IJCAI-97

Wrapper Induction for Information Extraction

Nicholas Kushmerick **Daniel S. Weld**
Department of Computer Science & Engineering
University of Washington, Box 352350
Seattle WA 98195–2350 USA
{nick, weld}@cs.washington.edu

Robert Doorenbos
NETbot, Inc.
4530 Union Bay Pl. NE, Suite 208
Seattle WA 98105 USA
bobd@netbot.com

Abstract

Many Internet information resources present relational data—telephone directories, product catalogs, *etc.* Because these sites are formatted for people, mechanically extracting their content is difficult. Systems using such resources typically use hand-coded *wrappers*, procedures to extract data from information resources. We introduce *wrapper induction*, a method for automatically constructing wrappers, and identify HLRT, a wrapper class that is efficiently learnable, yet expressive enough to handle 48% of a recently surveyed sample of Internet resources. We use PAC analysis to bound the problem's sample complexity, and show that the system degrades gracefully with imperfect labeling knowledge.

1 Introduction

The Internet contains many sources of relational data. For example, when queried with a name, email address services return $\langle$name, email$\rangle$ pairs. But because these sites are designed for people, the content is formatted for human browsing (*e.g.* an HTML page), rather than for use by a program. Therefore, software systems using such resources (*e.g.*, heterogeneous database systems [Chawathe *et al.*, 1994; Arens *et al.*, 1996] or software agents [Etzioni & Weld, 1994; Kirk *et al.*, 1995]) must translate query responses to relational form.

Wrappers are commonly used as such translators. A wrapper is a procedure, specific to a single information resource, that translates a query response to relational form. Wrappers are typically hand-coded; unfortunately, hand-coding is tedious and error-prone.

We seek an automated solution to this problem of constructing wrappers. Natural language processing has been used for similar information-extraction tasks; see [Cowie & Lehnert, 1996] for a recent summary. But many information resources do not exhibit the rich grammatical structure such techniques are designed to exploit. Moreover, linguistic approaches tend to be slow, while ideally wrappers should execute quickly, because they are used on-line to satisfy users' queries.

Wrapper induction is a new technique for automatically constructing wrappers. Our system learns a wrapper by generalizing from example query responses. A PAC model bounds the number of examples needed to generate a satisfactory wrapper. The inductive algorithm requires an oracle to label examples; we solve this *labeling problem* [Etzioni, 1996] by composing oracles from heuristic knowledge, and we demonstrate that our system degrades gracefully with imperfect heuristics.

We identify HLRT, a class of wrappers which is efficiently learnable, yet expressive enough to handle numerous actual Internet information resources. HLRT is designed for resources that display their content in a tabular layout. HLRT wrappers scan their input for substrings that delimit the information to be extracted. Though our focus is on Internet resources, these learned delimiters need not be HTML tags, but can be arbitrary text.

HLRT corresponds essentially to a class of finite-state automata, so wrapper induction is similar to FSA induction (*e.g.*, [Angluin, 1982]). Since FSAs run in linear time, HLRT satisfies the desire that wrappers be fast. However, since wrappers are used for parsing (rather than just classification), the learned FSA must have a specific state topology. Existing FSA induction algorithms do not make such guarantees, so we have developed a new algorithm targeted specifically at HLRT.

We make the following contributions. *First*, we formalize the wrapper construction problem as that of inductive generalization. *Second*, we identify the HLRT wrapper class, which is efficiently learnable yet reasonably expressive. *Third*, we show how to compose the required oracle from (possibly imperfect) heuristics.

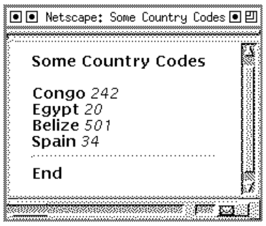
We proceed as follows. In Sec. 2, we describe wrappers. In Sec. 3, we cast wrapper construction as inductive generalization; we then spell out this framework by describing how to learn HLRT (Sec. 4), applying the PAC framework (Sec. 5), and presenting a modular approach to building oracles (Sec. 6). Sec. 7 provides an empirical evaluation of our approach. Finally, Sec. 8 describes related work.

2 Wrappers

A wrapper is a procedure for extracting tuples from a particular information source. Formally, a wrapper is a function from a page[1] to the set of tuples it contains.

[1] We use the term *page* generically, referring to whatever query response is returned by an information resource.

(a)



(b)

```
<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>
```

(c)

ExtractCCs(page P)
  skip past first occurence of <P> in P
  while next <B> is before next <HR> in P
    for each $\langle \ell_k, r_k \rangle \in \{\langle$<B>, </B>$\rangle, \langle$<I>, </I>$\rangle\}$
      skip past next occurence of ℓ_k in P
      extract attribute from P to next occurence of r_k
  return extracted tuples

(d)

ExecuteHLRT($\langle h, t, \ell_1, r_1, \ldots, \ell_K, r_K \rangle$, page P)
  skip past first occurence of h in P
  while next ℓ_1 is before next t in P
    for each $\langle \ell_k, r_k \rangle \in \{\langle \ell_1, r_1 \rangle, \ldots, \langle \ell_K, r_K \rangle\}$
      skip past next occurence of ℓ_k in P
      extract attribute from P to next occurence of r_k
  return extracted tuples

Figure 1: *(a) A fictitious example query response page; (b) the* HTML *from which it was rendered; (c) a wrapper for this resource; and (d) the* HLRT *wrapper template.*

For example, consider a fictitious information resource that provides a tabular list of countries and their telephone country codes. Suppose that in response to a query, the resource responds as displayed in Fig. 1(a), which was rendered from the HTML shown in (b). Many kinds of wrappers could be written; we consider wrappers that use the positions of particular strings to delimit the extracted text. From (b), it appears that this resource renders tuples by surrounding countries with <B> and </B>, and country codes with <I> and </I>.[2] So one candidate wrapper relies on these four delimiters.

But note that this simple left-right (LR) strategy fails, because not all occurrences of <B>$\cdots$</B> indicate a country. However, the string <P> can be used to distinguish the head of the page from the tuples proper. Similarly, <HR> separates the last tuple from the tail. Fig. 1(c) shows ExtractCCs, a wrapper based on this more sophisticated head-left-right-tail (HLRT) approach.

We focus on wrappers that are structurally similar to ExtractCCs. Fig. 1(d) shows the template for HLRT wrappers. Note that instantiating the template with the six strings $\langle$<P>,<HR>,<B>,</B>,<I>,</I>$\rangle$ yields ExtractCCs.

[2]Note that though this example involves HTML tags such as <B>, our system does not require the use of HTML; any text fragment (such as just B>) that reliably delimits the attribute is acceptable.

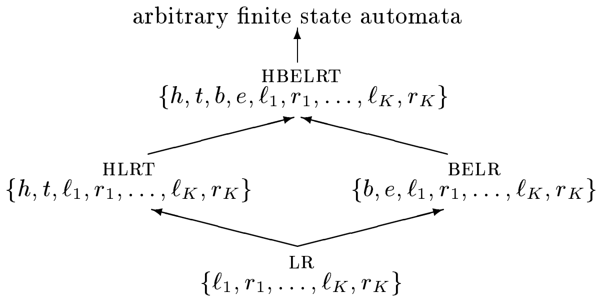


Figure 2: *A partial hierarchy of wrapper biases. Arrows indicate that one bias is more expressive than another.*

Formally, an HLRT wrapper for a domain with K attributes per tuple is encoded as a vector of $2K+2$ strings $\langle h, t, \ell_1, r_1, \ldots, \ell_K, r_K \rangle$. One string ($h$) marks the end of the header, another (t) marks the start of the tail, and two strings (ℓ_k and r_k) delimit each of the K attributes.

We focus on HLRT, but alternatives abound. Fig. 2 illustrates a partial hierarchy of wrapper classes. LR is less expressive than HLRT; for example, the country/code resource can be wrapped by HLRT but not LR. BELR's b and e mark the beginning and end of each *tuple*, rather than page's body. In the extreme, arbitrary finite-state automata could be used as wrappers. In [Kushmerick, 1997], we analyze this hierarchy in detail.

3 Constructing wrappers by induction

The *wrapper construction* problem is the following: given a supply of example query responses, learn a wrapper for the information resource that generated them. For the country/code resource, the problem is to induce the ExtractCCs procedure, given a supply of HTML pages similar to that shown in Fig. 1(b).

Induction thus provides a natural framework for formalizing wrapper construction. Induction is the task of generalizing from labeled examples to a hypothesis, a function for labeling instances. For our problem:

Instances correspond to pages—*e.g.* Fig. 1(b).

Labels correspond to pages' tuples—*e.g.* the example page is labeled as containing $\{\langle$Congo,242$\rangle$, $\langle$Egypt,20$\rangle$, $\langle$Belize,501$\rangle$, $\langle$Spain,34$\rangle\}$.

Hypotheses correspond to HLRT wrapper template parameters—*e.g.* $\langle$<P>,<HR>,<B>,</B>,<I>,</I>$\rangle$ is the encoding of ExtractCCs.

Oracles correspond to sources of example query responses and their labels. We split the traditional oracle (which returns a single labeled instance) into two parts. PageOracle generates example pages, and LabelOracle produces correct labels for these instances. PageOracle is specific to a particular information resource, while LabelOracle is composed from heuristics that are reusable across domains.

PAC analysis is used to terminate the learning process, so the system takes as input accuracy (ϵ) and

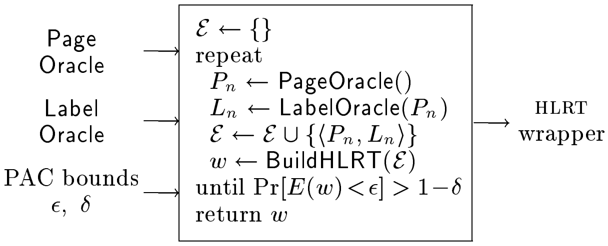


Figure 3: *The wrapper induction algorithm.*

confidence (δ) parameters.

With this framework in place, we now present the wrapper induction algorithm; see Fig. 3. Wrapper induction proceeds by accumulating a set $\mathcal{E}$ of labeled example pages. On each iteration, BuildHLRT is called with $\mathcal{E}$, which returns wrapper w. Learning stops when w satisfies the PAC bound. In Secs. 4–6, we describe the algorithm's main components:

BuildHLRT constructs an HLRT wrapper from a set of labeled example pages; see Sec. 4.

$\Pr[E(w) < \epsilon] > 1-\delta$ is a PAC-theoretic termination condition, testing whether enough examples have been seen to be confident that a satisfactory wrapper has been learned; see Sec. 5.

LabelOracle is a function from a page to a label. In Sec. 6 we describe how to compose a correct labeling oracle from (possibly imperfect) heuristic knowledge.

4 Building HLRT wrappers

BuildHLRT takes as input a set of labeled pages, and returns an HLRT wrapper that is *consistent* with each labeled page. A wrapper is consistent with a labeled page if it generates the label for the page. Fig. 4 shows the BuildHLRT algorithm.

BuildHLRT reasons about the conditions that must hold if wrapper $\langle h, t, \ell_1, r_1, \ldots, \ell_K, r_K \rangle$ is to be consistent. For example, in Fig. 1(b), the string <I> is a valid value for ℓ_2, because <I> actually precedes every instance of the second attribute. Such constraints apply to each r_k, and to each ℓ_k for $k > 1$.

BuildHLRT is complicated by the fact that ℓ_1, t, and h interact. For example, to determine whether <B> is acceptable as ℓ_1 (even though the head and tail contain <B>), BuildHLRT must find an h and t such that <B> does in fact reliably mark the start of the first attribute. In this case, $h =$ <P> and $t =$ <HR> are satisfactory. Lines *(a–d)* in Fig. 4 capture the constraints that ℓ_1, t, and h must satisfy. BuildHLRT examines all possible combinations of ℓ_1, h, and t, stopping when it finds values that jointly satisfy these constraints.

To summarize, BuildHLRT iterates over all choices for the $2K+2$ delimiters, stopping when a consistent wrapper is encountered. BuildHLRT's search is made more efficient by decomposing the constraint satisfaction problem into three independant subproblems: finding values for (1) the r_k; (2) the ℓ_k ($k > 1$); and (3) h, t, and ℓ_1.

```
BuildHLRT(labeled pages ℰ = {..., ⟨P_n, L_n⟩, ...})
  Note that each label L_n partitions page P_n into its
  attributes, separated by the strings between tuples
  and between the K attributes within a tuple.
 for k = 1 to K
   r_k ← any common prefix of the strings following each
        (but not contained in any) attribute k
 for k = 2 to K
   ℓ_k ← any common suffix of the strings preceding each
        attribute k
 for each common suffix ℓ_1 of the pages' heads
  for each common substring h of the pages' heads
   for each common substring t of the pages' tails
    if (a) h precedes ℓ_1 in each of the pages' heads; and
       (b) t precedes ℓ_1 in each of the pages' tails; and
       (c) t occurs between h and ℓ_1 in no page's head; and
       (d) ℓ_1 doesn't follow t in any inter-tuple separator
    then return ⟨h, t, ℓ_1, r_1, ..., ℓ_K, r_K⟩
```

Figure 4: *The* BuildHLRT *algorithm.*

In [Kushmerick, 1997], we prove that: *(1)* BuildHLRT is *sound* (if BuildHLRT returns a wrapper, then it is consistent) and *complete* (if a consistent wrapper exists, BuildHLRT finds it); and *(2)* under reasonable assumptions, BuildHLRT runs in time $O(KNMS^3)$, where each tuple has K attributes, the shortest of the N example pages has length S, and M is maximum number of tuples in any single example.

Appendix A formally describes the conditions under which an HLRT wrapper is consistent with a labeled page.

5 PAC analysis

PAC analysis answers the question, 'How many examples must a learner see to be confident that its hypothesis is good enough—*i.e.*, to be *probably approximately correct*?'; see [Kearns & Vazirani, 1994] for an introduction. A PAC model defines an error metric over hypothesis: $E(w)$ is the probability that hypothesis w will incorrectly label the next instance. The learning task is then analyzed in order to bound the number of examples which ensure that $\Pr[E(w) > \epsilon] < \delta$, for any given *accuracy* parameter ϵ and *confidence* parameter δ. In [Kushmerick, 1997], we prove the following theorem.

Theorem 1 (**HLRT sample complexity**) *Suppose* BuildHLRT($\mathcal{E}$) *returns wrapper w, where $\mathcal{E}$ contains collectively T tuples, each with K attributes. If*

$$\left(1 - 2\left(1 - \frac{\epsilon}{2}\right)^T\right)^{2K} \left(1 - 2\left(1 - \frac{\epsilon}{2}\right)^{|\mathcal{E}|}\right)^2 > 1-\delta,$$

then $\Pr[E(w) > \epsilon] < \delta$, for any $0 < \epsilon < 1$ and $0 < \delta < 1$.

For example, with $\epsilon = \delta = 0.1$, $K = 4$, and an average of 5 tuples/page, BuildHLRT must examine at least 72 examples to satisfy the PAC criteria.

This bound is relatively tight compared to typical PAC results. For example, the number of possible HLRT wrappers is infinite, but our bound does not depend on the number of wrappers. Thus clearly the stated bound is tighter than obtainable under simple PAC models (*e.g.*, [Valiant, 1984; Blumer *et al.*, 1987]), in which sample complexity grows with the number of hypotheses. The bound is also tighter than obtainable using Vapnik-Chervónenkis analysis [Haussler, 1988]. To understand these results, recall that BuildHLRT is essentially computing common prefixes and suffixes of sets of strings, which are highly constrained after relatively few examples; see [Kushmerick, 1997] for a detailed discussion.

6 Composing oracles

A key to induction is an oracle that labels examples. So far, we have assumed that LabelOracle is provided as input. We now describe how to compose LabelOracle from modular heuristic knowledge, which we call *recognizers*. A recognizer finds *instances* of a particular attribute on a page. For example, a country name recognizer would identify the four countries contained in Fig. 1(b)'s HTML. These recognized instances are then *corroborated* to label the entire page. For example, given a recognizer for countries and another for country codes, corroboration produces an oracle that labels pages containing pairs of these attributes.

Corroboration is trivial if each recognizer is perfect.[3] But an important feature of our approach is that it handles imperfect recognizers. Recognizers are either *perfect* (accept all positive instances and reject all negative instances of their target attribute), *incomplete* (reject all negative instances but reject some positive instances), *unsound* (accept all positive instances but accept some negative instances), or *unreliable* (reject some positive instances and accept some negative instances).

We require that each recognizer be annotated with the kind of error it makes. We expect this annotation to be natural for many kinds of recognizers. For example, a company name recognizer based on Fortune-500 data is incomplete, while a country code recognizer accepting any digit sequence is unsound. The intent is that recognizers are reusable across domains; a company name recognizer, for example, can be used with any information resource displaying companies.

Recall that LabelOracle is a function from a page to a label for the page. A label is an array, where rows correspond to tuples, and columns are attributes. A recognizer is a function from a page to a set of *instances* (subsequences of the page). The set of recognized instances is a column of the overall label array. The *corroboration problem*, then, is to build the entire label array from the individual columns. Note that the attributes' ordering within tuples is not part of the input; the corroboration algorithm must determine this ordering.

[3]Wrappers are needed even with perfect recognizers, because recognizers might be slow, while wrappers must be fast.

(a)

Ctry
incomp
10–15
50–55

\+ (b)

Code
perfect
18–20
38–40
58–60

\+ (c)

Cap
unsound
5–7
19–25
22–28
42–48
44–49
59–65
62–68
70–75

$\Rightarrow$ (d)

Ctry	**Code**	**Cap**
10–15	18–20	22–28
?	38–40	42–48⊕44–49
50–55	58–60	62–68⊕70–75

Figure 5: *Corroborating (a–c) yields (d).*

The basic idea of corroboration is that the location of some instances greatly constrains the possible location of others. Suppose recognizer A is unsound and identifies an instance at position 10–20, while perfect recognizer B finds an instance at 14–16. Since attributes never overlap, the A at 10–20 must be a false positive (FP) and thus must be ignored, while the B at 14–16 is a true positive (TP).

In the remainder of this section, we describe corroboration by walking through an example, present the Corrob corroboration algorithm, and describe how our work is extended to handle imperfect recognizers.

Example. Fig. 5 extends the country/code example to include an additional attribute, the country's capital.[4] Corroboration begins by noting the type of error made by each recognizer: in this simple example, assume the recognizer for the codes is perfect, but the country recognizer is incomplete and the capital recognizer is unsound.

Next, note that since the **Code** recognizer is perfect, all **Code** instances are TPs. Thus **Code** can be simply copied to the label array. The incomplete **Ctry** column is almost as easy: it is copied verbatim, but **Code** is used to align each **Ctry** instance. This leaves a "hole" in the **Ctry** column. Next, the corroborator processes the unsound **Cap** column. 5–7 must be a FP, because were it a TP, **Code** would have included an additional instance prior to 5. Next, since 19–25 overlaps with **Code**'s 18–20, 19–25 must be a FP. Since 22–28 is the only remaining possibility for the first tuple's **Cap** instance, 22–28 must be a TP. However, for the second tuple, there is no way to choose between 42–48 and 44–49: one must be a TP and the other a FP, but there is no way to decide. Corroboration thus uses 42–48⊕44–49, indicating that *exactly one* of the two instances is a TP. Finally, corroboration rejects 59–65 because it overlaps with 58–60.

The Corrob algorithm. Fig. 6 shows Corrob, an algorithm for corroborating imperfect recognizers, for the case when at least one recognizer is perfect.[5] As indicated in the example, Corrob builds the label array by

[4]The indices in this example do *not* match Fig. 1(b).

[5]People can be used as perfect recognizers, though we seek to automate wrapper construction as much as possible.

Corrob(recognizers $\{\ldots, R_k, \ldots\}$, instances $\{\ldots, I_j^k, \ldots\}$)
Notation: I_j^k *is the* $j^{\underline{\text{th}}}$ *instance recognized by* R_k.
$A \leftarrow$ BuildArray($\{I_j^k : R_k$ is perfect or incomplete$\}$)
for each I_j^k such that R_k is unsound or unreliable
if I_j^k is possibly a TP (based on the TPs in A), then
$m \leftarrow$ RowOf(I_j^k, A)
$A_{m,k} \leftarrow A_{m,k} \oplus I_j^k$
return A
BuildArray(necessarily TP instances $\{\ldots, I_j^k, \ldots\}$)
Build array with each I_j^k installed in the correct cell.
RowOf(possibly TP instance I_j^k, array A)
Determine the row m of A to which I_j^k belongs, which is always determined when at least one R_k is perfect.

Figure 6: *The* Corrob *algorithm.*

first installing instances recognized by perfect or incomplete recognizers. These TPs are then used to categorize the remaining instances as either *necessarily* FPs (meaning they can be ignored), or *possibly* TPs (meaning they are inserted using '$\oplus$'). In [Kushmerick, 1997], we describe Corrob in more detail and prove that it is correct.

Handling mistakes. Note that Corrob's output might contain attributes that are *missing* ('?' indicates attributes falsely rejected by their recognizers) or *ambiguous* ('$\oplus$' indicates under-constrained attributes). But BuildHLRT assumes that LabelOracle produces a perfect label. We now describe how to extend our work to handle this discrepancy.

Missing attributes require only minor changes to BuildHLRT: the algorithm must simply generalize from fewer examples. For example, recall the original country/code resource in Fig. 1. Suppose that corroboration yields a label that is correct except that the country `Congo` is missing. In this case, when learning ℓ_1, the algorithm generalizes from just the three occurrences of `</I><BR>⇓<B>`[6] that precede the recognized instances of the first attribute, and the algorithm may well fail to generate the correct wrapper. Only if the first country name is correctly recognized on a subsequent example page will BuildHLRT realize that ℓ_1 must be a suffix of `<HTML><TITLE>Some Country···Codes</B><P>⇓<B>` as well as of `</I><BR>⇓<B>`.

The PAC model must also be extended to accommodate missing attributes. To do so, we generalize Thm. 1 so that, instead of assuming exactly N head and tail and T left and right delimiter examples, the model counts the actual numbers, based on the non-missing attributes. So, in the previous example, there are four examples for l_2, r_2 and t, but only three for l_1 and r_1, and zero for h.

Ambiguity requires more substantial changes. There are two kinds of ambiguity. First, as described earlier, Corrob uses '$\oplus$' to indicate that more than one recognized instance is consistent with a particular cell in the label array. Second, recall that Corrob must determine the ordering of attributes within tuples. But the recognized instances may be consistent with more than one ordering. For example, in Fig. 5, a valid label exists for the ordering (**Cap**, **Ctry**, **Code**) as well as for (**Ctry**, **Code**, **Cap**). (We previously ignored ordering ambiguity to simplify the presentation of Corrob.)

We extend BuildHLRT to handle both types of ambiguity as follows. An ambiguous label actually corresponds to a set of unambiguous labels, one for each way to resolve each ambiguity. Exactly one such label is correct; the rest either contain FPs or corresponds to an incorrect attribute ordering. Faced with ambiguity, BuildHLRT iterates over each possibility, stopping when a wrapper can be induced.

Clearly the number of such possibilities grows exponentially in the number of ambiguities. In practice this growth is tolerable for both unsound and incomplete recognizers, even with very high error rates; see Sec. 7. However, Corrob is impractical for unreliable recognizers, because the number of ambiguities grows quickly with an unreliable recognizer's error rate.

We extend the PAC model to handle ambiguous labels by accounting for situations in which BuildHLRT considers an incorrect way to resolve a label's ambiguity, and yet a consistent wrapper exists anyway. For example, in Fig. 5(d), if the second tuple's **Cap** attribute is actually 42–48 rather than 44–49, but BuildHLRT tries 44–49 first and successfully finds a wrapper, then BuildHLRT has probably made a mistake. Similarly, if BuildHLRT considers the ordering (**Cap**, **Ctry**, **Code**) before (**Ctry**, **Code**, **Cap**), then BuildHLRT is probably wrong if it finds a consistent wrapper.

We model this effect by assuming that such a situation happens with probability at most μ per opportunity, and thus the left-hand side of the bound in Thm. 1 is multiplied by $(1-\mu)^R$, where R is the number of opportunities that a mistake of this type could have occurred as BuildHLRT was enumerating the possible labels. In practice, we find that μ is extremely close to zero and R is relatively small, and thus ambiguity has a negligible effect on the PAC results. In [Kushmerick, 1997], we compare this noise model to others in the PAC literature.

7 Empirical evaluation

In this section, we present preliminary evidence demonstrating the feasibility of HLRT learning. Our Lisp implementation requires between 4 and 40 SGI Indy CPU seconds per example page, depending on the domain. Normalizing for the number of attributes (K) and the size of the example pages, our system requires about 0.21 CPU sec. per attribute per KB of example data.

Our first experiment verifies the utility of the HLRT bias. Learnability aside, can a significant fraction of interesting information resources be wrapped by HLRT? We surveyed 100 Internet resources selected randomly from an independent organization's index (`search.com`), and found that 48% can be wrapped by HLRT. We take this result to be evidence that HLRT is genuinely useful.

Our second experiment measures the robustness of the

[6]'⇓' indicates a carriage return character.

system to the recognizers' error rates. We tested our system on (i) the OKRA email service, `okra.ucr.edu/okra`; and (ii) the BIGBOOK telephone directory, `bigbook.com`.

By hand, we constructed perfect recognizers for each attribute; OKRA has four attributes and BIGBOOK has six. As a baseline, we ran our system with these perfect recognizers. We then increased the error rates up to 40% (creating both incomplete and unsound recognizers for each attribute) and increased the number of imperfect recognizers from zero until all but one were imperfect.[7] We tested our system using two termination conditions: (a) we ran the system until the PAC criteria was satisfied (for $\epsilon = \delta = 0.1$); and (b) we required that the learned wrapper be 100% correct on a suite of test pages.

Fig. 7 shows the number of pages needed to induce a wrapper, as a function of the error rate, for each termination condition, and for each domain. Each curve within a graph represents a different number of imperfect recognizers. For example, the points marked "perfect" represent trials in which all recognizers are perfect, while the points marked "30% error rate of each recognizer" on the "2 imperfect recognizers" curves indicate trials in which two of the recognizers are imperfect (yielding either 30% FPs or 30% FNs) while the remaining recognizers are perfect. Thus in each graph, increasing the abscissa or examining curves with additional imperfect recognizers corresponds to trials in which the recognizers make more mistakes.

Figs. 7(i–ii.b) indicate that, from a practical perspective, relatively few examples are needed before the system learns the correct wrapper; across all conditions, about 4.9 examples suffice for OKRA and 29 for BIGBOOK. We conclude that the number of examples required is small enough that HLRT wrapper induction is practical, even for extremely high recognizer error rates.

Figs. 7(i–ii.a) show that the PAC bound is relatively loose. Across all conditions, about 105 examples are needed required to satisfy the PAC criteria. Thus the PAC bound is too loose by about an order of magnitude. We conclude that the current PAC model is too weak to tightly constrain the induction process. Nevertheless, since wrapper construction is intended to be an off-line process, the bound is not so loose as to be useless.

Finally, we have developed WIEN (pronounced "Vienna"), a wrapper induction environment. Using a Web browser, a user shows WIEN an example information resource page, and then uses the mouse to label the page. WIEN then tries to learn a wrapper for the resource. When the user shows WIEN a second example, it uses the learned wrapper to automatically label the new example. The user then corrects any mistakes, and WIEN generalizes from both examples. This process repeats until the user is satisfied. WIEN provides a complete implementation of BuildHLRT, though the user is assumed to label pages perfectly, so WIEN implements neither attribute recognition nor corroboration.

[7]Recall that Corrob is impractical for unreliable recognizers and requires at least one perfect recognizer.

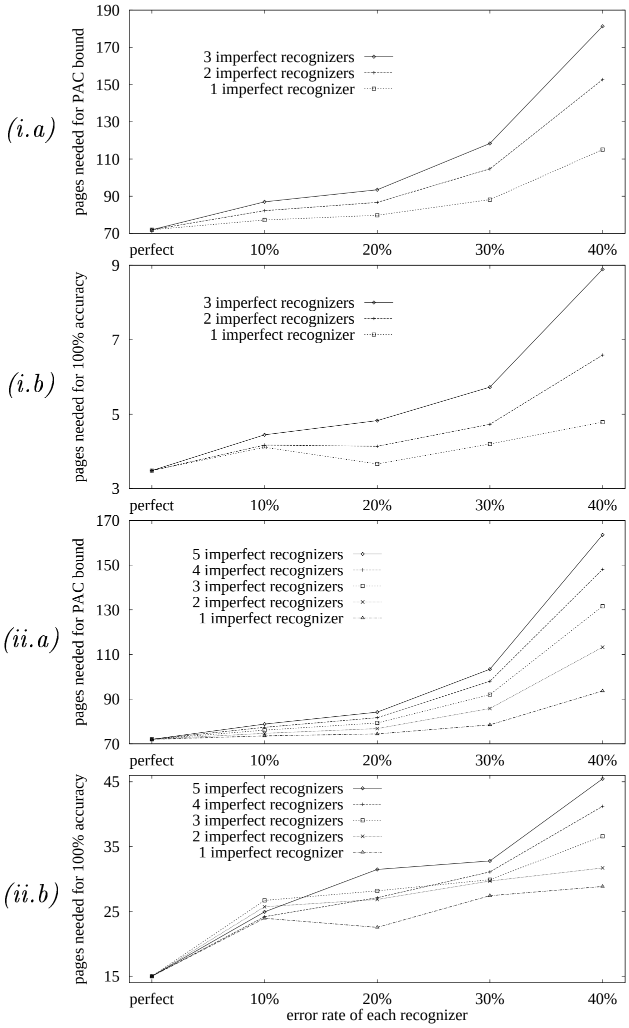


Figure 7: *Effect on learning curve of recognizer error, in the (i)* OKRA *and (ii)* BIGBOOK *domains, for the (a) PAC and (b) 100% accurate termination conditions.*

WIEN and the experimental data are available at `www.cs.washington.edu/homes/nick/research/wrappers`.

8 Related work

As suggested at the outset, wrapper construction is motivated by the software engineering issues involved with deploying software systems that rely on external information resources; examples include [Chawathe *et al.*, 1994; Etzioni & Weld, 1994; Arens *et al.*, 1996; Kirk *et al.*, 1995]. While data interchange protocols (*e.g.* KQML [Finin *et al.*, 1994]) have been proposed to address these issues, they require cooperation on the part of information providers, and such cooperation is rare.

From a formal perspective, in Sec. 1 we discussed the relationship between HLRT and FSA induction.

From an application perspective, our work is similar to [Ashish & Knoblock, 1997]. Their system learns a more expressive wrapper class than HLRT, but relies on many heuristics that are specific to HTML. In contrast, our systems treats HTML tags just as ordinary text. Moreover, their system requires human intervention to correct its mistakes, while our corroboration process is intended to correct mistakes automatically. A second related application is SHOPBOT [Doorenbos *et al.*, 1997]. Though in many respects SHOPBOT is more ambitious, its wrapper language is less expressive than HLRT.

Finally, our recognition knowledge is similar to work on semantically labeling natural text, such as the MUC-6 "Named Entity" task [DARPA, 1995], though relatively little work has been done on corroborating multiple such knowledge sources.

9 Conclusions

Wrapper induction is a new technique for automatically constructing wrappers. We have made three contributions. First, we have formalized the wrapper construction problem as induction. Second, we have defined the HLRT bias, which is efficiently learnable in this framework. Third, we have shown how to use heuristic knowledge to compose the algorithm's oracle. Though our work has involved primarily Internet information resources, we expecte that our results are applicable to similar information-extraction tasks in other domains.

We intend to extend our framework in several ways. In addition to the biases shown in Fig. 2, we want to design wrappers that can handle non-tabular pages, such as pages organized hierarchically. The research issues involve exploring the tradeoff between expressiveness and learnability. We also hope to tighten the PAC model so it is more useful in practice as well as more predictive of observed learning curves.

Acknowledgments. This research is 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. Thanks to the UW CS&E AI group, Steve Minton and David Smith for helpful discussion, and to Boris Bak for help with the `search.com` survey.

A HLRT consistency conditions

In this Appendix, we list the conditions under which HLRT wrapper $w = \{h, t, \ell_1, r_1, \ldots, \ell_K, r_K\}$ outputs label $L = \{\langle\langle b_{1,1}, e_{1,1}\rangle, \ldots, \langle b_{1,K}, e_{1,K}\rangle\rangle, \ldots, \langle\langle b_{M,1}, e_{M,1}\rangle, \ldots, \langle b_{M,K}, e_{M,K}\rangle\rangle\}$ for page P. This notation indicates that that P contains M tuples having K attributes each, where the $k^{\underline{\text{th}}}$ attribute of the $m^{\underline{\text{th}}}$ tuple <u>b</u>egins at index $b_{m,k}$ of P and <u>e</u>nds at $e_{m,k}$. Note that L partitions P as follows ('$\cdot$' indicates concatenation): $P = S_{0,K}\cdot A_{1,1}\cdot S_{1,1}\cdot A_{1,2}\cdot \cdots \cdot S_{1,K-1}\cdot A_{1,K}\cdot S_{1,K}\cdot \cdots \cdot A_{M,1}\cdot S_{M,1}\cdot A_{M,2}\cdot \cdots \cdot S_{M,K-1}\cdot A_{M,K}\cdot S_{M,K}$. The $A_{m,k}$ are the attribute values: $A_{m,k} = P[b_{m,k}, e_{m,k}]$. The $S_{m,k}$ separate the tuples, and attributes within a tuple: $S_{m,k} = P[e_{m,k}, b_{m,k+1}]$ (except that $S_{0,K} = P[0, b_{1,1}]$, $S_{M,K} = P[e_{M,K}, |P|]$, and $S_{m,K} \equiv S_{m+1,0} = P[e_{m,K}, b_{m+1,1}]$).

Under this notation, w is consistent with P and L iff:

1. the ℓ_k immediately precede their attributes
   $(S_{0,K}/h)/\ell_1 \neq \sharp \;\wedge\; \forall_{m,k>1}\, |S_{m,k-1}/\ell_k| = 0$
2. the r_k follow (but don't occur within) their attributes
   $\forall_{m,k}\; r_k\cdot((A_{m,k}\cdot S_{m,k})/r_k) = S_{m,k}$
3. h occurs in the head and t occurs in the tail
   $S_{0,K}/h \neq \sharp \;\wedge\; S_{M,K}/t \neq \sharp$
4. t never precedes ℓ_1 in an inter-tuple separator
   $\forall_{m<M}\; S_{m,K}/t \neq \sharp \;\Rightarrow\; |\ell_1| > |t\cdot(S_{m,K}/t)|$
5. t doesn't occur between h and ℓ_1 in the head
   $(S_{0,K}/h)/t \neq \sharp \;\Rightarrow\; |\ell_1| > |t\cdot((S_{0,K}/h)/t)|$
6. t precedes ℓ_1 in the tail
   $S_{M,K}/\ell_1 \neq \sharp \;\Rightarrow\; |t\cdot(S_{M,K}/t)| > |\ell_1\cdot(S_{M,K}/\ell_1)|$

(where s/s' is the substring of s after the first occurrence of s', with $s/s' = \sharp$ indicating that s doesn't contain s').

References

[Angluin, 1982] D. Angluin. Inference of reversible languages. *J. ACM*, 29(3):741–65, 1982.

[Arens *et al.*, 1996] Y. Arens, C. Knoblock, C. Chee, & C. Hsu. SIMS: Single interface to multiple sources. TR RL-TR-96-118, USC Rome Labs, 1996.

[Ashish & Knoblock, 1997] N. Ashish & C. Knoblock. Semi-automatic wrapper generation for Internet information sources. In *Proc. Cooperative Information Systems*, 1997.

[Blumer *et al.*, 1987] A. Blumer, A. Ehrenfeucht, D. Haussler, & M. Warmuth. Occam's razor. *Information Processing*, 24(6):377–80, 1987.

[Chawathe *et al.*, 1994] 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. IPSJ Conf*, 1994.

[Cowie & Lehnert, 1996] J. Cowie & W. Lehnert. Information extraction. *C. ACM*, 39(1):80–101, 1996.

[DARPA, 1995] DARPA. *Proc. 6th Message Understanding Conference.* Morgan Kaufmann, San Francisco, 1995.

[Doorenbos *et al.*, 1997] R. Doorenbos, O. Etzioni, & D. Weld. A scalable comparison-shopping agent for the World-Wide Web. In *Proc. Autonomous Agents*, 1997.

[Etzioni & Weld, 1994] O. Etzioni & D. Weld. A softbot-based interface to the Internet. *C. ACM*, 37(7):72–6, 1994.

[Etzioni, 1996] O. Etzioni. The World Wide Web: quagmire or gold mine? *C. ACM*, 37(7):65–8, 1996.

[Finin *et al.*, 1994] T. Finin, R. Fritzson, D. McKay, & R. McEntire. KQML: A language and protocol for knowledge and information exchange. In *Knowledge Building and Knowledge Sharing.* Ohmsha and IOS Press, 1994.

[Haussler, 1988] D. Haussler. Quantifying inductive bias. *Artificial Intelligence*, 36(2):177–221, 1988.

[Kearns & Vazirani, 1994] M. Kearns & U. Vazirani. *An introduction to computational learning theory.* MIT, 1994.

[Kirk *et al.*, 1995] T. Kirk, A. Levy, Y. Sagiv, & D. Srivastava. The Information Manifold. In *AAAI Spring Symposium: Information Gathering from Heterogeneous, Distributed Environments*, pp. 85–91, 1995.

[Kushmerick, 1997] N. Kushmerick. *Wrapper Construction for Information Extraction.* PhD thesis, Univ. of Washington, 1997. In preparation.

[Valiant, 1984] L. Valiant. A theory of the learnable. *C. ACM*, 27(11):1134–42, 1984.