Wrapper induction: Efficiency and expressiveness (Extended abstract)

NICHOLAS KUSHMERICK
School of Computer Applications, Dublin City University
nick@compapp.dcu.ie

Abstract

Recently, many systems have been built that automatically interact with Internet information resources. However, these resources are usually formatted for use by people; *e.g.*, the relevant content is embedded in HTML pages. *Wrappers* are often used to extract a resource's content, but hand-coding wrappers is tedious and error-prone. We advocate *wrapper induction*, a technique for automatically constructing wrappers. We have identified several wrapper classes that can be learned quickly (most sites require only a handful of examples, consuming a few CPU seconds of processing), yet which are useful for handling numerous Internet resources (70% of surveyed sites can be handled by our techniques).

Introduction

The Internet presents a stunning variety of on-line information resources: telephone directories, retail product catalogs, weather forecasts, and many more. Recently, there has been much interest in systems (such as software agents (Etzioni & Weld 1994; Kwok & Weld 1996) or information-integration systems (Chawathe *et al.* 1994; Kirk *et al.* 1995)) that automatically access such resources, manipulating their content on a user's behalf.

Unfortunately, this content is usually formatted for people rather than machines, and no provision is made for automating the process. 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.

Fig. 1 provides an example of the sort of information resource with which we are concerned. When the form in (a) is submitted, the resource responds as shown in (b), which was rendered from the HTML source P_{CC} shown in (c). A system seeking information about countries and their country codes must extract from P_{CC} the label L_{CC}, shown in (d). A page's label represents the relevant content. One way to perform this extraction task is to invoke the special-purpose wrapper procedure $\mathsf{ccwrap}_{\mathrm{LR}}$, shown in (e). $\mathsf{ccwrap}_{\mathrm{LR}}$ requires that the site's responses adhere to a uniform formatting convention: countries must be rendered in bold, while country codes must be in italics. $\mathsf{ccwrap}_{\mathrm{LR}}$ operates by scanning the raw HTML document for particular strings ('<B>', '</B>', '<I>' and '</I>') that identify the parts of the raw HTML document to be extracted.

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 construct such a wrapper for each resource. Unfortunately, this hand-coding process is error-prone and time-consuming.

As an alternative, we advocate ***wrapper induction*** ((Kushmerick 1998; 1997; Kushmerick, Weld, & Doorenbos 1997); see also (Ashish & Knoblock 1997)), 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 pairs such as $\langle P_{\mathrm{CC}}, L_{\mathrm{CC}}\rangle$, our wrapper induction algorithm generates $\mathsf{ccwrap}_{\mathrm{LR}}$.

This extended abstract merely summarizes our work. For details, see (Kushmerick 1998; 1997; Kushmerick, Weld, & Doorenbos 1997).

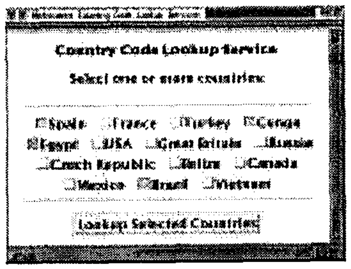
Evaluation

In machine-learning applications, the key to effective learning is to bias the learning algorithm. In this context, biases correspond to particular ***wrapper classes.*** We have identified six such classes. For example, $\mathsf{ccwrap}_{\mathrm{LR}}$ is an instance of the Left-Right (LR) class. For each class $\mathcal{W}$, we present an algorithm $\mathsf{learn}_{\mathcal{W}}$ for learning wrappers in class $\mathcal{W}$. As we are interested in tradeoffs between the classes, we compare them in several ways:

I – EXPRESSIVENESS: How useful are the classes for handling actual Internet resources? To what extent can sites handled by one class be handled by another? We answer these questions using a combination of empirical and analytical techniques.

I-1 – COVERAGE: We conducted a survey of actual Internet sites, to determine which can be handled by each class. To summarize, our classes can handle 70% of the sites in total; see Fig. 9.

(a)



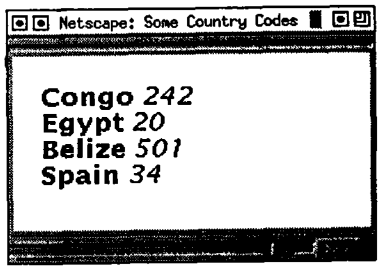
Country Code Lookup Service

Select one or more countries:

Spain France Turkey Congo
Egypt USA Great Britain Russia
Czech Republic Belize Canada
Mexico Israel Vietnam

Lookup Selected Countries

(b)



Netscape: Some Country Codes

Congo *242*
Egypt *20*
Belize *501*
Spain *34*

(c) P_{CC} =

```
<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>
```

(d) $L_{CC} = \left\{ \begin{array}{c} \langle\text{'Congo', '242'}\rangle, \\ \langle\text{'Egypt', '20'}\rangle, \\ \langle\text{'Belize', '501'}\rangle, \\ \langle\text{'Spain', '34'}\rangle \end{array} \right\}$

(e)

```
procedure ccwrap_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 $\ell_k$; 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 pairs $\{\ldots, \langle$country, code$\rangle, \ldots\}$
```

Figure 1: *(a)* A fictitious Internet site providing information about countries and their telephone country codes; *(b)* an example response page; *(c)* the HTML page P_{CC} from which (b) was rendered; *(d)* P_{CC}'s label L_{CC}; and *(e)* the ccwrap_{LR} procedure, which generates L_{CC} from P_{CC}.

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.

II – EFFICIENCY: Our expressiveness results demonstrate the usefulness of our wrapper classes, but can they learned quickly? We decompose this question into two parts: how many examples are needed, and how much processing is required per example?

II-1 – SAMPLE COST: Intuitively, the more examples provided to the $\text{learn}_{\mathcal{W}}$ algorithm, the more likely that the learned 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 problems. To summarize, we find that 2–3 examples are needed on average; see Fig. 9.

II-1-b – SAMPLE COMPLEXITY: We also developed a PAC (Kearns & Vazirani 1994) 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 are (with high probability) correct.

II-2 – INDUCTION COST: While sample cost measures the number of examples required, we are also concerned with the per-example processing time.

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. 9.

II-2-b – TIME COMPLEXITY: We have also performed a complexity analysis on the $\text{learn}_{\mathcal{W}}$ algorithms. As stated in Theorem 2, most of our wrapper classes can be learned in time that grows as a small-degree polynomial of the relevant problem parameters.

The wrapper induction problem

We begin with a formal statement of the learning task with which we are concerned. As shown in Fig. 2, an *information resource* 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 $\mathcal{Q}$. As we are concerned mainly with the responses, we ignore $\mathcal{Q}$.

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 P_{CC} in Fig. 1(c). Note that our techniques are motivated by, but do not rely on, HTML; our system does not use any HTML-specific knowledge or constraints.

We adopt a simple relational data model. Associated with each information resource is a set of K *attributes*, each representing a column in the relational model. In the example, $K = 2$.

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. Tuples represent rows in the relational model.

$$\begin{tikzcd} \substack{\text{query}\\ Q\in\mathcal{Q}} \arrow[r, Rightarrow] & \boxed{\substack{\text{resource}\\ \mathcal{S}}} \arrow[r, Rightarrow] & \substack{\text{page}\\ P\in\Sigma^*} \arrow[r, Rightarrow] & \boxed{\substack{\text{wrapper}\\ W\in\mathcal{W}}} \arrow[r, Rightarrow] & \substack{\text{label}\\ L\in\mathcal{L}} \end{tikzcd}$$

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

Page P_{CC}'s label comprises four tuples, the first of which is ⟨'Congo', '242'⟩.

The *label* of a page is the set of tuples it contains. For example, the label L_{CC} of the example country/code page is shown in Fig. 1(d). The symbol $\mathcal{L}$ in Fig. 2 refers to the set of all labels.

A *wrapper* W is a function from a page to a label; the notation $W(P) = L$ indicates that the result of applying wrapper W to page P is label L. At this level of abstraction, a wrapper is simply an arbitrary procedure. Of course, we will devote considerable attention to particular classes of wrappers. A *wrapper class* $\mathcal{W}$ is simply a set of wrappers.

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 interesting in wrappers from some class $\mathcal{W}$. 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 induction) we demand that W generate the correct label for each training example.

More precisely, 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 $\langle P_n, L_n\rangle \in \mathcal{E}$.

The LR wrapper class

The ccwrap$_{\text{LR}}$ procedure (Fig. 1(e)) illustrates a "programming idiom"—using left- and right-hand delimiters to extract the relevant fragments—that can be generalized beyond the country/code site. The Left-Right (LR) wrapper class is one way to formalize this programming idiom. As shown in Fig. 3, LR is a generalization of ccwrap$_{\text{LR}}$ that allows (1) the delimiters to be arbitrary strings (instead of the specific values ℓ_1 = '<B>', r_1 = '</B>', *etc.*); and (2) any number K of attributes $\{A_1, \ldots, A_K\}$ (rather than exactly two).

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 exec$_{\text{LR}}$ is invoked with the parameters $K = 2$, ℓ_1 = '<B>', r_1 = '</B>', ℓ_2 = '<I>' and r_2 = '</I>', then exec$_{\text{LR}}$ behaves like ccwrap$_{\text{LR}}$.

Notice that the behavior of ccwrap$_{\text{LR}}$ can be entirely described in terms 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.

The LR wrapper induction problem thus becomes 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 examples. Fig. 4 lists learn$_{\text{LR}}$, an algorithm that solves problems of this form.

learn$_{\text{LR}}$ operates by considering each of the $2K$ delimiters in turn. For each delimiter, a set cands$_x$ of candidates are considered. learn$_{\text{LR}}$ selects the first candidate satisfying valid$_x$.[1]

What are the candidates for each delimiter? Consider ℓ_1, which indicates the left-hand side of the first attribute. ℓ_1 must be a common suffix of the text fragments that precede the first attribute. Thus the suffixes of the shortest such fragment constitute a satisfactory pool of candidates in which to find ℓ_1. Candidates for the other delimiters are obtained similarly.

Under what conditions are candidates valid? To ensure that the learned wrapper is consistent with every example, we must ensure that when exec$_{\text{LR}}$ searches for ℓ_1, it finds the delimiter where it "belongs"—*i.e.*, immediately preceding each instance of the first attribute. Thus ℓ_1 must be a suffix of the text preceding each instance of the first attribute, across all tuples in each example. Moreover, ℓ_1 must not occur "upstream" from its correct position, and so we demand that ℓ_1 occur *only* as a suffix of this text; we call such a suffix a *proper suffix.*

Beyond LR

The LR wrapper class requires that resources format their pages in a fairly simpler manner. Of course, not all resources obey such restrictions. There may simply be no LR wrapper that extracts just the relevant content. For example, if page P_{CC} had contained bold text that is not a country (perhaps in heading such as "···<B>Some Country Codes</B>···"), then it is possible that no LR wrapper exists which extracts the correct information while not getting confused by this header.

HLRT. The Head-Left-Right-Tail (HLRT) wrapper class is designed to avoid getting confused by distracting material in a page's head and tail. An HLRT wrap-

[1]Earlier, we stated that a label is a set of K-tuples. The attribs, heads, tails and seps procedures in Fig. 4 are stated using an equivalent but more precise notation. A label for page P is a collection of integers $b_{m,k}$ and $e_{m,k}$, where $b_{m,k}$ is the index into P of the beginning of the m^{th} value for attribute A_k, and $e_{m,k}$ is the end index.

```
procedure exec_LR(wrapper $\langle \ell_1, r_1, \ldots, \ell_K, r_K \rangle$, page $P$)
 while there are more occurrences in $P$ of $\ell_1$
  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 next occurrence of $\ell_k$; save position as start of next value $A_k$
   scan in $P$ to next occurrence of $r_k$; save position as end of next value $A_k$
 return extracted tuples $\{\ldots, \langle A_1, \ldots, A_k \rangle, \ldots\}$
```

Figure 3: The exec_{LR} procedure specifies how an LR wrapper is executed.

```
procedure learn_LR(examples $\mathcal{E}$)
 for each $1 \leq k \leq K$
  for each $u \in \text{cands}_\ell(k, \mathcal{E})$: if $\text{valid}_\ell(u, k, \mathcal{E})$ then $\ell_k \leftarrow u$ and terminate this loop
 for each $1 \leq k \leq K$
  for each $u \in \text{cands}_r(k, \mathcal{E})$: if $\text{valid}_r(u, k, \mathcal{E})$ then $r_k \leftarrow u$ and terminate this loop
 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 $\text{neighbors}_\ell(k, \mathcal{E})$
procedure cands_r(index $k$, examples $\mathcal{E}$)
 return the set of all prefixes of the shortest string in $\text{neighbors}_r(k, \mathcal{E})$
procedure valid_ℓ(candidate $u$, index $k$, examples $\mathcal{E}$)
 for each $s \in \text{neighbors}_\ell(k, \mathcal{E})$: if $u$ is not a proper suffix of $s$ then return FALSE
 if $k = 1$ then for each $s \in \text{tails}(\mathcal{E})$: if $u$ is a substring of $s$ then return FALSE
 return TRUE
procedure valid_r(candidate $u$, index $k$, examples $\mathcal{E}$)
 for each $s \in \text{attribs}(k, \mathcal{E})$: if $u$ is a substring of $s$ then return FALSE
 for each $s \in \text{neighbors}_r(k, \mathcal{E})$: if $u$ is not a prefix of $s$ then return FALSE
 return TRUE
procedure 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 neighbors_ℓ(index $k$, examples $\mathcal{E}$)
 if $k = 1$ then return $\text{seps}(K, \mathcal{E}) \cup \text{heads}(\mathcal{E})$ else return $\text{seps}(k-1, \mathcal{E})$
procedure neighbors_r(index $k$, examples $\mathcal{E}$)
 if $k = K$ then $\text{seps}(K, \mathcal{E}) \cup \text{tails}(\mathcal{E})$ else return $\text{seps}(k, \mathcal{E})$
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 $\{P_n[e_{|L|,K}, |P_n|] \mid \langle P_n, \{\ldots, \langle \ldots, \langle b_{|L|,K}, e_{|L|,K} \rangle \rangle \} \rangle \in \mathcal{E}\}$
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 $\cup_{\langle P_n, L_n \rangle \in \mathcal{E}} \{P_n[e_{m,k}, b_{m,k+1}] \mid \langle \ldots, \langle b_{m,k}, e_{m,k} \rangle, \ldots \rangle \in L_n\}$
```

Figure 4: The learn_{LR} algorithm.

per 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. The wrapper starts processing the page after h, and stops when t is encountered. The $\text{exec}_{\text{HLRT}}$ procedure specifies the execution of an HLRT wrapper, while $\text{learn}_{\text{HLRT}}$ is the class's learning algorithm; see Fig. 5.

OCLR. The Open-Close-Left-Right (OCLR) wrapper class is designed to avoid distracting material between tuples. An OCLR wrapper is a vector of $2K + 2$ strings $\langle o, c, \ell_1, r_1, \ldots, \ell_K, r_K \rangle$. OCLR operates like LR, except that the wrapper skips forward to the opening delimiter o before processing each tuple, and also skips forward to the closing delimiter c after each tuple. The OCLR class is captured by the $\text{exec}_{\text{OCLR}}$ and $\text{learn}_{\text{OCLR}}$ procedures; see Fig. 6

HOCLRT. The HOCLRT class combines the functionality of HLRT and OCLR; see Figure 7.

N-LR and N-HLRT. The classes discussed so far concern "tabular" resources, but our techniques can be applied to non-tabular resources as well. One example of non-tabular structure is a *nested* page, in which attributes are organized hierarchically, like a book's table of contents. The Nested-Left-Right (N-LR) and Nested-Head-Left-Right-Tail (N-HLRT) wrapper classes are straightforward extensions of LR and HLRT to handle nested resources.

Space limitations preclude further details. The point is that we have defined several wrapper classes, all based on the idea of delimiter strings. Each class $\mathcal{W}$ is specified in terms of the $\text{exec}_{\mathcal{W}}$ procedure, and the $\text{learn}_{\mathcal{W}}$ learning algorithms are derived by reasoning about the behavior of $\text{exec}_{\mathcal{W}}$. In each case, $\text{learn}_{\mathcal{W}}$ is similar to

```
procedure exec_HLRT(wrapper $\langle h, t, \ell_1, r_1, \ldots, \ell_K, r_K \rangle$, page $P$)
  scan in $P$ to the first occurrence of $h$
  while the next occurrence of $t$ in $P$ occurs before the next occurrence of $\ell_1$
    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 $\ell_k$; save position as start of next $A_k$
      scan in $P$ to the next occurrence of $r_k$; save position as end of next $A_k$
  return extracted label $\{\ldots, \langle A_1, \ldots, A_K \rangle, \ldots\}$
procedure learn_HLRT(examples $\mathcal{E}$)
  $\langle \cdot, r_1, \ldots, \ell_K, r_K \rangle \leftarrow$ learn_LR($\mathcal{E}$)
  for each $u_{\ell_1} \in$ cands_ℓ($1, \mathcal{E}$)
    for each $u_h \in$ cands_h($\mathcal{E}$)
      for each $u_t \in$ cands_t($\mathcal{E}$)
        if valid_{ℓ1,h,t}($u_{\ell_1}, u_h, u_t, \mathcal{E}$) then
          $\ell_1 \leftarrow u_{\ell_1}$, $h \leftarrow u_h$, $t \leftarrow u_t$, and terminate these three loops
  return HLRT wrapper $\langle h, t, \ell_1, r_1, \ldots, \ell_K, r_K \rangle$
```

Figure 5: The exec$_{\mathrm{HLRT}}$ and learn$_{\mathrm{HLRT}}$ procedures.

```
procedure exec_OCLR(wrapper $\langle o, c, \ell_1, r_1, \ldots, \ell_K, r_K \rangle$, page $P$)
  while there are more occurences of $o$ in $P$
    scan to the next occurence of $o$ in $P$
    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 occurence of $\ell_k$; save position as start of next $A_k$
      scan in $P$ to the next occurence of $r_k$; save position as end of next $A_k$
    scan to the next occurence of $c$ in $P$
  return extracted label $\{\ldots, \langle A_1, \ldots, A_K \rangle, \ldots\}$
procedure learn_OCLR(examples $\mathcal{E}$)
  $\langle \cdot, r_1, \ldots, \ell_K, r_K \rangle \leftarrow$ learn_LR($\mathcal{E}$)
  for each $u_{\ell_1} \in$ cands_ℓ($1, \mathcal{E}$)
    for each $u_o \in$ cands_{o,c}($\mathcal{E}$)
      for each $u_c \in$ cands_{o,c}($\mathcal{E}$)
        if valid_{ℓ1,o,c}($u_{\ell_1}, u_o, u_c, \mathcal{E}$) then
          $\ell_1 \leftarrow u_{\ell_1}$, $o \leftarrow u_o$, $c \leftarrow u_c$, and terminate these three loops
  return OCLR wrapper $\langle o, c, \ell_1, r_1, \ldots, \ell_K, r_K \rangle$
```

Figure 6: The exec$_{\mathrm{OCLR}}$ and learn$_{\mathrm{OCLR}}$ procedures.

```
procedure exec_HOCLRT(wrapper $\langle h, t, o, c, \ell_1, r_1, \ldots, \ell_K, r_K \rangle$, page $P$)
  scan to the first occurence in $P$ of $h$
  while the next occurence of $o$ in $P$ occurs before the next occurence of $t$
    scan to the next occurence of $o$ in $P$
    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 occurence of $\ell_k$; save position as start of next $A_k$
      scan in $P$ to the next occurence of $r_k$; save position as end of next $A_k$
    scan to the next occurence of $c$ in $P$
  return extracted label $\{\ldots, \langle A_1, \ldots, A_K \rangle, \ldots\}$
procedure learn_HOCLRT(examples $\mathcal{E}$)
  $\langle \cdot, r_1, \ldots, \ell_K, r_K \rangle \leftarrow$ learn_LR($\mathcal{E}$)
  for each $u_{\ell_1} \in$ cands_ℓ($1, \mathcal{E}$)
    for each $u_o \in$ cands_{o,c}($\mathcal{E}$)
      for each $u_c \in$ cands_{o,c}($\mathcal{E}$)
        for each $u_h \in$ cands_h($\mathcal{E}$)
          for each $u_t \in$ cands_t($\mathcal{E}$)
            if valid_{ℓ1,h,t,o,c}($u_{\ell_1}, u_h, u_t, u_o, u_c, \mathcal{E}$) then
              $\ell_1 \leftarrow u_{\ell_1}$, $o \leftarrow u_o$, $c \leftarrow u_c$, $h \leftarrow u_h$, $t \leftarrow u_t$ and terminate these five loops
  return HOCLRT wrapper $\langle h, t, o, c, \ell_1, r_1, \ldots, \ell_K, r_K \rangle$
```

Figure 7: The exec$_{\mathrm{HOCLRT}}$ and learn$_{\mathrm{HOCLRT}}$ procedures.

learn$_{\mathrm{LR}}$: a set of candidates cands$_x$ is identifying for each delimiter, and then the candidates are evaluated using valid$_x$.

However, there is an important difference. For LR, all of the delimiters are *mutually independent*: the validity of a particular candidate for any delimiters does not depend on the choice of any other delimiters. For example, whether '<B>' is valid for ℓ_1 does not depend on the validity of '</I>' for r_2.

This independence property does not hold for the other five classes. Specifically, in HLRT, h, t, and ℓ_1 interact; o, c and ℓ_1 interact in OCLR; h, t, o, c and ℓ_1 interact in HOCLRT; and all delimiters interact in N-LR and N-HLRT. Thus rather than evaluating these delimiters' candidates separately, the learn$_{\mathcal{W}}$ procedures must enumerate all *combinations* of candidates for these delimiters. An important theoretical implication of this observation is that the classes differ with respect to how quickly they can be learned; see Theorem 2.

Empirical results

Our learning algorithms have been fully implemented; our experiments were run on a 233 MHz Pentium II using Allegro Common Lisp. We ran our algorithms on a collection of 30 Internet sites randomly selected from "www.search.com". We retrieved a sample of each site's pages, and then repeated the following process 30 times. We split the sample pages into a training and test set, and then provided each learning algorithm with one training example, then two, then three, *etc.*, until the resulting wrapper performed perfectly on the test pages.

Our empirical coverage, sample cost and induction cost results are shown in Fig. 9. For each of the $30{\cdot}6 = 180$ site/class combinations, we indicate whether the class can handle the site, and if so the number of examples and CPU time needed for perfect learning. With respect to coverage, our wrapper classes can handle between 13% and 57% of the sites, with 70% of the sites handled in total.

With respect to efficiency, in nearly all cases, just a few examples are required to learn a wrapper that works perfectly on the test pages, and each example consumes a fraction of a CPU second per example; overall, each learning episode usually takes less than one second.

In some cases (a few sites for HLRT and HOCLRT, and all sites for N-LR and N-HLRT), learn$_{\mathcal{W}}$ run very slowly. Specifically, while HOCLRT wrapper can usually be learned rapidly, the average HOCLRT processing time in Fig. 9 must be taken with a grain of salt, since two sites where our implementation performs poorly are excluded.

There are two explanations for these results. First, our learning algorithms require exponential time for N-LR and N-HLRT; see Theorem 2. Second, wrapper induction is essentially problem of search in the space of possible wrappers. This space is huge, often containing 10^{15} potential wrappers. Our learning algorithms implicitly use a particular search control policy, and our negative results indicate that it probably could be improved. (Our algorithms work as well as they do because the space of potential wrappers usually contains many valid wrappers.)

Analytical results

Relative expressiveness. We have explored the extent to which wrappers from one class can mimic another. To formalize this investigation, let $\Pi = \{\ldots, \langle P, L\rangle, \ldots\}$ be the set of all page/label pairs. Note that a wrapper class is equivalent to a subset of Π: a class corresponds to those page/label pairs for which a consistent wrapper exists in the class. The notation $\Pi(\mathcal{W})$ indicates the subset of Π which can be handled by $\mathcal{W}$: $\Pi(\mathcal{W}) = \{\langle P, L\rangle \in \Pi \mid \exists_{W \in \mathcal{W}}\, W(P) = L\}$.

$\Pi(\mathcal{W})$ is a natural basis on which to reason about relative expressiveness: if $\Pi(\mathcal{W}_1) \subset \Pi(\mathcal{W}_2)$, then $\mathcal{W}_2$ is more expressive than $\mathcal{W}_1$, because any page that can be wrapped by $\mathcal{W}_1$ can also be wrapped by $\mathcal{W}_2$. Our analysis indicates that the relationships between the six classes are fairly complicated:

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 shown in Fig. 8.*

Induction time complexity. We are interested in a complexity analysis of the learn$_{\mathcal{W}}$ algorithm. That is, if $\mathcal{E} = \{\ldots, \langle P_n, L_n\rangle, \ldots\}$ is a set of examples, we are interested in the time to execute learn$_{\mathcal{W}}(\mathcal{E})$, for each class $\mathcal{W}$.

THEOREM 2 *The running-time complexity of the learning algorithms is as follows:*

wrapper class	*complexity*
LR	$O\left(KM^2\lvert\mathcal{E}\rvert^2V^2\right)$
HLRT	$O\left(KM^2\lvert\mathcal{E}\rvert^4V^6\right)$
OCLR	$O\left(KM^4\lvert\mathcal{E}\rvert^2V^6\right)$
HOCLRT	$O\left(KM^4\lvert\mathcal{E}\rvert^4V^{10}\right)$
N-LR	$O\left(M^{2K}\lvert\mathcal{E}\rvert^{2K+1}V^{2K+2}\right)$
N-HLRT	$O\left(M^{2K+2}\lvert\mathcal{E}\rvert^{2K+3}V^{2K+4}\right)$

where K is the number of attributes per tuple, $\lvert\mathcal{E}\rvert$ is the number of examples, $M = \sum_n \lvert L_n\rvert$ is the total number of tuples, and $V = \max_n \lvert P_n\rvert$ is the length of the longest example.

While these bounds involve high degree polynomials, our empirical results indicate that (except for exec$_{\text{N-LR}}$ and exec$_{\text{N-HLRT}}$) our algorithms are usually fast in practice.

Discussion

We have defined the wrapper induction problem, and identified a family of six delimiter-based wrapper classes. Each class is defined in terms of a vector of delimiters used to identify the text fragments to be extracted. Learning wrappers for each class

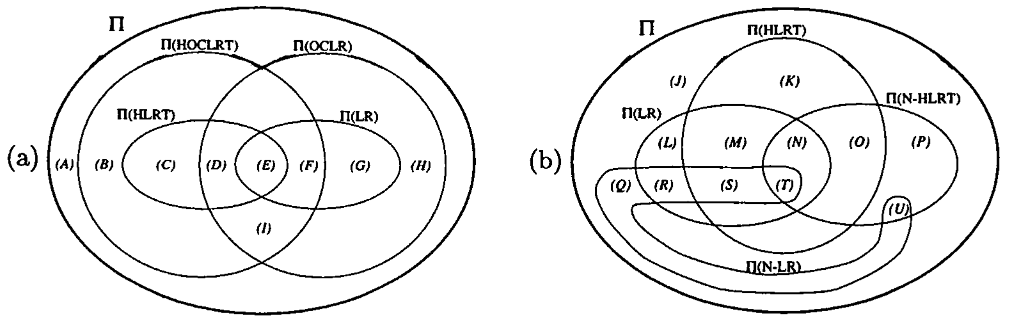


Figure 8: The relative expressiveness of (a) LR, HLRT, OCLR and HOCLRT; and (b) LR, HLRT, N-LR and N-HLRT.

involves enumerating and validating a set of candidates for each delimiter. We have demonstrated—both empirically and analytically—that most of our classes are reasonably useful, yet can be learned quickly. While wrapper induction constitutes the main technical focus of our work, we have also investigated related issues, such as automatically labeling the example pages using noisy heuristics; see (Kushmerick 1997; Kushmerick, Weld, & Doorenbos 1997) for details.

Our approach is most closely related to (Ashish & Knoblock 1997); their system uses HTML-specific heuristics and can handle more complicated pages, while we do not rely on HTML but our wrapper classes are less expressive. (Soderland 1997) describes an HTML-specific information-extraction system. Our effort is also allied with other work on learning to interact with Web sites; see (Doorenbos, Etzioni, & Weld 1997; Perkowitz & Etzioni 1995). At the highest level, we are influenced by the huge literature on information-gathering systems and software agents, such as (Chawathe *et al.* 1994; Etzioni & Weld 1994; Kirk *et al.* 1995; Kwok & Weld 1996).

Many problems remain. Earlier we mentioned search control. Also, while fairly expressive, our wrapper classes lack some of the sophistication needed by information-integration systems, such as handling missing attributes. Finally, it would be interesting to extend our results to the wrapper classes identified by others in the wrapper community.

Acknowledgements. Much of this research was conducted in collaboration with Dan Weld in the Department of Computer Science and Engineering at the University of Washington, and was funded by ONR Grant N00014-94-1-0060, NSF Grant IRI-9303461, ARPA / Rome Labs grant F30602-95-1-0024, and a gift from Rockwell International Palo Alto Research.

References

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

Chawathe, S.; Garcia-Molina, H.; Hammer, J.; Ireland, K.; Papakonstantinou, Y.; Ullman, J.; and Widom, J. 1994. The TSIMMIS project: Integration of heterogeneous information sources. In *Proc. 10th Meeting of the Information Processing Soc. of Japan*, 7–18.

Doorenbos, R.; Etzioni, O.; and Weld, D. 1997. A scalable comparison-shopping agent for the World-Wide Web. In *Proc. Autonomous Agents*, 39–48.

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

Kearns, M., and Vazirani, U. 1994. *An introduction to computational learning theory.* MIT.

Kirk, T.; Levy, A.; Sagiv, Y.; and Srivastava, D. 1995. The Information Manifold. In *AAAI Spring Symposium: Information Gathering from Heterogeneous, Distributed Environments*, 85–91.

Kushmerick, N.; Weld, D.; and Doorenbos, R. 1997. Wrapper Induction for Information Extraction. In *Proc. 15th Int. Joint Conf. AI.*

Kushmerick, N. 1997. *Wrapper Induction for Information Extraction.* Ph.D. Dissertation, Univ. of Washington.

Kushmerick, N. 1998. Wrapper induction: Efficiency and expressiveness. Submitted to *Artificial Intelligence.*

Kwok, C., and Weld, D. 1996. Planning to gather information. In *Proc. 13th Nat. Conf. AI.*

Perkowitz, M., and Etzioni, O. 1995. Category translation: Learning to understand information on the Internet. In *Proc. 14th Int. Joint Conf. AI*, 930–6.

Soderland, S. 1997. Learning to Extract Text-based Information from the World Web. In *Proc. 3rd Int. Conf. Knowledge Discovery and Data Mining.*

	title	URL	LR	HLRT	OCLR	HOCLRT	N-LR	N-HLRT
			wrapper class					
1	Computer ESP	`www.computeresp.com`	2.0 • 0.10	2.0 • 0.26	2.0 • 0.11	2.0 • 0.25	×	×
2	CNN/Time AllPolitics	`allpolitics.com`	×	×	×	×	×	×
3	Film.com	`www.film.com/admin/search.htm`	2.0 • 0.14	2.0 • 3.00	2.0 • 0.19	2.0 • 2.72	×	×
4	Yahoo People Search	`www.yahoo.com/search/people`	2.0 • 0.02	2.0 • 3.63	2.0 • 0.04	2.0 • 107	√	√
5	Cinemachine	`www.cinemachine.com`	2.0 • 0.03	2.2 • 1.94	2.0 • 0.05	2.1 • 2.09	×	√
6	PharmWeb	`www.pharmweb.net`	×	×	×	×	×	×
7	TravelData...	`www.ultranet.com/biz/inns/search-form`	×	×	×	×	×	√
8	NEWS.COM	`www.news.com`	2.0 • 0.10	4.4 • 113	2.0 • 0.32	4.6 • 26.7	×	√
9	I'net Travel Network	`www.itn.net`	×	×	×	×	×	×
10	Time World Wide	`pathfinder.com/time`	4.4 • 0.14	5.7 • 0.98	3.9 • 0.20	4.1 • 1.07	×	×
11	I'net Address Finder	`www.iaf.net`	×	×	×	×	×	×
12	Expedia World Guide	`www.expedia.com/pub/genfts.dll`	×	2.0 • 0.33	×	2.0 • 0.36	×	√
13	thrive@pathfinder	`pathfinder.com/thrive/index.html`	2.0 • 0.13	2.0 • 1.23	2.0 • 0.19	2.0 • 1.19	×	×
14	Monster Job...	`www.monster.com`	×	7.0 • 397	×	9.0 • 277	×	√
15	NewJour	`gort.ucsd.edu/newjour`	2.0 • 0.02	2.0 • 0.04	2.0 • 0.03	2.0 • 0.04	×	√
16	Zipper	`www.voxpop.org/zipper`	×	×	×	×	×	×
17	Coolware Classifieds...	`www.jobsjobsjobs.com`	×	×	×	×	×	√
18	Ultimate Band List	`ubl.com`	×	×	×	×	×	√
19	Shops.Net	`shops.net`	2.0 • 0.04	2.0 • 0.91	2.0 • 0.06	√	√	√
20	Democratic Party	`www.democrats.org`	2.0 • 0.05	2.0 • 0.08	2.0 • 0.08	2.0 • 0.09	√	√
21	Works of Shakespeare	`the-tech.mit.edu/Shakespeare/works.html`	×	×	×	×	×	×
22	Bible...	`etext.virginia.edu/rsv.browse.html`	2.0 • 0.10	2.0 • 0.60	2.0 • 0.26	2.0 • 0.66	×	√
23	Virtual Garden	`pathfinder.com/vg`	2.0 • 0.13	2.0 • 0.37	3.1 • 0.18	√	×	√
24	Foreign Languages...	`www.travlang.com`	×	×	×	×	×	×
25	U.S. Tax Code	`www.fourmilab.ch/ustax/ustax.html`	2.0 • 0.02	2.0 • 0.09	2.0 • 0.03	2.0 • 0.09	×	√
26	CD Club...	`www.cd-clubs.com`	×	×	×	×	×	×
27	Expedia Currency...	`www.expedia.com/pub/curcnvt.dll1`	2.0 • 0.05	2.0 • 11.8	2.0 • 0.23	2.0 • 11.4	×	√
28	Cyberider Cycling	`blueridge.infomkt.ibm.com/bikes`	2.0 • 0.26	×	2.0 • 0.28	×	√	×
29	Security APL...	`qs.secapl.com`	×	×	×	×	×	×
30	Congressional...	`voter96.cqalert.com/cq_job.htm`	6.6 • 0.01	6.4 • 0.01	5.3 • 0.01	6.2 • 0.01	×	×
		wrapper class coverage	53%	57%	53%	57%	13%	50% *(total: 70%)*
		average (excluding timeouts)	2.4 • 0.08	2.9 • 31.5	2.4 • 0.14	3.1 • 28.7		

Figure 9: Empirical results. "×" indicates that the class can not handle the site; "$n \cdot t$" indicates that n examples are required for learning a wrapper that performs perfectly on the test problems, and that each example consumes t CPU seconds (*e.g.*, the top-left cell is "2.0 • 0.10", indicating that 2.0 examples were needed to learn an LR wrapper for site 1, and that $\mathsf{learn}_{\mathrm{LR}}$ takes 0.1 CPU seconds per example, so that overall the learning episode takes $2.0 \cdot 0.1 = 0.2$ seconds); and "√" indicates that the class can handle the site, but our (unoptimized) implementation requires more than 15 minutes of CPU time per example.