ELSEVIER

Artificial Intelligence 118 (2000) 15–68

Artificial Intelligence

www.elsevier.com/locate/artint

Wrapper induction: Efficiency and expressiveness

Nicholas Kushmerick [1]

Department of Computer Science, University College Dublin, Dublin 4, Ireland

Received 30 May 1998; received in revised form 10 March 1999

Abstract

The Internet presents numerous sources of useful information—telephone directories, product catalogs, stock quotes, event listings, etc. Recently, many systems have been built that automatically gather and manipulate such information on a user's behalf. However, these resources are usually formatted for use by people (e.g., the relevant content is embedded in HTML pages), so extracting their content is difficult. Most systems use customized *wrapper* procedures to perform this extraction task. Unfortunately, writing wrappers is tedious and error-prone. As an alternative, we advocate *wrapper induction*, a technique for automatically constructing wrappers. In this article, we describe six wrapper classes, and use a combination of empirical and analytical techniques to evaluate the computational tradeoffs among them. We first consider *expressiveness*: how well the classes can handle actual Internet resources, and the extent to which wrappers in one class can mimic those in another. We then turn to *efficiency*: we measure the number of examples and time required to learn wrappers in each class, and we compare these results to PAC models of our task and asymptotic complexity analyses of our algorithms. Summarizing our results, we find that most of our wrapper classes are reasonably useful (70% of surveyed sites can be handled in total), yet can rapidly learned (learning usually requires just a handful of examples and a fraction of a CPU second per example).

Keywords: Information extraction; Wrapper induction; Machine learning; Internet information integration; Information agents

1. Introduction

The Internet presents a stunning variety of on-line information resources: telephone directories, retail product catalogs, weather forecasts, airline schedules, event schedules,

[1] Email: nick@ucd.ie.

0004-3702/00/$ – see front matter
PII: S0004-3702(99)00100-9

and many more. Recently, there has been much interest in systems (such as software agents [30,36,49,66] or information-integration systems [8,18,19,45,55]) that automatically access such resources, manipulating their content on a user's behalf.

Numerous technical problems arise when building such a system. These challenges have lead to work on resource discovery [13], Web query languages [48,57], semi-structured data models [1,15], query planning [24,36,55], reasoning about local completeness [28,53] and ontological [31,56] knowledge, and handling heterogeneous identifiers [20,59].

In this article, we address yet another challenge: we would like to build systems that make use of the Internet's content, but much of this content is formatted for people rather than machines. Specifically, the content is often embedded in an HTML page, and an information-integration system must extract the relevant text, while discarding irrelevant material such as HTML tags or advertisements. In this article, we describe techniques that enable information-integration systems to automatically make use of such valuable but obscured information.

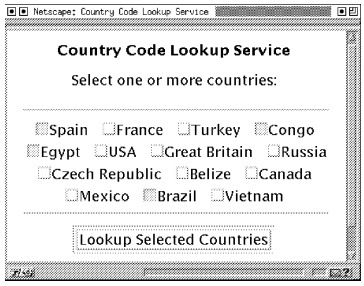
Fig. 1 provides a concrete example of the sort of information resource with which we are concerned. Consider a fictitious Internet site that provides information about countries and their telephone country codes. When the form in Fig. 1(a) is submitted, the resource responds as shown in Fig. 1(b), which was rendered from the HTML shown in Fig. 1(c). Of course, this raw HTML is of little use to a system seeking information about countries and their country codes. Such a system must extract the response's actual content; see Fig. 1(d).

One way to perform this extraction task is to invoke the customized wrapper procedure $\mathsf{ccwrap}_{\mathrm{LR}}$, shown in Fig. 1(e). $\mathsf{ccwrap}_{\mathrm{LR}}$ has two nested loops; the outer 'while' loop extracts a country/code pair, and the inner 'for' loop extracts these two attributes in sequence. $\mathsf{ccwrap}_{\mathrm{LR}}$ is 'hard-wired' to the country/code site, with the inner loop iterating exactly twice for each iteration of the outer loop.

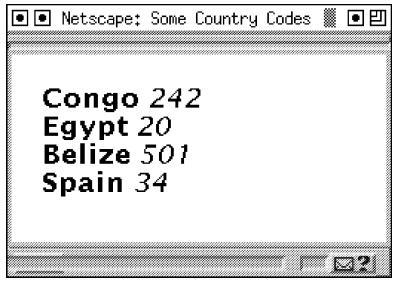
The $\mathsf{ccwrap}_{\mathrm{LR}}$ procedure works because the site exhibits a uniform formatting convention: **countries** are rendered in bold, while country *codes* are in italics. $\mathsf{ccwrap}_{\mathrm{LR}}$ operates by scanning the HTML document for particular strings ('<B>', '</B>', '<I>' and '</I>') that identify the text fragments to be extracted. These strings are identified by $\mathsf{ccwrap}_{\mathrm{LR}}$ as ℓ_1, r_1, ℓ_2 and r_2, respectively. The notation ℓ_k ($k \in \{1, 2\}$) indicates that the string delimits the *left*-hand edge of an attribute to be extracted, while r_k indicates a *right* delimiter.

When given a page such as P_{CC}, $\mathsf{ccwrap}_{\mathrm{LR}}$ sequentially scans the entire page. The outer loop checks whether there are additional country/code pairs to extract, by looking for the delimiter '<B>' in the unscanned portion of the page. As long as the beginning of a country is found, the inner loop is invoked to extract the appropriate page substrings.

Where does the $\mathsf{ccwrap}_{\mathrm{LR}}$ wrapper come from? Few Internet sites publish their formatting conventions, and thus the designer of an information-gathering system must manually construct such a wrapper for each resource. While an individual wrapper is usually structurally quite simple, hand-coding the details is tedious and error-prone. Moreover, we are interested in the scaling issues that arise as we build systems that integrate information from hundreds or thousands of Internet sources. Excite's 'Jango' shopping agent, for example, relied on several hundred wrappers, each with a mean time to failure of about one month [74]. Finally, most sites periodically change their formatting



(a)



(b)

```
<HTML><TITLE>Some Country Codes</TITLE><BODY>
<B>Congo</B> <I>242</I><BR>
<B>Egypt</B> <I>20</I><BR>
<B>Belize</B> <I>501</I><BR>
<B>Spain</B> <I>34</I><BR>
</BODY></HTML>
```

(c)

$$\left\{ \begin{array}{c} \langle \text{`Congo'}, \text{`242'} \rangle, \\ \langle \text{`Egypt'}, \text{`20'} \rangle, \\ \langle \text{`Belize'}, \text{`501'} \rangle, \\ \langle \text{`Spain'}, \text{`34'} \rangle \end{array} \right\}$$

(d)

procedure $\text{ccwrap}_{\text{LR}}$(page P)
  while there are more occurrences in P of '<B>'
    for each $\langle \ell_k, r_k \rangle \in \{\langle$'<B>', '</B>'$\rangle, \langle$'<I>', '</I>'$\rangle\}$
      scan in P to next occurrence of ℓ_k; save position as start of kth attribute
      scan in P to next occurrence of r_k; save position as end of kth attribute
  return extracted $\{\ldots, \langle \text{country}, \text{code} \rangle, \ldots\}$ pairs

(e)

Fig. 1. A fictitious Internet site providing information about countries and their telephone country codes: (a) the search form; (b) an example response page; (c) P_{CC}, the HTML page for (b); (d) the response's content; and (e) the $\text{ccwrap}_{\text{LR}}$ procedure, which generates (d) from (c).

conventions, which usually breaks an existing wrapper [51]. For these reasons, wrapper programming and maintenance is a serious knowledge-engineering.

To facilitate wrapper construction and maintenance, we advocate *wrapper induction* [50,52], a technique for automatically learning wrappers. Wrapper induction involves generalizing from a set of examples of a resource's pages, each annotated with the text fragments to be extracted. For example, given a set of ⟨page, content⟩ pairs such as ⟨Fig. 1(c), Fig. 1(d)⟩, our wrapper induction algorithm generates $\text{ccwrap}_{\text{LR}}$.

As in many machine-learning applications, the key to effective learning is to *bias* the learning algorithm [58]. In our work, biases correspond to *wrapper classes*. For example, $\mathsf{ccwrap}_{\mathrm{LR}}$ is an instance of a wrapper class we call Left-Right (LR), which extracts the text indicated by specific left- and right-hand delimiters. In this article we describe LR as well as several wrapper classes that extend it in various ways.

While the particular classes we have identified are interesting in their own right, in this article we focus on the computational tradeoffs among them. We have compared our wrapper classes using a combination of empirical and analytical techniques. Our evaluation can be characterized in terms of the following hierarchical organization:

I — EXPRESSIVENESS: The first issue concerns how useful the wrapper classes are for handling actual Internet resources, and the extent to which sites handled by one class can be handled by others.

I-1 — COVERAGE: We conducted a survey of actual Internet sites, to determine which can be handled by each class. Unlike similar information-extraction and -retrieval tasks, we are interested only in wrappers that exhibit 100% precision and recall. Thus rather than measuring the accuracy of a wrapper class, we are interested in *coverage*, the fraction of Internet sites for which there exists a 100%-accurate wrapper in the class. Our classes can handle 70% of the sites in total; see Fig. 15 in Section 5.1.

I-2 — RELATIVE EXPRESSIVENESS: A more formal question is the extent to which wrappers in one class can mimic those in another. The relationships turn out to be rather subtle; see Theorem 1 in Section 5.2.

II — EFFICIENCY: Our expressiveness results demonstrate the usefulness of our wrapper classes, but can they be learned quickly? We decompose this second aspect of our analysis into two parts: how many examples are needed, and how much computation is required?

II-1 — SAMPLE COST: Intuitively, the more examples provided to the learner, the more likely that the wrapper is correct. We assessed the number of examples required both empirically and analytically.

II-1-a — EMPIRICAL RESULTS: We measured the number of examples needed to learn a wrapper that performs perfectly on a suite of test pages. We find that 2–3 examples usually suffice; see Fig. 18 in Section 6.1.1.

II-1-b — SAMPLE COMPLEXITY: We also developed a PAC [7,73] model of our learning task, which formalizes the intuition that more examples improves learning. We have derived bounds on the number of examples needed to ensure (with high probability) that learned wrappers rarely (with low probability) make mistakes. We have shown that the number of examples required is polynomial in the relevant parameters; see Theorem 2 in Section 6.1.2.

II-2 — INDUCTION COST: While sample cost measures the number of examples required, we are also concerned with the time to process the examples.

II-2-a — EMPIRICAL RESULTS: When tested on actual Internet sites, our learning algorithms usually require less than one CPU second per example; see Fig. 19 in Section 6.2.1.

II-2-b — TIME COMPLEXITY: We have also performed a complexity analysis on our algorithms. As stated in Theorem 3 in Section 6.2.2, most of our wrapper classes can be learned in polynomial time.

The remainder of this article is organized as follows. We begin in Section 2 with a formal characterization of wrappers and our wrapper induction task. In Section 3 we describe the LR class just mentioned, and in Section 4 we describe five variants of LR. We then evaluate these six classes as described: in Section 5 we discuss expressiveness (issue I above), and in Section 6 we turn to efficiency (issue II). Wrapper induction requires that the example pages be properly labeled prior to learning; in Section 7 we briefly introduce *corroboration*, our work on automating this page-labeling step. We conclude with a discussion of related (Section 8) and future (Section 9) research.

2. Wrapper induction

The wrapper induction problem is framed in terms of a simple model of information extraction; see Fig. 2.

Resources, queries, and pages. As shown in Fig. 2, an *information resource* $\mathcal{S}$ is a function from a *query* Q to a *response page* P.

Query Q describes the desired information, in terms of an expression in some query language $\mathcal{Q}$. For typical Internet resources, the query is represented by the arguments to a CGI script; alternatively, $\mathcal{Q}$ might be SQL or KQML. (We are concerned mainly with the response pages, and so will largely ignore $\mathcal{Q}$. This focus is motivated by our assumption that the issues related to learning to extract information from the responses can be decoupled from the issues related to learning to pose queries. Of course, learning to pose queries is an important research issue; see [23] for some interesting progress.)

Response page P is the resource's answer to the query. We take P to be a string over some alphabet Σ. Typically, Σ is the ASCII character set, and the pages are HTML documents. For example, earlier we saw the query response in Fig. 1(c); for convenience, we will hereafter refer to this page as P_{CC}. Note that our techniques are motivated by, but do not rely on, the use of HTML. For example, the responses might be natural language text, or obey a standard such as KIF or XML.

Attributes and tuples. We adopt a standard relational data model. Associated with each information resource is a set of K distinct *attributes*, each representing a column in the relational model. In the country/code example, there are $K = 2$ attributes.

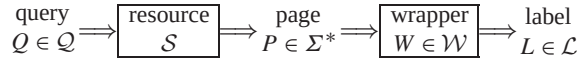


Fig. 2. A simple model of information extraction: resources map queries to pages, and wrappers map pages to labels.

A *tuple* is a vector $\langle A_1, \ldots, A_K \rangle$ of K strings; $A_k \in \Sigma^*$ for each k. String A_k is the *value* of tuple's kth attribute. Whereas attributes represent columns in the relational model, tuples represent rows. The example country/code page P_{CC} contains four tuples, the first of which is $\langle$'Congo', '242'$\rangle$. Note that attributes values must not overlap.

Content and labels. The *content* of a page is the set of tuples it contains. For example, the content of the example country/code page is shown in Fig. 1(d).

The literal string notation used in Fig. 1(d) is adequate, but since pages have unbounded length, we use instead a cleaner and more concise representation of a page's content. Rather than listing the attributes explicitly, a page's *label* represents the content in terms of a set of indices into the page. Note that representing a pages content with such indices is visually simpler than, but computationally equivalent to, the literal string notation.

For example, the label for the example country/code page P_{CC} is

$$L_{\text{CC}} = \left\{ \begin{array}{c} \big\langle \langle 50, 55\rangle,\ \langle 63, 66\rangle \big\rangle, \\ \big\langle \langle 78, 83\rangle,\ \langle 91, 93\rangle \big\rangle, \\ \big\langle \langle 105, 111\rangle,\ \langle 119, 122\rangle \big\rangle, \\ \big\langle \langle 134, 139\rangle,\ \langle 147, 149\rangle \big\rangle \end{array} \right\}.$$

Label L_{CC} indicates that the example country/code page contains four tuples, where each tuple consists of $K = 2$ attributes values. Each value is represented by a pair of integers. Consider the first pair, $\langle 50, 55\rangle$. These integers indicate that the first attribute of the first tuple is the substring between positions 50 and 55 (i.e., the string 'Congo'); inspection of Fig. 1(c) reveals that these integers are correct. Similarly, the last pair, $\langle 147, 149\rangle$, indicates that the last attribute's country code occurs between positions 147 and 149 (i.e., the string '34').

More generally, the content of page P is represented as the label

$$L = \left\{ \begin{array}{c} \big\langle \langle b_{1,1}, e_{1,1}\rangle, \quad \ldots, \quad \langle b_{1,k}, e_{1,k}\rangle, \quad \ldots, \quad \langle b_{1,K}, e_{1,K}\rangle \big\rangle, \\ \vdots \\ \big\langle \langle b_{m,1}, e_{m,1}\rangle, \quad \ldots, \quad \langle b_{m,k}, e_{m,k}\rangle, \quad \ldots, \quad \langle b_{m,K}, e_{m,K}\rangle \big\rangle, \\ \vdots \\ \big\langle \langle b_{|L|,1}, e_{|L|,1}\rangle, \ \ldots, \ \langle b_{|L|,k}, e_{|L|,k}\rangle, \ \ldots, \ \langle b_{|L|,K}, e_{|L|,K}\rangle \big\rangle \end{array} \right\}.$$

Label L encodes the content of page P. The page contains $|L| > 0$ tuples, each of which has $K > 0$ attributes. The integers $1 \leqslant k \leqslant K$ are the attributes indices, while the integers $1 \leqslant m \leqslant |L|$ index tuples within the page. Each pair $\langle b_{m,k}, e_{m,k}\rangle$ encodes a single attribute value. The value $b_{m,k}$ is the index in P of the *beginning* of the kth attribute value in the mth tuple. Similarly, $e_{m,k}$ is *end* index of the kth attribute value in the mth tuple. Thus, the kth attribute of the mth tuple occurs between positions $b_{m,k}$ and $e_{m,k}$ of page P. For example, the pair $\langle b_{3,2}, e_{3,2}\rangle = \langle 147, 149\rangle$ above encodes the second (country code) attribute of the example page's fourth tuple.

The symbol $\mathcal{L}$ in Fig. 2 refers to the (infinite) set of all labels.

Wrappers and wrapper classes. As shown in Fig. 2, a *wrapper* W is a function from a page to a label; the notation $W(P) = L$ indicates that the result of invoking wrapper W on page P is label L.

At this level of abstraction, a wrapper is simply an arbitrary procedure, but in this article we examine several classes of wrappers. Formally, a *wrapper class* $\mathcal{W}$ is simply a set of wrappers. The classes we consider are infinite, comprising all ways to "instantiate" a "template" for writing wrappers in each class.

The wrapper induction problem. Finally, we are in a position to state our task: we want to learn a wrapper for information resource $\mathcal{S}$, and we will be interested in wrappers from some class $\mathcal{W}$.

Intuitively, the input to our learning system is a sample of $\mathcal{S}$'s pages and their associated labels, and the output should be a wrapper $W \in \mathcal{W}$. Ideally, we want W to output the appropriate label for all of $\mathcal{S}$'s pages. In general we can not make such a guarantee, so (in the spirit of inductive learning) we demand that W generate the correct labels for a given set of training examples.

More formally, the *wrapper induction problem* (with respect to a particular wrapper class $\mathcal{W}$) is as follows:

input: a set $\mathcal{E} = \{\ldots, \langle P_n, L_n\rangle, \ldots\}$ of *examples*, where each P_n is a page, and each L_n is a label;

output: a wrapper $W \in \mathcal{W}$, such that $W(P_n) = L_n$ for every $\langle P_n, L_n\rangle \in \mathcal{E}$.

3. The LR wrapper class

The $\mathsf{ccwrap}_{\mathrm{LR}}$ procedure (Fig. 1(e)) illustrates a "programming idiom"—using left- and right-hand delimiters to extract the relevant fragments—that is useful for resources other than just the country/code site. The Left-Right (LR) wrapper class is one way to formalize this idiom. As shown in Fig. 3, LR is a generalization of $\mathsf{ccwrap}_{\mathrm{LR}}$ that allows:

(1) the delimiters to be arbitrary strings (instead of the specific values '<B>', '</B>', etc.); and
(2) any number K of attributes $A_1, \ldots, A_K$ (rather than exactly two).

Note that although the delimiters in this example are entire HTML tags, our techniques do not require this. For example, the left/right delimiters '<A href="' and '">' could

```
procedure exec_LR(wrapper ⟨ℓ_1, r_1, ..., ℓ_K, r_K⟩, page P)
  m ← 0
  while there are more occurrences in P of ℓ_1                          [i]
    m ← m + 1
    for each ⟨ℓ_k, r_k⟩ ∈ {⟨ℓ_1, r_1⟩, ..., ⟨ℓ_K, r_K⟩}
      scan in P to the next occurrence of ℓ_k; save position as b_{m,k}  [ii]
      scan in P to the next occurrence of r_k; save position as e_{m,k}  [iii]
  return label {..., ⟨⟨b_{m,1}, e_{m,1}⟩, ..., ⟨b_{m,K}, e_{m,K}⟩⟩, ...}
```

Fig. 3. The $\mathsf{exec}_{\mathrm{LR}}$ procedure specifies how an LR wrapper is executed.

be used to extract URLs. Indeed, the text might not be HTML at all: '`(`' and '`)`' might be valid area code delimiters for phone numbers such as '`(206)367-2578`'.

The $\mathsf{exec}_{\mathrm{LR}}$ routine specifies how LR wrappers behave. Earlier we stated that the $W(P)$ is the label that results from invoking wrapper W on page P; $\mathsf{exec}_{\mathrm{LR}}$ is simply a procedure for computing $W(P)$ from W and P, for the case when W is an LR wrapper.

The values of $\ell_1, \ldots, \ell_K$ indicate the left-hand attribute delimiters, while $r_1, \ldots, r_K$ indicate the right-hand delimiters. For example, if $\mathsf{exec}_{\mathrm{LR}}$ is invoked with the parameters $K = 2$, $\ell_1 =$ '`<B>`', $r_1 =$ '`</B>`', $\ell_2 =$ '`<I>`' and $r_2 =$ '`</I>`', then $\mathsf{exec}_{\mathrm{LR}}$ behaves like $\mathsf{ccwrap}_{\mathrm{LR}}$.[2]

Notice that the behavior of the country/code wrapper $\mathsf{ccwrap}_{\mathrm{LR}}$ can be entirely encapsulated in terms of a vector of four strings ⟨'`<B>`', '`</B>`', '`<I>`', '`</I>`'⟩.

More generally, any LR wrapper for a site containing K attributes is equivalent to a vector of $2K$ strings $\langle \ell_1, r_1, \ldots, \ell_K, r_K \rangle$, and any such vector can be interpreted as an LR wrapper. Given this equivalence, we use the notation $\langle \ell_1, r_1, \ldots, \ell_K, r_K \rangle$ as a shorthand for the LR wrapper obtained by partially evaluating $\mathsf{exec}_{\mathrm{LR}}$ with the given delimiters.

LR is very simple; indeed one might wonder whether it is so simple as to be useless. Before describing our algorithm for automatically constructing LR wrappers, it is worthwhile to look ahead to our main empirical results. We find that LR is reasonably useful (it can handle 53% of a surveyed collection of Internet sites; Section 5.1), yet LR wrappers can be learned in just a few seconds (Section 6.2.1), based on just a handful of examples (Section 6.1.1).

Since an LR wrapper is simply a vector $\langle \ell_1, r_1, \ldots, \ell_K, r_K \rangle$, the LR wrapper induction problem thus is one of identifying $2K$ delimiter strings $\langle \ell_1, r_1, \ldots, \ell_K, r_K \rangle$, on the basis of a set $\mathcal{E} = \{\ldots, \langle P_n, L_n \rangle, \ldots\}$ of example pages and their labels. More precisely, we must solve the following constraint satisfaction problem (CSP):

variables: delimiters $\ell_1, r_1, \ldots, \ell_K, r_K$;

domains: each delimiter is an arbitrary string;

constraints: $W(P_n) = L_n$ for every $\langle P_n, L_n \rangle \in \mathcal{E}$, where LR wrapper $W = \langle \ell_1, r_1, \ldots, \ell_K, r_K \rangle$.

In the remainder of this section we describe $\mathsf{learn}_{\mathrm{LR}}$, an algorithm that solves problems of this form.

3.1. Delimiter candidates

We begin by noting that the domains of the $2K$ variables are tightly constrained by the examples $\mathcal{E}$. At the very least, the delimiters must be substrings of the examples. Of course, we can do much better. On the basis of just the single example $\langle P_{\mathrm{CC}}, L_{\mathrm{CC}} \rangle$, we know that r_2 (the right-hand delimiter for the code attribute) must be a prefix of '`</I><BR>`⇓`</BODY></HTML>`'[3]. To see this, note that if r_2 is not a prefix of this

[2] Note that $\mathsf{ccwrap}_{\mathrm{LR}}$ is described somewhat informally; for example, a wrapper is supposed to output a label consisting of $\langle b_{m,k}, e_{m,k} \rangle$ pairs, but $\mathsf{ccwrap}_{\mathrm{LR}}$ does not explicitly mention these indices. The intent is that $\mathsf{exec}_{\mathrm{LR}}$ is both a generalization and a more precise specification of $\mathsf{ccwrap}_{\mathrm{LR}}$.

[3] The symbol '⇓' indicates a new-line character.

string, then every wrapper with this delimiter will, at the very least, fail to extract '34' as the code attribute for P_{CC}'s fourth tuple. Thus the candidates for r_2 are all prefixes of '</I>
⇓</BODY></HTML>'.

A similar analysis applies to all $2K$ delimiters. In detail, the candidates for the delimiters are generated as follows:

Candidates for the ℓ_k. Consider ℓ_2, the left-hand delimiter for the code attribute. Recall the fragments 'Congo</B> <I>', 'Egypt</B> <I>', etc. that precede the country codes in Fig. 1(c). Given these fragments, we know that ℓ_2 must be a suffix of '</B> <I>'. Thus the candidates for ℓ_2 are the eight non-empty suffixes of this string.

Delimiter ℓ_1 is more complicated, because the strings prior to the first attribute occur between the first attribute and the last attribute of the previous tuple, as well as between the start of the page and the first tuple. In the example, the strings under consideration are '<HTML><TITLE>Some Country Codes</TITLE><BODY>⇓<B>' and '</I>
⇓<B>'. Clearly ℓ_1 must be a suffix of these strings. Thus the candidates for ℓ_1 can be generated by enumerating the suffixes of one such fragment. (For efficiency, the shortest string yields the fewest candidates.)

To generalize this discussion, we have concluded that the candidates for delimiter ℓ_k given the example set $\mathcal{E}$—written $\mathsf{cands}_\ell(k,\mathcal{E})$—are generated by enumerating the suffixes of the shortest string occurring to the left of each instance of attribute k in each example. (As mentioned in the previous paragraph, the case $k=1$ is special: we must enumerate the suffixes of the shortest string either between adjacent tuples or before the first tuple.) For example, if $\mathcal{E}=\{\langle P_{\mathrm{CC}}, L_{\mathrm{CC}}\rangle\}$, then we have:

$$\mathsf{cands}_\ell(1,\mathcal{E}) = \left\{ \begin{array}{l} \text{'</I></BR>⇓<B>', '/I></BR>⇓<B>', 'I></BR>⇓<B>',} \\ \text{'></BR>⇓<B>', '</BR>⇓<B>', '/BR>⇓<B>', 'BR>⇓<B>',} \\ \text{'R>⇓<B>', '>⇓<B>', '⇓<B>', '<B>', 'B>', '>'} \end{array} \right\}, \tag{1}$$

$$\mathsf{cands}_\ell(2,\mathcal{E}) = \left\{ \begin{array}{l} \text{'</B> <I>', '/B> <I>', 'B> <I>', '> <I>', ' <I>',} \\ \text{'<I>', 'I>', '>'} \end{array} \right\}.$$

Candidates for the r_k. The candidates for the right-hand delimiters are generated similarly, but there are two differences. First, the strings under consideration occur to the right of the appropriate attribute (rather than to the left). Second, r_k must be a prefix (not a suffix) of these strings. For example, the delimiter r_1 must be a prefix of the string '</B> <I>', while r_2 must be a prefix of both '</I>
⇓<B>' and '</I>
⇓</BODY></HTML>'.

More generally, the candidates for delimiter r_k given the example set $\mathcal{E}$—written $\mathsf{cands}_r(k,\mathcal{E})$—are generated by enumerating the prefixes of the shortest string occurring to the right of each instance of attribute k in each example. (As discussed above, ℓ_1 is is a

special case. Similarly, r_K is a special case: we must enumerate the prefixes of the shortest string occurring either between adjacent tuples or after the last tuple.) For example:

$$\mathsf{cands}_r(1,\mathcal{E}) = \left\{ \begin{array}{l} \text{`</B> <I>'},\ \text{`</B> <I'},\ \text{`</B> <'},\ \text{`</B> '},\ \text{`</B>'}, \\ \text{`</B'},\ \text{`</'},\ \text{`<'} \end{array} \right\},$$

$$\mathsf{cands}_r(2,\mathcal{E}) = \left\{ \begin{array}{l} \text{`</I></BR>}\Downarrow\text{<B>'},\ \text{`</I></BR>}\Downarrow\text{<B'},\ \text{`</I></BR>}\Downarrow\text{<'}, \\ \text{`</I></BR>}\Downarrow\text{'},\ \text{`</I></BR>'},\ \text{`</I></BR'},\ \text{`</I></B'}, \\ \text{`</I></'},\ \text{`</I><'},\ \text{`</I>'},\ \text{`</I'},\ \text{`</'},\ \text{`<'} \end{array} \right\}. \quad (2)$$

3.2. Delimiter independence

Given these candidates for each delimiter, a naïve algorithm for learning an LR wrapper is the following:

procedure $\mathsf{learn}_{\mathrm{LR}}$(examples $\mathcal{E}$)—*naïve version*

(1) Generate the candidate sets $\mathsf{cands}_\ell(k,\mathcal{E})$ and $\mathsf{cands}_r(k,\mathcal{E})$ for each delimiter.
(2) Enumerate the cross product of these candidate sets; each element $W = \langle \ell_1, r_1, \ldots, \ell_K, r_K \rangle$ of this cross product is a wrapper. Halt if W is satisfactory, i.e., $\mathsf{exec}_{\mathrm{LR}}(W, P_n) = L_n$ for every $\langle P_n, L_n \rangle \in \mathcal{E}$.

Unfortunately, this algorithm is slow: it runs in time proportional to the product of the number of candidates for each delimiter, and each delimiter can have many candidates.

We can devise a faster algorithm by observing that the $2K$ delimiters are *mutually independent*, in that whether a candidate is valid for a particular delimiter in no way depends on any other delimiters. For example, we can evaluate whether '`</I>`' is satisfactory for r_2 without reasoning about any of the other delimiters.

To see that this independence property holds, recall the $\mathsf{exec}_{\mathrm{LR}}$ procedure. At each point in its execution, $\mathsf{exec}_{\mathrm{LR}}$ is searching its input page P for exactly one of the $2K$ delimiters. If any of these searches fails to identify the correct location in P, then the label output by $\mathsf{exec}_{\mathrm{LR}}$ will be incorrect. But whether these searches return the right answer depends only on the delimiter under consideration and the example pages—not on the other delimiters.

Put another way, once we've committed to a particular candidate for some delimiter, there is no way the candidate can be made invalid, no matter what candidates are selected for the other delimiters. The contrapositive of this assertion also makes intuitive sense: if a candidate is invalid, there is no way to repair it, no matter how carefully we select candidates for other delimiters. Note that this independence property is guaranteed; it is not merely a heuristic that facilitates learning.

The significance of this observation is that we can decompose the original $2K$-variable CSP problem into $2K$ subproblems, and solve each in isolation. In pseudo-code, our improved LR wrapper induction algorithms is as follows:

procedure $\mathsf{learn}_{\mathrm{LR}}$(examples $\mathcal{E}$)—*efficient version*

(1) Generate the candidate sets $\mathsf{cands}_\ell(k,\mathcal{E})$ and $\mathsf{cands}_r(k,\mathcal{E})$ for each delimiter.
(2) For each delimiter, select a valid candidate.

This algorithm is much faster than the original naïve algorithm: it runs in time proportional to the sum (rather than product) of the number of candidates for each delimiter.

3.3. Candidate validity

Of course the second step of this improved algorithm requires that we precisely characterize the conditions under which a delimiter candidate is valid.

Consider first the r_k delimiters. The $\mathsf{exec}_{\mathrm{LR}}$ procedure searches for r_k during the execution of line [*iii*] in Fig. 3. The algorithm has identified the beginning of some instance of the kth attribute in line [*ii*], and is trying to locate the end of the instance. Thus a candidate u for delimiter r_k must satisfy two constraints:

Constraint $\mathcal{C}_r^{\mathrm{A}}$: u must not be a substring of any instance of attribute k in any of the example pages.

Constraint $\mathcal{C}_r^{\mathrm{B}}$: u must be a prefix of the text that occurs immediately following each instance of attribute k in every example page.

If these constraints are violated by a candidate u for delimiter r_k, then every wrapper that includes the assignment $r_k = u$ will fail for at least one of the examples $\mathcal{E}$. If constraint $\mathcal{C}_r^{\mathrm{A}}$ is violated, then attribute k will be too short; if $\mathcal{C}_r^{\mathrm{B}}$ is violated, it will be too long.

We can summarize this discussion as follows. We are interested in the conditions that must hold if some candidate u is to be valid as a value for delimiter r_k, with respect to a given set of examples $\mathcal{E}$. We will refer to these conditions as $\mathsf{valid}_r(u, k, \mathcal{E})$. We have seen that $\mathsf{valid}_r(u, k, \mathcal{E})$ holds if and only if candidate u satisfies constraints $\mathcal{C}_r^{\mathrm{A}}$ and $\mathcal{C}_r^{\mathrm{B}}$ for delimiter r_k with respect to example set $\mathcal{E}$. Returning to the example, if we apply the valid_r test to the candidates generated by cands_r (Eq. (2)), we have:

$$\begin{aligned}
\mathsf{valid}_r(\text{`</B> <I>'}, 1, \mathcal{E}) &= \text{TRUE} & \mathsf{valid}_r(\text{`</I></BR>⇓<B>'}, 2, \mathcal{E}) &= \text{FALSE}\\
\mathsf{valid}_r(\text{`</B> <I'}, 1, \mathcal{E}) &= \text{TRUE} & \mathsf{valid}_r(\text{`</I></BR>⇓<B'}, 2, \mathcal{E}) &= \text{FALSE}\\
\mathsf{valid}_r(\text{`</B> <'}, 1, \mathcal{E}) &= \text{TRUE} & \mathsf{valid}_r(\text{`</I></BR>⇓<'}, 2, \mathcal{E}) &= \text{TRUE}\\
\mathsf{valid}_r(\text{`</B> '}, 1, \mathcal{E}) &= \text{TRUE} & \mathsf{valid}_r(\text{`</I></BR>⇓'}, 2, \mathcal{E}) &= \text{TRUE}\\
\mathsf{valid}_r(\text{`</B>'}, 1, \mathcal{E}) &= \text{TRUE} & \mathsf{valid}_r(\text{`</I></BR>'}, 2, \mathcal{E}) &= \text{TRUE}\\
\mathsf{valid}_r(\text{`</B'}, 1, \mathcal{E}) &= \text{TRUE} & \mathsf{valid}_r(\text{`</I></BR'}, 2, \mathcal{E}) &= \text{TRUE}\\
\mathsf{valid}_r(\text{`</'}, 1, \mathcal{E}) &= \text{TRUE} & \mathsf{valid}_r(\text{`</I></B'}, 2, \mathcal{E}) &= \text{TRUE}\\
\mathsf{valid}_r(\text{`<'}, 1, \mathcal{E}) &= \text{TRUE} & \mathsf{valid}_r(\text{`</I></'}, 2, \mathcal{E}) &= \text{TRUE}\\
& & \mathsf{valid}_r(\text{`</I><'}, 2, \mathcal{E}) &= \text{TRUE}\\
& & \mathsf{valid}_r(\text{`</I>'}, 2, \mathcal{E}) &= \text{TRUE}\\
& & \mathsf{valid}_r(\text{`</I'}, 2, \mathcal{E}) &= \text{TRUE}\\
& & \mathsf{valid}_r(\text{`</'}, 2, \mathcal{E}) &= \text{TRUE}\\
& & \mathsf{valid}_r(\text{`<'}, 2, \mathcal{E}) &= \text{TRUE}
\end{aligned}$$

What are the constraints on the ℓ_k? The $\mathsf{exec}_{\mathrm{LR}}$ procedure searches for delimiter ℓ_k under two different circumstances. First, at line [*ii*] in Fig. 3 the algorithm has just located

the end of the previous attribute and is searching for the beginning of the kth attribute by scanning forward for ℓ_k. Thus ℓ_k must be a proper suffix [4] of the text occurring between each instance of attribute k and the previous attribute.

The second reference to ℓ_k occurs in line $[i]$ of $\mathsf{exec}_{\mathrm{LR}}$; note that this constraint applies only to ℓ_1. At this point $\mathsf{exec}_{\mathrm{LR}}$ is checking to see whether there are additional tuples to be extracted. So we require that ℓ_1 not be a substring of any example's "tail" (i.e., the text occurring after the last tuple).

More precisely, we have shown that a candidate u for delimiter ℓ_k must satisfy two constraints:

Constraint $\mathcal{C}^{\mathrm{A}}_{\ell}$: u must be a proper suffix of the text that occurs immediately before each instance of attribute k in every example page.

Constraint $\mathcal{C}^{\mathrm{B}}_{\ell}$: for ℓ_1, u must not be a substring of any example page's tail.

If these constraints are violated, then every wrapper that includes the assignment $\ell_k = u$ will disagree with the examples $\mathcal{E}$. If constraint $\mathcal{C}^{\mathrm{A}}_{\ell}$ is violated, then at least one of the starting indices $b_{m,k}$ computed by $\mathsf{exec}_{\mathrm{LR}}$ will be incorrect (either less or greater than the correct value, or undefined, depending how u violates constraint $\mathcal{C}^{\mathrm{A}}_{\ell}$). If $\mathcal{C}^{\mathrm{B}}_{\ell}$ is violated, then $\mathsf{exec}_{\mathrm{LR}}$ will attempt to extract too many examples from the page for which u violates $\mathcal{C}^{\mathrm{B}}_{\ell}$.

To summarize, we are interested in the conditions that must hold if some candidate u is to be valid as a value for delimiter ℓ_k, with respect to example set $\mathcal{E}$. We will refer to these conditions as $\mathsf{valid}_{\ell}(u, k, \mathcal{E})$. We have seen that $\mathsf{valid}_{\ell}(u, k, \mathcal{E})$ holds if and only if candidate u satisfies constraints $\mathcal{C}^{\mathrm{A}}_{\ell}$ and $\mathcal{C}^{\mathrm{B}}_{\ell}$ for delimiter ℓ_k with respect to $\mathcal{E}$. Returning to the example, we have:

$$\begin{aligned}
\mathsf{valid}_{\ell}(\text{`</I></BR>⇓<B>'}, 1, \mathcal{E}) &= \textsc{false} & \mathsf{valid}_{\ell}(\text{`</B> <I>'}, 2, \mathcal{E}) &= \textsc{true}\\
\mathsf{valid}_{\ell}(\text{`/I></BR>⇓<B>'}, 1, \mathcal{E}) &= \textsc{false} & \mathsf{valid}_{\ell}(\text{`/B> <I>'}, 2, \mathcal{E}) &= \textsc{true}\\
\mathsf{valid}_{\ell}(\text{`I></BR>⇓<B>'}, 1, \mathcal{E}) &= \textsc{false} & \mathsf{valid}_{\ell}(\text{`B> <I>'}, 2, \mathcal{E}) &= \textsc{true}\\
\mathsf{valid}_{\ell}(\text{`></BR>⇓<B>'}, 1, \mathcal{E}) &= \textsc{false} & \mathsf{valid}_{\ell}(\text{`> <I>'}, 2, \mathcal{E}) &= \textsc{true}\\
\mathsf{valid}_{\ell}(\text{`</BR>⇓<B>'}, 1, \mathcal{E}) &= \textsc{false} & \mathsf{valid}_{\ell}(\text{`<I>'}, 2, \mathcal{E}) &= \textsc{true}\\
\mathsf{valid}_{\ell}(\text{`/BR>⇓<B>'}, 1, \mathcal{E}) &= \textsc{false} & \mathsf{valid}_{\ell}(\text{`<I>'}, 2, \mathcal{E}) &= \textsc{true}\\
\mathsf{valid}_{\ell}(\text{`BR>⇓<B>'}, 1, \mathcal{E}) &= \textsc{false} & \mathsf{valid}_{\ell}(\text{`I>'}, 2, \mathcal{E}) &= \textsc{false}\\
\mathsf{valid}_{\ell}(\text{`R>⇓<B>'}, 1, \mathcal{E}) &= \textsc{false} & \mathsf{valid}_{\ell}(\text{`>'}, 2, \mathcal{E}) &= \textsc{false}\\
\mathsf{valid}_{\ell}(\text{`>⇓<B>'}, 1, \mathcal{E}) &= \textsc{true}\\
\mathsf{valid}_{\ell}(\text{`⇓<B>'}, 1, \mathcal{E}) &= \textsc{true}\\
\mathsf{valid}_{\ell}(\text{`<B>'}, 1, \mathcal{E}) &= \textsc{true}\\
\mathsf{valid}_{\ell}(\text{`B>'}, 1, \mathcal{E}) &= \textsc{true}\\
\mathsf{valid}_{\ell}(\text{`>'}, 1, \mathcal{E}) &= \textsc{false}
\end{aligned}$$

[4] String s is a *proper suffix* of string s' if s is a suffix of s' and moreover s occurs in s' *only* as a suffix. For example, 'cde' is a proper suffix of 'deabcde', while 'de' is not.

3.4. *The* $\mathsf{learn}_{\mathrm{LR}}$ *algorithm*

With this background in place, we are in a position to precisely describe $\mathsf{learn}_{\mathrm{LR}}$, an algorithm for learning LR wrappers. Fig. 4 lists the algorithm, as well as the cands_x, valid_x and related subroutines.

As described earlier, $\mathsf{learn}_{\mathrm{LR}}$ operates by considering each delimiter in turn. For each delimiter, the algorithm enumerates the candidates $\mathsf{cands}_x(k,\mathcal{E})$ $(x \in \{\ell, r\})$, stopping when it identifies a candidate u satisfying $\mathsf{valid}_x(u,k,\mathcal{E})$. After a candidate for each

procedure $\mathsf{learn}_{\mathrm{LR}}$(examples $\mathcal{E}$)
  for each $1 \leqslant k \leqslant K$
    for each $u \in \mathsf{cands}_\ell(k,\mathcal{E})$: if $\mathsf{valid}_\ell(u,k,\mathcal{E})$ then $\ell_k \leftarrow u$ and terminate this loop [*i*]
  for each $1 \leqslant k \leqslant K$
    for each $u \in \mathsf{cands}_r(k,\mathcal{E})$: if $\mathsf{valid}_r(u,k,\mathcal{E})$ then $r_k \leftarrow u$ and terminate this loop [*ii*]
  return LR wrapper $\langle \ell_1, r_1, \ldots, \ell_K, r_K \rangle$

procedure cands_ℓ(index k, examples $\mathcal{E}$)
  return the set of all suffixes of the shortest string in $\mathsf{neighbors}_\ell(k,\mathcal{E})$ [*iii*]

procedure cands_r(index k, examples $\mathcal{E}$)
  return the set of all prefixes of the shortest string in $\mathsf{neighbors}_r(k,\mathcal{E})$

procedure valid_ℓ(candidate u, index k, examples $\mathcal{E}$)
  for each $s \in \mathsf{neighbors}_\ell(k,\mathcal{E})$: if u is not a proper suffix of s then return FALSE $\mathcal{C}_\ell^{\mathrm{A}}$
  if $k = 1$ then for each $s \in \mathsf{tails}(\mathcal{E})$: if u is a substring of s then return FALSE $\mathcal{C}_\ell^{\mathrm{B}}$
  return TRUE

procedure valid_r(candidate u, index k, examples $\mathcal{E}$)
  for each $s \in \mathsf{attribs}(k,\mathcal{E})$: if u is a substring of s then return FALSE $\mathcal{C}_r^{\mathrm{A}}$
  for each $s \in \mathsf{neighbors}_r(k,\mathcal{E})$: if u is not a prefix of s then return FALSE $\mathcal{C}_r^{\mathrm{B}}$
  return TRUE

procedure $\mathsf{attribs}$(index k, examples $\mathcal{E}$)
  return $\cup_{\langle P_n, L_n\rangle \in \mathcal{E}} \{P_n[b_{m,k}, e_{m,k}] \mid \langle \ldots, \langle b_{m,k}, e_{m,k}\rangle, \ldots\rangle \in L_n\}$

procedure $\mathsf{neighbors}_\ell$(index k, examples $\mathcal{E}$)
  if $k = 1$ then return $\mathsf{seps}(K,\mathcal{E}) \cup \mathsf{heads}(\mathcal{E})$ else return $\mathsf{seps}(k-1,\mathcal{E})$

procedure $\mathsf{neighbors}_r$(index k, examples $\mathcal{E}$)
  if $k = K$ then return $\mathsf{seps}(K,\mathcal{E}) \cup \mathsf{tails}(\mathcal{E})$ else return $\mathsf{seps}(k,\mathcal{E})$

procedure heads(examples $\mathcal{E}$)
  return $\{P_n[1, b_{1,1}] \mid \langle P_n, \{\langle\langle b_{1,1}, e_{1,1}\rangle, \ldots\rangle, \ldots\}\rangle \in \mathcal{E}\}$

procedure tails(examples $\mathcal{E}$)
  return $\{P_n[e_{|L_n|,K}, |P_n|] \mid \langle P_n, \{\ldots, \langle \ldots, \langle b_{|L_n|,K}, e_{|L_n|,K}\rangle\rangle\}\rangle \in \mathcal{E}\}$

procedure seps(index k, examples $\mathcal{E}$)
  if $k = K$ then
    return $\cup_{\langle P_n, L_n\rangle \in \mathcal{E}} \{P_n[e_{m,K}, b_{m+1,1}] \mid \langle \ldots, \langle b_{m,K}, e_{m,K}\rangle\rangle \in L_n \wedge m < |L_n|\}$
  else
    return $\cup_{\langle P_n, L_n\rangle \in \mathcal{E}} \{P_n[e_{m,k}, b_{m,k+1}] \mid \langle \ldots, \langle b_{m,k}, e_{m,k}\rangle, \ldots\rangle \in L_n\}$

Fig. 4. The $\mathsf{learn}_{\mathrm{LR}}$ algorithm.

delimiter has been validated, $\mathsf{learn}_{\mathrm{LR}}$ simply assembles the delimiters. No additional verification is necessary, because the constraints enforced by the valid_x subroutines ensure that the learned wrapper is satisfactory.

For example, if we invoke $\mathsf{learn}_{\mathrm{LR}}$ with the single example $\mathcal{E} = \{\langle P_{\mathrm{CC}}, L_{\mathrm{CC}}\rangle\}$, then the learning algorithm outputs the wrapper:

$$\ell_1 = \text{`>⇓<B>'} \qquad r_1 = \text{`</B> <I>'}$$
$$\ell_2 = \text{`</B> <I>'} \qquad r_2 = \text{`</I>
⇓<'}$$

(assuming that the candidates are considered as ordered in Eqs. (1) and (2)).[5]

As shown in Fig. 4, $\mathsf{learn}_{\mathrm{LR}}$ invokes several subroutines. We have already discussed the cands_x and valid_x subroutines. To review: $\mathsf{cands}_r(k, \mathcal{E})$ generates a set of candidates for delimiter r_k; $\mathsf{cands}_\ell(k, \mathcal{E})$ generates the candidates for ℓ_k; $\mathsf{valid}_r(u, k, \mathcal{E})$ verifies whether candidate u is acceptable for delimiter r_k; and $\mathsf{valid}_\ell(u, k, \mathcal{E})$ verifies candidates for ℓ_k.

Subroutines cands_x and valid_x require access to particular fragments of the example pages. For example, to verify constraint $\mathcal{C}^{\mathrm{A}}_\ell$, valid_ℓ must reason about the page fragments that correspond to the attributes of the examples. This access is provided by four additional subroutines:

$\mathsf{attribs}(k, \mathcal{E})$: returns a set containing all values of the kth attribute in each example. If $\mathcal{E} = \{\langle P_{\mathrm{CC}}, L_{\mathrm{CC}}\rangle\}$, then:

$$\mathsf{attribs}(1, \mathcal{E}) = \{\text{`Congo'}, \text{`Egypt'}, \text{`Belize'}, \text{`Spain'}\},$$
$$\mathsf{attribs}(2, \mathcal{E}) = \{\text{`242'}, \text{`20'}, \text{`501'}, \text{`34'}\}.$$

Note that each example $\langle P_n, L_n\rangle \in \mathcal{E}$ provides $|L_n|$ values of each attribute, and thus $|\mathsf{attribs}(k, \mathcal{E})| = \sum_n |L_n|$.

$\mathsf{heads}(\mathcal{E})$: returns the fragments of each page before the first tuple. For example:

$$\mathsf{heads}(\mathcal{E}) = \{\text{`<HTML><TITLE>Some County Codes</TITLE><BODY>⇓<B>'}\}.$$

Note that each example provides one head: $|\mathsf{heads}(\mathcal{E})| = |\mathcal{E}|$.

$\mathsf{tails}(\mathcal{E})$: returns the fragments of each page following the last tuple. For example:

$$\mathsf{tails}(\mathcal{E}) = \{\text{`</I>
⇓</BODY></HTML>'}\}.$$

Note that $|\mathsf{tails}(\mathcal{E})| = |\mathcal{E}|$.

[5] Although these delimiters are different than $\mathsf{ccwrap}_{\mathrm{LR}}$'s, it is straightforward to verify that both wrappers are correct for the example $\langle P_{\mathrm{CC}}, L_{\mathrm{CC}}\rangle$. Which delimiters are better is a subtle matter. For example, if the learned wrapper should be as robust as possible, then perhaps $\mathsf{learn}_{\mathrm{LR}}$ should prefer short candidates. On the other hand, perhaps robustness is not desired, since the system using the wrapper learns later that the site has changed format; in this case, perhaps $\mathsf{learn}_{\mathrm{LR}}$ should prefer long candidates. Though important, these concerns are beyond the scope of this article.

$\mathsf{seps}(k, \mathcal{E})$: returns the fragments of each page between the kth and $((k \bmod K) + 1)$th attributes. For example:

$$\mathsf{seps}(1, \mathcal{E}) = \{\text{`</B> <I>'}, \text{`</B> <I>'}, \text{`</B> <I>'}, \text{`</B> <I>'}\},$$

$$\mathsf{seps}(2, \mathcal{E}) = \{\text{`</I><BR>⇓<B>'}, \text{`</I>
⇓<B>'}, \text{`</I><BR>⇓<B>'}, \text{`</I>
⇓<B>'}\}.$$

Notice that $\mathsf{seps}(K, \mathcal{E})$ does not include $\mathsf{tails}(\mathcal{E})$. Except for this case, each example $\langle P_n, L_n \rangle \in \mathcal{E}$ provides $|L_n|$ separators: $|\mathsf{attribs}(k, \mathcal{E})| = \sum_n |L_n|$ $(k < K)$, and $|\mathsf{attribs}(K, \mathcal{E})| = \sum_n (|L_n| - 1)$.

Note that the fragments returned by these subroutines are readily generated, since each example in $\mathcal{E}$ consists of a page together with its label.

Finally, the sets returned by seps, tails and heads provide relatively low-level access to the relevant substrings of the examples. The $\mathsf{neighbors}_x$ subroutine provides a useful higher level of abstraction. Specifically, $\mathsf{neighbors}_\ell(k, \mathcal{E})$ returns all strings to the left of the kth attributes, whether these strings are in the heads or the bodies of the pages. Similarly, $\mathsf{neighbors}_r(k, \mathcal{E})$ returns all strings to the right of the kth attribute, whether in the tails or bodies. For example:

$$\mathsf{neighbors}_\ell(1, \mathcal{E}) = \left\{ \begin{array}{l} \text{`</I><BR>⇓<B>'}, \text{`</I>
⇓<B>'}, \text{`</I><BR>⇓<B>'}, \\ \text{`<HTML><TITLE>Some County Codes</TITLE>} \\ \text{ <BODY>⇓<B>'} \end{array} \right\},$$

$$\mathsf{neighbors}_\ell(2, \mathcal{E}) = \{\text{`</B> <I>'}, \text{`</B> <I>'}, \text{`</B> <I>'}, \text{`</B> <I>'}\},$$

$$\mathsf{neighbors}_r(1, \mathcal{E}) = \{\text{`</B> <I>'}, \text{`</B> <I>'}, \text{`</B> <I>'}, \text{`</B> <I>'}\},$$

$$\mathsf{neighbors}_r(2, \mathcal{E}) = \left\{ \begin{array}{l} \text{`</I><BR>⇓<B>'}, \text{`</I>
⇓<B>'}, \text{`</I><BR>⇓<B>'}, \\ \text{`</I>
⇓</BODY></HTML>'} \end{array} \right\}.$$

4. Beyond LR

We have introduced the LR wrapper class; in this section we describe five variants. Since we will describe each class in the same way that we described LR, a brief review is in order.

An LR wrapper is specified in terms of a vector $\langle \ell_1, r_1, \ldots, \ell_K, r_K \rangle$ of $2K$ delimiters; the $\mathsf{exec}_{\mathrm{LR}}$ procedure specifies how these delimiters are interpreted. We also described $\mathsf{learn}_{\mathrm{LR}}$, an algorithm for learning LR wrappers. The key to $\mathsf{learn}_{\mathrm{LR}}$ is that it identifies each of the $2K$ delimiters independently. For each delimiter, $\mathsf{learn}_{\mathrm{LR}}$ considers candidates from the set generated by the cands_x procedure. Each such candidate is then tested using the valid_x procedures. The valid_x procedures ensure that the candidates satisfy the appropriate constraints: $\mathcal{C}_\ell^{\mathrm{A}}$ and $\mathcal{C}_\ell^{\mathrm{B}}$ for the ℓ_k, and $\mathcal{C}_r^{\mathrm{A}}$ $\mathcal{C}_r^{\mathrm{B}}$ for the r_k.

In the remainder of this section, we describe five classes that extend LR in various ways. For each such wrapper class $\mathcal{W}$, we:

- motivate and describe the differences between $\mathcal{W}$, LR, and the other classes;
- define class $\mathcal{W}$ in terms of a vector of delimiter strings and a procedure $\mathsf{exec}_{\mathcal{W}}$;

- define the set cands_x of candidates for each delimiter;
- define the constraints that the delimiters must satisfy, and describe a valid_x procedure that verifies whether these constraints are satisfied; and
- define the $\mathsf{learn}_{\mathcal{W}}$ procedure, which selects candidates from the cands_x procedures and tests them using them valid_x procedures.

4.1. The HLRT wrapper class

The LR wrapper class requires that resources format their pages in a very simple manner. Specifically, there must exist delimiters that reliably indicate the left- and right-hand sides of the fragments to be extracted. Of course, not all resources obey such restrictions. For example, Fig. 5 shows a variant of the country/code example. Notice that the page in Fig. 5(b)—which we will refer to as P^*_{CC}—contains additional text rendered in bold. It is straightforward to show that no LR wrapper can handle page P^*_{CC}. The difficulty is that there is no ℓ_1 delimiter that reliably discriminates between bold country text and bold irrelevant text.

However, the $\mathsf{ccwrap}_{\mathrm{HLRT}}$ procedure (Fig. 5(c)) can handle P^*_{CC}. This wrapper operates by searching for two additional delimiters, '<P>' and '<HR>'. The *head* delimiter '<P>' indicates the beginning of the page's body. The *tail* delimiter '<HR>' indicates the end of the page's body. $\mathsf{ccwrap}_{\mathrm{HLRT}}$ is an instance of the HLRT wrapper class, just as $\mathsf{ccwrap}_{\mathrm{LR}}$ exemplifies LR.

More formally, a Head-Left-Right-Tail (HLRT) wrapper is a vector of $2K+2$ strings $\langle h, t, \ell_1, r_1, \ldots, \ell_K, r_K \rangle$. Like LR, HLRT wrapper use $2K$ delimiters to determine the left- and right-hand sides of the fragments to be extracted. In addition, HLRT wrappers include a head delimiter h and a tail delimiter t. For example, $\mathsf{ccwrap}_{\mathrm{HLRT}}$ corresponds to the HLRT wrapper ⟨'<P>', '<HR>', '<B>', '</B>', '<I>', '</I>'⟩.

Just as $\mathsf{exec}_{\mathrm{LR}}$ defines the meaning of an LR wrapper, the $\mathsf{exec}_{\mathrm{HLRT}}$ procedure (Fig. 5(d)) specifies the behavior of an HLRT wrapper. $\mathsf{exec}_{\mathrm{HLRT}}$ operates by first skipping over the head of the page by searching for the first occurrence of the head delimiter h. HLRT wrappers then operate much like LR wrappers, using the ℓ_k and r_k delimiters to extract each attribute in term. However, LR and HLRT wrappers use a different termination criterion: rather than stopping when there are no more occurrences of ℓ_1, HLRT wrappers halt when t occurs before the next occurrence of ℓ_1.

Having defined the HLRT wrapper class, we now describe $\mathsf{learn}_{\mathrm{HLRT}}$, an algorithm for learning HLRT wrappers; see Fig. 6. As with LR, the learning task is to find a set of delimiters that are consistent with a set of examples. All but three of the $2K+2$ delimiters $\langle h, t, \ell_1, r_1, \ldots, \ell_K, r_K \rangle$ can be learned using the original $\mathsf{learn}_{\mathrm{LR}}$ algorithm; the exceptions are h, t and ℓ_1. Thus line [i] of $\mathsf{learn}_{\mathrm{HLRT}}$ in Fig. 6 simply invokes $\mathsf{learn}_{\mathrm{LR}}$.[6]

Recall that all $2K$ LR delimiters are mutually independent. In contrast, h, t and ℓ_1 *interact*, in the sense that whether a particular candidate is valid for one of these three

[6] To simplify the presentation, we describe the $\mathsf{learn}_{\mathrm{HLRT}}$ algorithm as invoking $\mathsf{learn}_{\mathrm{LR}}$, and then discarding and re-learning ℓ_1. Of course HLRT's *raison d'être* is that a consistent LR wrapper consistent might not exist. Specifically, there may be no valid ℓ_1. Therefore, $\mathsf{learn}_{\mathrm{HLRT}}$ must pass a flag to $\mathsf{learn}_{\mathrm{LR}}$ instructing it to ignore ℓ_1. While important, we will not clutter our description of the algorithms with this detail.

(a)

```
<HTML><TITLE>Some Country Codes</TITLE><BODY>
<B>Some Country Codes</B><P>
<B>Congo</B> <I>242</I><BR>
<B>Egypt</B> <I>20</I><BR>
<B>Belize</B> <I>501</I><BR>
<B>Spain</B> <I>34</I><BR>
<HR><B>End</B></BODY></HTML>
```

(b)

procedure $\mathsf{ccwrap}_{\mathrm{HLRT}}$(page P)
  scan to the first occurrence in P of '<P>'
  while the next occurrence of '<B>' in P occurs before the next occurrence of '<HR>'
    for each $\langle \ell_k, r_k \rangle \in \{\langle$'<B>', '</B>'$\rangle, \langle$'<I>', '</I>'$\rangle\}$
      scan in P to next occurrence of ℓ_k; save position as start of kth attribute
      scan in P to next occurrence of r_k; save position as end of kth attribute
  return extracted $\{\ldots, \langle \text{country}, \text{code} \rangle, \ldots\}$ pairs

(c)

procedure $\mathsf{exec}_{\mathrm{HLRT}}$(wrapper $\langle h, t, \ell_1, r_1, \ldots, \ell_K, r_K \rangle$, page P)
  $m \leftarrow 0$
  scan in P to the first occurrence of h [i]
  while the next occurrence of ℓ_1 in P occurs before the next occurrence of t [ii]
    $m \leftarrow m + 1$
    for each $\langle \ell_k, r_k \rangle \in \{\langle \ell_1, r_1 \rangle, \ldots, \langle \ell_K, r_K \rangle\}$
      scan in P to the next occurrence of ℓ_k; save position as $b_{m,k}$ [iii]
      scan in P to the next occurrence of r_k; save position as $e_{m,k}$
  return label $\{\ldots, \langle \langle b_{m,1}, e_{m,1} \rangle, \ldots, \langle b_{m,K}, e_{m,K} \rangle \rangle, \ldots\}$

(d)

Fig. 5. A variant on the country/code resources in Fig. 1: (a) an example response page; (b) P^*_{CC}, the HTML page for (a); (c) the HLRT wrapper $\mathsf{ccwrap}_{\mathrm{HLRT}}$; and (d) the $\mathsf{exec}_{\mathrm{HLRT}}$ procedure.

```
procedure learn_HLRT(examples E)
  ⟨·, r_1, ..., ℓ_K, r_K⟩ ← learn_LR(E)                                   [i]
  for each u_ℓ1 ∈ cands_ℓ(1, E)                                           [ii]
    for each u_h ∈ cands_h(E)                                             [iii]
      for each u_t ∈ cands_t(E)                                           [iv]
        if valid_ℓ1,h,t(u_ℓ1, u_h, u_t, E) then                           [v]
          ℓ_1 ← u_ℓ1, h ← u_h, t ← u_t, and terminate these three loops   [vi]
  return HLRT wrapper ⟨h, t, ℓ_1, r_1, ..., ℓ_K, r_K⟩

procedure cands_h(examples E)
  return the set of all substrings of the shortest string in heads(E)

procedure cands_t(examples E)
  return the set of all substrings of the shortest string in tails(E)

procedure valid_ℓ1,h,t(candidates u_ℓ1, u_h, u_t, examples E)
  for each s ∈ heads(E)
    if u_h is not a substring of s then return FALSE                      C^A_ℓ1,h,t
    if u_ℓ1 is not a proper suffix of scan(s, u_h) then return FALSE      C^B_ℓ1,h,t
    if u_t occurs before u_ℓ1 in scan(s, u_h), then return FALSE          C^C_ℓ1,h,t
  for each s ∈ tails(E)
    if u_t is not a substring of s then return FALSE                      C^D_ℓ1,h,t
    if u_ℓ1 occurs before u_t in s then return FALSE                      C^E_ℓ1,h,t
  for each s ∈ seps(K, E)
    if u_ℓ1 is not a proper suffix of s then return FALSE                 C^F_ℓ1,h,t
    if u_t occurs before u_ℓ1 in s then return FALSE                      C^G_ℓ1,h,t
  return TRUE

procedure scan(strings s_1, s_2)
  return the suffix of s_1 following the first occurrence of s_2
  (e.g., scan('abcdefcdgh', 'cd') = 'cdefcdgh')
```

Fig. 6. The $\mathsf{learn}_{\mathrm{HLRT}}$ algorithm.

delimiters depends on the choice for the other two. For example, is '<B>' valid for ℓ_1? The answer depends on the choice for h and t. If $h =$ '<HTML>', then '<B>' is not valid for ℓ_1, because $\mathsf{exec}_{\mathrm{LR}}$ will not skip the irrelevant bold text '<B>Some Country Codes</B>'. On the other hand, if $h =$ '<P>', then $\ell_1 =$ '<B>' causes no problems. Similarly, ℓ_1 and t interact: $\ell_1 =$ '<B>' is acceptable if $t =$ '<HR>', but unacceptable if $t =$ '</HTML>'.

The ramification of this discussion is that, unlike r_1, ℓ_2, r_2, ..., ℓ_K and r_K, candidates for the three delimiters h, t and ℓ_1 must be considering jointly. As shown in lines [*ii–vi*], $\mathsf{learn}_{\mathrm{HLRT}}$ uses a triply-nested loop to enumerate all combinations of candidates for h, t and ℓ_1. As with $\mathsf{exec}_{\mathrm{LR}}$, candidates for ℓ_1 are generated using the cands_ℓ procedure. Candidates for h are computed with the cands_h procedure. Since h must be a substring of every example's head, $\mathsf{cands}_h(\mathcal{E})$ simply returns the substrings of the shortest head in $\mathcal{E}$. Similarly, the candidates for t are generated by $\mathsf{cands}_t(\mathcal{E})$, which generates the substrings of the shortest tail in $\mathcal{E}$.

To complete the description of $\mathsf{learn}_{\mathrm{HLRT}}$, we must discuss $\mathsf{valid}_{\ell_1,h,t}$, which determines whether a particular combination of candidates u_h, u_t, and u_{ℓ_1} for h, t, and ℓ_1 (respectively) are satisfactory. By examining $\mathsf{exec}_{\mathrm{HLRT}}$, we can see that the constraints are as follows:

Constraint $\mathcal{C}^{\mathrm{A}}_{\ell_1,h,t}$**:** u_h must be a substring of every page's head (otherwise $\mathsf{exec}_{\mathrm{HLRT}}$ line [*i*] will fail).[7]

Constraint $\mathcal{C}^{\mathrm{B}}_{\ell_1,h,t}$**:** u_{ℓ_1} must be a proper suffix of the portion of each page's head after the first occurrence of u_h (otherwise $\mathsf{exec}_{\mathrm{HLRT}}$ line [*iii*] will fail for $m = k = 1$).

Constraint $\mathcal{C}^{\mathrm{C}}_{\ell_1,h,t}$**:** u_t must not occur between the first occurrence of h in any page and the subsequent occurrence of ℓ_1 (otherwise $\mathsf{exec}_{\mathrm{HLRT}}$ will terminate at line [*ii*] without extracting anything).

Constraint $\mathcal{C}^{\mathrm{D}}_{\ell_1,h,t}$**:** u_t must be a substring of every page's tail (otherwise $\mathsf{exec}_{\mathrm{HLRT}}$ line [*ii*] will never terminate).

Constraint $\mathcal{C}^{\mathrm{E}}_{\ell_1,h,t}$**:** u_{ℓ_1} must not occur before t in every page's tail (otherwise $\mathsf{exec}_{\mathrm{HLRT}}$ line [*iii*] will iterate too many times).

Constraint $\mathcal{C}^{\mathrm{F}}_{\ell_1,h,t}$**:** u_{ℓ_1} must be a proper suffix of the text between tuples in every page (otherwise $\mathsf{exec}_{\mathrm{HLRT}}$ line [*iii*] will fail for $k = 1$).

Constraint $\mathcal{C}^{\mathrm{G}}_{\ell_1,h,t}$**:** u_t must not occur before u_{ℓ_1} in the text between tuples in any page (otherwise $\mathsf{exec}_{\mathrm{HLRT}}$ line [*ii*] will terminate early).

Before proceeding, let us illustrate $\mathsf{learn}_{\mathrm{HLRT}}$ with the modified country/code example in Fig. 5. If $\mathcal{E}$ contains just the single example P^*_{CC} and its label, then we have that:

$$\mathsf{cands}_h(\mathcal{E}) = \begin{cases} \text{the } 87 \cdot 86/2 = 3741 \text{ substrings of the 87 character string} \\ \text{`}\texttt{<HTML><TITLE>Some Country Codes</TITLE><BODY>}\Downarrow \\ \texttt{<B>Some Country Codes</B><P>}\Downarrow\texttt{<B>}\text{'} \end{cases}$$

$$\mathsf{cands}_t(\mathcal{E}) = \begin{cases} \text{the } 41 \cdot 40/2 = 820 \text{ substrings of the 41 character string} \\ \text{`}\texttt{</I>
}\Downarrow\texttt{<HR><B>End</B></BODY></HTML>}\text{'} \end{cases}$$

($\mathsf{cands}_\ell(1, \mathcal{E})$ returns 13 candidates, as listed in Eq. (1)). $\mathsf{learn}_{\mathrm{HLRT}}$ enumerates the $3741 \cdot 820 \cdot 13 = 39,879,060$ ways to choose one candidate from each set, stopping when one such combination satisfies $\mathsf{valid}_{\ell_1,h,t}$. Fortunately, many of the combinations are valid, so $\mathsf{learn}_{\mathrm{HLRT}}$ soon terminates, returning a wrapper such as $\mathsf{ccwrap}_{\mathrm{HLRT}}$. (Without specifying the order in which candidates are considered, we cannot say which wrapper is returned.)

[7] This notation is somewhat imposing; the idea is simply to unambiguously refer to the various constraints. Thus, $\mathcal{C}^{\mathrm{A}}_{\ell_1,h,t}$ refers to "part A" of the constraints that apply to delimiters ℓ_1, h and t, just as $\mathcal{C}^{\mathrm{A}}_{\ell}$ refers to "part A" of the constraints that apply to the ℓ_k delimiters.

4.2. The OCLR wrapper class

The LR wrapper class is quite restrictive, and HLRT is one of many ways to extend the LR class; the Open-Close-Left-Right (OCLR) class is an alternative.

Instead of using head and tail delimiters to indicate the body of the page, the OCLR class uses *open* and *close* delimiters to indicate the beginning and end of each tuple in the page. For example, Fig. 7 lists an example resource that is well-suited to OCLR, as well as $\mathsf{ccwrap}_{\mathrm{OCLR}}$, an example of the OCLR class.

Wrapper $\mathsf{ccwrap}_{\mathrm{OCLR}}$ operates by using the delimiters delimiters '<LI>···
' to find tuples within each page, just as '<B>···</B>' indicates countries and '<I>···</I>' indicates codes within a single tuple. '<LI>' is the *open* delimiter, and '
' is the *close* delimiter.

```
<HTML><TITLE>Some Country Codes</TITLE><BODY>
<B>Some Country Codes</B><P><UL>
<LI><B>Congo</B> <I>242</I><BR>
<LI><B>Egypt</B> <I>20</I><BR>
<LI><B>Belize</B> <I>501</I><BR>
<LI><B>Spain</B> <I>34</I><BR>
</UL><HR><B>End</B></BODY></HTML>
```

(a)

procedure $\mathsf{ccwrap}_{\mathrm{OCLR}}$(page P)
  while there are more occurrences of '<LI>' in P
    scan to the next occurrence of '<LI>' in P
    for each $\langle \ell_k, r_k \rangle \in \{\langle$'<B>', '</B>'$\rangle, \langle$' <I>', '</I>'$\rangle\}$
      scan in P to next occurrence of ℓ_k; save position as start of kth attribute
      scan in P to next occurrence of r_k; save position as end of kth attribute
    scan to the next occurrence of '
' in P
  return extracted $\{\ldots, \langle \text{country, code} \rangle, \ldots\}$ pairs

(b)

procedure $\mathsf{exec}_{\mathrm{OCLR}}$(wrapper $\langle o, c, \ell_1, r_1, \ldots, \ell_K, r_K \rangle$, page P)
  $m \leftarrow 0$
  while there are more occurrences of o in P [i]
    $m \leftarrow m + 1$
    scan to the next occurrence of o in P [ii]
    for each $\langle \ell_k, r_k \rangle \in \{\langle \ell_1, r_1 \rangle, \ldots, \langle \ell_K, r_K \rangle\}$
      scan in P to the next occurrence of ℓ_k; save position as $b_{m,k}$ [iii]
      scan in P to the next occurrence of r_k; save position as $e_{m,k}$
    scan to the next occurrence of c in P [iv]
  return label $\{\ldots, \langle \langle b_{m,1}, e_{m,1} \rangle, \ldots, \langle b_{m,K}, e_{m,K} \rangle \rangle, \ldots\}$

(c)

Fig. 7. The OCLR wrapper class: (a) a second variant on the Fig. 1's country/code resources; (b) the OCLR wrapper $\mathsf{ccwrap}_{\mathrm{OCLR}}$; and (c) $\mathsf{exec}_{\mathrm{OCLR}}$, a generalization of $\mathsf{ccwrap}_{\mathrm{OCLR}}$.

Fig. 7 also lists $\mathsf{exec}_{\mathrm{OCLR}}$, a generalization of $\mathsf{ccwrap}_{\mathrm{OCLR}}$. As indicated, an OCLR wrapper is a vector of $2K+2$ strings $\langle o, c, \ell_1, r_1, \ldots, \ell_K, r_K\rangle$, where o is the open delimiter and c is the close delimiter. For example, $\mathsf{ccwrap}_{\mathrm{OCLR}}$ uses $o =$ '<LI>' and $c =$ '
'.

Given this specification of the OCLR wrapper class, we are in a position to describe $\mathsf{learn}_{\mathrm{OCLR}}$, an algorithm for learning OCLR; see Fig. 8. As with the classes discussed so far, $\mathsf{learn}_{\mathrm{OCLR}}$ operates by generating and evaluating a set of candidates for each of the $2K+2$ delimiters $\langle o, c, \ell_1, r_1, \ldots, \ell_K, r_K\rangle$.

Specifically, like LR and HLRT, the $2K-1$ delimiters r_1, ℓ_2, r_2, …, ℓ_K and r_K are independent of each other and o, c and ℓ_1. Thus line [i] of $\mathsf{learn}_{\mathrm{OCLR}}$ simply invokes $\mathsf{learn}_{\mathrm{LR}}$ to determine r_1, ℓ_2, r_2, …, ℓ_K and r_K.

Recall that for HLRT, delimiters h, t and ℓ_1 interact. Similarly, for OCLR, delimiters o, c and ℓ_1 interact. For example, $\ell_1 =$ '<B>' is valid if $o =$ '<LI>' and $c =$ '
', but not if $o =$ '<' and $c =$ '>'. Thus just as $\mathsf{learn}_{\mathrm{HLRT}}$ uses triply-nested loops to enumerate the candidates for h, t and ℓ_1, $\mathsf{learn}_{\mathrm{OCLR}}$ uses a similar loop structure for o, c, and ℓ_1.

What are the candidates for o and c? Notice that these two delimiters must both occur between each tuple in each example. The invocation $\mathsf{seps}(K, \mathcal{E})$ (see Fig. 4) returns the set strings between the tuples. The candidates for both o and c are denoted $\mathsf{cands}_{o,c}(\mathcal{E})$ in Fig. 8; $\mathsf{cands}_{o,c}(\mathcal{E})$ enumerates the substrings of the shortest member of $\mathsf{seps}(K, \mathcal{E})$ (i.e., the shortest inter-tuple separator).

```
procedure learn_OCLR(examples E)
  ⟨·, r_1, ..., ℓ_K, r_K⟩ ← learn_LR(E)                                   [i]
  for each u_ℓ1 ∈ cands_ℓ(1, E)                                           [ii]
    for each u_o ∈ cands_o,c(E)                                           [iii]
      for each u_c ∈ cands_o,c(E)                                         [iv]
        if valid_ℓ1,o,c(u_ℓ1, u_o, u_c, E) then                           [v]
          ℓ_1 ←u_ℓ1, o ←u_o, c ←u_c, and terminate these three loops      [vi]
  return OCLR wrapper ⟨o, c, ℓ_1, r_1, ..., ℓ_K, r_K⟩

procedure cands_o,c(examples E)
  return the set of all substrings of the shortest string in seps(K, E)

procedure valid_ℓ1,o,c(candidates u_ℓ1, u_o, u_c, examples E)
  for each s ∈ heads(E)
    if u_o is not a substring of s then return FALSE                      C^A_ℓ1,o,c
    if u_ℓ1 is not a proper suffix of scan(s, u_o) then return FALSE      C^B_ℓ1,o,c
  for each s ∈ tails(E)
    if u_c is not a substring of s then return FALSE                      C^C_ℓ1,o,c
    if u_o occurs after u_c in s then return FALSE                        C^D_ℓ1,o,c
  for each s ∈ seps(K, E)
    if u_o is not a substring of s then return FALSE                      C^E_ℓ1,o,c
    if u_c is not a substring of s then return FALSE                      C^F_ℓ1,o,c
    if u_ℓ1 is not a proper suffix of scan(scan(s, u_c), u_o) then return FALSE   C^G_ℓ1,o,c
  return TRUE
```

Fig. 8. The $\mathsf{learn}_{\mathrm{OCLR}}$ algorithm.

Finally, the $\mathsf{valid}_{\ell_1,o,c}$ procedure must evaluate a triplet of candidates u_{ℓ_1}, u_o and u_c, for ℓ_1, o and c (respectively). As shown in Fig. 8, $\mathsf{valid}_{\ell_1,o,c}$ implements the following constraints, which derive from an examination of $\mathsf{exec}_{\mathrm{OCLR}}$:

Constraint $\mathcal{C}^{\mathrm{A}}_{\ell_1,o,c}$**:** u_o must be a substring of every page's head (otherwise $\mathsf{exec}_{\mathrm{OCLR}}$ line [*i*] will fail).

Constraint $\mathcal{C}^{\mathrm{B}}_{\ell_1,o,c}$**:** u_{ℓ_1} must be a proper suffix of the portion of each page's head after the first occurrence of o (otherwise $\mathsf{exec}_{\mathrm{OCLR}}$ line [*iii*] will fail for $m = k = 1$).

Constraint $\mathcal{C}^{\mathrm{C}}_{\ell_1,o,c}$**:** u_c must be a substring of every page's tail (otherwise $\mathsf{exec}_{\mathrm{OCLR}}$ line [*iv*] will fail).

Constraint $\mathcal{C}^{\mathrm{D}}_{\ell_1,o,c}$**:** u_o must not occur after u_c in any page's tail (otherwise $\mathsf{exec}_{\mathrm{OCLR}}$ line [*i*] will extract too many tuples).

Constraint $\mathcal{C}^{\mathrm{E}}_{\ell_1,o,c}$**:** u_o must be a substring of the text between tuples in every page (otherwise $\mathsf{exec}_{\mathrm{OCLR}}$ line [*ii*] will fail).

Constraint $\mathcal{C}^{\mathrm{F}}_{\ell_1,o,c}$**:** u_c must be a substring of the text between tuples in every page (otherwise $\mathsf{exec}_{\mathrm{OCLR}}$ line [*iv*] will fail).

Constraint $\mathcal{C}^{\mathrm{G}}_{\ell_1,o,c}$**:** u_{ℓ_1} must be a proper suffix of the text that occurs after o, in the text that occurs after c, in the text between tuples, in every page (otherwise $\mathsf{exec}_{\mathrm{OCLR}}$ line [*iii*] will fail).

As shown in Fig. 8, the $\mathsf{valid}_{\ell_1,o,c}$ procedure implements these seven constraints.

4.3. The HOCLRT wrapper class

We have introduced OCLR and HLRT, two variants on the simple LR wrapper class. As shown in Fig. 9, the Head-Open-Close-Left-Right-Tail (HOCLRT) class combines the functionality of HLRT and OCLR. An HOCLRT wrapper is a vector $\langle h, t, o, c, \ell_1, r_1, \ldots, \ell_K, r_K \rangle$ of $2K + 4$ delimiters. For example, $\mathsf{ccwrap}_{\mathrm{HOCLRT}}$ in Fig. 9(a) corresponds to the wrapper ⟨'<P>', '<HR>', '<LI>', '
', '<B>', '</B>', '<I>', '</I>'⟩.

The $\mathsf{exec}_{\mathrm{HOCLRT}}$ procedure (Fig. 9(b)) is a generalization of $\mathsf{ccwrap}_{\mathrm{HOCLRT}}$. The head delimiter h is used to determine the end of the page's head; the opening and closing delimiters o and c demarcate individual tuples, and the tail delimiter t indicates that the page contains no more tuples.

Fig. 10 lists $\mathsf{learn}_{\mathrm{HOCLRT}}$, an algorithm for learning HOCLRT wrappers. Recall that for LR, all of the delimiters are mutually independent; while for HLRT, ℓ_1, h and t must be learned jointly, and for OCLR, ℓ_1, o and c must be learned jointly. As expected, the five HOCLRT delimiters ℓ_1, h, t, o and c must be learned jointly. Like $\mathsf{learn}_{\mathrm{HLRT}}$ and $\mathsf{learn}_{\mathrm{OCLR}}$, the $\mathsf{learn}_{\mathrm{HOCLRT}}$ algorithm first invokes $\mathsf{learn}_{\mathrm{LR}}$ to learn r_1, ℓ_2, r_2, …, ℓ_K and r_K. $\mathsf{learn}_{\mathrm{HOCLRT}}$ then uses a quintuply-nested loop structure to enumerate all combinations

```
procedure ccwrap_HOCLRT(page P)
  scan to the first occurrence in P of '<P>'
  while the next occurrence of '<LI>' in P occurs before the next occurrence of '<HR>'
    scan to the next occurrence of '<LI>' in P
    for each ⟨ℓ_k, r_k⟩ ∈ {⟨'<B>', '</B>'⟩, ⟨'<I>', '</I>'⟩}
      scan in P to next occurrence of ℓ_k; save position as start of kth attribute
      scan in P to next occurrence of r_k; save position as end of kth attribute
    scan to the next occurrence of '<BR>' in P
  return extracted {..., ⟨country, code⟩, ...} pairs
```

(a)

```
procedure exec_HOCLRT(wrapper ⟨h, t, o, c, ℓ_1, r_1, ..., ℓ_K, r_K⟩, page P)
  scan to the first occurrence in P of h                                   [i]
  m ← 0
  while the next occurrence of o in P occurs before the next occurrence of t   [ii]
    m ← m + 1
    scan to the next occurrence of o in P                                  [iii]
    for each ⟨ℓ_k, r_k⟩ ∈ {⟨ℓ_1, r_1⟩, ..., ⟨ℓ_K, r_K⟩}
      scan in P to the next occurrence of ℓ_k; save position as b_{m,k}    [iv]
      scan in P to the next occurrence of r_k; save position as e_{m,k}
    scan to the next occurrence of c in P                                  [v]
  return label {..., ⟨⟨b_{m,1}, e_{m,1}⟩, ..., ⟨b_{m,K}, e_{m,K}⟩⟩, ...}
```

(b)

Fig. 9. The HOCLRT wrapper class: (a) $\mathsf{ccwrap}_{\mathrm{HOCLRT}}$, an HOCLRT wrapper for Fig. 7's variant of the country/code resources; and (b) the $\mathsf{exec}_{\mathrm{HOCLRT}}$ procedure, a generalization of $\mathsf{ccwrap}_{\mathrm{HOCLRT}}$.

of candidates for ℓ_1, h, t, o and c. As before, the candidates for ℓ_1 are provided by $\mathsf{cands}_{\ell}(1, \mathcal{E})$; for h by $\mathsf{cands}_{h}(\mathcal{E})$; for t by $\mathsf{cands}_{t}(\mathcal{E})$; and for o and c by $\mathsf{cands}_{o,c}(\mathcal{E})$.

Only the $\mathsf{valid}_{\ell_1,h,t,o,c}$ subroutine in Fig. 10 remains to be explained. An examination of $\mathsf{exec}_{\mathrm{HOCLRT}}$ reveals that the following constraints must be satisfied for candidates u_{ℓ_1}, u_o, u_c, u_h and u_t to be consistent with a given set of examples pages:

Constraint $\mathcal{C}^{\mathrm{A}}_{\ell_1,h,t,o,c}$: u_h must occur in every page's head (otherwise $\mathsf{exec}_{\mathrm{HOCLRT}}$ line [i] will fail).

Constraint $\mathcal{C}^{\mathrm{B}}_{\ell_1,h,t,o,c}$: u_o must occur in every page's head (otherwise $\mathsf{exec}_{\mathrm{HOCLRT}}$ line [ii] will fail).

Constraint $\mathcal{C}^{\mathrm{C}}_{\ell_1,h,t,o,c}$: u_{ℓ_1} must be a proper suffix of the text occurring after u_o, in the text occurring after u_h, in every head (otherwise $\mathsf{exec}_{\mathrm{HOCLRT}}$ line [iv] will fail for $m = k = 1$).

Constraint $\mathcal{C}^{\mathrm{D}}_{\ell_1,h,t,o,c}$: u_t must not occur before u_{ℓ_1}, in the text occurring after u_o, in the text occurring after u_h, in every head (otherwise $\mathsf{exec}_{\mathrm{HOCLRT}}$ line [ii] will halt without extracting any tuples).

```
procedure learn_HOCLRT(examples E)
  ⟨·, r_1, ..., ℓ_K, r_K⟩ ← learn_LR(E)
  for each u_ℓ1 ∈ cands_ℓ(1, E)
    for each u_o ∈ cands_o,c(E)
      for each u_c ∈ cands_o,c(E)
        for each u_h ∈ cands_h(E)
          for each u_t ∈ cands_t(E)
            if valid_ℓ1,h,t,o,c(u_ℓ1, u_h, u_t, u_o, u_c, E) then
              ℓ_1 ←u_ℓ1, o ←u_o, c ←u_c, h ←u_h, t ←u_t and terminate these five loops
  return HOCLRT wrapper ⟨h, t, o, c, ℓ_1, r_1, ..., ℓ_K, r_K⟩

procedure valid_ℓ1,h,t,o,c(candidates u_ℓ1, u_h, u_t, u_o, u_c, examples E)
  for each s ∈ heads(E)
    if u_h is not a substring of s then return FALSE                                   C^A_ℓ1,h,t,o,c
    if u_o is not a substring of s then return FALSE                                   C^B_ℓ1,h,t,o,c
    if u_ℓ1 is not a proper suffix of scan(scan(s, u_h), u_o) then return FALSE         C^C_ℓ1,h,t,o,c
    if u_t occurs before u_ℓ1 in scan(scan(s, u_h), u_o) then return FALSE              C^D_ℓ1,h,t,o,c
  for each s ∈ tails(E)
    if u_t is not a substring of s then return FALSE                                   C^E_ℓ1,h,t,o,c
    if u_c is not a substring of s then return FALSE                                   C^F_ℓ1,h,t,o,c
    if u_o occurs before u_t in scan(s, u_c) then return FALSE                         C^G_ℓ1,h,t,o,c
  for each s ∈ seps(K, E)
    if u_o is not a substring of s then return FALSE                                   C^H_ℓ1,h,t,o,c
    if u_c is not a substring of s then return FALSE                                   C^I_ℓ1,h,t,o,c
    if u_ℓ1 is not a proper suffix of scan(scan(s, u_c), u_o) then return FALSE         C^J_ℓ1,h,t,o,c
    if u_t occurs before u_ℓ1 in scan(scan(s, u_c), u_o) then return FALSE              C^K_ℓ1,h,t,o,c
  return TRUE
```

Fig. 10. The $\text{learn}_{\text{HOCLRT}}$ algorithm.

Constraint $\mathcal{C}^{\text{E}}_{\ell_1,h,t,o,c}$: u_t must occur in every page's tail (otherwise $\text{exec}_{\text{HOCLRT}}$ line [*ii*] will fail).

Constraint $\mathcal{C}^{\text{F}}_{\ell_1,h,t,o,c}$: u_c must occur in every page's tail (otherwise $\text{exec}_{\text{HOCLRT}}$ line [*v*] will fail).

Constraint $\mathcal{C}^{\text{G}}_{\ell_1,h,t,o,c}$: u_t must occur before u_o in the text occurring after u_c, in every page's tail (otherwise $\text{exec}_{\text{HOCLRT}}$ line [*ii*] will extract too many tuples).

Constraint $\mathcal{C}^{\text{H}}_{\ell_1,h,t,o,c}$: u_o must be a substring of the text between each tuple in each page (otherwise $\text{exec}_{\text{HOCLRT}}$ line [*ii*] will fail).

Constraint $\mathcal{C}^{\text{I}}_{\ell_1,h,t,o,c}$: u_c must be a substring of the text between each tuple in each page (otherwise $\text{exec}_{\text{HOCLRT}}$ line [*v*] will fail).

Constraint $\mathcal{C}^{\text{J}}_{\ell_1,h,t,o,c}$: u_{ℓ_1} must be a proper suffix of the text occurring after u_o, in the text occurring after u_c, in the text between each tuple in each page (otherwise $\text{exec}_{\text{HOCLRT}}$ line [*iv*] will fail for $k = 1$).

Constraint $\mathcal{C}^{\mathrm{K}}_{\ell_1,h,t,o,c}$: u_t must occur after u_{ℓ_1} in the text occurring after u_o, in the text occurring after u_c, in the text between each tuple in each page (otherwise $\mathsf{exec}_{\mathrm{HOCLRT}}$ line [*iv*] will fail for $k = 1$).

The $\mathsf{valid}_{\ell_1,h,t,o,c}$ procedure in Fig. 10 implements these eleven constraints.

4.4. Nested documents

The wrapper classes introduced so far (LR, OCLR, HLRT and HOCLRT) assume that documents are structured in a relational or *tabular* manner. Of course, many documents are not tabular. The remaining two wrapper classes are concerned with *hierarchically nested* (or just "nested") documents, which represent one way to relax the tabularity assumption. In this section, we describe documents with nested structure; in Sections 4.5 and 4.6 we introduce wrapper classes for extracting such structure.

While a rectangular table is the prototypical example of a document exhibiting tabular structure, nested documents have a tree-like structure. Consider the document in Fig. 11(a) from a telephone directory in which a person can have any number of addresses, and an address can have any number of telephones. Unlike the previous examples, this document does not contain HTML tags, and therefore demonstrates that our wrapper induction techniques are not limited to HTML text.

In a document with nested structure, values for K attributes are presented, with the information organized hierarchically rather than in a table. Attributes residing "below" attribute k represent additional information or details about the object represented by attributes 1 to k. For each attribute, there may be any number (possibly zero) of values for the given attribute. The only constraint is that values can be provided for attribute k only if values are also provided for attributes 1 to $k-1$.

In the telephone directory example, there are $K = 3$ attributes: people's names, addresses, and telephone numbers. The constraint that each attribute can have any number of values corresponds to the fact that a person can have any number of addresses, and an address can have any number of telephone numbers. The constraint that attribute k can have a value only if attributes 1 to $k-1$ have values corresponds to the fact that there can be no "floating" telephone number without an associated address, or a "floating" address without an associated person.

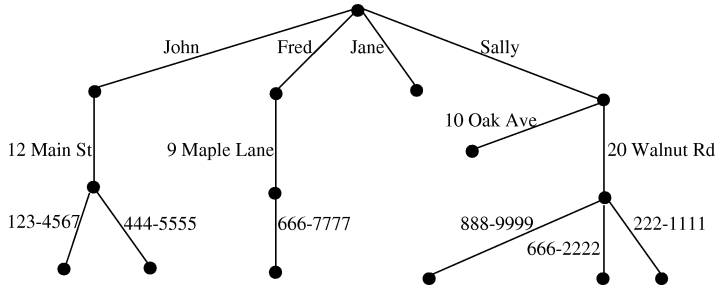
The information-extraction model introduced in Section 2 used a natural definition of a tabular document's content; namely, the the substrings of the document corresponding to the attribute values for each tuple. For nested documents, we extend this idea in a straightforward manner. The content of a nested document is a tree of depth of most K. Edges encode the attribute values, while nodes group related attribute values. For example, the content of the telephone directory document is shown in Fig. 11(b).

More formally, a nested document's label is a tree, encoded according to following recursive definition:

$$\mathsf{label} \Rightarrow \{\ldots, \mathsf{node}, \ldots\}$$
$$\mathsf{node} \Rightarrow [\langle b, e\rangle; \mathsf{label}]\,.$$

```
name:  John
     address:  12 Main St
         phone:  123-4567
         phone:  444-5555
name:  Fred
     address:  9 Maple Lane
         phone:  666-7777
name:  Jane
name:  Sally
     address:  10 Oak Ave
     address:  20 Walnut Rd
         phone:  888-9999
         phone:  666-2222
         phone:  222-1111
```

(a)



(b)

$$\left\{ \begin{array}{l} \left[\langle 7,10\rangle ; \left\{ \left[\langle 22,31\rangle ; \left\{ \begin{array}{l} [\langle 42,49\rangle ; \{\}] , \\ [\langle 60,67\rangle ; \{\}] \end{array} \right\} \right] \right\} \right] , \\ [\langle 75,78\rangle ; \{[\langle 90,101\rangle ; \{[\langle 112,119\rangle ; \{\}]\}]\}] , \\ [\langle 127,130\rangle ; \{\}] , \\ \left[\langle 138,142\rangle ; \left\{ \begin{array}{l} [\langle 154,163\rangle ; \{\}] , \\ \left[\langle 175,186\rangle ; \left\{ \begin{array}{l} [\langle 197,204\rangle ; \{\}] , \\ [\langle 215,222\rangle ; \{\}] , \\ [\langle 233,240\rangle ; \{\}] \end{array} \right\} \right] \end{array} \right\} \right] \end{array} \right\}$$

(c)

Fig. 11. An example of a document with nested structure: (a) the document (note that this document does not contain HTML tags); (b) a tree representing the document's content; and (c) the document's label (see text for details).

That is, a label structure is a set of zero or more nodes. A node structure consists of an interval $\langle b, e\rangle$ and a LABEL structure, where $\langle b, e\rangle$ are the indices of the text that labels the edge to the node's parent, and label represents its children.

The label for the telephone directory example is shown in Fig. 11(c), assuming that nested entries are indented with a single tab character per level, and each line ends with a single new-line character (recall that this document is plain text and does not contain HTML tags). For instance, the pair $\langle 7, 10\rangle$ in the first row indicates that characters 7–10 of the document are the first name in the directory, i.e., '`John`'. Similarly, characters 138–142 are the last name, '`Sally`'. `Sally` has two addresses: the first, from 154–163, has no telephone; the second, from 175–186, has three telephone numbers (from 197–204, 215–222 and 233–240).

Note that a tabular structure is a special case of a nested structure, in which the root has one child for each tuple, all other interior nodes have exactly one child, and every leaf has depth K. We will overload the term "label" to mean either a tabular or a nested label; the interpretation will be clear from context. Note that this "reduction" from tabular to nested wrapper *outputs* does not imply a reduction between wrapper *classes*, as we show in Section 5. For example, in Section 4.5 we define the N-LR wrapper class which extracts nested structures, but there exists documents for which an LR wrapper exists that extracts the correct tabular structure, but for which there does not exist an N-LR wrapper that extracts the nested structure to which this tabular structure is equivalent.

4.5. The N-LR wrapper class

N-LR is a simple wrapper class for extracting nested structure. Like the other classes, associated with each delimiter is a left-hand delimiter ℓ_k and a right-hand delimiter r_k. In the tabular wrappers, after extracting the value of the kth attribute for some particular tuple, then the wrapper extracts the $(k+1)$st attribute value (or the first, in the case of $k = K$). Nested-structure wrappers generalize this procedure: after extracting the kth attribute, the next extracted value could belong to attributes $k+1, k, k-1, k-2, \ldots, 2$ or 1. The N-LR wrapper class uses the relative position of the ℓ_k delimiters to indicate how the page should be interpreted: the next value to be extracted is indicated by which ℓ_k delimiters occurs next. For example, if an N-LR wrapper has extracted up to position 1350, and there is a name starting at position 1450 and an address starting at position 1400, then the wrapper will extract the address next, since $1400 < 1450$.

More precisely, an N-LR wrapper is a vector of $2K$ strings $\langle \ell_1, r_1, \ldots, \ell_K, r_K\rangle$. The $\mathsf{exec}_{\text{N-LR}}$ procedure (Fig. 12(a)) uses these delimiters as just described. Line [i] is the key to $\mathsf{exec}_{\text{N-LR}}$. The algorithm determines which attribute (among the $k+1$ possibilities) k' occurs next. The termination condition (line [ii]) is satisfied whenever none of the $\ell_{k'}$ occur—i.e., when the wrapper has reached the end of the page. Note that if line [i] of $\mathsf{exec}_{\text{N-LR}}$ were replaced by "$k \leftarrow (k \bmod K) + 1$", then $\mathsf{exec}_{\text{N-LR}}$ would be identical to $\mathsf{exec}_{\text{LR}}$.

For example, the N-LR wrapper

$$\langle \text{`}\texttt{name: }\text{'}, \text{`}\Downarrow\text{'}, \text{`}\texttt{address: }\text{'}, \text{`}\Downarrow\text{'}, \text{`}\texttt{phone: }\text{'}, \text{`}\Downarrow\text{'}\rangle$$

extracts the content from the telephone directory document. As a second example, the N-LR wrapper $\langle$'`<B>`', '`</B>`', '`<I>`', '`</I>`'$\rangle$ extracts the desired structure from the country/code page P_{CC} shown in Fig. 1. Of course this resource is tabular rather than nested and thus does not expose the full power of N-LR. As mentioned above, tabular structure is

```
procedure exec_N-LR(wrapper ⟨ℓ_1, r_1, ..., ℓ_K, r_K⟩, page P)
  k ← 0
  loop
    k ← the index k' ∈ [1, min(1 + k, K)] such that ℓ_k' occurs first in P      [i]
    exit loop if no such k exists                                               [ii]
    b ← the position of ℓ_k in P
    e ← the position of r_k following ℓ_k in P
    save ⟨b, e⟩ as indices of next value of kth attribute
  return nested label {..., [⟨b, e⟩; {...}], ...}
```

(a)

```
procedure learn_N-LR(examples E)
  for each u_r1 ∈ cands_r^N(1, E)
    ...
      for each u_rK ∈ cands_r^N(K, E)
        for each u_ℓ1 ∈ cands_ℓ^N(1, E)
          ...
            for each u_ℓK ∈ cands_ℓ^N(K, E)
              W ← ⟨u_ℓ1, u_r1, ..., u_ℓK, u_rK⟩
              if ∀_⟨P_n, L_n⟩∈E exec_N-LR(W, P_n) = L_n then return W            [i]
```

(b)

Fig. 12. The N-LR wrapper class: (a) the $\mathsf{exec}_{\text{N-LR}}$ procedure defines how an N-LR wrapper is executed; and (b) the $\mathsf{learn}_{\text{N-LR}}$ algorithm learns N-LR wrappers.

a special case of nested structure, and so the invocation $\mathsf{exec}_{\text{N-LR}}(\langle$'<B>', '</B>', '<I>', '</I>'$\rangle, P_{\text{CC}})$ returns a degenerate tree that is equivalent to label L_{CC}.

As with the other classes, we are interested in automatically learning N-LR wrappers. Recall that LR's delimiters are mutually independent, while some of HLRT's, OCLR's and HOCLRT's delimiters interact. The situation for N-LR is even worse: *all* $2K$ N-LR delimiters interact. To see this, recall that after extracting a value for attribute k, line $[i]$ of $\mathsf{exec}_{\text{N-LR}}$ is looking for $k+1$ delimiters ($\ell_1, \ell_2, \ldots, \ell_k$ and ℓ_{k+1}). In short, the choice of each ℓ_k depends on the choice for the other $k+1$ delimiters. The result of this analysis is that, unlike LR, HLRT, OCLR and HOCLRT, the $2K$-variable CSP for learning N-LR is not decomposed into $2K$ subproblems, and so $\mathsf{learn}_{\text{N-LR}}$ uses an exponential-time generate-and-test algorithm; see Fig. 12(b). [8]

The candidates for each delimiter are generated by the $\mathsf{cands}_x^{\text{N}}$ procedures, trivial generalizations of the cands_x procedures that handle nested rather than tabular labels.

[8] Actually, the r_k delimiters are independent, so the $2K$-variable CSP can be decomposed into $K+1$ subproblems: finding the r_k individually, and jointly finding the ℓ_k. We ignore this more sophisticated approach, though see Section 4.7 for a discussion of related issues.

```
procedure exec_N-HLRT(wrapper ⟨h, t, ℓ_1, r_1, ..., ℓ_K, r_K⟩, page P)
  k ← 0
  scan in P to the first occurrence of h
  loop
    k ← the index k' ∈ [1, min(1 + k, K)] such that ℓ_k' occurs first in P      [i]
    exit loop if t occurs before ℓ_k in P                                       [ii]
    b ← the position of ℓ_k in P
    e ← the position of r_k following ℓ_k in P
    save ⟨b, e⟩ as indices of next value of kth attribute
  return nested label {..., ⟨⟨b, e⟩, {...}⟩, ...}
```

(a)

```
procedure learn_N-HLRT(examples E)
  for each u_r1 ∈ cands_r^N(1, E)
    ...
      for each u_rK ∈ cands_r^N(K, E)
        for each u_ℓ1 ∈ cands_ℓ^N(1, E)
          ...
            for each u_ℓK ∈ cands_ℓ^N(K, E)
              for each u_h ∈ cands_h^N(E)
                for each u_t ∈ cands_t^N(E)
                  W ← ⟨u_h, u_t, u_ℓ1, u_r1, ..., u_ℓK, u_rK⟩
                  if ∀⟨P_n, L_n⟩∈E exec_N-HLRT(W, P_n) = L_n then return W
```

(b)

Fig. 13. The N-HLRT wrapper class: (a) $\mathsf{exec}_{\text{N-HLRT}}$ defines how an N-HLRT wrapper is executed; and (b) $\mathsf{learn}_{\text{N-HLRT}}$ is an algorithm for learning N-HLRT.

4.6. The N-HLRT wrapper class

The HLRT and N-LR class are two straightforward ways to extend the LR class to handle more complicated pages. The sixth class we describe, N-HLRT, combines the functionality of N-LR and HLRT. An N-HLRT wrapper is a vector of $2K + 2$ strings $\langle h, t, \ell_1, r_1, \ldots, \ell_K, r_K \rangle$. The execution of such a wrapper is defined by $\mathsf{exec}_{\text{N-HLRT}}$; see Fig. 13(a).

As with N-LR, the N-HLRT delimiters interact, and thus learning is a matter of generating and testing the set of viable wrappers; see the $\mathsf{learn}_{\text{N-HLRT}}$ in Fig. 13(b). $\mathsf{learn}_{\text{N-HLRT}}$ operates just like $\mathsf{exec}_{\text{N-LR}}$, except that two additional delimiters—h and t—must also be considered.

4.7. Pushing tests into generation

As described in Section 3.2, the LR wrapper classes can be learned efficiently be two $2K$-variable CSP to which LR wrapper induction corresponds can be decomposed into $2K$

independent one-variable CSPs. We have shown that this independence property does not hold completely for the other wrapper classes. For example, for HLRT, the ℓ_1, h and t delimiters must jointly satisfy constraints $\mathcal{C}^{\mathrm{A}}_{\ell_1,h,t}$–$\mathcal{C}^{\mathrm{G}}_{\ell_1,h,t}$. With the constraints formalized in this manner, we described the $\mathsf{learn}_{\mathcal{W}}$ algorithms as straightforward generate-and-test over the space of delimiter candidates.

However, additional optimization is possible. For example, in $\mathsf{learn}_{\mathrm{HLRT}}$, only constraint $\mathcal{C}^{\mathrm{C}}_{\ell_1,h,t}$ mentions all three variables u_{ℓ_1}, u_h and u_t, while $\mathcal{C}^{\mathrm{A}}_{\ell_1,h,t}$ mentions only u_h, $\mathcal{C}^{\mathrm{D}}_{\ell_1,h,t}$ mentions only u_t, and $\mathcal{C}^{\mathrm{F}}_{\ell_1,h,t}$ mentions only u_{ℓ_1}. This observation suggests an improved $\mathsf{learn}_{\mathrm{HLRT}}$ algorithm that verifies one of $\mathcal{C}^{\mathrm{A}}_{\ell_1,h,t}$, $\mathcal{C}^{\mathrm{D}}_{\ell_1,h,t}$ or $\mathcal{C}^{\mathrm{F}}_{\ell_1,h,t}$ first, $\mathcal{C}^{\mathrm{C}}_{\ell_1,h,t}$ last, and the remaining constraints in between. However, the optimal ordering depends on the particular training examples $\mathcal{E}$, and we have not developed a provably-optimal algorithm for adaptively selecting the order in which constraints are verified.

We have also neglected the related issue of search control. The $\mathsf{learn}_{\mathcal{W}}$ algorithms as stated test candidates in an arbitrary order, but the algorithms could use heuristics to order the candidates. For example, presumably very long and very short candidates are less likely likely to be valid, and therefore the algorithm could order the candidates in terms of their difference from the mean length.

Constraint ordering and search control are two ways that the generate-and-test $\mathsf{learn}_{\mathcal{W}}$ algorithms can be made more efficient by pushing the tests into the generation process. While important, we leave further study of these issues to future work.

4.8. Review

In the next two sections, we evaluate the efficiency and expressiveness of the six wrapper classes we have defined. Before proceeding, a brief review is in order.

We have defined the LR, HLRT, OCLR, HOCLRT, N-LR and N-HLRT wrapper classes. Each class $\mathcal{W}$ is characterized by a vector of delimiters, the "interpretation" or "semantics" of which is defined by the corresponding $\mathsf{exec}_{\mathcal{W}}$ procedure. We illustrated each class by describing $\mathsf{ccwrap}_{\mathcal{W}}$, a wrapper in the class for the example country/code resource.

We also presented the $\mathsf{learn}_{\mathcal{W}}$ algorithms, which learn a wrapper for class $\mathcal{W}$. The various $\mathsf{learn}_{\mathcal{W}}$ all use the cands_x subroutines (which generate a set of candidates for each delimiter), and the valid_x subroutines (which verify whether candidates are consistent with the examples). These subroutines in turn rely on the heads, tails, $\mathsf{attribs}$, seps, $\mathsf{neighbors}_x$ and scan lower-level subroutines, which access various portions of the examples.

Finally, we discussed techniques for improving the efficiency of the $\mathsf{learn}_{\mathcal{W}}$ wrapper.

5. Expressiveness

It is important—from both a practical and theoretical perspective—to evaluate and compare our wrapper classes. In this and the following section, we perform a detailed analysis of the classes according to the criteria listed in Section 1. We begin with *expressiveness*. The fundamental issues are how well the classes handle actual Internet sites, and the relationship between the sites wrappable by each class.

5.1. Empirical results: Coverage survey

We begin with an empirical investigation of the *coverage* of our wrapper classes. Roughly, we are interested in the fraction of Internet sites can be handled by each class. In a nutshell, we surveyed a large collection of resources, and found that the majority (70% in total) of the resources can be covered by our six wrapper classes.

We examined sites listed at www.search.com, an index of 448 Internet sites.[9] A wide variety of topics are included: from the Abele Owners' Network ("over 30,000 properties nationwide; the national resource for homes sold by owner") to Zipper ("find the name of your representative or senator, along with the address, phone number, email, and Web page"). While the Internet obviously contains more than 448 sites, we expect that this index is representative of the kind of sites that information integration systems might use.

To perform the survey, we first randomly selected 30 (6.7%) of the sites from www.search.com's index. Fig. 14 lists the surveyed sites, as well as the number of attributes (K) extracted from each; K ranges from two to eighteen.

Next, for each of the thirty sites, we gathered the responses to ten sample queries. The queries were chosen by hand to be appropriate to the resource. For example, for site 1 (a computer hardware vendor), the sample queries were 'pentium pro', 'newton', 'hard disk', 'cache memory', 'macintosh', 'server', 'mainframe', 'zip', 'backup' and 'monitor'. Our intent was to solicit "normal" rather than unusual responses (e.g., error responses, pages containing no data, etc.).[10] To complete the survey, we determined how to fill in a thirty-by-six matrix, indicating for each resource whether it can be handled by each of the wrapper classes. To fill in this matrix, we labeled the examples by hand, and then used the $\mathsf{learn}_{\mathcal{W}}$ algorithms to try to learn a wrapper in class $\mathcal{W}$ that is consistent with the resource's ten examples.

One possible concern with this survey was that we might inadvertently bias the experiment in favor of sites that are more amenable to simple delimiter-based wrappers. To maintain objectivity, we randomly selected sites listed as www.search.com, an independent organization. Also, the difficulty of wrapping the site does not depend on the queries because a site either is or is not wrappable by a given wrapper class, regardless of the query. Therefore the fact that we hand-selected the queries did not bias the experiment.

Our results are listed in Fig. 15. '$\times$' indicates that there does not exist a wrapper in the class that can handle the site, while '$\surd$' indicates that the class can handle the site. To count as handling a site, a wrapper must be 100% accurate on the ten sample pages.

Fig. 16 summarizes Fig. 15. Each line in the table indicates the coverage of one or more wrapper classes. For example, the first line indicates that 70% of the surveyed sites can be handled by one or more of the six wrapper classes, while other lines show that the individual classes cover between 13% and 57% of the sites. We also report the coverage for several groups of wrapper classes. The groups are organized hierarchically: the first

[9] The site www.search.com is constantly updating its index. The survey was conducted in July 1997; some sites might have disappeared or changed significantly since then.

[10] While learning to handle such exceptional situations is important, our work does not address this problem; see [23] for some interesting progress in this area.

	resource	URL	K
1	Computer ESP	http://www.computeresp.com	4
2	CNN/Time AllPolitics Search	http://allpolitics.com/	4
3	Film.com Search	http://www.film.com/admin/search.htm	6
4	Yahoo People Search: Telephone/Address	http://www.yahoo.com/search/people/	4
5	Cinemachine: The Movie Review Search Engine	http://www.cinemachine.com/	2
6	PharmWeb's World Wide List of Pharmacy Schools	http://www.pharmweb.net/	13
7	TravelData's Bed and Breakfast Search	http://www.ultranet.com/biz/inns/search-form.html	4
8	NEWS.COM	http://www.news.com/	3
9	Internet Travel Network	http://www.itn.net/	13
10	Time World Wide	http://pathfinder.com/time/	4
11	Internet Address Finder	http://www.iaf.net/	6
12	Expedia World Guide	http://www.expedia.com/pub/genfts.dll	2
13	thrive@pathfinder	http://pathfinder.com/thrive/index.html	4
14	Monster Job Newsgroups	http://www.monster.com/	3
15	NewJour: Electronic Journals & Newsletters	http://gort.ucsd.edu/newjour/	2
16	Zipper	http://www.voxpop.org/zipper/	11
17	Coolware Classifieds Electronic Job Guide	http://www.jobsjobsjobs.com	2
18	Ultimate Band List	http://ubl.com	2
19	Shops.Net	http://shops.net/	5
20	Democratic Party Online	http://www.democrats.org/	6
21	Complete Works of William Shakespeare	http://the-tech.mit.edu/Shakespeare/works.html	5
22	Bible (Revised Standard Version)	http://etext.virginia.edu/rsv.browse.html	3
23	Virtual Garden	http://pathfinder.com/vg/	3
24	Foreign Languages for Travelers Site Search	http://www.travlang.com/	4
25	U.S. Tax Code On-Line	http://www.fourmilab.ch/ustax/ustax.html	2
26	CD Club Web Server	http://www.cd-clubs.com/	5
27	Expedia Currency Converter	http://www.expedia.com/pub/curcnvt.dll	6
28	Cyberider Cycling WWW Site	http://blueridge.infomkt.ibm.com/bikes/	3
29	Security APL Quote Server	http://qs.secapl.com/	18
30	Congressional Quarterly's On The Job	http://voter96.cqalert.com/cq_job.htm	8

Fig. 14. The surveyed information resources.

site	LR	HLRT	OCLR	HOCLRT	N-LR	N-HLRT
1	✓	✓	✓	✓	×	×
2	×	×	×	×	×	×
3	✓	✓	✓	✓	×	×
4	✓	✓	✓	✓	✓	✓
5	✓	✓	✓	✓	×	✓
6	×	×	×	×	×	×
7	×	×	×	×	×	✓
8	✓	✓	✓	✓	×	✓
9	×	×	×	×	×	×
10	✓	✓	✓	✓	×	×
11	×	×	×	×	×	×
12	×	✓	×	✓	×	✓
13	✓	✓	✓	✓	×	×
14	×	✓	×	✓	×	✓
15	✓	✓	✓	✓	×	✓
16	×	×	×	×	×	×
17	×	×	×	×	×	✓
18	×	×	×	×	×	✓
19	✓	✓	✓	✓	✓	✓
20	✓	✓	✓	✓	✓	✓
21	×	×	×	×	×	×
22	✓	✓	✓	✓	×	✓
23	✓	✓	✓	✓	×	✓
24	×	×	×	×	×	×
25	✓	✓	✓	✓	×	✓
26	×	×	×	×	×	×
27	✓	✓	✓	✓	×	✓
28	✓	×	✓	×	✓	×
29	×	×	×	×	×	×
30	✓	✓	✓	✓	×	×
coverage	16 (53%)	17 (57%)	16 (53%)	17 (57%)	4 (13%)	15 (50%)	*total:* 21 (70%)

Fig. 15. Coverage results: the surveyed sites that can be handled by each wrapper class.

wrapper class(es)	coverage (%)
LR ∪ HLRT ∪ OCLR ∪ HOCLRT ∪ N-LR ∪ N-HLRT	70
LR ∪ HLRT ∪ OCLR ∪ HOCLRT	60
LR ∪ OCLR	53
LR	53
OCLR	53
HLRT ∪ HOCLRT	57
HLRT	57
HOCLRT	57
N-LR ∪ N-HLRT	53
N-LR	13
N-HLRT	50
N-LR ∪ N-HLRT but not LR ∪ HLRT ∪ OCLR ∪ HOCLRT	25

Fig. 16. A summary of Fig. 15.

split distinguishes between tabular and nested classes, then between "HT" and "non-HT" classes, and finally the "OC" and "non-OC" classes.

Notice that the two "OC" classes (OCLR and HOCLRT) handle exactly the same sites as their "non-OC" counterparts (LR and HLRT, respectively). We are interested in "OC" wrappers because the original Metacrawler [66] used them [65]. In Section 5.2, we show that there exist sites that can be handled by OCLR but not LR, and by HOCLRT but not HLRT. These empirical results suggest that this theoretical result has modest practical significance.

A second observation is that the N-LR and N-HLRT classes perform worst. Recall that we introduced the N-LR and N-HLRT wrapper classes in order to handle the resources whose content exhibited a nested rather than tabular structure. The last line of Fig. 16 shows how successful we were: we find that N-LR and N-HLRT cover 25% of the sites that the other four classes can not handle. We conclude that, despite their relatively poor showing overall, N-LR and N-HLRT do indeed provide expressiveness not available with the other four classes.

The six wrapper classes can handle a total of 70% of the surveyed sites. The remaining 30% have several characteristics that complicate wrapping. Sites 6, 9, 11, 16 and 24 illustrate the common problem of missing attributes: if countries appear as '`<B>Ireland</B>`' but are sometimes absent, then a wrapper cannot simply search for '`<B>···</B>`'. Sites 2, 21 and 29 illustrate the second common problem of requiring disjunction: a site might display countries as either '`<B>Ireland</B>`' or '`<I>Greece</I>`', and so a wrapper must search for a disjunctive pattern. Finally, site 26 renders information in fixed-width columns, but our delimiter-based wrappers require specific constant strings.

5.2. Formal results: Relative expressiveness

While the empirical coverage results are important, we also sought a more theoretical understanding of the expressiveness tradeoffs between the various classes. Our analysis is couched in terms of *relative expressiveness*, the extent to which the functionality of

wrappers in one class can be mimicked by those in another. For example, we have already seen that both LR and HLRT wrappers exist for the country/code resource in Fig. 1, while LR can not handle the variant in Fig. 5.

To formalize this investigation, let $\Pi = \{\ldots, \langle P, L\rangle, \ldots\} \subset \Sigma^* \times \mathcal{L}$ be the *resource space*, the set of all page/label pairs $\langle P, L\rangle$.[11] Conceptually, Π includes pages from all information resources, whether regularly structured or unstructured, tabular or nested, and so forth. For each such information resource, Π contains all of the resource's pages, and each page P included in Π is paired with its label L.

Note that a wrapper class can be identified with a subset of Π: a class corresponds to those page/label pairs for which a consistent wrapper exists in the class. For a wrapper class $\mathcal{W}$, the notation $\Pi(\mathcal{W})$ indicates the subset of Π that $\mathcal{W}$ can handle:

$$\Pi(\mathcal{W}) = \{\langle P, L\rangle \in \Pi \mid \exists_{W\in\mathcal{W}}\, W(P) = L\}.$$

The $\Pi(\mathcal{W})$ formalism provides a natural way to compare the relative expressiveness of wrapper classes. If $\Pi(\mathcal{W}_1) \subset \Pi(\mathcal{W}_2)$, then $\mathcal{W}_2$ is more expressive than $\mathcal{W}_1$, in the sense that any page that can be wrapped by $\mathcal{W}_1$ can also be wrapped by $\mathcal{W}_2$.

With six wrapper classes, there are potentially 2^6 distinct regions in a Venn diagram of the $\Pi(\cdot)$ sets. To simplify this analysis, we provide relative expressiveness results for two groups of wrapper classes: (a) LR, HLRT, OCLR and HOCLRT; and (b) LR, HLRT, N-LR, and N-HLRT. Our results are captured by Theorem 1 and Fig. 17, which graphically depicts the overlap between the $\Pi(\cdot)$ regions.

Theorem 1. *The relationships between* Π(LR), Π(HLRT), Π(OCLR) *and* Π(HOCLRT), *and between* Π(LR), Π(HLRT), Π(N-LR) *and* Π(N-HLRT), *are as depicted in Fig.* 17.

Proof (*Sketch*). To see that these relationships hold, it suffices to show that:

(1) there exists at least one page/label pair $\langle P, L\rangle \in \Pi$ in each the regions in Fig. 17;
(2) OCLR subsumes LR: $\Pi(\text{LR}) \subset \Pi(\text{OCLR})$;
(3) HOCLRT subsumes HLRT: $\Pi(\text{HLRT}) \subset \Pi(\text{HOCLRT})$;
(4) every pair in N-LR and HLRT is also in LR: $(\Pi(\text{N-LR}) \cap \Pi(\text{HLRT})) \subset \Pi(\text{LR})$; and
(5) every pair in N-HLRT and LR is also in HLRT: $(\Pi(\text{N-HLRT}) \cap \Pi(\text{LR})) \subset \Pi(\text{HLRT})$.

Note that these five conditions jointly imply Theorem 1. To establish (1), we demonstrate a synthetic document that satisfies the required conditions. For example, there exists LR, OCLR and HOCLRT wrappers, but not an HLRT wrapper, that extracts $\langle\langle$`A11`, `A12`$\rangle$, $\langle$`A21`, `A22`$\rangle$, $\langle$`A31`, `A32`$\rangle\rangle$ from the synthetic document '`ho[A11](A12) cox[A21](A22)co[A31](A32)c`'. For each region, we create such a synthetic example. Then, for each wrapper class, we either demonstrate a wrapper that handles the example, or exhaustively enumerate the set of wrappers to show that none exists. To establish (2)–(5) we analyze the relevant $\text{exec}_{\mathcal{W}}$ procedures to show how to construct a wrapper in one class from a wrapper in another. For example, to establish (2), we show that for every pair $\langle P, L\rangle \in \Pi$, if LR wrapper $W = \langle \ell_1, r_1, \ldots, \ell_K, r_K\rangle$ satisfies $W(P) = L$,

[11] Recall from Section 2 that Σ is the alphabet from which pages are composed, and $\mathcal{L}$ is the set of all labels.

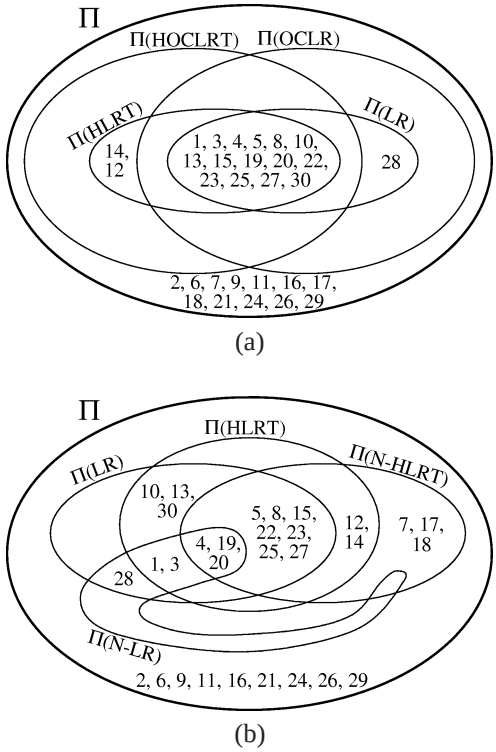


Fig. 17. The relative expressiveness of (a) LR, HLRT, OCLR and HOCLRT; and (b) LR, HLRT, N-LR and N-HLRT. Regions labeled $\Pi(\mathcal{W})$ indicate the subset of the resource space Π that can be wrapped by class $\mathcal{W}$, and integers indicate the location in Π of the surveyed sites listed in Fig. 14.

then OCLR wrapper $W' = \langle \ell_1, \phi, \ell_1, r_1, \ldots, \ell_K, r_K \rangle$ satisfies $W'(P) = L$ (where ϕ is the empty string), thereby establishing that $\Pi(\text{LR}) \subset \Pi(\text{OCLR})$. To see this, note that if an OCLR wrapper has $o \equiv \ell_1$ and $c \equiv \phi$, then $\text{exec}_{\text{OCLR}}$ reduces to exec_{LR}. (End of Theorem 1 proof sketch; see [50] for details.) □

One possibly counterintuitive implication of Theorem 1 is that the LR class is not subsumed by the HLRT class. For example, site 28 can be handled by LR but not HLRT. One might expect that an HLRT wrapper can always be constructed to mimic the behavior of any given LR wrapper. To do so, the head delimiter can simply be set to the empty string. However, the tail delimiter must be set some non-empty page fragment, and in general such a delimiter might not exist. For similar reasons, the OCLR wrapper class is not subsumed by the HOCLRT class. One could simplify these results by defining wrapper classes that allow an "artificial" end-of-file value for the tail delimiter t.

Note that Theorem 1 is a formal rather than an empirical assertion. In practice, some of the expressiveness differences are more significant than others. For example, according to Theorem 1, there exists sites that can only be wrapped by HOCLRT, but our survey did not reveal any. To shed light on the empirical relevance of Theorem 1, Fig. 17

indicates the location in Π of each surveyed site. For example, site 28 is a member of $\Pi(\text{LR}) \setminus \Pi(\text{HOCLRT})$ in (a), and $(\Pi(\text{LR}) \cap \Pi(\text{N-LR})) \setminus \Pi(\text{HLRT})$ in (b), because it can be handled by LR, OCLR and N-LR, but none of the other three classes (see line 28 of Fig. 15).

As described earlier, one major gap is that the surveyed sites do not illustrate the expressiveness differences between LR and OCLR, or between HLRT and HOCLRT. We conclude that, while interesting from a theoretical perspective, the "OC" functionality has modest practical significance. For the nested classes (N-LR and N-HLRT), the surveyed sites are distributed in most of the regions in Fig. 17.

6. Efficiency

In the previous section, we discussed the expressiveness of our six wrapper classes. Our main empirical result is that, while some fare better than others, most of the classes can handle numerous actual Internet sites. We now discuss efficiency: can wrapper induction be performed quickly? We divide our evaluation into two parts.

First (Section 6.1), we analyze the number of examples required for effective learning. If our system were to require thousands of examples before it could identify the correct wrapper, then it would be useless in practice. To summarize our results, we find that, on the contrary, a handful of examples usually suffice.

Second (Section 6.2), we analyze the computation required to learn from these examples. If our system were to require days of CPU time to process the examples, then it would be impractical, even if relatively few examples are needed for correct generalization. Looking ahead to our results, we find that in most cases our implementation requires a fraction of a CPU second per example.

6.1. Sample cost

The input to the learning algorithms $\mathsf{learn}_{\mathcal{W}}$ is a set $\mathcal{E}$ of examples. Each example $\langle P_n, L_n \rangle \in \mathcal{E}$ consumes various resources: page P_n must be fetched over the network and stored locally, and label L_n must be generated, which might require substantial processing time and consultation with a person.[12] Thus network bandwidth, processor time, memory, and human intervention are all consumed by $\mathcal{E}$. For simplicity, we ignore these details, and simply count $|\mathcal{E}|$, the number of training examples. Intuitively, as $|\mathcal{E}|$ increases, the wrapper output by $\mathsf{learn}_{\mathcal{W}}(\mathcal{E})$ should be increasingly likely to be correct. We have used a combination of empirical and analytical techniques to determine how large $|\mathcal{E}|$ must be for satisfactory performance.

6.1.1. Empirical results: Number of examples required

We have implemented the $\mathsf{learn}_{\mathcal{W}}$ algorithm for all six wrapper classes.[13] Our goal is to determine the minimum sample size $|\mathcal{E}|$ needed for effective generalization. We ran

[12] We have also investigated techniques for automatically labeled pages [50, Chapter 6].

[13] We implemented our system in Common Lisp on a 233 MHz Pentium II, with relatively little attention paid to optimizations.

our learning algorithm on the 30 sites in Fig. 14. For each site, we gathered a set of examples, and then split the examples into training and test sets. 65% of the collected examples were (potentially) used for training and the remaining 35% were used for testing; the samples were re-split randomly for each trial. We gave our learning algorithms one example from the training set, then two examples, then three, and so on, stopping when the learned wrapper performed perfectly on the test set. This process was repeated 30 times per site/class pair.

Our results are listed in Fig. 18. As with Fig. 15, "×" indicates that the given class can not wrap the given site. For the remaining site/class combinations, we list the number of examples needed to learn a wrapper that performed perfectly on the test pages. We find that 2–3 examples suffice in most cases.

Some cells not marked "×" contain only "√" instead of a page count—two sites for HOCLRT, and all sites for N-LR and N-HLRT. As with Fig. 15, "√" indicates that the class can handle the site. However, our implementation of $\mathsf{learn}_{\mathcal{W}}$ requires more than 15 minutes of CPU time, making it infeasible to run the full experiment. We discuss the complexity of our learning algorithms in Section 6.2.

6.1.2. Formal results: PAC model

Our experimental results demonstrate that in practice relatively few examples are needed to learn a high-quality wrapper. These experiments can be thought of as an empirical investigation of our task's *sample complexity*, the number of examples needed to perform a particular learning task to some specified criterion. We have also pursued a more theoretical investigation of our task's sample complexity. In this section we describe a *probably approximately correct* model of our wrapper induction problem; see [7] for a survey of the relevant literature. The PAC model gives a bound on the number of examples required to ensure (with probability exceeding a user-specified threshold) that the learned wrapper is wrong only rarely (with probability bounded by a second user-specified threshold).

The PAC model is based on the assumption that the pages—both the training and test pages—are drawn from a stationary though arbitrary and unknown probability distribution $\mathcal{D}$. The wrapper induction task for class $\mathcal{W}$ is to identify the *target* wrapper $W_T \in \mathcal{W}$, given a set $\mathcal{E} = \{\ldots, \langle P_n, L_n\rangle, \ldots\}$ of examples. Conceptually, $\mathcal{E}$ is generated by repeatedly drawing a page P_n according to distribution $\mathcal{D}$, and obtaining the label L_n that W_T would generate for P_n. As is standard in supervised learning, we assume that the learner can access W_T only indirectly, via an *oracle* from which the labels are obtained.

To learn a wrapper in class $\mathcal{W}$, we provide $\mathcal{E}$ to our $\mathsf{learn}_{\mathcal{W}}$ procedure, obtaining wrapper $W \in \mathcal{W}$ as output, and we want W to approximate W_T. How should we compare W and W_T? Ideally, $W = W_T$, though in general we can not guarantee this. The PAC model compares W and W_T based on W's *error*. The error $\mathsf{error}(W)$ of W is defined as the probability of observing an example page P drawn from distribution $\mathcal{D}$ such that $W(P) \neq W_T(P)$:

$$\mathsf{error}(W) \equiv \Pr_{P\in\mathcal{D}}\big[W(P) \neq W_T(P)\big].$$

Notice that (as expected) $\mathsf{error}(W_T) = 0$. Also, while $\mathsf{error}(W)$ is implicitly a function of $\mathcal{D}$ and W_T, to simplify the notation we do not indicate this dependence explicitly.

site	LR	HLRT	OCLR	HOCLRT	N-LR	N-HLRT	
1	2.0	2.0	2.0	2.0	×	×	
2	×	×	×	×	×	×	
3	2.0	2.0	2.0	2.0	×	×	
4	2.0	2.0	2.0	2.0	√	√	
5	2.0	2.2	2.0	2.1	×	√	
6	×	×	×	×	×	×	
7	×	×	×	×	×	√	
8	2.0	4.4	2.0	4.6	×	√	
9	×	×	×	×	×	×	
10	4.4	5.7	3.9	4.1	×	×	
11	×	×	×	×	×	×	
12	×	2.0	×	2.0	×	√	
13	2.0	2.0	2.0	2.0	×	×	
14	×	7.0	×	9.0	×	√	
15	2.0	2.0	2.0	2.0	×	√	
16	×	×	×	×	×	×	
17	×	×	×	×	×	√	
18	×	×	×	×	×	√	
19	2.0	2.0	2.0	√	√	√	
20	2.0	2.0	2.0	2.0	√	√	
21	×	×	×	×	×	×	
22	2.0	2.0	2.0	2.0	×	√	
23	2.0	2.0	3.1	√	×	√	
24	×	×	×	×	×	×	
25	2.0	2.0	2.0	2.0	×	√	
26	×	×	×	×	×	×	
27	2.0	2.0	2.0	2.0	×	√	
28	2.0	×	2.0	×	√	×	
29	×	×	×	×	×	×	
30	6.6	6.4	5.3	6.2	×	×	
mean	2.4	2.9	2.4	3.1			*mean of means:* 2.7
median	2.0	2.0	2.0	2.0			*mean of medians:* 2.0

Fig. 18. The number of examples required to learn a wrapper that performs perfectly on the test pages.

The PAC model treats W as a good approximation to W_T to the extent that $\mathsf{error}(W)$ approaches zero. Specifically, we assume a user-supplied *accuracy* parameter $0 < \varepsilon < 1$, and we want to ensure that $\mathsf{error}(W) < \varepsilon$, no matter how close ε is to zero.

In general we can not guarantee that this relationship will hold, because the examples $\mathcal{E}$ might be misleading. The best we can do is to make the probability that the error is small be as close to one as requested. A second *reliability* parameter $0 < \delta < 1$ serves this purpose.

The PAC model demands that, for any values of ε and δ, the learner will (with probability at least $1-\delta$) output a wrapper W satisfying $\mathsf{error}(W) < \varepsilon$.

How will the learner satisfy this criterion? The idea is that the learner can request a sample of a given size—i.e., it has control over $|\mathcal{E}|$. Presumably, the learner would request more examples as ε and δ approach zero. Naturally, the learner has no access to W_T or $\mathcal{D}$ when deciding how many examples to request.

The PAC model for a particular wrapper classes $\mathcal{W}$ thus boils down to the following analysis: determine the number $|\mathcal{E}|$ such that, for any distribution $\mathcal{D}$, any target wrapper $W_T \in \mathcal{W}$, and any $0 < \varepsilon, \delta < 1$, if our $\mathsf{learn}_{\mathcal{W}}$ algorithm is provided with $|\mathcal{E}|$ examples and returns a wrapper W, then we have that $\mathsf{error}(W) < \varepsilon$ with probability at least $1-\delta$—in short, W is probably approximately correct.

To develop a PAC model of our wrapper induction task, we begin with the well-known PAC bound for the case in which there are a finite number of possible targets. It is straightforward to show that if $\mathcal{W}$ has finite cardinality, and wrapper $W \in \mathcal{W}$ agrees with the target wrapper on $|\mathcal{E}|$ examples (i.e., $W(P_n) = L_n = W_T(P_n)$ for all $\langle P_n, L_n\rangle \in \mathcal{E}$), then the chance that $\mathsf{error}(W) > \varepsilon$ is at most $|\mathcal{W}|(1-\varepsilon)^{|\mathcal{E}|}$. The learning algorithm can thus satisfy the PAC criterion by ensuring that $|\mathcal{W}|(1-\varepsilon)^{|\mathcal{E}|} < \delta$, which is easily shown to be satisfied for

$$|\mathcal{E}| > \frac{1}{\varepsilon}\big(\ln|\mathcal{W}| - \ln\delta\big). \tag{3}$$

What is $|\mathcal{W}|$, for each of our six classes? Since a wrapper is just a vector of arbitrary strings, there are an infinite number of wrappers in each class. However, after observing just a single example, the number of "feasible" wrappers becomes finite; for example, at the very least every delimiter must be a substring of this first example.

Consider first the LR class. We can bound |LR| as follows. The set of wrappers considered by $\mathsf{learn}_{\mathrm{LR}}$ (Fig. 4) is the cross product of each delimiter's candidate set. The candidates for ℓ_k are generated by the cands_ℓ subroutine, and r_k's candidates are generated by cands_r. We can bound $|\mathsf{cands}_\ell(k,\mathcal{E})|$ and $|\mathsf{cands}_r(k,\mathcal{E})|$ in terms of $R = \min_n |P_n|$, the length of the shortest example. R is an upper bound on both $|\mathsf{cands}_\ell(k,\mathcal{E})|$ and $|\mathsf{cands}_r(k,\mathcal{E})|$, for each value of k. To see this, note that the candidates for ℓ_k are the suffixes of the shortest string occurring immediately prior to an instance of the kth attribute (see line [*iii*] in Fig. 4), and since this shortest string has length at most R, there can be at most R such candidates. A similar argument applies to each r_k. Therefore we have that:

$$|\mathrm{LR}| = \prod_{k=1}^{K}\big|\mathsf{cands}_\ell(k,\mathcal{E})\big| \times \prod_{k=1}^{K}\big|\mathsf{cands}_r(k,\mathcal{E})\big| \leqslant \prod_{k=1}^{K} R \times \prod_{k=1}^{K} R \;=\; R^{2K}.$$

Substituting this bound on |LR| for $|\mathcal{W}|$ in Eq. (3), we arrive at the PAC model for the LR class. To satisfy the PAC criterion, $\mathsf{learn}_{\mathrm{LR}}$ must examine at least $|\mathcal{E}|$ examples, where:

$$|\mathcal{E}| \geqslant \frac{1}{\varepsilon}\big(\ln|\mathrm{LR}| - \ln\delta\big) \leqslant \frac{1}{\varepsilon}\big(\ln R^{2K} - \ln\delta\big) = \frac{1}{\varepsilon}\big(2K\ln R - \ln\delta\big).$$

It is straightforward to extend these ideas to the other six wrapper classes. The number of wrappers in each class can be calculated by multiplying together the number of candidates for each of the class's delimiters. We have seen that the number of candidates for each ℓ_k

and r_k is simply R. There are $R(R-1)/2$ substrings of a string of length R. Therefore there are $R(R-1)/2$ candidates for each of h, t, o and c, since can be an arbitrary substring of the shortest page. Thus we have that

$$|\text{LR}|, |\text{N-LR}| \leqslant R^{2K},$$

$$|\text{HLRT}|, |\text{OCLR}|, |\text{N-HLRT}| \leqslant R^{2K}\left(\frac{R(R-1)}{2}\right)^2 = \frac{R^{2K+2}}{4}\left(R^2-2R+1\right),$$

$$|\text{HOCLRT}| \leqslant R^{2K}\left(\frac{R(R-1)}{2}\right)^4 = \frac{R^{2K+4}}{16}\left(R^4-4R^3+6R^2-4R+1\right).$$

To conclude the development of our PAC model for the six classes, we can substitute each of these bounds on $|\mathcal{W}|$ into Eq. (3), thereby proving the following theorem.

Theorem 2. *Suppose we give the learning algorithm* $\mathsf{learn}_{\mathcal{W}}$ (*for any of our six wrapper classes* $\mathcal{W}$) *a set* $\mathcal{E} = \{\ldots, \langle P_n, L_n\rangle, \ldots\}$ *of examples, and it returns a wrapper* $W \in \mathcal{W}$. *For any distribution* $\mathcal{D}$, *for any values of* $0 < \varepsilon, \delta < 1$, *and for any target* $W_T \in \mathcal{W}$, *if* $|\mathcal{E}|$ *satisfies the following condition*:

Wrapper class(es)	*Sample complexity*
LR, N-LR	$\lvert\mathcal{E}\rvert \geqslant \frac{1}{\varepsilon}(2K\ln R - \ln\delta)$
HLRT, OCLR, N-HLRT	$\lvert\mathcal{E}\rvert \geqslant \frac{1}{\varepsilon}((2K+2)\ln R + \ln(R^2-2R+1) - \ln(4\delta))$
HOCLRT	$\lvert\mathcal{E}\rvert \geqslant \frac{1}{\varepsilon}((2K+4)\ln R + \ln(R^4-4R^3+6R^2-4R+1) - \ln(16\delta))$

(*where* $R = \min_n |P_n|$), *then the probability that* $\mathsf{error}(W) < \varepsilon$ *is at least* $1-\delta$.

For example, if there are $K = 4$ attributes per tuple, the shortest example page has length $R = 10,000$, and $\varepsilon = \delta = 1/20$, then the $\mathsf{learn}_{\mathcal{W}}$ algorithms must examine at least the following number of examples to satisfy the PAC criterion:

Wrapper class(es)	Predicted minimum sample size
LR, N-LR	$\lvert\mathcal{E}\rvert \geqslant 1534$
HLRT, OCLR, N-HLRT	$\lvert\mathcal{E}\rvert \geqslant 2243$
HOCLRT	$\lvert\mathcal{E}\rvert \geqslant 2952$

Compared to our empirical results in the previous section, our PAC bounds appear to be too loose by about two orders of magnitude. We have investigated a variety of ways to tighten these bounds [50]. For example, we have shown that an HLRT, OCLR or N-HLRT wrapper is PAC if:

$$(4K-2)\left(1-\frac{\varepsilon}{4K-2}\right)^M + \frac{R^3(R-1)^2}{4}\left(1-\frac{\varepsilon}{2}\right)^{|\mathcal{E}|} < \delta,$$

where $M = \sum_n |L_n|$ is the total number of tuples across the examples in $\mathcal{E}$. Using the above parameters, and assuming an average of 5 tuples per example (so that $M = 5|\mathcal{E}|$), the model predicts that 898 examples are required, a 60% savings. This bound is still very loose, and we leave the problem of tightening it further to future work.

6.2. Induction cost

Our evaluation of the sample cost of wrapper induction reveals that relatively few examples are needed in practice for effective learning. But a learning system that processes examples slowly will still perform poorly, even if few examples are required. Thus are we also analyze the cost of processing the examples.

In this section, we begin with empirical evidence that our system usually runs quite quickly, and then provide complexity analyses of our learning algorithms.

6.2.1. Empirical results: Per-example processing time

We used the experimental approach described in Sections 5.1 and 6.1.1 to measure the per-example processing cost. To review, for each of the thirty Internet sites that can be handled by each wrapper class, we give our system one example, then two, etc., until it generates a wrapper that performs perfectly on a suite of test pages.

Fig. 19 lists the number of CPU seconds per example required to learn a wrapper in this experiment. As in Fig. 18, our system sometimes runs very slowly (more than 15 CPU minutes); these cases are listed as ">900". We also listed the means and medians for each class. Across all sites and wrapper classes, the median time per example is 0.58 CPU seconds, though the mean figure of 40.8 more accurately reflects the fact that the learning algorithms occasionally run very slowly. Our complexity results in Section 6.2.2 describe the parameters on which the algorithms' running times depend. (When calculating the means for HOCLRT, we use the value 900 seconds for sites 19 and 23, so the "true" mean is actually greater than the reported values. Also, these statistics are not calculated for N-LR and N-HLRT since they would be meaningless.)

We have reported the CPU time *per example* because the number of examples required depends on the site. The total CPU time can be obtained by simply multiplying the corresponding cells in Figs. 18 and 19. The results are as follows (times are in CPU seconds):

- minimum total time: 0.04,
- maximum total time: 2779,
- mean total time: 144.5,
- median total time: 0.74.

6.2.2. Formal results: Complexity analysis

Our empirical results suggest that our wrapper induction algorithms usually run quite quickly, often consuming just a fraction of a CPU second per example. To get a deeper understanding of these results—and particularly to understand why the algorithms occasionally run very slowly—we have also investigated the computational complexity the $\mathsf{learn}_{\mathcal{W}}$ algorithms.

Let $\mathcal{E} = \{\ldots, \langle P_n, L_n\rangle, \ldots\}$ be a set of examples. We are interested in a bound on the time to execute $\mathsf{learn}_{\mathcal{W}}(\mathcal{E})$. Our analysis is stated in terms of the following parameters: $|\mathcal{E}|$ (the number of examples), K (the number of attributes per tuple), $M = \sum_n |L_n|$ (the total number of tuples in the examples), and $V = \max_n |P_n|$ (the length of the longest example).

site	LR	HLRT	OCLR	HOCLRT	N-LR	N-HLRT	
1	0.10	0.26	0.11	0.25	×	×	
2	×	×	×	×	×	×	
3	0.14	3.00	0.19	2.72	×	×	
4	0.02	3.63	0.04	107	>900	>900	
5	0.03	1.94	0.05	2.09	×	>900	
6	×	×	×	×	×	×	
7	×	×	×	×	×	>900	
8	0.10	113	0.32	26.7	×	>900	
9	×	×	×	×	×	×	
10	0.14	0.98	0.20	1.07	×	×	
11	×	×	×	×	×	×	
12	×	0.33	×	0.36	×	>900	
13	0.13	1.23	0.19	1.19	×	×	
14	×	397	×	277	×	>900	
15	0.02	0.04	0.03	0.04	×	>900	
16	×	×	×	×	×	×	
17	×	×	×	×	×	>900	
18	×	×	×	×	×	>900	
19	0.04	0.91	0.06	>900	>900	>900	
20	0.05	0.08	0.08	0.09	>900	>900	
21	×	×	×	×	×	×	
22	0.10	0.60	0.26	0.66	×	>900	
23	0.13	0.37	0.18	>900	×	>900	
24	×	×	×	×	×	×	
25	0.02	0.09	0.3	0.9	×	>900	
26	×	×	×	×	×	×	
27	0.05	11.8	0.23	11.4	×	>900	
28	0.26	×	0.28	×	>900	×	
29	×	×	×	×	×	×	
30	0.01	0.01	0.01	0.01	×	×	
mean	0.08	31.5	0.14	131.3			*mean of means:* 40.8
median	0.08	0.91	0.12	1.19			*mean of medians:* 0.58

Fig. 19. The number of CPU seconds per example required for learning, for each class that can handle each site.

Consider first the LR class. We will derive the complexity of $\mathsf{learn}_{\mathrm{LR}}$ in a bottom-up fashion, reasoning about the complexity of the algorithm's subroutines and then composing these results to obtain the $\mathsf{learn}_{\mathrm{LR}}$'s overall complexity.

We begin with the lowest-level subroutines: attribs, seps, heads, tails and $\mathsf{neighbors}_x$. attribs and seps both run in time $\mathrm{O}(M)$, since they involve iterating over every example tuple. heads and tails both run in time $\mathrm{O}(|\mathcal{E}|)$, since there is one head and tail per example.

The $\mathsf{neighbors}_x$ subroutines invoke seps, tails and heads in turn, and therefore runs in time $O(M + |\mathcal{E}|)$.

The cands_x subroutines enumerate the suffixes (or prefixes) of the shortest of the $O(M + |E|)$ strings returned by $\mathsf{neighbors}_x$. Since each candidate has length $O(V)$, we have that cands_x runs in time $O(M + |\mathcal{E}| + V)$.

The valid_x subroutines search for delimiter candidates in the strings returned by tails, $\mathsf{attribs}$, and $\mathsf{neighbors}_x$. Since the candidates and strings being searched all have length bounded by V, each such search can be performed in time $O(V)$ using efficient techniques [47]. Therefore the overall complexity of the valid_x subroutines is $O(V(M + |\mathcal{E}|))$.

Finally, we are in a position to evaluate the overall complexity of $\mathsf{learn}_{\mathrm{LR}}$. This learning algorithm first learns the left-hand delimiters and then the right-hand delimiters. To learn each of the K delimiters of each kind, $\mathsf{learn}_{\mathrm{LR}}$ tests the candidates generated by cands_x using valid_x. Since there are $O(M + |\mathcal{E}| + V)$ candidates for each delimiter, and since each call to valid_x takes time $O(V(M + |\mathcal{E}|))$, we have that $\mathsf{learn}_{\mathrm{LR}}$ takes time $O(K(M + |\mathcal{E}| + V)V(M + |\mathcal{E}|)) = O(KM^2|\mathcal{E}|^2V^2)$.

The results of such an analysis for the other three tabular classes (HLRT, OCLR and HOCLRT) are similar. Learning these classes is harder, since their learning algorithms use nested loops to search for a satisfactory combination of candidates for ℓ_1, h, t, o and c. The nested classes (N-LR and N-HLRT) have the worst complexity, since their learning algorithms have a deeply nested loop structure.

Theorem 3. *The invocation* $\mathsf{learn}_{\mathcal{W}}(\mathcal{E})$ *takes time that grows as follows*:

Wrapper class	*Complexity*
LR	$O(KM^2\|\mathcal{E}\|^2V^2)$
HLRT	$O(KM^2\|\mathcal{E}\|^4V^6)$
OCLR	$O(KM^4\|\mathcal{E}\|^2V^6)$
HOCLRT	$O(KM^4\|\mathcal{E}\|^4V^{10})$
N-LR	$O(M^{2K}\|\mathcal{E}\|^{2K+1}V^{2K+2})$
N-HLRT	$O(M^{2K+2}\|\mathcal{E}\|^{2K+3}V^{2K+4})$

where $V = \max_n |P_n|$, $M = \sum_n |L_n|$, *and each tuple contains* K *attributes.*

These results are mildly encouraging: the tabular classes can all be learned in polynomial time. Moreover, K is usually relatively small. Thus the exponential results for the nested classes are somewhat attenuated.

Nevertheless, the degrees of the polynomials are fairly ominous: a degree-ten polynomial is unlikely to be useful in practice, even if the parameters are fairly small. Moreover, the parameters are *not* normally small; e.g., in our experiments, V ranges from 899 to 57,116 characters. It is therefore not surprising that in some cases (two sites for HOCLRT, and all sites for N-LR and N-HLRT) our algorithm runs so slowly.

Perhaps a more pressing question is why our system *ever* runs quickly. The answer is that there are usually many satisfactory wrappers for a given site: the search space is large, but densely filled with goal states. For example, for site 4, $\mathsf{learn}_{\mathrm{LR}}$ explores

a space containing 63,680,400 potential wrappers, of which 9,192,960 (14%) are valid. An interesting direction of future work would be to develop search heuristics for this space.

7. Corroboration

The learning algorithms described in Sections 3 and 4 are steps toward our goal of automatic wrapper construction, but the *labeling problem* [27] remains: our learning algorithms require not just the example pages (e.g., Fig. 1(c)) but also a description of the information to be extracted (Fig. 1(d)).

So far, we have assumed that a person labels the examples. This approach reduces the task of hand-coding a wrapper to the task of hand-labeling a set of examples. This reduction might make the person's job much easier, as he can focus on the attributes to be extracted rather than low-level HTML-specific details. Nevertheless, since our goal is to automate the wrapper construction task, we have also explored ways to automatically label the examples.

To address this issue, we have developed a technique for automatically labeling example documents [50, Chapter 6]. Our *corroboration* algorithm takes as input as set of *recognizers*, domain-specific heuristics for identifying instances of the attributes to be extracted. In the country/code example, our system would take as input procedures for recognizing the instances of countries ('`Congo`', '`Egypt`', etc.) and the codes ('`242`', '`20`', etc.).

The required recognition heuristics might be very primitive—e.g., using the regular expression '`[1-9][0-9]+`' to identify country codes. At the other extreme, recognition might require natural language processing, or the querying of other information resources—e.g., asking an already-wrapped resource to determine whether a particular text fragment is a person's name.

Once the instances of each attribute have been identified, corroboration involves combining the results for the entire page. If the recognition heuristics are perfect, then this integration step is trivial. Note, though, that perfect recognizers do not obviate the need for wrapper induction, because while the recognizers might be perfect, they might also be very slow and thus be unable to deliver the fast performance required for an on-line information-integration system.

An important feature of our corroboration system is that it can handle recognized mistakes. For example, the country recognizer might find some text fragments that are not in fact countries (i.e., it might exhibit false positives), or it might ignore some countries (false negatives). Our corroboration algorithm can make use of recognizers even when they make such mistakes. Specifically, the algorithm requires at least one correct recognizer, and also that each recognizer exhibits false positives or false negatives, but not both.

To handle noisy recognizers, our corroboration algorithm computes a set of labels that are consistent with the recognized instances. The algorithm uses the required perfect recognizer as an "anchor" to detect false positives. For example, suppose the country recognizer is correct, but the country-code recognizer makes false positives. If the country recognizer reports that there are two countries at indices 10–20 and 30–40, while the country-code recognizer reports three codes at indices 5–8, 25–28 and 45–48, then the

corroboration algorithm can discard the code at 5–8: it must be a false positive because it implies a country before 5, but the country recognizer does not make mistakes.

While in this example the correct label can be recovered, in other situations the recognized instances are inherently ambiguous. For example, if the code recognizer reported codes at 25–28, 45–48 and 50–55, then the corroboration algorithm concludes that 25–28, and *either* 45–48 *or* 50–55, are correct. When faced with such ambiguity, the corroboration algorithm generates a set of labels, one for each way to select one instance from each ambiguous set.

While false positives lead to multiple consistent labels, false negatives result in "holes" in a label. If the code recognizer produced false negatives, then it might report just a single code 25–28. In this case, the corroboration algorithm determines that code 25–28 must correspond to country 10–20 (since $20 < 25 < 28 < 40$). However, the algorithm can find no code corresponding to country 30–40, and so simply leaves the corresponding label cell empty. Such empty cells result in fewer training examples during learning, but our experiments demonstrate that the number of additional training documents required scales well the the rate of false negatives.

Finally, our corroboration algorithm is not given as input the order in which the attributes occur; the algorithm repeats the above process for all attribute orderings.

The output of the corroboration algorithm is a set of labels, each consistent with the recognized instances, but only one of which is correct. To learn a wrapper from a set of examples, the corroboration algorithm is invoked one each example, resulting in a set of candidate labels for each example. The next step is to select one label for each example. To do so we exploit an additional heuristic: the correct labels must not only be consistent with the recognized instances, but also there must exist a wrapper that can correctly extracted the given labels. For example, if one of the candidate labels involves (incorrectly) marking both '`Congo</`' and '`Ireland`' as countries, then the learning algorithm will be unable to find a valid r_1 delimiter, because r_1 must be a prefix of both '`B>`$\cdots$' (the text after '`Congo</`') and '`</B>`$\cdots$' (the text after '`Ireland`'), but these strings have no common prefix. In principle it is possible that a label is incorrect and yet there exists a wrapper that can extract this invalid content, but in our experiments this rarely happens.

In [50, Chapter 6] we describe our corroboration algorithm in detail, and demonstrate empirically that our wrapper induction approach scales well, even when all but one of the recognizers make up to 40% errors. In some domain-specific it can be difficult to develop recognizer heuristics that make only one-sided errors. Nevertheless, we conclude that our corroboration algorithm represents interesting progress toward the goal of fully-automatic wrapper induction.

8. Related work

Our approach to automatic wrapper construction draws on ideas from numerous research areas. After briefly discussing systems that use wrappers, and work on learning models of information sources, we discuss research on trainable information extraction systems, of which our wrapper induction technique is an instance.

Systems that use wrappers. Our concern with wrappers is motivated by the diverse research on software agents (e.g., [14,29]) and information integration (e.g., [38,44,54, 75]). We were strongly influenced by the University of Washington "Softbot" project [23,26,30,66,67]; related projects include ARIADNE [46], CARNOT [21], DISCO [32], GARLIC [18], HERMES [2], the Information Manifold [55], TSIMMIS [19], FUSION [68], BargainFinder [49], and the Knowledge Broker [4]. The details vary widely, but these systems all need a library of wrappers for accessing the information sources they exploit.

There has been substantial research on specialized programming languages and graphical user interfaces to assist in manually writing such wrappers [3,25,37,39,42,63, 69]. These projects rely on humans rather than learning techniques to generate wrappers. We see wrapper induction as a complementary effort: current wrapper induction algorithms generate wrappers expressible in subsets of these languages, and an important direction for future work is to learn more expressive subsets.

Emerging standards such as XML will simplify the extraction of structured information from heterogeneous sources. However, few sites currently use such standards, and legacy data will be with us for years. Moreover, XML forces information consumers to accept the ontological decisions of the data exporters. For example, integration is difficult if one site splits people's names into first and last names, while another combines them. We conclude that the thorny problem of wrapper construction and maintenance will remain for some time. On the other hand, XML is an ideal mechanism for standardizing wrapper outputs. Furthermore, as Knoblock and Minton observed [54], wrapper induction algorithms may be able to use XML as a source of supervised training data.

Learning models of information sources. Wrapper induction is one aspect of the larger problem of learning models of information sources; examples include SHOPBOT and ILA. SHOPBOT [23] uses HTML-specific heuristics to learn how to pose queries to on-line product catalogs. SHOPBOT also uses heuristics to extract particular pieces of information, such as product prices. ILA [61] learns an information source's schema in terms of its background knowledge. Suppose ILA knows that email(`Jane`) = `jane@z.com`, email(`Fred`) = `fred@z.com`, secretary(`Jane`) = `Fred`, office(`Fred`) = `Rm29`, and phone(`Rm29`) = `567-9876`. ILA can learn about a site by querying with `Jane`; if it observes ⟨`Jane`, `fred@z.com`, `567-9876`⟩, ILA hypothesizes that the site returns tuples of the form ⟨*person*, email(secretary(*person*)), phone(office(*person*))⟩.

Trainable information extraction systems. Information extraction (IE) is the task of identifying fragments in a document that constitute its core semantic content; see [22,40] for surveys. The key challenge to IE is scalability, the capacity to rapidly reconfigure an IE system as new information sources become available, or existing sources change their format or disappear. Current work in scalable IE systems has focused on the use of machine learning techniques to automatically acquire and maintain domain-specific extraction knowledge. A wrapper is thus a special-purpose IE system designed for documents from a particular Internet site, and wrapper induction is a machine learning technique for maintaining wrapper libraries in a scalable fashion.

There has been substantial work on trainable IE systems in recent years. This research has tended to be split between two communities: the natural language processing

community has focused on free text [16,43,62,72], while the information integration and software agent communities have focused on structured Internet documents [9,11,41,52, 60]. This distinction has started to blur, as researchers have started to evaluate their systems on both structured and natural text [33,70,71]. We now discuss these systems in more detail, roughly in order from Internet-specific to free-text systems.

Ashish and Knoblock [9] describe a semi-automatic technique for wrapper induction that uses HTML-specific heuristics to generate plausible segmentations of a document, and plausible items for extraction within a segment. After a human corrects the choices if necessary, they are compiled into N-HLRT-like wrappers, where the delimiters can be regular expressions instead of constants; these regular expressions imported from the hand-coded heuristics rather than learned.

Hsu and Dung [41] present SOFTMEALY. Their wrapper language is more expressive than HOCLRT, allowing

(1) disjunction (attribute edges can be delimited by more than one delimiter, whereas we assume exactly one delimiter per attribute edge);
(2) multiple attribute orders within tuples;
(3) missing attributes; and
(4) extraction to be driven by features of the candidate for extraction (e.g., "extract starting at the next '<B>' if the next word is capitalized").

Hsu and Dung report that SOFTMEALY can wrap the 30% of the sites surveyed in Section 5.1 that our six classes can not handle [41].

Muslea et al. [60] describe STALKER, an algorithm for learning a wrapper language that, like SOFTMEALY, allows disjunction and reordered or missing attributes. The main contribution of Muslea et al. is that their language permits an arbitrary sequence of "landmarks" (e.g., "extract at the first '<B>' following the next '<HR>' "). This feature can be thought of as a generalization of the "OC" and "HL" functionality we describe. In an empirical comparison, Muslea et al. report that STALKER is 4-fold slower $\mathsf{learn}_{\mathrm{HLRT}}$ for one domain, 12-fold faster in a second, and our six classes are not expressive enough to handle several other domains [60].

The previous systems all take advantage of HTML annotations by, for example, using HTML tags as delimiters or landmarks. However, our system and several others do not depend on HTML; they will learn a non-HTML delimiter as long as it reliably identifies the items to be extracted. In contrast, Bauer and Dengler [11] describe TRIAS, a wrapper induction system that relies more heavily on HTML. Wrappers in their language operate directly on a document's HTML parse tree. The advantage of TRIAS is that it can be less sensitive to changes in document formatting outside the relevant fragments of the parse tree; the disadvantage is that the techniques are inapplicable to non-HTML documents. For example, TRIAS can not handle the document in Fig. 11, since it does not contain HTML tags.

The systems discussed so far create wrappers in specialized wrapper languages. The LR wrapper class, for example, corresponds to all ways to instantiate the $\mathsf{exec}_{\mathrm{LR}}$ "template". An alternative is to encode documents in a first-order relational representation, and use inductive logic programming to learn wrappers corresponding to arbitrary first-order theories over the representation. These techniques have arisen mainly in the natural language processing community, because wrappers for free-text documents often must rely

on complex chains of relationships between tokens. In practice, a completely general ILP approach is infeasible, so researchers have investigated a variety of special cases.

- Freitag [33,34] describes the SRV system. SRV's representation "simulates" a document's token sequence—i.e., rules can be triggered by the token that occurs before or after a given token, and these conditions can be chained arbitrarily. SRV also encodes "low-level" features (e.g., whether a token is capitalized or contains numbers), and a set of domain-specific features for handling HTML text. SRV searches for rules in a top-down fashion, gradually adding constraints to rules that are maximally general. Freitag has also shown that performance improves when SRV is combined with other learning algorithms in a "multistrategy learning" approach [35].
- Soderland [72] describes CRYSTAL, which learns information extraction rules triggered by part-of-speech and lexical information. CRYSTAL uses a bottom-up search, gradually relaxing rules that are maximally specific. Soderland later developed WEBFOOT [70], an extension to CRYSTAL for structured HTML documents. WEBFOOT that uses hand-coded heuristics to partition HTML documents into sentence-like segments.
- Califf and Mooney [16,17] describe RAPIER. Like CRYSTAL, RAPIER uses part-of-speech and lexical information, though RAPIER searches for extraction rules in a bi-directional fashion.

Soderland [71] describes WHISK, which learns to extract from documents with varying degrees of structure—grammatical text, telegraphic or ungrammatical text (e.g., weather reports or apartment listings), and highly structured text with HTML annotations (e.g., the sort of documents we used to evaluate our system). WHISK's learned rules correspond to a restricted class of regular expressions; such rules are more expressive than the six wrapper classes we have described, and less expressive than the relational rules learned by SRV, CRYSTAL and RAPIER. The main weakness of WHISK is that it operates only on a single sentence or segment at a time, and so its extraction decisions are based on a limited context. Soderland demonstrates WHISK handling a site that HLRT can wrap, but notes that WHISK runs slower than $\mathsf{learn}_{\mathrm{HLRT}}$ on this problem because its search space is much larger [71].

WEBFOOT, SRV and WHISK suggest that a trainable information extraction systems need not be confined to one sort of document, such as grammatical text or rigidly structured HTML text. Rather, trainable IE systems can gain leverage from the regularities that happen to correctly indicate the fragments to be extracted in a particular domain, whether these regularities arise from "low-level" features (e.g., constant delimiters or landmarks, or capitalization information) or "linguistic" features (e.g., part-of-speech or lexical information).

Finally, our work is related to the literature on grammar induction (e.g., [6]). For each wrapper class $\mathcal{W}$, the $\mathsf{exec}_{\mathcal{W}}$ procedure uses a finite amount of state for parsing, augmented with additional book-keeping state for storing the extracted information. Though unbounded, this book-keeping state is distinct from the state used for parsing, and thus our wrappers are formally equivalent to regular grammars. However, we can not use existing grammar induction algorithms, because our wrappers are used for parsing, not just classification. They can not simply examine a query response and confirm that it came from a particular site. Rather, a specific sort of examination must occur; namely, one that

involves scanning the page so as to identify the fragments to be extracted. We therefore require that the learned grammar have a particular structure. Learning algorithms have been developed for several classes of regular grammars (e.g., reversible grammars [5]), but we do not know of algorithms that deliver the particular state topology we require.

9. Discussion and future work

In this article, we have introduced wrapper induction, a technique for automatically constructing the information-extraction procedures required by many kinds of information-manipulation systems. Our work can be summarized in terms of our two main contributions.

- First, we posed the task of automatic wrapper construction as one of inductive learning, where instances correspond to pages, labels correspond to the pages' content, and hypotheses correspond to wrappers. A learning approach is crucial to maintaining large wrapper libraries for the Internet, since new sources continually appear, and existing sources disappear or regularly change their formatting conventions. We have identified several wrapper classes that are reasonably useful, but that can usually be learned relatively quickly.
- Second, using a combination of empirical and analytic techniques, we explored the computational tradeoffs between the classes. Our evaluation revealed several subtleties regarding the expressiveness and efficiency of our classes.

While the six wrapper classes we identified (particularly HLRT) are interesting in their own right, our main motivation has been not to propose a definitive wrapper language, but rather to develop a framework within which to investigate the wrapper induction problem.

From our experience with the six wrapper classes, we can make two general conclusions about families of related wrapper classes. First, one should not underestimate the effectiveness of very simple classes. LR, for example, can be learned quickly from few examples, and yet can handle some sites (e.g., site 28 in Fig. 15) that defeat more sophisticated classes such as HLRT, HOCLRT and N-HOCLRT. A second conclusion is that even simple extensions can have dramatic computational results. For example, LR can be learned in time *independent* of the number of attributes K, while our N-LR learning algorithms runs in time *exponential* in K.

We are currently investigating several extensions to the techniques described in this article. Our results for the LR, HLRT, OCLR and HOCLRT classes are fairly satisfying, but N-LR and N-HLRT fare less well. While they represent an interesting first step, these classes do not provide the functionality needed to handle some kinds of nested documents, and they are very hard to learn. We are investigating further variants of the LR class in order to further explore the tradeoffs between expressiveness and efficiency.

We are also examining ways to speed up the six learning algorithms. While our empirical results are satisfactory for LR and OCLR, our HLRT and HOCLRT algorithms occasionally run very slowly. One possibility is to develop heuristics to speed search in the enormous space of potential wrappers. The constraint satisfaction literature may well provide useful ideas for eliminating large portions of this space.

Our PAC models are clearly too loose, and tightening the models would be an interesting direction for future work. Note that the models makes worst-case assumptions about the learning task. Specifically, they assume that the distribution $\mathcal{D}$ over examples is arbitrary. A standard technique for tightening a PAC model is to assume that $\mathcal{D}$ has certain properties [10,12]. Shuurmans and Greiner [64] suggest another strategy: by replacing the "batch" model on inductive learning with a "sequential" model in which the PAC-theoretic analysis is repeated as each example is observed, many fewer examples are predicted. It would be interesting to apply these approaches to our task.

Finally, we have focused exclusively on extraction, but "industrial-strength" wrappers must deal with a host of complications, such as caching, parallel network access, transient network faults, and incremental extraction as documents arrive over the network. From a research perspective, one of the most interesting challenges is that existing wrapper induction systems ignores the fact that the formatting conventions on which wrappers rely can change unexpectedly. The implicit strategy is to learn a new wrapper from scratch, rather than repair the broken wrapper. As a preliminary step to addressing this wrapper maintenance task, we have investigated the problem of verifying whether a wrapper is correct [51].

Acknowledgements

This research was conducted while the author was at the Department of Computer Science and Engineering at the University of Washington; many thanks to Dan Weld. Alan Smeaton, Ion Muslea and the anonymous reviewers gave helpful comments. This research was funded in part by ONR Grant N00014-94-1-0060, by NSF Grant IRI-9303461, by ARPA / Rome Labs grant F30602-95-1-0024, and by a gift from Rockwell International Palo Alto Research.

References

[1] S. Abiteboul, Querying semi-structured data, in: Proc. Internat. Conf. Database Theory, 1996, pp. 1–18.
[2] S. Adali, K. Candan, Y. Papakonstantinou, V. Subrahmanian, Query caching and optimization in distributed mediator systems, in: Proc. ACM SIGMOD Conference on Management of Data, Montreal, Quebec, 1996.
[3] B. Adelberg, NoDoSE—A tool for semi-automatically extracting structured and semistructured data from text documents, in: Proc. ACM SIGMOD Conference on Management of Data, Seattle, WA, 1998.
[4] J. Andreoli, U. Borghoff, R. Pareschi, The constraint-based knowledge broker mode: Semantics, implementation and analysis, J. Symbolic Comput. 21 (4) (1996) 635–667.
[5] D. Angluin, Inference of reversible languages, J. ACM 29 (3) (1982) 741–765.
[6] D. Angluin, Learning regular sets from queries and counterexamples, Inform. and Comput. 75 (1987) 87–106.
[7] D. Angluin, Computational learning theory: Survey and selected bibliography, in: Proc. 24th ACM Symposium on the Theory of Computing, 1992, pp. 351–369.
[8] Y. Arens, C. Knoblock, C. Chee, C. Hsu, SIMS: Single interface to multiple sources, TR RL-TR-96-118, USC Rome Labs, 1996.
[9] N. Ashish, C. Knoblock, Semi-automatic wrapper generation for Internet information sources, in: Proc. Cooperative Information Systems, 1997.
[10] P. Bartlett, R. Williamson, Investigating the distributional assumptions of the PAC learning model, in: Proc. 4th Workshop Computational Learning Theory, 1991, pp. 24–32.

[11] M. Bauer, D. Dengler, TrIAs—An architecture for trainable information assistants, in: Proc. Workshop on AI and Information Integration, AAAI-98, Madison, WI, 1998.
[12] G. Benedek, A. Itai, Learnability by fixed distributions, in: Proc. 1st Workshop Computational Learning Theory, 1988, pp. 80–90.
[13] M. Bowman, P. Danzig, U. Manber, F. Schwartz, Scalable Internet discovery: Research problems and approaches, Comm. ACM 37 (8) (1994) 98–107.
[14] J. Bradshaw (Ed.), Intelligent Agents, MIT Press, Cambridge, MA, 1997.
[15] P. Buneman, Semistructured data, in: Proc. 16th ACM Symp. Principles of Database Systems, Tucson, AZ, 1997.
[16] M. Califf, R. Mooney, Relational learning of pattern-match rules for information extraction, in: Proc. Workshop in Natural Language Learning, Conference Assoc. Computational Linguistics, 1997.
[17] M. Califf, R. Mooney, Relational learning of pattern-match rules for information extraction, in: Proc. AAAI-99, Orlando, FL, 1999.
[18] M. Carey, L. Haas, P. Schwarz, M. Arya, W. Cody, R. Fagin, M. Flickner, A. Luniewski, W. Niblack, D. Petkovic, J. Thomas, J. Williams, E. Wimmers, Towards heterogeneous multimedia information systems: The Garlic approach, in: Proc. 5th Internat. Workshop of Research Issues in Data Engineering: Distributed Object Management, 1995, pp. 124–131.
[19] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, J. Widom, The TSIMMIS project: Integration of heterogeneous information sources, in: Proc. 10th Meeting of the Information Processing Soc. of Japan, 1994, pp. 7–18.
[20] W. Cohen, Integration of heterogeneous databases without common domains using queries based on textual similarity, in: Proc. ACM SIGMOD Conference on Management of Data, Seattle, WA, 1998.
[21] C. Collet, M. Huhns, W. Shen, Resource integration using a large knowledge base in CARNOT, IEEE Computer (1991).
[22] J. Cowie, W. Lehnert, Information extraction, Comm. ACM 39 (1) (1996) 80–91.
[23] R. Doorenbos, O. Etzioni, D. Weld, A scalable comparison-shopping agent for the World-Wide Web, in: Proc. Internat. Conference on Autonomous Agents, Marina del Rey, CA, 1997, pp. 39–48.
[24] O. Duschka, A. Levy, Recursive plans for information gathering, in: Proc. IJCAI-97, Nagoya, Japan, 1997.
[25] D. Embley, D. Campbell, Y. Jiang, Y.-K. Ng, R. Smith, S. Liddle, D. Quass, A conceptual-modeling approach to extracting data from the web, in: Proc. Internat. Conference Conceptual Modeling, 1998.
[26] O. Etzioni, Moving up the information food chain: Softbots as information carnivores, in: Proc. AAAI-96, Portland, OR, 1996.
[27] O. Etzioni, The World Wide Web: Quagmire or gold mine?, Comm. ACM 37 (7) (1996) 65–68.
[28] O. Etzioni, K. Golden, D. Weld, Sound and efficient closed-world reasoning for planning, Artificial Intelligence 89 (1–2) (1997) 113–148.
[29] O. Etzioni, P. Maes, T. Mitchell, Y. Shoham (Eds.), Working Notes of the AAAI Spring Symposium on Software Agents, AAAI Press, Menlo Park, CA, 1994.
[30] O. Etzioni, D. Weld, A softbot-based interface to the Internet, Comm. ACM 37 (7) (1994) 72–76.
[31] D. Fensel, M. Erdmann, R. Studer, Ontobroker: The very high idea, in: Proc. 11th Internat. Florida AI Research Conference, 1998.
[32] D. Florescu, L. Rashid, P. Valduriez, Using heterogeneous equivalences for query rewriting in multi-database systems, in: Proc. Cooperative Information Systems, 1995.
[33] D. Freitag, Information extraction from HTML: Application of a general machine learning approach, in: Proc. AAAI-98, Madison, WI, 1998.
[34] D. Freitag, Machine learning for information extraction in informal domains, Ph.D. Thesis, Carnegie Mellon University, Pittsburgh, PA, 1998.
[35] D. Freitag, Multistrategy learning for information extraction, in: Proc. Internat. Conference Machine Learning, Madison, WI, 1998.
[36] M. Friedman, D. Weld, Efficiently executing information-gathering plans, in: Proc. IJCAI-97, Nagoya, Japan, 1997, pp. 785–791.
[37] J.-B. Gruser, L. Raschid, M. Vidal, L. Bright, Wrapper generation for web accessible data sources, in: Proc. Conference Cooperative Information Systems, 1998.
[38] A. Gupta (Ed.), Integration of Information Systems: Bridging Heterogeneous Databases, IEEE Press, 1989.

[39] J. Hammer, H. Garcia-Molina, J. Cho, R. Aranha, A. Crespo, Extracting semistructured information from the web, in: Proc. Workshop on Management of Semistructured Data, 1997.
[40] J. Hobbs, The generic information extraction system, in: Proc. 4th Message Understanding Conference, 1992.
[41] C. Hsu, M. Dung, Generating finite-state transducers for semistructured data extraction from the web, J. Information Systems 23 (8) (1998).
[42] G. Huck, P. Frankhauser, K. Aberer, E. Neuhold, Jedi: Extracting and synthesizing information from the web, in: Proc. Conference Cooperative Information Systems, 1998.
[43] S. Huffman, Learning information extraction patterns from examples, in: S. Wermter, E. Riloff, G. Scheler (Eds.), Connectionist, Statistical and Symbolic Approaches to Learning for Natural Language Processing, Springer, Berlin, 1996.
[44] A. Knoblock, A. Levy, O. Duschka, D. Florescu, N. Kushmerick (Eds.), Proc. Workshop on AI and Information Integration, AAAI-98, Madison, WI, 1998.
[45] C. Knoblock, Planning, executing, sensing, and replanning for information gathering, in: Proc. IJCAI-95, Montreal, Quebec, 1995, pp. 1686–1693.
[46] C. Knoblock, S. Minton, J. Ambite, N. Ashish, P. Modi, I. Muslea, A. Philpot, S. Tejada, Modeling web sources for information integration, in: Proc. AAAI-98, Madison, WI, 1998.
[47] D. Knuth, J. Morris, V. Pratt, Fast pattern matching in strings, SIAM J. Computing 6 (2) (1977) 323–350.
[48] D. Konopnicki, O. Schmeuli, W3QL: A query system for the World Wide Web, in: Proc. Internat. Conference Very Large Data Bases, Zurich, Switzerland, 1995.
[49] B. Krulwich, The BargainFinder agent: Comparison price shopping on the Internet, in: J. Williams (Ed.), Bots and Other Internet Beasties, Chapter 13, SAMS.NET, 1996.
[50] N. Kushmerick, Wrapper induction for information extraction, Ph.D. Thesis, University of Washington, Seattle, WA, 1997.
[51] N. Kushmerick, Regression testing for wrapper maintenance, in: Proc. AAAI-99, Orlando, FL, 1999, pp. 74–79.
[52] N. Kushmerick, D. Weld, R. Doorenbos, Wrapper induction for information extraction, in: Proc. IJCAI-97, Nagoya, Japan, 1997, pp. 729–735.
[53] A. Levy, Obtaining complete answers from incomplete databases, in: Proc. 22nd VLDB Conference, Bombay, India, 1996.
[54] A. Levy, C. Knoblock, S. Minton, W. Cohen, Trends and controversies: Information integration, IEEE Intelligent Systems 13 (5) (1998).
[55] A. Levy, A. Rajaraman, J. Ordille, Query-answering algorithms for information agents, in: Proc. AAAI-96, Portland, OR, 1996.
[56] S. Luke, L. Spector, D. Rager, J. Hendler, Ontology-based web agents, in: Proc. First Internat. Conference on Autonomous Agents, Marina del Rey, CA, 1997.
[57] A. Mendelzon, G. Mihaila, T. Milo, Querying the World Wide Web, Internat. J. Digital Libraries 1 (1) (1997) 54–67.
[58] T. Mitchell, The need for biases in learning generalizations, Technical Report CBM-TR-117, Department of Computer Science, Rutgers University, 1980.
[59] A. Monge, C. Elkan, The field matching problem: Algorithms and applications, in: Proc. 2nd Internat. Conference Knowledge Discovery and Data Mining, 1996.
[60] I. Muslea, S. Minton, C. Knoblock, A hierarchical approach to wrapper induction, in: Proc. 3rd Internat. Conference Autonomous Agents, Seattle, WA, 1999.
[61] M. Perkowitz, O. Etzioni, Category translation: Learning to understand information on the Internet, in: Proc. IJCAI-95, Montreal, Quebec, 1995, pp. 930–936.
[62] E. Riloff, Automatically constructing a dictionary for information extraction tasks, in: Proc. AAAI-93, Washington, DC, 1993, pp. 811–816.
[63] A. Sahuguet, F. Azavant, WysiWyg Web wrapper factory, in: Proc. World Wide Web Conference, Toronto, Ont., 1999.
[64] D. Schuurmans, R. Greiner, Practical PAC learning, in: Proc. IJCAI-95, Montreal, Quebec, 1995, pp. 1169–1175.
[65] E. Selberg, May 1997, Personal communication.

[66] E. Selberg, O. Etzioni, The metacrawler architecture for resource aggregation on the web, IEEE Expert 12 (1) (1997) 8–14.
[67] J. Shakes, M. Langheinrich, O. Etzioni, Dynamic reference sifting: A case study in the homepage domain, in: Proc. 6th World Wide Web Conference, Santa Clara, CA, 1997.
[68] A. Smeaton, F. Crimmins, Relevance feedback and query expansion for searching the web: A model for searching a digital library, in: Proc. 1st European Conference Digital Libraries, 1997, pp. 99–112.
[69] D. Smith, M. Lopez, Information extraction for semi-structured documents, in: Proc. Workshop on Management of Semistructured Data, 1997.
[70] S. Soderland, Learning to extract text-based information from the world web, in: Proc. 3rd Internat. Conference Knowledge Discovery and Data Mining, 1997.
[71] S. Soderland, Learning information extraction rules for semi-structured and free text, Machine Learning 34 (1999).
[72] S. Soderland, D. Fisher, J. Aseltine, W. Lehnert, CRYSTAL: Inducing a conceptual dictionary, in: Proc. IJCAI-95, Montreal, Quebec, 1995, pp. 1314–1321.
[73] L. Valiant, A theory of the learnable, Comm. ACM 27 (11) (1984) 1134–1142.
[74] D. Weld, April 1998, Personal communication.
[75] G. Wiederhold, Intelligent Information Integration, Kluwer, Dordrecht, 1996.