training and test examples, and the average number of occurrences of each item in the given set of documents. Each of the sources above is a list of k-tuples, where k is the number of leaves from Table 1; consequently, in all four cases, the learning problem consists of finding one list extraction rule (i.e., a rule that can extract the whole list from the page), one list iteration rule, and k item extraction rules (one rule for each of the k leaves).

As we noticed that, in practice, a user rarely has the patience of labeling more than a dozen training examples, the main point of our experiments was to verify whether or not STALKER can generate high accuracy rules based just on a few training examples. Our experimental setup was the following: we started with one randomly chosen training example, learned an extraction rule, and tested it against all the unseen examples. We repeated these steps 500 times, and we averaged the number of test examples that were correctly extracted. Then we repeated the same procedure with 2, 3, ..., and 10 training examples. As we will see later in this section, STALKER usually requires less then 10 examples to obtain a 97% average accuracy over 500 trials.

We must emphasize that the 97% average accuracy means that out of the 500 learned rules, about 485 were capable of extracting correctly *all* the relevant data, while the other 15 rules were erroneous. This behavior has a simple explanation: as most information sources allow some variations in the document format, in the rare cases when a training set does not include the whole spectrum of variations or, even worse, when all the training examples are instances of the same type of document, the learned extraction rules perform poorly on some of the unseen examples.

In Table 2 we show some illustrative figures for WIEN and STALKER based on their respective performances on the four

[2] All WIEN sources can be obtained from the RISE [14] repository.

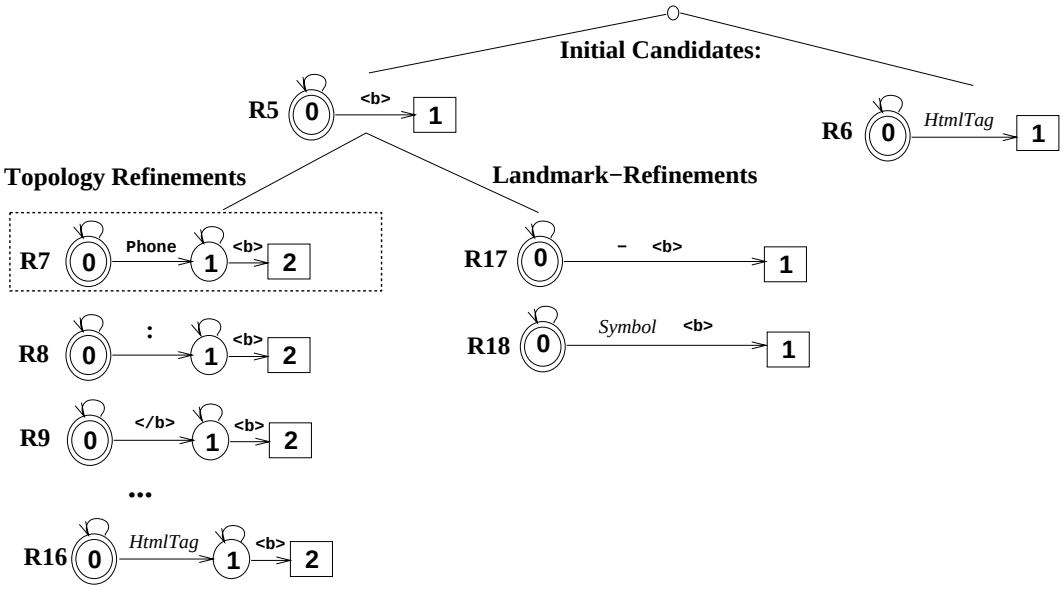


Figure 7: **Rule induction (second iteration).**

SR	Missing Items	Various Orders	WIEN Exs	WIEN CPU	STALKER Exs	STALKER CPU
S1	-	-	46	5 sec	1	19 sec
S2	-	-	274	83 sec	8	7 sec
S3	√	√	-	-	10	202 sec
S4	√	-	-	-	10	708 sec

Table 2: **Experimental data.**

test domains. Note that these numbers can not be used for a rigorous comparison for several reason. First, the WIEN data was collected for 100% accurate extraction rules, while we stopped the experiments after reaching either the 97% accuracy threshold or after training on 10 examples. Second, the two systems were not run on the same machine. Finally, as WIEN considers that a training example is a completely labeled document and each document contains several instances of the same item, we converted Kushmerick's original numbers by multiplying them with the average number of occurrences per page (remember that in the STALKER framework each occurrence of an item within a document is considered a distinct training example).

The results from Table 2 deserve a few comments. First, STALKER needs only a few training examples to wrap **S1** and **S2** with a 97% accuracy, while WIEN finds perfect rules, but requires up to two orders of magnitude more examples. Second, even for a significantly more difficult source like **S3**, which allows both missing items and items that appear in various orders (in fact, **S3** also allows an item to have several occurrences within the same tuple!), STALKER can learn extraction rules with accuracies ranging between 85% and 100%. Third, based on as few as 10 examples, STALKER could wrap **S4** with a median correctness of 79% over all 18 relevant items. This last figure is reasonable considering that some of the documents in **S4** contain so many errors and formatting exceptions that one of the authors (I. Muslea), who was given access to *all* available documents, required several hours to handcraft a set of extraction rules that were 88% correct. Last but not least, STALKER is reasonably fast: the easier sources **S1** and **S2** are completely

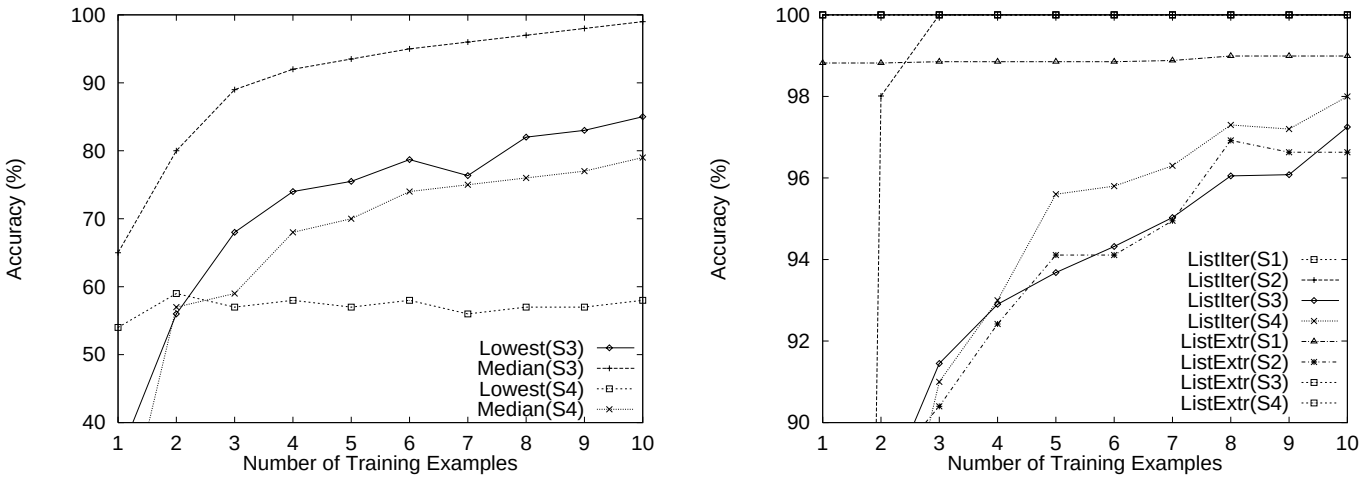
SRC	Extraction Task	Correctness	Nmb. Examples
S1	Score	100%	1
	Email	100%	1
	Name	100%	1
	FirstEntered	100%	1
	List Extraction	98.89%	10
	List Iteration	100%	1
S2	Name	100%	3
	Address	100%	3
	City	100%	4
	State	99.63%	10
	AreaCode	98.38%	10
	Phone	98.38%	10
	List Extraction	96.63%	10
	List Iteration	100%	3

Table 3: **Correctness levels for S1 and S2.**

wrapped in less then 20 seconds, while the more difficult sources take less than 40 seconds *per item*.

In Table 3, we provide detailed data about each extraction task in **S1** and **S2**; more precisely, for each extraction rule learned by STALKER, we show its accuracy and the number of examples required to reach it. As the documents in **S1** have an extremely regular structure, except for the list extraction rule, all other rules have a 100% accuracy based on a single training example! The source **S2** is more difficult to wrap: even though half of the rules have a 100% accuracy, they can be induced only based on three or four examples. Furthermore, the other four rules can not achieve a 100% accuracy even based on 10 training examples.

In order to better understand STALKER's behavior on difficult extraction tasks, in Figure 9(a) we show the learning curves for the lowest and median accuracy extraction tasks for both **S3** and **S4**. For **S3**, the hardest extraction task can be achieved with a 85% accuracy based on just 10 training examples; furthermore, the 99% accuracy of the median difficulty tasks tells us that half of the items can be extracted with an accuracy of at least 99%. Even though STALKER can not generate high accuracy rules for all the items in **S4**,



(a) The median and lowest accuracy tasks for **S3** and **S4**. (b) List extraction and list iteration tasks.

Figure 9: Learning curves for the illustrative sources.

our hierarchical approach, which extracts the items independently of their siblings in the $\mathcal{EC}$ tree, allows the user to extract at least the items for which STALKER generates accurate rules.

Finally, in Figure 9(b) we present the learning curves for the list extraction and list iterations tasks for all four sources.[3] It is easy to see that independently of how difficult it is to induce all the extraction rules for a particular source, learning list extraction and list iteration rules is a straightforward process that converges quickly to an accuracy level above 95%. This fact strengthens our belief that the $\mathcal{EC}$ formalism is extremely useful to the break down of a hard problem into several easier ones.

Based on the results above, we can draw two important conclusions. First of all, since most of the relevant items are relatively easy to extract based on just a few training examples, we can infer that our hierarchical approach to wrapper induction was beneficial in terms of reducing the amount of necessary training data. Second, the fact that even for the hardest items in **S4** we can find a correct rule (remember that the low correctness comes from averaging correct rules with erroneous ones) means that we can try to improve STALKER's behavior based on active learning techniques [13] that would allow the algorithm to select the few relevant cases that would lead to a correct rule.

8 Related Work

Research on learning extraction rules has occurred mainly in two contexts: creating wrappers for information agents and developing general purpose information extraction systems for natural language text. The former are primarily used for semistructured information sources, and their extraction rules rely heavily on the regularities in the structure of the documents; the latter are applied to free text documents and use extraction patterns that are based on syntactic and semantic information.

With the increasing interest in accessing Web-based information sources, a significant number of research projects depend on wrappers to retrieve the relevant data. A wide variety of languages have been developed for manually writing wrappers (i.e., where the extraction rules are written by a human expert), from procedural languages [2] and Perl scripts [7] to pattern matching [5] and LL(k) grammars [6]. Even though these systems offer fairly expressive extraction languages, the manual wrapper generation is a tedious, time consuming task that requires a high level of expertise; furthermore, the rules have to be rewritten whenever the sources suffer format changes. In order to help the users cope with these difficulties, Ashish and Knoblock [1] proposed an expert system approach that uses a fixed set of heuristics of the type "look for bold or italicized strings".

The wrapper induction techniques introduced in WIEN [12] are better fit to frequent format changes because they rely on learning techniques to generate the extraction rules. Compared to the manual wrapper generation, Kushmerick's approach has the advantage of dramatically reducing both the time and the effort required to wrap a source; however, his extraction language is significantly less expressive than the ones provided by the manual approaches. In fact, the WIEN extraction language is a 1-disjunctive $\mathcal{LA}$ that is interpreted as a $SkipTo()$ and does not allow the use of wildcards. There are several other important differences between STALKER and WIEN. First, as WIEN learns the landmarks by searching *common prefixes* at the *character level*, it needs more training examples than STALKER. Second, WIEN cannot wrap sources in which some items are missing or appearing in various orders. Last but not least, STALKER can handle $\mathcal{EC}$ trees of arbitrary depths, while WIEN's approach to nested documents turned out to be prohibitive in terms of CPU time.

SoftMealy [9] uses a wrapper induction algorithm that generates extraction rules expressed as finite transducers. The SoftMealy rules are more general than the WIEN ones because they use wildcards and they can handle both missing items and items appearing in various orders. The SoftMealy extraction language is a k-disjunctive $\mathcal{LA}$, where each disjunct is either a $SkipTo()NextLandmark()$ or a single $SkipTo()$. As SoftMealy does not use neither multiple $SkipTo()$s nor $SkipUntil()$s, it follows that its extraction rules are strictly less expressive than STALKER's. Finally, SoftMealy has one additional drawback: in order to deal with missing items and various orderings of items, SoftMealy

[3] The learning curves for **ListExtr(S3)**, **ListExtr(S4)**, and **ListIter(S1)** are identical because all three learning tasks reach a 100% accuracy after seeing a single example.

has to see training examples that include *each possible ordering* of the items.

In contrast to information agents, most general purpose information extraction systems are focused on unstructured text, and therefore the extraction techniques text are based on linguistic constraints. However, there are three such systems that are somewhat related to STALKER: WHISK [15], Rapier [4], and SRV [8]. The extraction rules induced by Rapier and SRV can use the landmarks that immediately precede and/or follow the item to be extracted, while WHISK is capable of using multiple landmarks. But, similarly to STALKER and unlike WHISK, Rapier and SRV extract a particular item independently of the other relevant items. It follows that WHISK has the same drawback as SoftMealy: in order to handle correctly missing items and items that appear in various orders, WHISK must see training examples for each possible ordering of the items. None of these three can handle embedded data, though all use powerful linguistic constraints that are beyond STALKER's capabilities.

9 Conclusions and Future Work

The primary contribution of our work is to turn a potentially hard problem – learning extraction rules – into a problem that is extremely easy in practice (i.e., typically very few examples are required). The number of required examples is small because the $\mathcal{EC}$ description of a page simplifies the problem tremendously: as the Web pages are intended to be human readable, the $\mathcal{EC}$ structure is generally reflected by actual landmarks on the page. STALKER merely has to find the landmarks, which are generally in the close proximity of the items to be extracted. In other words, given our $\mathcal{SLG}$ formalism, the extraction rules are typically very small, and, consequently, they are easy to induce.

We plan to continue our work on several directions. First, we plan to use *unsupervised learning* in order to narrow the landmark search-space. Second, we would like to use *active learning* techniques to minimize the amount of labeling that the user has to perform. Third, we hope to create a polynomial-time version of STALKER and to provide PAC-like guarantees for the new algorithm.

Acknowledgments

This work was supported in part by USC's Integrated Media Systems Center (IMSC) - an NSF Engineering Research Center, by the National Science Foundation under grant number IRI-9610014, by the U.S. Air Force under contract number F49620-98-1-0046, by the Defense Logistics Agency, DARPA, and Fort Huachuca under contract number DABT63-96-C-0066, and by a research grant from General Dynamics Information Systems. The views and conclusions contained in this paper are the authors' and should not be interpreted as representing the official opinion or policy of any of the above organizations or any person connected with them.

References

[1] Ashish, N., and Knoblock, C. Semi-automatic wrapper generation for internet information sources. *Proceedings of Cooperative Information Systems* (1997).

[2] Atzeni, P., and Mecca, G. Cut and paste. *Proceedings of 16th ACM SIGMOD Symposion on Principles of Database Systems* (1997).

[3] Atzeni, P., Mecca, G., and Merialdo, P. Semi-structured and structured data in the web: going back and forth. *Proceedings of ACM SIGMOD Workshop on Management of Semi-structured Data* (1997), 1–9.

[4] Califf, M., and Mooney, R. Relational learning of pattern-match rules for information extraction. *Working Papers of the ACL-97 Workshop in Natural Language Learning* (1997), 9–15.

[5] Chawathe, S., Garcia-Molina, H., Hammer, J., Ireland, K., Papakonstantinou, Y., Ullman, J., and Widom., J. The tsimmis project: integration of heterogeneous information sources. *10th Meeting of the Information Processing Society of Japan* (1994), 7–18.

[6] Chidlovskii, B., Borghoff, U., and Chevalier, P. Towards sophisticated wrapping of web-based information repositories. *Proceedings of 5th International RIAO Conf.* (1997), 123–35.

[7] Cohen, W. A web-based information system that reasons with structured collections of text. *Proceedings of Autonomous Agents AA-98* (1998), 400–407.

[8] Freitag, D. Information extraction from html: Application of a general learning approach. *Proceedings of the Fifteenth Conference on Artificial Intelligence AAAI-98* (1998), 517–523.

[9] Hsu, C. Initial results on wrapping semistructured web pages with finite-state transducers and contextual rules. *AAAI-98 Workshop on AI and Information Integration* (1998), 66–73.

[10] Kirk, T., Levy, A., SAgiv, Y., and Srivastava, D. The information manifold. *AAAI Spring Symposium: Information Gathering from Heterogeneous Distributed Environments* (1995), 85–91.

[11] Knoblock, C., Minton, S., Ambite, J., Ashish, N., Margulis, J., Modi, J., Muslea, I., Philpot, A., and Tejada, S. Modeling web sources for information integration. *Proceedings of the Fifteenth National Conference on ASrtificial Intelligence* (1998), 211–218.

[12] Kushmerick, N. Wrapper induction for information extraction. *PhD Thesis, Dept. of Computer Science, U. of Washington, TR UW-CSE-97-11-04* (1997).

[13] RayChaudhuri, T., and Hamey, L. Active learning-approaches and issues. *Journal of Intelligent Systems 7* (1997), 205–243.

[14] RISE. Rise: A repository of online information sources used in information extraction tasks. *[http://www.isi.edu/ muslea/RISE/index.html] Information Sciences Institute / USC* (1998).

[15] Soderland, S. Learning information extraction rules for semi-structured and free text. *http://www.cs.washington.edu/homes/soderlan/WHISK.ps* (1998).