

# Learning Page-Independent Heuristics for Extracting Data from Web Pages

William W. Cohen  
AT&T Shannon Laboratories  
180 Park Ave  
Florham Park, NJ 07974  
wcohen@research.att.com

Wei Fan  
Department of Computer Science  
Columbia University  
New York, NY 10027  
wfan@cs.columbia.edu

## Abstract

One bottleneck in implementing a system that intelligently queries the Web is developing “wrappers”—programs that extract data from Web pages. Here we describe a method for learning general, page-independent heuristics for extracting data from HTML documents. The input to our learning system is a set of working wrapper programs, paired with HTML pages they correctly wrap. The output is a general procedure for extracting data that works for many formats and many pages. In experiments with a collection of 84 constrained but realistic extraction problems, we demonstrate that 30% of the problems can be handled perfectly by learned extraction heuristics, and around 50% can be handled acceptably. We also demonstrate that learned page-independent extraction heuristics can substantially improve the performance of methods for learning page-specific wrappers.

**Keywords:** information integration, machine learning, extraction.

## 1 Introduction

A number of recent systems operate by taking information from the Web, storing it in some sort of knowledge base, and then allowing a user to query that knowledge base [14, 7, 11, 8, 13, 15, 19]. One bottleneck in building such an *information integration system* is developing “wrappers”—programs that convert Web pages into an appropriate format for the knowledge base. Because data can be presented in many different ways on the Web, and Web pages frequently change format, building and maintaining these wrappers is often time-consuming and tedious.

A number of proposals have been made for reducing the cost of building wrappers. Data exchange standards like XML have promise; unfortunately, XML is not yet widely used, and

one might expect that Web information sources using “legacy” formats like HTML will be common for some time. Some researchers have proposed special languages for writing wrappers [9, 5], or semi-automated tools for wrapper construction [1]. Others have implemented systems that allow wrappers to be trained from examples [12, 10, 16]. Although languages and learning methods for wrapper construction are useful, they do not entirely eliminate the human effort involved in “wrapping” a Web site; for example, if learning is used, it is still necessary for a human to label the examples given to the learner.

Most of the data extracted by wrappers is originally encoded in HTML, and is very regular and repetitive in format: generally, the pages being wrapped are well-structured tables and lists. An alternative research program would be to develop general, page-independent, heuristics for recognizing (and extracting data from) tables and lists in HTML documents. However, developing general-purpose, reliable, heuristics for table- and list-recognition is non-trivial, as users often do not implement tables and lists using the appropriate HTML constructs (*e.g.*, `<table>`, `<ul>`, `<dl>`). Furthermore, any such heuristics might well require substantial effort to maintain as HTML and conventions for using it continue to evolve.

In this paper, we seek to *learn* general, page-independent heuristics for extracting data from HTML documents. The input to our learning system is a set of working wrapper programs, paired with HTML pages they correctly wrap. The output is a general, page-independent procedure for extracting data—a procedure that works for many formats and many pages. New pages that are correctly wrapped by the learned procedure can be incorporated into a knowledge base with minimal human effort; it is only necessary to indicate where in the knowledge base the extracted information should be stored. Our method thus differs from earlier methods for learning wrappers, in which the goal of learning was a wrapper for pages with a single specific format, and a new training process is needed for each page format.

Below we will describe our learning method, and evaluate it on a collection of 84 extraction problems encountered in building applications for the information integration system WHIRL [4, 5]. We first identify two types of extraction problems, namely extraction of *simple lists* and *simple hotlists*; these were the most common types of extraction problems for WHIRL, together comprising about 75% of the implemented wrappers. We then explain how this extraction problem can be reduced to a more conventional classification problem, thus allowing existing learners to be used to learn extraction heuristics. We demonstrate that around 30% of the benchmark problems can be handled perfectly by learned extraction heuristics, and around 50% of the benchmarks can be handled reasonably well. We also show that the learned heuristics are domain-independent.

We also evaluate a hybrid system that combines learned page-independent extraction heuristics with a more conventional wrapper-learning approach, in which the learner is re-trained for each page format. We show that incorporating page-independent heuristics leads to improved performance: for instance, the hybrid system gets acceptable performance on 80% of the benchmarks after seeing only 6 training examples, where as the conventional system requires 12 training examples to do as well.

## 2 Extraction as classification

### 2.1 Extraction as tree rewriting

In earlier work [5] we described a special-purpose language for writing wrappers. A program in this language manipulates an HTML parse tree (that is, a tree with nodes labeled by tag names like `body`, `table`, and `ul`), primarily by deleting and relabeling nodes. A wrapper in this language converts the parse tree for a Web page into another tree labeled with terms from the knowledge base, which can be stored directly in the knowledge base. This paradigm for extraction is clearly not sufficient for all purposes, since it is impossible to extract along boundaries not indicated by HTML markup commands; for instance, it is impossible to separate names from affiliations in the HTML page shown on the left-hand side of Figure 1. In practise, however, the language is almost always expressive enough to wrap the inputs of WHIRL.<sup>1</sup>

As a first step in understanding the problems involved in automatically learning wrappers, we collected 111 different wrapper programs, all written as part of the WHIRL project, but written over several months in several different domains. Each of these wrappers was paired with a single sample page that it correctly wrapped. This was as complete a sample of working conversion programs as we could assemble; the only existing wrappers that were discarded were ones for which no correctly-wrapped HTML pages were available, due to changes in the associated Web sites. Of these 111 wrappers, 84 (or nearly 75%) fell into two special classes, which we call *simple lists* and *simple hotlists*.

In a page containing a *simple list*, the structure extracted is a one-column relation containing a set of strings  $s_1, \dots, s_N$ , and each  $s_i$  is all the text that falls below some node  $n_i$  in the parse tree. In a *simple hotlist*, the extracted structure is a two-column relation, containing a set of pairs  $\langle s_1, u_1 \rangle, \dots, \langle s_n, u_N \rangle$ ; each  $s_i$  is all the text that falls below some node  $n_i$  in the parse tree; and each  $u_i$  is a URL that is associated with some HTML anchor element  $a_i$  that appears somewhere inside  $n_i$ . Figure 1 shows the HTML source for a simple list and a simple hotlist, together with the data that is extracted from each. Notice that, although we use the term “list”, it is not important that the information is presented in a format that looks like a list to the user; for instance, our example for a “simple list” is formatted as a table with two columns.

Henceforth we will concern ourselves only with the problem of extracting simple lists and simple hotlists. We will evaluate all methods on the collection of 84 benchmark problems described above. In the benchmarks, we ignore certain operations performed by the wrappers. Some wrappers included filter predicates, which allow them to ignore certain table entries; for instance, a wrapper might extract a pair  $\langle s, u \rangle$  only if the URL  $u$  contains the substring “.ps”. These filter predicates were removed, since this sort of filtering can be performed just as easily after extraction using mechanisms in the knowledge base. A few wrappers were also associated with `sed` scripts, which are applied (by the interpreter for the tree-rewriting language) to a Web page before parsing.<sup>2</sup> Additional preprocessing steps are applied to every

---

<sup>1</sup>It should be noted, however, that WHIRL includes certain soft-matching facilities that make it very tolerant of inexact extraction. The language might well be less useful if used to build wrappers for a more brittle information integration system.

<sup>2</sup>Most of these scripts were written to avoid bugs in the HTML parser. We are currently using the HTML

### A Simple List

**HTML Source:**

```
<html><head>...</head>
<body>
<h1>Editorial Board Members</h1>
<table> <tr>
  <td>G. R. Emlin, Lucent</td>
  <td>Harry Q. Bovik, Cranberry U</td>
</tr> <tr>
  <td>Bat Gangley, UC/Bovine</td>
  <td>Pheobe L. Mind, Lough Tech</td>
</tr> <tr>
...
</table>
...
```

**Extracted data:**

G. R. Emlin, Lucent
Harry Q. Bovik, Cranberry U
Bat Gangley, UC/Bovine
...

### A Simple Hotlist

**HTML Source:**

```
<html><head>...</head>
<body>
<h1>My Publications</h1>
<ul>
<li>Optimization of fuzzy neural
  networks using distributed parallel
  case-based genetic knowledge discovery
  (<a href="buzz.ps">postscript</a>,
  <a href="buzz.pdf">PDF</a>)</li>
<li>A linear-time version of GSAT
  (<a href="peqnp.ps">postscript</a>)</li>
...

```

**Extracted data:**

Optimization ... (postscript,PDF)	buzz.ps
Optimization ... (postscript, PDF)	buzz.pdf
A linear-time version of ...	peqnp.ps
...	...

Figure 1: A simple list, a simple hotlist, and the data that would be extracted from each.

Web page; for example, relative URLs are always translated to absolute ones. We simplified the problem by assuming that all preprocessing steps are known, including any page-specific sed scripts—*i.e.*, we paired each wrapper with a preprocessed Web page input.

## 2.2 Extraction of lists as classification

Most existing learning systems learn to classify: that is, they learn to associate a *class label* from some small, fixed, set with an unlabeled *instance*. To use a classification learner on an extraction problem, it is necessary to re-cast extraction as a labeling task.

It is straightforward to use labels to encode the output of a wrapper for a simple lists or a simple hotlists. Since each data item extracted by the wrapper corresponds directly to a node in the parse tree, one can encode output of wrapper by appropriately labeling parse tree nodes. To encode a simple list, label a node  $n_i$  as “positive” if it is associated with some extracted string  $s_i$ , and “negative” otherwise. For instance, in the parse tree for the simple list in Figure 1, every `td` node would be labelled as positive. Given a correctly labeled tree, data can be extracted by simply finding each positive node  $n_i$ , and extracting all the text below it as  $s_i$ , the  $i$ -th entry in the extracted list.

---

parser included in the Perl `libwww` package, which is fairly robust, but far from perfect. It is fairly hard to write a completely general HTML parser, since HTML elements are often not explicitly closed, and the rules for implicitly closing unclosed elements are complex.

Encoding a simple hotlist with parse tree labels can be done in a similar way. For a simple hotlist, label a node  $n_i$  as “positive” if it is associated with some extracted string  $s_i$  or some extracted URL  $u_i$ , and “negative” otherwise. For instance, in the parse tree for the simple hotlist in Figure 1, every **a** (anchor) node would be labelled as positive, as well as every **li** node. To extract data from a correctly labeled tree, one examines each outermost positively labeled node  $y_i$ , and does the following. If  $y_i$  contains some positive node  $z_i$ , then for each such  $z_i$ , extract the pair  $\langle s_i, u_i \rangle$ , where  $s_i$  consists of all text below  $y_i$ , and  $u_i$  is the **href** attribute of  $z_i$ . If  $y_i$  does not contain any positive nodes, then treat  $y_i$  as both the “text node” and the “anchor node”: that is, extract the pair  $\langle s_i, u_i \rangle$ , where  $s_i$  consists of all text below  $y_i$ , and  $u_i$  is the **href** attribute of  $y_i$  (which must be an anchor).

To summarize, for simple lists and hotlists, the task of extracting data from a Web page can be re-cast as the task of labeling each node in the HTML parse tree for the page. By the same token, a wrapper can be represented as a procedure for labeling parse tree nodes. Such a node-labeling procedure can be learned from a sample of correctly labeled parse tree nodes. A set of correctly labeled parse tree nodes, in turn, can be easily generated given an existing wrapper and a page that is correctly wrapped.

We thus propose the following procedure for learning general, page-independent extraction procedures. Begin with a set of wrappers  $w_1, \dots, w_N$  that correctly wrap the Web pages  $p_1, \dots, p_N$ . For each  $w_i, p_i$  pair, find the parse tree for  $p_i$ , and label nodes in that tree according to  $w_i$ . This results in a set of labeled parse tree nodes  $\langle n_{i,1}, \ell_{i,1} \rangle, \dots, \langle n_{i,m_i}, \ell_{i,m_i} \rangle$ , which are added to a data set  $S$ . Finally, use  $S$  to train some classification learner. The output of the learner is a node-labeling procedure  $h$ , which is a function

$$h : \text{parse-tree-node} \longrightarrow \{\text{positive, negative}\}$$

The learned function  $h$  can then be used to label the parse tree nodes of new Web pages, and thereby to extract data from these pages. It remains to describe the learning method, and the way in which parse tree nodes are encoded for the learner.

## 2.3 Features and learning methods

In most of our experiments we used the rule learning system RIPPER [2]. RIPPER has some advantages for this problem: in particular, it handles the “set-valued” features (described below) directly, and is efficient enough to use on problems of this scale. (About 65,000 parse-tree node examples are generated from the 84 wrapper/page pairs). The main reason for using RIPPER, however, was that we were familiar with it; as we show later, other learning systems achieve comparable results on this problem.

RIPPER, like most classification learners, requires an example to be represented as a vector of relatively simple features. The value of each feature is either a real number, or else a symbolic feature—an atomic symbol from a designated set, like  $\{\text{true, false}\}$ . The primitive tests allowed for a real-valued feature are of the form  $f \geq \theta$  or  $f \leq \theta$ , where  $f$  is a feature and  $\theta$  is a constant number, and the primitive tests allowed for a symbolic feature are of the form  $f = a_i$ , where  $f$  is a feature and  $a_i$  is a possible value for  $f$ . RIPPER also allows *set-valued features* [3]. The value of a set-valued feature is a set of atomic symbols, and tests on set-valued features are of the form  $a_i \in f$ , where  $f$  is the name of feature and  $a_i$  is a

possible value (e.g., `ul`  $\in$  `ancestorTagNames`). For two-class problems of this sort, RIPPER uses a number of heuristics to build a disjunctive normal form formula that generalizes the positive examples. This formula is usually thought of as a set of *rules*, each rule having the form “label an instance ‘positive’ if  $t_1$  and  $t_2$  and ...”, where each  $t_i$  in the rule is a primitive test on some feature.

After some thought, we devised the following nineteen features to describe a parse tree node. The *tag name* is the HTML tag name associated with the node, such as `a`, `p`, `br`, and `html`. This is an informative feature: some tags such as `head` are always negative, while, other tags such as the anchor tag `a` are often positive. We measured the size of the string directly associated<sup>3</sup> with a node in two ways: the *text length* is the total size of all the text associated with a node, and the *non-white text length* is similar to text length but ignores blanks, tabs, and line returns. We also measured the length of text contained in the subtree rooted at the current node by the features *recursive text length* and *recursive non-white text length*; these features are important because they measure the size of the string  $s_i$  that would be extracted if the node were marked as positive. The features *set of ancestor tag names*, *depth*, *number of children*, *number of siblings*, *parent tag name*, *set of child tag names* and *set of descendent tag names* are other natural and easily-computed features. Since the size of parse trees varies considerably, we also normalize many of the above features by the total number of nodes or by the maximal node degree.

The final features we designed are intended to detect and quantify repeated structure in the parse tree; intuitively, this is important, because positive nodes often reside on a structure frequently repeated in the tree. The repetitive aspect of a structure can often be detected by looking at the sequence of node tags that appear in paths through the tree; for instance, in the parse tree for bibliography page of Figure 1, there are many paths labeled `li-a` that originate at the `ul` node.

To measure this sort of repetition, let  $tag(n)$  denote the tag associate with a node  $n$ , and define the *tag sequence position of  $n$* ,  $p(n)$ , as the sequence of tags encountered in traversing the path from the root of the parse tree to  $n$ : that is,  $p(n) = \langle html, \dots, tag(n) \rangle$ . If  $p(n_1) = \langle t_1, \dots, t_k \rangle$  and  $p(n_2) = \langle t_1, \dots, t_k, t_{k+1}, \dots, t_m \rangle$ , then we say the tag sequence position  $p(n_1)$  is a *prefix* of  $p(n_2)$ ; if additionally  $p(n_1)$  is strictly shorter than  $p(n_2)$ , then we say that  $p(n_1)$  is a *proper prefix* of  $p(n_2)$ .

We use the *node prefix count for  $n$*  as a way of measuring the degree to which  $n$  participates in a repeated substructure. The *node prefix count for  $n$* ,  $p_{count}(n)$ , is the number of leaf nodes  $l$  in the tree such that the tag sequence position of  $n$  is a prefix of the tag sequence of  $l$ : more formally,  $p_{count}(n) = |\{l : p(n) \text{ is a tag sequence prefix of } p(l), l \text{ is a leaf}\}|$ . The *node suffix count for  $n$* ,  $s(n)$ , is closely related; it is defined as the number of leaf nodes  $l$  with tag sequence positions of which  $p(n)$  is a proper prefix. We normalize both  $p_{count}(n)$  and  $s_{count}(n)$  by the total number of paths in the tree.

These last features are clearly engineered, but are based on clear and plausible intuitions; the tree-rewriting language specifies nodes to be re-written in terms of their tag sequence

---

<sup>3</sup>Text is considered to be “directly associated” with a node if it is contained in the HTML element associated with the node, and not contained in any smaller HTML element. For instance, if  $n$  is the first `li` node in the parse tree for the simple hotlist of Figure 1, the text “Optimization of ... discovery” is directly associated with  $n$ , but the text “(postscript)” is not.

- Rule 1.** Label a node “positive” if tagName= “a” and normalizedNodeSuffixCount  $\geq 0.445545$ .
- Rule 2.** Label a node “positive” if tagName= “a”, normalizedNodeSuffixCount  $\geq 0.2666355$ , and depth  $\leq 4$ .
- Rule 3.** Label a node “positive” if normalizedNodePrefixCount  $\geq 0.688645$ , tagName = “p”, and numSiblings  $\geq 317$ .
- Rule 4.** Label a node “positive” if tagName= “td” and normalizedNodePrefixCount  $\geq 0.981132$ .

Figure 2: Some extraction rules learned by RIPPER.

positions, and tools for writing programs in the tree-rewriting language are also based on looking for repeated tag sequences. Most of the other features were introduced with a fairly non-selective “brainstorming” process; here we were looking for features that were easy to compute and plausibly related to the classification task. No effort was made to select an optimal set of features, or to include any sort of strong knowledge of the type of extraction programs we expected to see.

As an illustration of the way in which these features are used in learning, Figure 2 shows some representative rules that appear in the hypothesis obtained by training RIPPER on all of the 84 wrapper/page pairs. Rule 1 says that a node should be labeled as “positive”—that is, part of a list to be extracted—if it is an anchor (**a**) element that repeats frequently, as measured by the normalized suffix count for the node. Rule 2 is similar; it marks anchor elements for extraction if they repeat somewhat less frequently, but are close to the root of the parse tree. Rule 3 marks paragraph (**p**) elements for extraction; since this type of element is less likely to be in a list, the rule requires very strong repetition, as measured both by the node prefix count, and by the number of siblings of the node. Rule 4 marks table entry elements (**td**) for extraction if they are part of a repeating structure that comprises most of the page. The complete hypothesis learned by RIPPER from this data contains a total of 36 such rules.

## 3 Experimental results

### 3.1 Leave-one-page-out experiments

We used the following method to evaluate the learned extraction heuristics. For each wrapper/Web page pair  $w_i, p_i$ , we trained the learner on a dataset constructed from all other wrapper/page pairs: that is, from the pairs  $\langle w_1, p_1 \rangle, \dots, \langle w_{i-1}, p_{i-1} \rangle, \langle w_{i+1}, p_{i+1} \rangle, \dots, \langle w_m, p_m \rangle$ . We then tested the learned extraction heuristics on data constructed from the single held-out page  $p_i$ , measuring the *recall* and *precision* of the learned classifier.<sup>4</sup>

---

<sup>4</sup>*Recall* is the fraction of positive labels that were labeled as positive by the classifier, and *precision* is the fraction of nodes labeled as positive by the classifier that were correctly labeled. In other words, if  $P_p$  is the set of nodes predicted to be positive by the learner, and  $P_w$  is the set of nodes that are labeled positive by the target wrapper program, then recall is  $|P_p \cap P_w|/|P_w|$ , and precision is  $|P_p \cap P_w|/|P_p|$ .

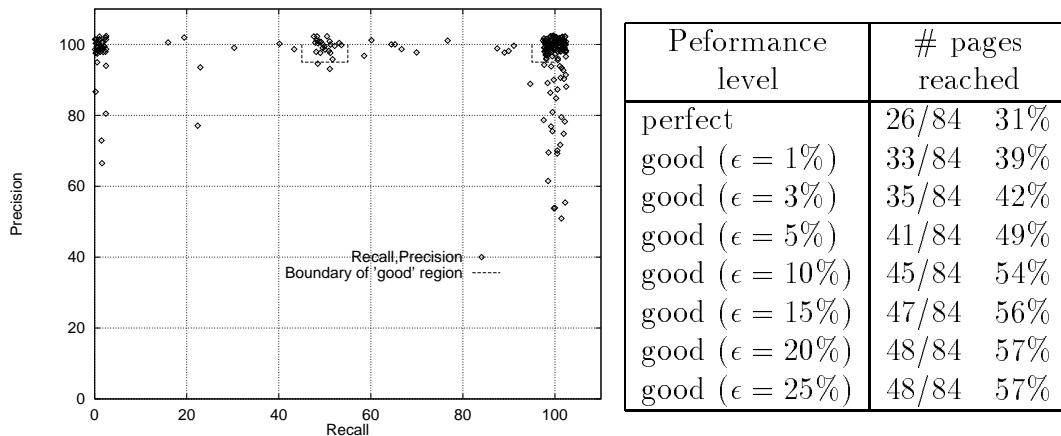


Figure 3: Performance of RIPPER in leave-one-page out experiments.

This results in 168 measurements, two for each page. Before attempting to summarize these measurements we will first present the raw data in detail. All the results of this “leave one page out” experiment (and two variants that will be described shortly) are shown in the scatterplot of Figure 3; here we plot for each page  $p_i$  a point where recall is the  $x$ -axis position and precision is the  $y$ -axis position. So that nearby points can be more easily distinguished, we added 5% noise to both coordinates.<sup>5</sup>

The scatter plot shows three distinct clusters. One cluster is near the point (100%,100%), corresponding to perfect agreement with the target wrapper program. The second cluster is near (0%,100%), and usually corresponds to a test case for which no data at all was extracted.<sup>6</sup> The third cluster is near (50%,100%) and represents an interesting type of error: for most pages in the cluster, the learned wrapper extracted the anchor nodes correctly, but incorrectly assumed that the text node was identical to the anchor node. We note that in many cases, the choice of how much context to include in the description  $s_i$  of a URL  $u_i$  is somewhat arbitrary, and hand-examination of a sample these results showed that the choices made by the learned system are usually not unreasonable; therefore it is probably appropriate to consider results in this cluster as qualified successes, rather than failures.

For an information integration system like WHIRL—one which is somewhat tolerant to imperfect extraction—many of these results would acceptably accurate. In particular, results near either the (100%,100%) or (50%,100%) clusters are probably good enough for WHIRL’s purposes. In aggregating these results, we thus considered two levels of performance. A learned extraction heuristic has *perfect* performance on a page  $p_i$  if recall and precision are both 1. An extraction heuristic has  $\epsilon$ -*good* performance on a page  $p_i$  if recall and precision are both at least  $1 - \epsilon$ , or if precision is at least  $1 - \epsilon$  and recall is at least  $1/2 - \epsilon$ . The table in Figure 3 shows the number of perfect and  $\epsilon$ -good results in conducting the leave-one-out

<sup>5</sup>More precisely, for each point  $(x, y)$  (where  $x$  is recall and  $y$  is precision on some problem) we plot  $(x + R_1, y + R_2)$ , where  $R_1$  and  $R_2$  are picked uniformly at random from the interval  $[-0.05, 0.05]$ . Note that adding noise in this way can produce impossible recall-precision combinations, such as a precision greater than 1, or a recall of 0 and precision of 0.9.

<sup>6</sup>If nothing as labeled as positive by the classifier, we score the precision as 100%, even though precision is strictly speaking undefined in this case.

		lv1-domain-out		lv1-page-out (baseline)		Intra-domain lv1-page-out	
Domain	#pages	#perfect	#good	#perfect	#good	#perfect	#good
birds	41	13	22	12	23	13	23
games	25	8	9	10	12	10	11
movies	9	1	3	3	4	5	6
news	9	2	2	1	2	0	0
Total	84	24	36	26	41	28	40
(as percent)		29%	43%	31%	49%	33%	48%

Table 1: Performance of RIPPER on leave-one-out variants, by domain

		lv1-domain-out		lv1-page-out		Intra-domain lv1-page-out	
Domain	#pages	#perfect	#good	#perfect	#good	#perfect	#good
birds	41	10	21	12	24	15	25
games	25	6	7	7	8	7	10
movies	9	1	1	3	5	8	8
news	9	4	4	4	4	0	0
Total	84	21	33	26	41	30	43
(as percent)		25%	39%	31%	49%	36%	51%

Table 2: Performance of CART on leave-one-out experiments

experiment above.

We will use  $\epsilon = 5\%$  as a baseline performance threshold for later experiments; however, as shown in Figure 3, the number of  $\epsilon$ -good pages does not change much as  $\epsilon$  is varied (because the clusters are so tight). We believe that this sort of aggregation is more appropriate for measuring overall performance than other common aggregation schemes, such as measuring average precision and recall. On problems like this, a system that finds perfect wrappers only half the time and fails abjectly on the remaining problems is much more useful than a system which is consistently mediocre.

### 3.2 Variations on leave-one-page-out

In the leave-one-page-out experiment, when extraction performance is tested on a page  $p_i$ , the learned extraction program has no knowledge whatsoever of  $p_i$  itself. However, it may well be the case that the learner has seen examples of only slightly different pages—for instance, pages from the same Web site, or Web pages from different sites that present similar information. So it is still possible that the learned extraction heuristics are to some extent specialized to the benchmark problems from which they were generated, and would work poorly in a novel application domain.

We explored this issue by conducting two variants of the leave-one-page-out experiment. The first variant is a “leave-one-domain-out” experiment. Here we group the pages by domain, and for each domain, test performance of the extraction heuristics obtained by

training on the other three domains. If the extraction heuristics were domain-specific, then one would expect to see markedly worse performance; in fact, the performance degrades only slightly. (Note also that less training data is available in the “leave-one-domain-out” experiments, another possible cause of degraded performance.) These results shown in the leftmost section of Table 1.

The second variant is presented in the rightmost section of Table 1, labeled as the “intra-domain leave-one-page-out” experiment. Here we again group the pages by domain, and perform a separate leave-one-page-out experiment for each domain. Thus, in this experiment the extraction heuristics tested for page  $p_i$  are learned from only the most similar pages—the pages from the same domain. In this variant, one would expect a marked *improvement* in performance if the learned extraction heuristics were very domain- or site-specific. In fact, there is little change. These experiments thus support the conjecture that the learned extraction are in fact quite general.

We also explored using classification learners other than RIPPER. Table 2 shows the results for the same set of experiments using CART, a widely used decision tree learner.<sup>7</sup> CART achieves performance generally comparable to RIPPER. We also explored using C4.5 [17] and an implementation of Naive Bayes; however, preliminary experiments suggested that their performance was somewhat worse than both RIPPER and CART.

## 4 Comparison to page-specific wrapper learning

As noted in the introduction, in previous research in learning wrappers, a new wrapper has been trained for each new page format, using examples specific to that page. One possible way of training such a wrapper-induction system is the following. The user first labels the first few items that should be extracted from the list starting from the top of the page. These are assumed to be a complete list of items to be extracted up to this point; that is, it is assumed that any unmarked text preceding the last marked item should *not* be extracted. The learning system then learns a wrapper from these examples, and uses it to extract data from the remainder of the page. The learned wrapper can also be used for other pages with the same format as the page used in training.

For simple lists and hotlists, the approach outlined above in Section 2.2 can also be used to learn page-specific wrappers. The only difference is in the way that a dataset is constructed, and the circumstances in which the learned wrapper is used; in learning a page-specific wrapper, all training examples come from the page  $p$  being wrapped, and the learned classifier is only used to label parse tree nodes from  $p$  (or other pages with the same format as  $p$ ).

We performed the following experiment to test the effectiveness of such a learning system. For a wrapper/page pair  $w_i, p_i$ , we used  $w_i$  and  $p_i$  to build a labeled parse tree. Then, we traversed the tree in a left-to-right, depth-first order, collecting all nodes up to and including the  $K$ -th node with a positive label into a sample  $S_{i,K}$ ; this simulates asking the user to

---

<sup>7</sup>We used the implementation of CART supplied with the IND package. Since CART doesn’t handle set-valued attribute, we encoded sets with a bit-vector of boolean attributes, one for each possible set value. After this transformation, there are 235 attributes in the data set for CART.

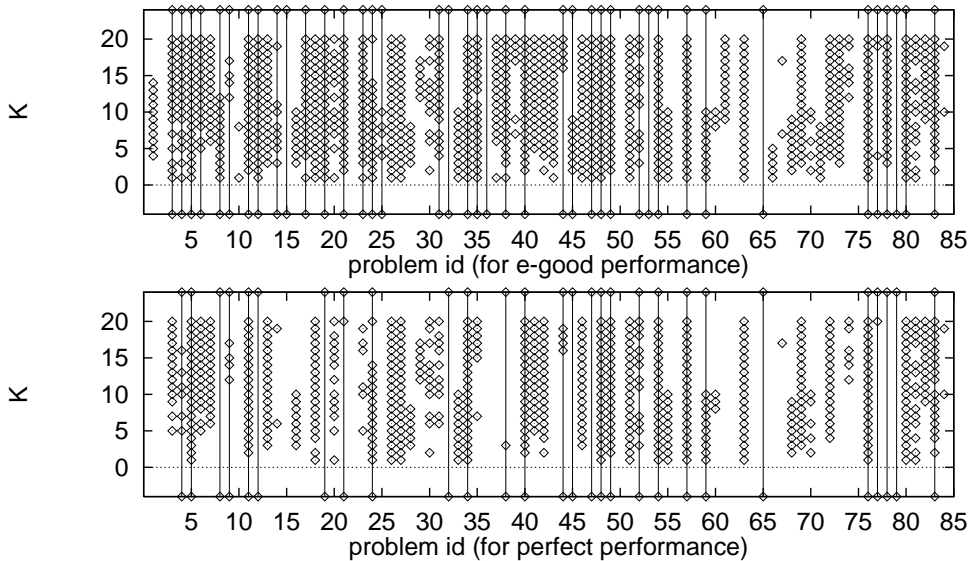


Figure 4: Details of the intra-page learning method *vs* the number of positive examples  $K$ . Each point is a successful result for the intra-page method. The vertical lines are drawn where the page-independent method succeeds. The top graph is for perfect performance, the bottom for good performance with  $\epsilon = 5\%$ .

label the first few sections of text to be extracted. We then trained a wrapper in the usual way on the sample  $S_{i,K}$ , and tested its performance on the remaining nodes. We repeated this procedure for all 84 wrapper/page pairs, and for  $K = 1, \dots, 20$ .

The results are shown in Figure 4. In each plot, we place a mark at row  $K$  and column  $i$  if the tested performance of the classifier trained on problem  $i$  with  $K$  positive examples exceeds some performance threshold; in the bottom plot, a mark is placed if performance is perfect, and in the top plot, a mark is placed if performance is  $\epsilon$ -good for  $\epsilon = 5\%$ . Also, a vertical line is placed at column  $i$  if the corresponding page-independent classifier exceeds the performance threshold; that is, if the extraction heuristic learned from all problems *other* than problem  $i$  had acceptable performance on problem  $i$ .

The figure illustrates a number of interesting points. One surprising point is that many of the extraction tasks handled correctly by the page-independent learned heuristics are difficult to learn using examples taken from the specific page being wrapped. For instance, problems 32, 65 and 72 are handled perfectly with the page-independent extractor; however, extraction performance with examples from these pages alone is not  $\epsilon$ -good, even with 20 positive examples. Thus the technique of learning page-independent extraction heuristics seem to be somewhat complementary to the technique of learning page-specific wrappers.

Another point is that, while labeling more examples does clearly help on average, performance improves erratically on many specific problems. In problems 29 and 30, for example, the learner appears to oscillate between performance that is perfect and performance that is not even  $\epsilon$ -good. Because the amount of training data on these problems is small, adding even a single new positive example—together with all negative examples that precede it—can change the statistics of the training data substantially.

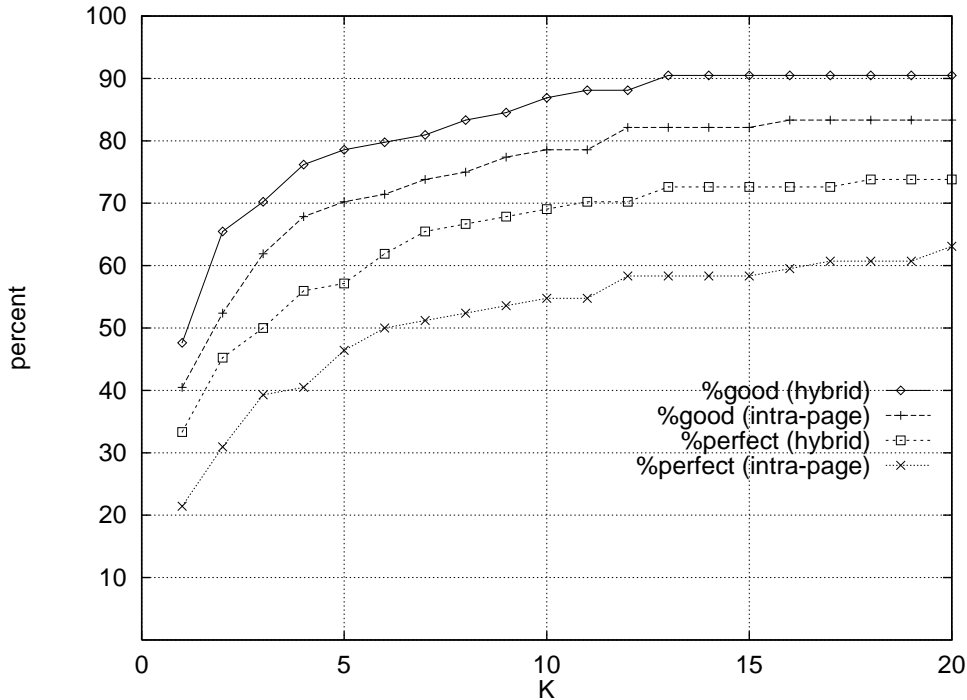


Figure 5: Performance of the intra-page learning method and the hybrid intra-page and page-independent learning method as the number of positive examples  $K$  is increased.

Based on these observations, we evaluated the performance of two possible methods for learning page-specific wrappers. In applying the *pure intra-page method* to problem  $i$ , the user is repeatedly prompted for the next positive example, until the wrapper learned from the existing dataset meets some performance goal. At this point learning stops. We simulated the intra-page method experimentally for two performance goals (perfection, and  $\epsilon$ -goodness) using the depth-first left-to-right tree traversal discussed above to simulate user input. The results are shown in the two curves labeled “intra-page” in Figure 5; for each value  $K$ , we show the percentage of benchmark problems that meet the performance goal with  $K$  or fewer examples.

In the *hybrid method*, the system first learns a page-independent classifier from all pages other than page  $i$ , and then attempts to wrap that page with the learned extraction heuristics. If the user accepts this wrapper—in simulation, if the performance goal is reached—then learning stops. Otherwise, the user is repeatedly prompted for the next positive example, as in intra-page learning, until the learned wrapper meets the performance goal. The results are shown in the two curves labeled “hybrid” in Figure 5; however, for each value  $K$ , we show the percentage of benchmark problems that meet the performance goal with  $K$  or fewer *user inputs*, rather than  $K$  or fewer examples. The first “user input” is the acceptance or rejection of the page-independent wrapper—this query is thus charged equally with labeling an example.

The graph shows that the hybrid method offers a substantial advantage over the pure intra-page method; in many situations, one would have to label double or triple the number of examples to obtain a comparable same set of wrappers. For instance, one can obtain perfect

wrappers for 60% of the problems with only 6 user inputs for the hybrid system, while for the intra-page system, 15-20 labeled examples are necessary; also, the hybrid system gets acceptable performance on 80% of the benchmarks after seeing only 6 training examples, while the intra-page system requires 12 training examples to do as well.

Similar results were obtained with CART (not shown), except that CART’s performance was worse for very small values of  $K$ .

## 5 Conclusion

We have described a method for learning general, page-independent heuristics for extracting data from (“wrapping”) HTML documents. The input to our learning system is a set of working wrapper programs, paired with HTML pages they correctly wrap. The output is a general, page-independent heuristic procedure for extracting data. Page formats that are correctly “wrapped” by the learned heuristics can be incorporated into a knowledge base with minimal human effort; it is only necessary to indicate where in the knowledge base the extracted information should be stored. In contrast, other wrapper-induction methods require a human teacher to train them on each new page format.

More specifically, we defined two common types of extraction problems—extraction of simple lists and simple hotlists—which together comprise nearly 75% of the wrappers required in experiments with WHIRL. We showed that learning these types of wrappers can be reduced to a classification problem. Using this reduction and standard-off-the-shelf learners, we were able to learn extraction heuristics which worked perfectly on about 30% of the problems in a large benchmark collection, and which worked well about about 50% of the problems. The extraction heuristics were also demonstrated to be domain-independent. Finally, we showed that page-independent extraction heuristics are complementary to more traditional methods for learning wrappers, and that a simple combination of these methods can substantially improve the performance of a page-specific wrapper learning method.

Ashish and Knoblock [1] propose heuristics for detecting hierarchical structure in an HTML document; obtaining this structure can facilitate programming a wrapper. These heuristics were designed manually, rather than acquired by learning as in our system.

One of the authors of this paper has also evaluated certain hand-coded heuristic methods for detecting lists and hotlists in HTML pages [6]. Briefly, this work considers a very restricted class of possible wrapper programs—one chosen by careful analysis of the 84 extraction problems used in this study. Given an HTML page, it is possible to enumerate all wrapper programs in the restricted class, and then rank the enumerated wrappers according to various heuristic measures. Cohen’s results show that some natural ranking heuristics perform relatively poorly on this task: for instance, the wrapper that extracts the longest list is correct only 18% of the time. However, more complex heuristics, encoded in a special-purpose logic, perform as well as or better than the learning approach discussed here. An advantage of the learning approach (relative to the ranking approach [6]) is that the learned heuristics are obtained without manual engineering, and hence can be more readily adapted to variations of the extraction problem. The learning approach is also applicable to a broader class of extraction programs.

Earlier work on automatic extraction of data from documents include heuristic methods for recognizing tables given a bitmap-level representation of a document [18]. These methods rely on finding rows and columns of white space in the bitmap; and would seem to be relatively expensive to apply directly to HTML documents. Our method also has the advantage that it can extract information not clearly visible to the human reader, such as URLs that appear only as attribute values in the HTML. The heuristics devised by Rus and Subramanian were also designed manually, rather than acquired by learning.

The results of this paper raise a number of questions for further research. The breadth of the system could be improved by devising labeling schemes for other sorts of extraction tasks. Performance on lists and hotlists could perhaps be improved by using more powerful learning methods, or learning methods that exploit more knowledge about this specific learning task. Finally, it might be useful to couple these automatic extraction methods with methods that automatically determine if a Web page contains a list or hotlist, and methods that automatically associate types with the extracted data. This combination of methods could further lower the cost of fielding an information integration system for a new domain.

## References

- [1] Naveen Ashish and Craig Knoblock. Wrapper generation for semistructured Internet sources. In Dan Suciu, editor, *Proceedings of the Workshop on Management of Semistructured Data*, Tucson, Arizona, May 1997. Available on-line from <http://www.research.att.com/~suciu/workshop-papers.html>.
- [2] William W. Cohen. Fast effective rule induction. In *Machine Learning: Proceedings of the Twelfth International Conference*, Lake Tahoe, California, 1995. Morgan Kaufmann.
- [3] William W. Cohen. Learning with set-valued features. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, Portland, Oregon, 1996.
- [4] William W. Cohen. Integration of heterogeneous databases without common domains using queries based on textual similarity. In *Proceedings of ACM SIGMOD-98*, Seattle, WA, 1998.
- [5] William W. Cohen. A Web-based information system that reasons with structured collections of text. In *Proceedings of Autonomous Agents-98*, St. Paul, MN, 1998.
- [6] William W. Cohen. Recognizing structure in web pages using similarity queries. Submitted for publication, 1999.
- [7] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, and J. Widom. The TSIMMIS approach to mediation: Data models and languages (extended abstract). In *Next Generation Information Technologies and Systems (NGITS-95)*, Naharia, Israel, November 1995.
- [8] Michael Genesereth, Arthur Keller, and Oliver Dushka. Infomaster: an information integration system. In *Proceedings of the 1997 ACM SIGMOD*, May 1997.

- [9] J. Hammer, H. Garcia-Molina, J. Cho, and A. Crespo. Extracting semistructured information from the Web. In Dan Suciu, editor, *Proceedings of the Workshop on Management of Semistructured Data*, Tucson, Arizona, May 1997. Available on-line from <http://www.research.att.com/~suciu/workshop-papers.html>.
- [10] Chun-Nan Hsu. Initial results on wrapping semistructured web pages with finite-state transducers and contextual rules. In *Papers from the 1998 Workshop on AI and Information Integration*, Madison, WI, 1998. AAAI Press.
- [11] Craig A. Knoblock, Steven Minton, Jose Luis Ambite, Naveen Ashish, Pragnesh Jay Modi, Ion Muslea, Andrew G. Philpot, and Sheila Tejada. Modeling web sources for information integration. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, Madison, WI, 1998.
- [12] Nicholas Kushmerick, Daniel S. Weld, and Robert Doorenbos. Wrapper induction for information extraction. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence*, Osaka, Japan, 1997.
- [13] Zoe Lacroix, Arnaud Sahuguet, and Raman Chandrasekar. User-oriented smart-cache for the web: what you seek is what you get. In *Proceedings of the 1998 ACM SIGMOD*, June 1998.
- [14] Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proceedings of the 22nd International Conference on Very Large Databases (VLDB-96)*, Bombay, India, September 1996.
- [15] Giansalvatore Mecca, Paolo Atzeni, Alessandro Masci, Paolo Merialdo, and Giuseppe Sindoni. The ARANEUS web-base management system. In *Proceedings of the 1998 ACM SIGMOD*, June 1998.
- [16] Ion Muslea, Steven Minton, and Craig Knoblock. Wrapper induction for semistructured, web-based information sources. In *Proceedings of the Conference on Automated Learning and Discovery (CONALD)*, 1998.
- [17] J. Ross Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann, 1994.
- [18] D. Rus and D. Subramanian. Customizing information capture and access. *ACM transactions on information systems*, 15(1):67–101, Jan 1997.
- [19] Anthony Tomasic, Remy Amouroux, Philippe Bonnet, and Olga Kapitskaia. The distributed information search component (Disco) and the World Wide Web. In *Proceedings of the 1997 ACM SIGMOD*, May 1997.