A Flexible Learning System for Wrapping Tables and Lists in HTML Documents

William W. Cohen
WhizBang Labs
4616 Henry Street
Pittsburgh, PA 15213
wcohen@whizbang.com

Matthew Hurst
WhizBang Labs
4616 Henry Street
Pittsburgh, PA 15213
mhurst@whizbang.com

Lee S. Jensen*
WhizBang Labs
3210 N. Canyon Road
Provo, UT 84604
lee.jensen@whizbang.com

ABSTRACT

A program that makes an existing website look like a database is called a *wrapper*. *Wrapper learning* is the problem of learning website wrappers from examples. We present a wrapper-learning system called WL^2 that can exploit several different representations of a document. Examples of such different representations include DOM-level and token-level representations, as well as two-dimensional geometric views of the rendered page (for tabular data) and representations of the visual appearance of text as it will be rendered. Additionally, the learning system is modular, and can be easily adapted to new domains and tasks. The learning system described is part of an "industrial-strength" wrapper management system that is in active use at WhizBang Labs. Controlled experiments show that the learner has broader coverage and a faster learning rate than earlier wrapper-learning systems.

Categories and Subject Descriptors

I.2.6 [**Learning**]: Concept learning; H.2.4 [**System**]: Textual databases; H.2.8 [**Database applications**]: Data mining

General Terms

Algorithms, Experimentation

Keywords

Record linkage, reference matching, canopy, learning

1. INTRODUCTION

Manywebsites contain large quantities of highly structured, database-like information. It is often useful to be able to access these websites programmatically, as if they were true databases. A program that accesses an existing website and makes that website act like a database is called a *wrapper*. *Wrapper learning* is the problem of learning website wrappers from examples [16, 22].

*Current address: NextPage Corporation, 3125 W. Executive Parkway, Lehi, UT, 84043, lee.jensen@nextpage.com.

WWW2002, May 7–11, 2002, Honolulu, Hawaii, USA.
ACM 1-58113-449-5/02/0005.

Check out this K00L Stuff!!!

"Actresses"			
Lucy	Lawless	images	links
Angelina	Jolie	images	links
...	...	...	...
"Singers"			
Madonna		images	links
Brittany	Spears	images	links
...	...	...	...

Last modified: 11/1/01.

Figure 1: A difficult page to wrap.

In this paper we will discuss some of the more important representational issues for wrapper learners, focusing on the specific problem of extracting text from web pages. We argue that pure DOM- or token-based representations of web pages are inadequate for the purpose of learning wrappers.

We then propose a learning system that can exploit multiple document representations. Additionally, this learning system is extensible: it can be easily "tuned" to a new domain by adding new learning components. In more detail, the system includes a single general-purpose "master learning algorithm" and a varying number of smaller, special-purpose "builders", each of which can exploit a different view of a document. Implemented builders make use of DOM-level and token-level views of a document; views that take more direct advantage of visual characteristics of rendered text, like font size and font type; and views that exploit a high-level geometric analysis of tabular information. Experiments show that the learning system achieves excellent results on real-world wrapping tasks, as well as on artificial wrapping tasks previously considered by the research community.

2. ISSUES IN WRAPPER LEARNING

One important challenge faced in wrapper learning is picking the representation for documents that is most suitable for learning. Most previous wrapper learning systems represent a document as a linear sequence of tokens or characters [22, 3]. Another possible scheme is to represent documents as trees, for instance using the document-object model (DOM). This representation is used by a handful of wrapper learning systems [7, 6] and many wrapper programming languages (*e.g.*, [27]).

Unfortunately, both of these representations are imperfect. In a website, regularities are most reliably observed in the view of the information seen by human readers–that is, in the rendered document. Since the rendering is a two-dimensional image, neither a linear representation nor a tree representation can encode it adequately.

One case in which this representational mismatch is important is the case of complex HTML tables. Consider the sample table of Figure 1. Suppose we wish to extract the third column of Figure 1. This set of items cannot easily be described at the DOM or token level: for instance, the best DOM-level description is probably "`td` nodes such that the sum of the column width of all left-sibling `td` nodes is 2, where column width is defined by the `colspan` attribute if it is present, and is defined to be one otherwise." Extracting the data items in the first column is also complex, since one must eliminate the "cut-in" table cells (those labeled "Actresses" and "Singers") from that column. Again, cut-in table cells have a complex, difficult-to-learn description at the DOM level ("`td` nodes such that no right-sibling `td` node contains visible text").

Another problemmatic case is illustrated by Figure 2. Here a rendering of a web page is shown, along with two possible HTML representations. In the first case, the HTML is very regular, and hence the artist names to be extracted can be described quite easily and concisely. In the second case, the underlying HTML is irregular, even though it has the same appearance when rendered. (Specifically, the author alternated between using the markup sequences ⟨i⟩⟨b⟩foo⟨/b⟩⟨/i⟩ and ⟨b⟩⟨i⟩bar⟨/i⟩⟨/b⟩ in constructing italicized boldfaced text.) This sort of irregularity is unusual in pages that are created by database scripts; however, it is quite common in pages that are created or edited manually.

In summary, one would like to be able to to concisely express concepts like "all items in the second column of a table" or "all italicized boldfaced strings". However, while these concepts can be easily described in terms of the rendered page, they may be hard to express in terms of a DOM- or token-level representation.

3. AN EXTENSIBLE WRAPPER LEARNING SYSTEM

3.1 Architecture of the Learning System

The remarks above are not intended to suggest that DOM and token representations are bad—in fact they are often quite good. We claim simply that neither is sufficient to successfully model all wrappers concisely. In view of this, we argue that an ideal wrapper-learning system will be able to exploit several different representations of a document—or more precisely, several different views of a single highly expressive baseline representation.

In this paper we will describe such a learning system, called the WhizBang Labs Wrapper Learner (WL^2). The basic idea in WL^2 is to express the bias of the learning system as an ordered set of "builders". Each "builder" is associated with a certain restricted language L. However, the builder for L is not a learning algorithm for L. Instead, to facilitate implementation of new "builders", a separate master learning algorithm handles most of the real work of learning, and builders need support only a small number of operations on L. Builders can also be constructed by composing other builders in certain ways. For instance, two builders for languages L_1 and L_2 can be combined to obtain builders for the language $(L_1 \circ L_2)$, or the language $(L_1 \wedge L_2)$.

We will describe builders for several token-based, DOM-based, and hybrid representations, as well as for representations based on properties of the expected rendering of a document. Specifically, we will describe builders for representations based on the expected formatting properties of text nodes (font, color, and so on), as well as representations based on the expected geometric layout of tables in HTML.

We finally note that an extendible learner has other advantages. One especially difficult type of learning problem is illustrated by the example page of Figure 3, where the task is to extract "office locations". Only two examples are available, and there are clearly many generalizations of these, such as: "extract all list items", "extract all list items starting with the letter P", etc. However, not all generalizations are equally useful. For instance, if a new office in "Mountain View, CA" were added to the web page, some generalizations would extract it, and some would not.

In order to obtain the most desirable of the many possible generalizations of the limited training data, most previous wrapper-learning systems have been carefully crafted for the task. Another advantage of an extensible learning architecture is that it allows a wrapper-learning system to be tuned in a principled way.

3.2 A Generic Representation for Structured Documents

We will begin with a general scheme for describing subsections of a document, and then define languages based on restricted views of this general scheme.

We assume that structured documents are represented with the *document object model* (DOM). (For pedagogical reasons we simplify this model slightly in our presentation.) A *DOM tree* is an ordered tree, where each node is either an *element node* or a *text node*. An *element node* has an ordered list of zero or more child nodes, and contains a string-valued *tag* (such as `table`, `h1`, or `li`) and also zero more string-valued *attributes* (such as `href` or `src`). A *text node* is normally defined to contain a single *text string*, and to have no children. To simplify the presentation, however, we will assume that a text node containing a string s of length k will have k "character node" children, one for each character in s.

Items to be extracted from a DOM tree are represented as *spans*. A *span* consists of two *span boundaries*, a *right boundary* and a *left boundary*. Conceptually, a boundary corresponds to a position in the structured document. We define a *span boundary* to be a pair (n,k), where n is a node and k is an integer. A span boundary points to a spot

WheezeBong.com: Contact info

Currently we have offices in two locations:

- Pittsburgh, PA
- Provo, UT

Figure 3: A sample web page. Notice that only two examples of "location" exist.

Rendered page:

My Favorite Musical Artists

- ***Muddy Waters***
- ***John Hammond***
- ***Ry Cooder***
- ...

Last modified: 11/1/01.

HTML implementation 1:

⟨h3⟩My Favorite Musical Artists⟨/h3⟩
⟨ul⟩
⟨li⟩⟨i⟩⟨b⟩Muddy Waters⟨/b⟩⟨/i⟩
⟨li⟩⟨i⟩⟨b⟩John Hammond⟨/b⟩⟨/i⟩
⟨li⟩⟨i⟩⟨b⟩Ry Cooder⟨/b⟩⟨/i⟩
⟨li⟩...
⟨/ul⟩
⟨p⟩
Last modified: 11/1/01

HTML implementation 2:

⟨h3⟩My Favorite Musical Artists⟨/h3⟩
⟨ul⟩
⟨li⟩⟨i⟩⟨b⟩Muddy Waters⟨/b⟩⟨/i⟩
⟨li⟩⟨b⟩⟨i⟩John Hammond⟨/i⟩⟨/b⟩
⟨li⟩⟨i⟩⟨b⟩Ry Cooder⟨/b⟩⟨/i⟩
⟨li⟩...
⟨/ul⟩
⟨p⟩
Last modified: 11/1/01

Figure 2: A rendered page, with two HTML implementations. The second implementation exhibits irregularity at the DOM level, even though the rendering has a regular appearance.

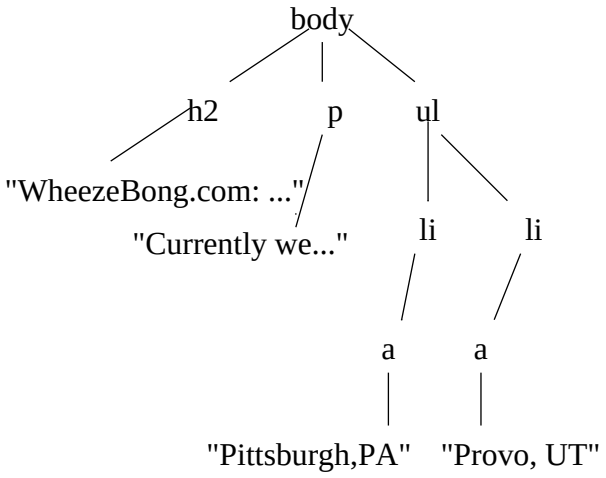

Figure 4: A sample DOM tree.

between the k-th and the $(k+1)$-th child of n. For example, if n_1 is the rightmost `text` node in Figure 4, then $(n_1, 0)$ is before the first character of the word "Provo", and $(n_1, 5)$ is after the last character of the word "Provo". The span with left boundary $(n_1, 0)$ and right boundary $(n_1, 5)$ corresponds to the text "Provo".

As another example, if n_2 is the leftmost `li` node in Figure 4, then the span from $(n_2, 0)$ to $(n_2, 1)$ contains the text "Pittsburgh, PA". It also corresponds to a single DOM node, namely, the leftmost anchor (`a`) node in the DOM tree. A span that corresponds to a single DOM node is called a *node span*.

3.3 A Generic Representation for Extractors

A *predicate* $p_i(s_1, s_2)$ is a binary relation on spans. To *execute a predicate* p_i *on span* s_1 means to compute the set $EXECUTE(p_i, s_1) = \{s_2 : p_i(s_1, s_2)\}$. For example, consider a predicate $p(s_1, s_2)$ which is defined to be true iff (a) s_1 contains s_2, and (b) s_2 is a node span corresponding to an element node with tag `li`. Let s_1 be a span encompassing the entire document of Figure 4. Then $EXECUTE(p, s_1)$ contains two spans, each corresponding to an `li` node in the DOM tree, one containing the text "Pittsburgh, PA", and one containing the text "Provo, UT".

We will assume here that every predicate is one-to-many and that membership in a predicate can be efficiently decided (i.e., given two spans s_1 and s_2, one can easily test if $p(s_1, s_2)$ is true.) We also assume that predicates are *executable*—i.e., that $EXECUTE(p, s)$ can be efficiently computed for any initial span s. The extraction routines learned by our wrapper induction system are represented as executable predicates. Since predicates are simply sets, it is possible to combine predicates by Boolean operations like conjunction or disjunction; similarly, one can naturally say that predicate p_i is "more general than" predicate p_j.

We note that these semantics can be used for many commonly used extraction languages, such as regular expressions and XPath queries.[1] Many of the predicates learned by the system are stored as equivalent regular expressions or XPath queries.

3.4 Representing Training Data

A wrapper induction system is typically trained by having a user identify items that should be extracted from a page. Since it is inconvenient to label all of a large page, a user should have the option of labeling some initial section of a page. To generate negative data, it is assumed that the user completely labeled the page or an initial section of it.

A *training set* $\mathcal{T}$ for our system thus consists of a set of triples $(Outer_1, Scope_1, InnerSet_1)$, $(Outer_2, Scope_2, InnerSet_2)$, ..., where in each pair $Outer_i$ is usually a span corresponding to a web page, $Scope_i$ is the part of $Outer_i$ that the user has completely labeled, and $InnerSet_i$ is the set of all spans that should be extracted from $Outer_i$.

Constructing positive data from a training set is trivial. The positive examples are simply all pairs $\{(Outer_i, Inner_{ij}) : Inner_{ij} \in InnerSet_i\}$. When it is convenient we will think of $\mathcal{T}$ as this set of pairs.

While it is not immediately evident how negative data can be constructed, notice that any hypothesized predicate p can be tested for consistency with a training set $\mathcal{T}$ by simply executing it on each outer span in the training set. The spans in the set $InnerSet_i - EXECUTE(p, Outer_i)$ are false negative predictions for p, and the false positive predictions for p are spans s in the set

$$\{s \in EXECUTE(p, Outer_i) - InnerSet_i : \text{contains}(Scope, s)\} \quad (1)$$

3.5 Designing a Bias

The bias of the learning system is represented by an ordered list of *builders*. Each builder $\mathcal{B}_L$ corresponds to a cer-

[1] XPath is a widely-used declarative language for addressing nodes in an XML (or XHTML) document[4].

tain restricted extraction language[2] L. To give two simple examples, consider these restricted languages:

- L_{bracket} is defined as follows. Each concept $c \in L_{\text{bracket}}$ is defined by a pair (ℓ, r), where ℓ and r are strings. Each pair corresponds to a predicate $p_{\ell,r}(s_1, s_2)$, which is true iff s_2 is contained in s_1; the string corresponding to s_2 is preceded by the string ℓ; and the string corresponding to s_2 is followed by the string r.

  For example, executing the predicate $p_{in,locations}$ on the span for the document of Figure 3 would produce a single span containing the text "two". L_{bracket} is one example of a language based on viewing the document as a sequence of tokens.

- L_{tagpath} is defined as follows. Each concept $c \in L_{\text{tagpath}}$ is defined by a sequence of strings, $t_1, \ldots, t_k$, and corresponds to a predicate $p_{t_1,\ldots,t_k}$. The predicate $p_{t_1,\ldots,t_k}(s_1, s_2)$ is true iff s_2 is a node span contained in s_1; the tag of the node n_2 corresponding to s_2 is t_k; and for $1 \leq j \leq k-1$, the tag of the j-th ancestor of n_2 is t_{k-j}.

  For example, executing the predicate $p_{\texttt{ul,li,a}}$ on the span for the document of Figure 3 would produce the two spans "Pittsburgh, PA" and "Provo, UT". L_{tagpath} is an example of a language based viewing the document as a DOM.

Each builder $\mathcal{B}_L$ must implement two operations. A builder must be able to compute the *least general generalization* (LGG) of a training set $\mathcal{T}$ with respect to L—i.e., the most specific concept $c \in L$ that covers all positive training examples in $\mathcal{T}$. Given an LGG concept c and a training set $\mathcal{T}$, a builder must also be able to *refine* c *with respect to* $\mathcal{T}$—i.e., to compute a set of concepts $c'_1, \ldots, c'_m$ such that each c'_k covers some but not all of the positive examples $(Outer_i, Inner_{ij}) \in \mathcal{T}$.

Below we will write these operations as $LGG_{\mathcal{B}}(\mathcal{T})$ and $REFINE_{\mathcal{B}}(c, \mathcal{T})$. We will also assume that there is a special "top predicate", written "`true`", which is always true (and hence is not executable.)

Other builders will be described below, in Sections 4.1, 4.2, and 4.3.

3.6 The Master Learning Algorithm

The master learning algorithm used in WL^2 is shown in Figure 5. It takes two inputs: a training set $\mathcal{T}$, and an ordered list of builders. The algorithm is based on FOIL [24, 26] and learns a DNF expression, the primitive elements of which are predicates. As in FOIL, the outer loop of the learning algorithm (the `learnPredicate` function) is a set-covering algorithm, which repeatedly learns a single "rule" p (actually a conjunction of builder-produced predicates) that covers some positive data from the training set, and then removes the data covered by p. The result of `learnPredicate` is the disjunction of these "rules".

The inner loop (the `learnConjunction` function) first evaluates all LGG predicates constructed by the builders. If any LGG is consistent with the data, then that LGG is returned. If more than one LGG is consistent, then the LGG produced by the earliest builder is returned. If no LGG is consistent, the "best" one is chosen as a the first condition in a "rule". Executing this "best" predicate yields a set of spans, some of which are marked as positive in $\mathcal{T}$, and some of which are negative. From this point the learning process is quite conventional: the rule is specialized by greedily conjoining builder-produced predicates together. The predicate choices made in the inner loop are guided by the same information-gain metric used in FOIL.

There are several differences between this learning algorithm and FOIL. One important difference is the initial computation of LGG's using each of the builders. In many cases some builder's LGG is consistent, so often the learning process is quite fast. Builders are also used to generate primitive predicates in the `learnConjunction` function, instead of instead of testing all possible primitive predicates as FOIL does. This is useful since there are some languages that are difficult to learn using FOIL's top-down

[2]More precisely, we will use $\mathcal{L}$ to denote both a set of predicates, and a notation for describing this set of predicates.

define learnPredicate($\mathcal{T}$,($\mathcal{B}_1, \ldots, \mathcal{B}_k$)):
  $p_* = \text{false}$
  while (there are positive examples in $\mathcal{T}$) **do**
    let $p = \text{learnConjunction}(\mathcal{T}, (\mathcal{B}_1, \ldots, \mathcal{B}_k))$
    let $p_* = p_* \vee p$
    // remove training examples covered by p
    for each $(Outer_i, InnerSet_i) \in \mathcal{T}$ **do**
      $InnerSet_i \leftarrow \{s_2 \in InnerSet_i : \neg p(Outer_i, s_2)\}$
    endfor
  endwhile
  return p_*
end definition

define learnConjunction($\mathcal{T}$,($\mathcal{B}_1, \ldots, \mathcal{B}_k$)):
  for $i = 1, \ldots, k$ **do**
    let $p_i = LGG_{\mathcal{B}_i}(\mathcal{T})$
    if (p_i is consistent with $\mathcal{T}$) **then**
      return p_i
    endif
  endfor
  // pick a predicate and generate candidate spans
  let $p_{i_{opt}}$ be the p_i that maximizes information gain on $\mathcal{T}$, breaking ties in favor of p_i's generated by earlier builders
  let POS and NEG be the true positive and false negative predictions of $p_{i_{opt}}$ on $\mathcal{T}$ (see Eq. 1)
  let $p = p_{i_{opt}}$
  // specialize the predicate p using POS, NEG
  while (NEG$\neq \emptyset$) **do**
    let $P = \bigcup_i \{p' : p' = LGG_{\mathcal{B}_i}(\text{POS})\} \cup \bigcup_i \{p' \in REFINE_{\mathcal{B}_i}(LGG_{\mathcal{B}_i}(\text{POS}), \text{POS})\}$
    let $p'_{i_{opt}}$ be the $p' \in P$ that maximizes information gain on POS,NEG, breaking ties in favor of p''s generated by earlier builders
    $p \leftarrow p \wedge p'_{i_{opt}}$
    // remove training examples not covered by p
    $\text{POS} \leftarrow \{(s_1, s_2) \in \text{POS} \cap p\}$
    $\text{NEG} \leftarrow \{(s_1, s_2) \in \text{NEG} \cap p\}$
  endwhile
  return p
end definition

Figure 5: The master learning algorithm.

approach. Extensive use of the *LGG* operation also tends to make learned rules fairly specific. This is advantageous in wrapper-learning since when a site changes format, it is usually the case that old rules will simply fail to extract any data; this simplifies the process of "regression testing" for wrappers [15].

Another difference is that WL^2 uses the ordering of the builders to prioritize the primitive predicates. Predicates generated by earlier builders are preferred to later ones, if their information gains are equal. Notice that because there are very few positive examples, there are many ties in the information-gain metric.

A final difference is the way in which negative data is generated. In our algorithm, negative data is generated after the first predicate of a "rule" is chosen, by executing the chosen predicate and comparing the results to the training set. After this generation phase, subsequent hypothesis predicates can be tested by simply matching them against positive and negative example pairs—a process which is usually much more efficient than execution.

3.7 Discussion and Related Work

A number of recent extraction systems work by generating and classifying candidate spans (*e.g.*, [9, 10]). Using *LGG* predicates to generate negative data is an variant of this approach: essentially, one *LGG* predicate is selected as a candidate span generator, and subsequent predicates are used to filter these candidates.

Certain other extraction systems cast extraction as an automata induction problem [11, 3]. As noted above, this sort of approach requires a commitment to one particular sequential view of the document—as a sequence of tokens. The approach taken here is somewhat more flexible, in that the document can be viewed (by different builders) as a DOM tree or as a token sequence.

Many of the ideas used in this learning system are adapted from work in inductive logic programming (ILP) [20, 8]. In particular, the approach of defining bias via a set of builders is reminiscent of earlier ILP work in declarative bias [5, 1]. The hybrid top-down/bottom-up learning algorithm is also broadly similar to some earlier ILP systems like CHILL [30]. The approach taken here avoids the computational complexities involved in ILP, while keeping much of the expressive power. We also believe that this approach to defining a learning system's bias is easier to integrate into a production environment than an approach based on a purely declarative bias language.

4. ADDITIONAL BUILDERS

4.1 Composite Builders

The builders described above are examples of *primitive builders*. It is also possible to construct new builders by combining other builders. In fact, one reason for using the only the *REFINE* and *LGG* operations in builders is that *LGG* and *REFINE* can often be defined compositionally.

One useful composite builder is a *chain builder*. Given two builders $\mathcal{B}_{L_1}$ and $\mathcal{B}_{L_2}$, a chain builder learns (roughly) the composition of L_1 and L_2.

For efficiency reasons we implemented a slightly restricted form of builder composition. A *chain builder* is a composite builder based on two builders and a user-provided *decomposition function* f_d. Intuitively, the decomposition function

Jobs at WheezeBong.com:

To apply:	Send c.v. via e-mail to headhunt@wheezebong.com or call (888)-555-BONG.

- Webmaster (New York). Perl,servlets a plus.
- Librarian (Pittsburgh). MLS required.
- Ditch Digger (Palo Alto). No experience needed.

Figure 6: An example web page.

takes as an argument the span s_2 to be extracted and returns an intermediate span s': i.e., $f_d(s_2) = s'$. The chain builder will learn concepts p of the form

$$p \equiv \{(s_1, s_2) : p_1(s_1, f_d(s_2)) \wedge p_2(f_d(s_2), s_2)\} \qquad (2)$$

where p_1 is in the language associated with $\mathcal{B}_1$ and p_2 is in the language associated with $\mathcal{B}_2$.

Given the decomposition function f_d, it is straightforward to define the necessary operations for a chain builder $\mathcal{B}_{1\circ 2,f_d}$ for two builders $\mathcal{B}_1$ and $\mathcal{B}_2$.

- $LGG_{\mathcal{B}_{1\circ 2,f_d}}(\mathcal{T})$ is computed as follows. The first step is to use f_d to decompose the training set into two training sets, one for p_1 and one for p_2. Each pair $Outer_i, Inner_{ij} \in \mathcal{T}$ will be replaced by a pair $(Outer_i, f_d(Inner_{ij}))$ in $\mathcal{T}_1$ and a pair $(f_d(Inner_{ij}), Inner_{ij})$ in $\mathcal{T}_2$.

  Given these training sets, one can next use $\mathcal{B}_1$ and $\mathcal{B}_2$ to compute the *LGG* for the composition. Let $p_1 = LGG_{\mathcal{B}_1}(\mathcal{T}_1)$ and $p_2 = LGG_{\mathcal{B}_2}(\mathcal{T}_2)$. Then $LGG_{\mathcal{B}_{1\circ 2,f_2}}(\mathcal{T}_1)$ is $p(s_1, s_2)$, where the set p is simply the set defined in Eq. 2.

- Let $p = (p_1 \circ p_2)_{f_d}$ denote the "composition" of p_1 and p_2 as defined in Eq 2, and let $\mathcal{T}_1$, $\mathcal{T}_2$ be as above. Then $REFINE_{\mathcal{B}_{1\circ 2,f_d}}((p_1 \circ p_2)_{f_d}, \mathcal{T}) = R_1 \cup R_2$ where $R_1 = \{(p_1 \circ p_2')_{f_d} : p_2' \in REFINE_{\mathcal{B}_2}(p_2, \mathcal{T}_2)\}$ and $R_2 = \{(p_1' \circ p_2)_{f_d} : p_1' \in REFINE_{\mathcal{B}_1}(p_1, \mathcal{T}_1)\}$

Less formally, refinements of the composition $(p_1 \circ p_2)_{f_d}$ are formed by refining either step of the chain (*e.g.*, p_1 or p_2).

Another combination is *conjunction*. Given builders B_{L_1} and B_{L_2}, it is straightforward to define a builder $B_{L_1 \wedge L_2}$ for the language of predicates of the form $p_1 \wedge p_2$ such that $p_1 \in L_1$ and $p_2 \in L_2$.

Another useful composite builder is a *filtered builder*. A filtered builder $\mathcal{B}_{q,L}$ extends a builder $\mathcal{B}_L$ with an arbitrary *training set query* q, and is defined as follows, where $c_\emptyset$ is a special null concept.

$$LGG_{\mathcal{B}_{q,L}}(\mathcal{T}) = \begin{cases} LGG_{\mathcal{B}_L}(\mathcal{T}) & \text{if } q(\mathcal{T}) \\ c_\emptyset & \text{otherwise} \end{cases}$$

$$REFINE_{\mathcal{B}_{q,L}}(c, \mathcal{T}) = \begin{cases} REFINE_{\mathcal{B}_L}(c, \mathcal{T}) & \text{if } q(\mathcal{T}) \\ \emptyset & \text{otherwise} \end{cases}$$

Informally, a filtered builder is "switched off" whenever the predicate q is not satisfied. Filtered builders can be used to introduce additional control information, for example, by restricting some builders to be run only on certain types of

extraction tasks, or only on certain very large (or very small) training sets.

The following examples help illustrate how composite builders might be used.

Example 1. Let $f_d^{container}(s_2)$ return the span corresponding to the smallest DOM node that contains s_2. Chaining together $\mathcal{B}_{L_{\text{tagpath}}}$ and $\mathcal{B}_{L_{\text{bracket}}}$ using the decomposition function $f_d^{container}$ is a new and more expressive extraction language. For instance, let the strings ℓ and r represent left and right parentheses, respectively. For the page of Figure 6, the composite predicate $p_{\texttt{ul,li}} \circ p_{\ell,r}$ would extract the locations from the job descriptions. Notice that $p_{\ell,r}$ alone would also pick out the area code "888".

Example 2. Let $f_d^{predecessor}(s_2)$ return the first "small" text node preceding s_2 (for some appropriate definition of "small"), and let L_{bow} be a language of bag-of-words classifiers for DOM nodes. For example, L_{bow} might include concepts like $p_{job,title}(s_1, s_2) \equiv$ "s_1 contains s_2 and s_2 contains the words 'job' and 'title'." Let L_{dist} contain classifiers that test the distance in the DOM tree between the nodes corresponding to s_1 and s_2. For example, L_{dist} might include concepts like $p_{1 \le d \le 3}(s_1, s_2) \equiv$ "there are between 1 and 3 nodes between s_1 and s_2 (in a postfix traversal of the tree)".

Chaining together $\mathcal{B}_{L_{\text{bow}}}$ and $\mathcal{B}_{L_{\text{dist}} \wedge L_{\text{tagpath}}}$ using the decomposition function $f_d^{predecessor}$ would lead to a builder that learns concepts such as the following $p(s_1, s_2)$:

> $p(s_1, s_2) \equiv s'$ is the first text node preceding s_2 that contains three or fewer words; s' contains the words "To" and "apply"; s_2 is between 1 and 4 nodes after s', and s_2 is reached from a_1 by a tagpath ending in `table`, `tr`, `td`.

For the sample page in Figure 6, this predicate might pick out the table cell containing the text: "Send c.v. via e-mail...".

4.2 Format-based Extraction

Figure 2 illustrates an important problem with DOM-based representations: while regularity in the DOM implies a regular appearance in the rendered document, regular documents may have very irregular DOM structures. In the figure, the markup sequences ⟨i⟩⟨b⟩foo⟨/b⟩⟨/i⟩ and ⟨b⟩⟨i⟩foo⟨/i⟩⟨/b⟩ both produce italicized boldfaced text, but have different token- and DOM-level representations. Alternating between them will lead to a document that is regular in appearance but irregular in structure. Our experience is that this sort of problem is quite common in small-to-medium sized web sites, where much of the content is hand-built or hand-edited.

Our solution to this problem is to construct builders that rely more directly on the appearance of rendered text. We achieve this with a mixture of document preprocessing and reasoning at learning time.

In a preprocessing stage, HTML is "normalized" by applying a number of transformations. For instance, the `strong` tag is replaced by the `b` tag, `em` is replaced by `i` tag, and constructs like `font=+1` are replaced by `font=`k (where k is the appropriate font-size based on the context of the node.) This preprocessing makes it possible to compute a number of "format features" quickly at each node that contains text. Currently these features include properties like font size, font color, font type, and so on.

A special builder then extracts nodes using these features. These properties are treated as binary features (*e.g.*, the property "font-size=3" is treated as a Boolean condition "fontSizeEqualsThree=true"). The format builder then produces as its LGG the largest common set of Boolean format conditions found for the inner spans in its training set. Refinement is implemented by adding a single feature to the LGG set.

4.3 Table-based Extraction

4.3.1 Recognizing Tables

Consider again the sample tables in Figure 1. We would like to provide the learner with the ability to form generalizations based on the geometry of the tables, rather than their HTML representation. This is important since text strings that are nearby in the rendered image (and thus likely to be closely related) need not be nearby in the HTML encoding.

The first step in doing this is to recognize "interesting" tables in a document. Specifically, we are interested in collections of data elements in which semantic relationships between these elements are indicated by geometric relationships—either horizontal or vertical alignment. These "interesting" tables must be distinguished from other uses of the HTML `table` element. (In HTML, tables are also used for arbitrary formatting purposes, for instance, to format an array of images.) For more detailed discussion refer to Hurst [13] or Wang [29].

To recognize this class of tables, we used machine learning techniques. Specifically, we learned to classify HTML `table` nodes as *data tables* ("interesting" tables) and *non-data tables*.[3]

We explored two types of features: those derived directly from the DOM view of the table, and those derived from an *abstract table model* built from the table. (The abstract table model is described below). The best classifier contains only the abstract table model features, which are: the number of rows and columns (discretized into the ranges 1, 2, 3, 4, 5, 6-10, and 11+); the proportion of cells with string content; and the proportion of *singular cells*. A singular cell is a cell which has unit size in terms of the logical grid on which the table is defined.

We collected a sample of 339 labeled examples. To evaluate performance, we averaged five trials in which 75% of the data was used for training and the remainder for testing. We explored several learning algorithms including multinomial Naive Bayes [17, 19], Maximum Entropy [23], Winnow [18, 2], and a decision tree learner modeled after C4.5 [25]. Of these, the Winnow classifier performs the best with a precision of 1.00, a recall 0.922, and an F-measure of 0.959.[4]

4.3.2 Exploiting Table Context

Table classification is not only the first step in table processing: it is also useful in itself. There are several builders that are more appropriate to apply outside a table than inside one, or vice versa. One example is builders like that of Example 2 in Section 4.1, which in Figure 6 learns to extract text shortly after the phrase "To apply:". This builder gen-

[3]Documents can contain subsections that appear to the reader as a single table, but in fact are not contained by a single `table` node. We will not consider this issue here.

[4]F-measure is the harmonic mean of recall and precision, i.e., $F = (2 \cdot \textit{recall} \cdot \textit{precision})/(\textit{recall} + \textit{precision})$.

Problem#	Problem Name	Examples Available	WIEN(=)	STALKER(≈)	WL^2(=)
S1	Okra	3335	46	1	1
S2	BigBook	4299	274	8	6
S3	AddressFinder	57	–	–	1
S4	QuoteServer	22	–	–	4

Table 1: Comparison of STALKER, WEIN, and WL^2. For the columns marked with a "≈", a wrapper with an accuracy of 97% or above is learned from k examples. For columns marked with a "=", a perfect wrapper is learned.

erally inappropriate inside a table—for instance, in Figure 1, it is probably not correct to generalize the example "Lawless" to "all tables cells appearing shortly after the string 'Lucy'".

A number of builders in WL^2 work like the builder of Example 2, in that the extraction is driven primarily by some nearby piece of text. These builders are generally restricted to apply only when they are outside a data table. This can be accomplished readily with filtered builders.

4.3.3 Modeling Tables

More complex use of tables in wrapper-learning requires knowledge of the geometry of the rendered table. To accomplish this, we construct an *abstract geometric model* of each data table. In an abstract geometric model, a table is assumed to lie on a grid, and every table cell is assumed to be a contiguous rectangle on the grid. An *abstract table model* is thus a set of *cells*, each of which is defined by the co-ordinates of the upper-left and lower-right corners, and a representation of the cell's contents. In the case of HTML tables, the contents are generally a single DOM node.

Since we aim to model the table *as perceived by the reader*, a table model cannot be generated simply by rendering the `table` node following the algorithm recommended by W3C [12]. Further analysis is required in order to capture additional table-like sub-structure visible in the rendered document. Examples of this type of structure include nested `table` elements, rows of `td` elements containing aligned list elements, and so on. Our table modeling system thus consists of several steps.

First, we generate a table model from a `table` node using a variation of the algorithm recommended by W3C. We then refine the resulting table model in the following ways.

Rationalization. HTML is often very noisy. In order to build a DOM structure it must first be cleaned up to produce syntactically correct HTML. This is done by the Tidy utility [28]. Due to the constraints of that task and the lack of adhesion to the correct use of `table` encoding in HTML, the Tidy step often generates extra table cells. These are detected and removed.

Complex cell analysis. Cells that contain structure which is common across a row (*e.g.*, nested tables, forced line breaks, `pre` encoded text, etc) are subdivided into appropriate sub-cells which are then inserted back into the table model.

Normalization. Any rows that have height greater than one are checked to ensure that they contain some unit height cells. If they do not, then the row height is reduced appropriately. An analogous process is used to normalize column width. This normalization is necessary when an explicit `rowspan` or `colspan` attribute is used to indicate multiple row or column spanning cells and the value of the attribute is higher than the total number of rows or columns actually spanned in the rendered table.

4.3.4 Exploiting the Table Models

To exploit the geometric view of a table that is encapsulated in an abstract table model, we choose certain properties to export to the learning system. Our goal was to choose a small but powerful set of features that could be *unambiguously* derived from tables. More powerful features from different aspects of the abstract table model were also considered, such as the classification of cells as data cells or header cells—however, determining these features would require a layer of classification and uncertainty, which complicates their use in wrapper-learning.

To export the table features to WL^2, we used the following procedure. When a page is loaded into the system, each `table` node is annotated with an attribute indicating the table's classification as a data table or non-table table. Each node in the DOM that acts as a cell in an abstract table is annotated with its logical position in the table model; this is expressed as two ranges, one for column position and one for row position. Finally, each `tr` node is annotated with an attribute indicating whether or not it contains a "cut-in" cell (like the "Actresses" and "Singers" cells in Figure 1.)

Currently this annotation is done by adding attributes directly to the DOM nodes. This means that builders can easily model table regularities by accessing attributes in the enriched, annotated DOM tree. Currently four types of "table builders" are implemented. The *cut-in header builder* represents sets of nodes by their DOM tag, and the bag of words in the preceding cut-in cell. For example, in the table of Figure 1, the bag of words "Actresses" and the tag `td` would extract the strings "Lucy", "Lawless", "images", "links", "Angelina", "Jolie", and so on. The *column header builder* and the *row header builder* are analogous. The fourth type of table builder is an extended version of the builder for the L_{tagpath} language, in which tagpaths are defined by a sequence of tags augmented with the values of the attributes indicating geometric table position and if a row is a cut-in. As an example, the "extended tagpath"

`table`,`tr`(cutIn='no'),`td`(colRange='2-2')

would extract the strings "Lawless", "Jolie", "Spears" (but not "Madonna", because her geometric column co-ordinates are "1-2", not "2-2".) Finally, the conjunction of this extended tagpath and the example cut-in expression above would extract only "Lawless" and "Jolie".

5. EXPERIMENTS

5.1 Comparison with Previous Work

To evaluate the learning system, we conducted a series of experiments. The first set of experiments compare WL^2 with previous wrapper-learning algorithms.

The discussion in this paper has been restricted to "binary extraction tasks", by which we mean tasks in which a yes/no decision is made for each substring in the document, indicating whether or not that substring should be extracted. There are several existing schemes for decomposing the larger problem of wrapping websites into a series of binary extraction problems [22, 14]. WL^2 is embedded in one such system. Thus, the basic evaluation unit is a "wrapper-learning problem", which can be broken into a set of "binary extraction problems".

Muslea *et al.* [21] provide a detailed comparison of STALKER and WIEN on a set of four sample wrapper-learning problems. STALKER [21] is a wrapper-learning system which learns wrappers expressed as "landmark automata". WEIN [16] is an earlier wrapper-learning system. The sample problems were chosen as representative of the harder extraction problems to which WIEN was applied.

In the experiments of Muslea *et al.*, STALKER is repeatedly run on a sample of k labeled records, for $k = 1, 2, .., 10$, and then tested on all remaining labeled records. The process of gradually incrementing k was halted when the wrapper's average accuracy is 97% or better (averaging over the different samples of k training examples). The value of k shown in the column labeled "STALKER($\approx$)" of Table 1 shows the number of examples required for STALKER to achieve 97% accuracy. (This value is taken from Muslea *et al.*) The value of k shown in the column labeled WIEN(=) is Muslea *et al*'s estimate of the number of examples needed by WIEN to learn an exact (100% accurate) wrapper. Note that neither WIEN nor STALKER successfully learns wrappers for problems S3 and S4.

To perform the same flavor of evaluation, we ran WL^2 on the same four problems. We wish to emphasize that WL^2 was developed using completely different problems as benchmarks, and hence these problems are a fair prospective test of the system. In the column labeled "WL^2(=)", we show the number of examples k required to obtain perfect accuracy on every binary extraction problems associated with a wrapper-learning task. Unlike Muslea *et al.* we did not average over multiple runs: however, informal experiments suggest that performance of WL^2 is quite stable if different subsets of the training data are used.[5]

Although no result is not shown in the table, WL^2 can also be used to learn approximate wrappers. On these problems, WL^2 learns 95%-accurate wrappers from only two examples for all of the problems from Muslea *et al* but one. The most "difficult" problem is S2, which requires six examples to find even an approximate wrapper. This is due to the fact that many fields on this web page are optional, and it requires several records before every field has been seen at least once.

We now turn to some more additional benchmark problems. Table 2 gives the performance of WL^2 on several real-world wrapper-learning problems, taken from two domains for which WL^2 has been used internally at WhizBang Labs. The first seven problems are taken from the domain of job postings. The last six problems are taken from the domain of continuing education courses. These problems were selected as representative of the more difficult wrapping problems encountered in these two domains. Each of these problems contains several binary extraction problems—a total of 34 problems all told.

[5]A second reason for picking a single sample is that the user interface imposes a default ordering on the pages of each type, and most users label pages following this ordering. Hence, by using the default ordering to select the training data, the experiments more closely model the data that would be seen in actual use.

Problem	WL^2	Problem	WL^2
JOB1	3	CLASS1	1
JOB2	1	CLASS2	3
JOB3	1	CLASS3	3
JOB4	2	CLASS4	3
JOB5	2	CLASS5	6
JOB6	9	CLASS6	3
JOB7	4		
median	2	median	3

Table 2: Performance of WL^2 on representative wrapping problems occurring in actual use.

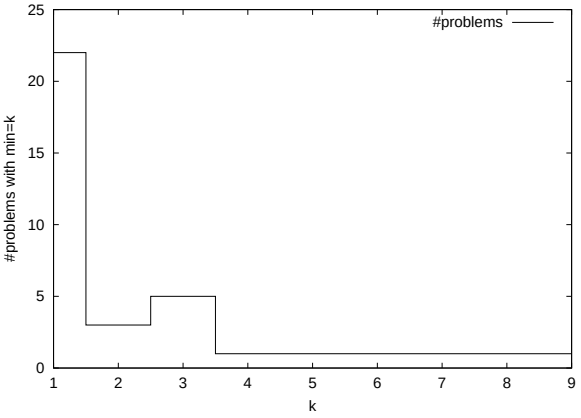

Figure 8: Histogram showing the minimum number of examples needed for each field-extraction problem.

5.2 Performance on Real-World Wrapping Tasks

Along with each problem we record the minimum number of labeled records needed to learn a wrapper with 100% accuracy. The largest number of examples needed is nine (for one field of an extremely irregular site) and the median number of examples is between 2 and 3. Figure 8 gives some additional detail: it plots the number of field-extraction problems that required a minimum of k labeled records, for value of k. About two-thirds of the binary extraction problems could be learned with *one* example, and about four-fifths could be learned with three examples.

In some cases, it is useful to obtain approximate wrappers, as well as perfect ones. To measure the overall quality of wrappers, we measured the *recall* and *precision* of the wrappers learned for each problem from k examples, for k =1, 2, 3, 5, 10, 15, and 20. Recall and precision were measured

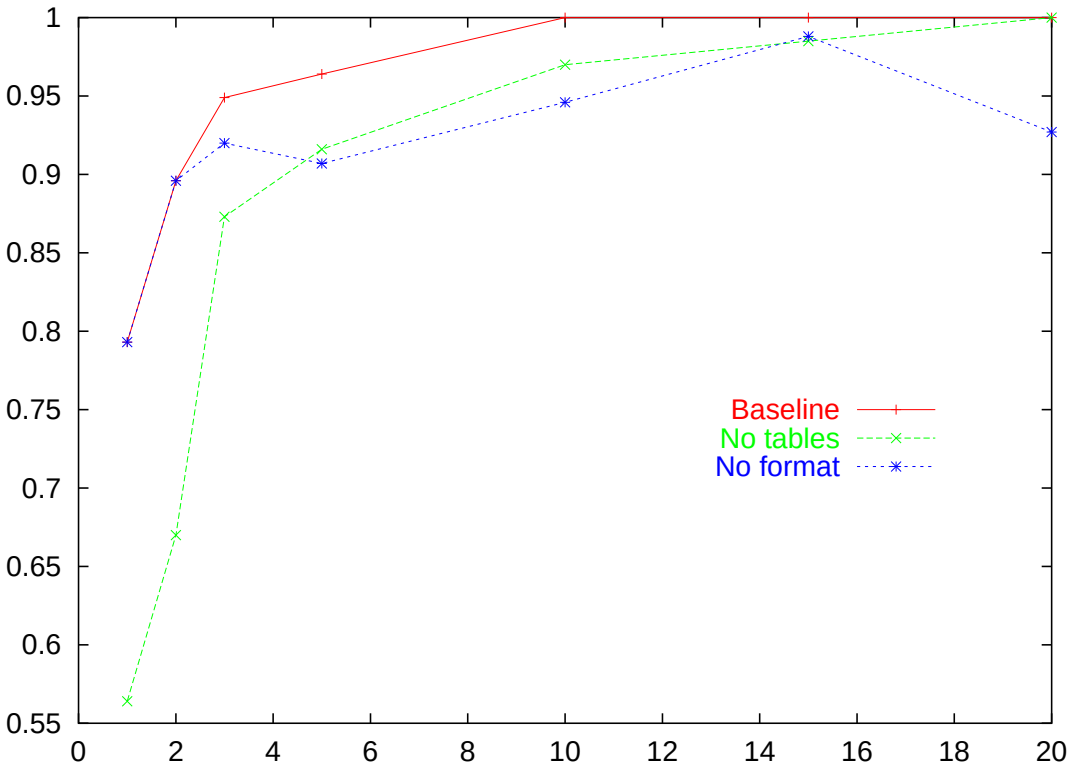

Figure 7: Baseline WL^2 system on WhizBang benchmark extraction problems, with and without table and format builders. The plot shows average F-measure on 13 sample problems as a function of the number of examples labeled.

by averaging across all individual field extraction problems associated with a wrapper-learning task. The learning system we use is strongly biased toward high-precision rules, so precision is almost always perfect, but recall varies from problem to problem. We then plotted the average F-measure across all problems as a function of k.

Figure 7 shows these curves for the baseline WL^2 system on the real-world wrapping tasks of Table 2. The curves marked "no format" and "no tables" show the performance of two restricted versions of the system: a version without the format-oriented builders of Section 4.2, and a version without the table-oriented builders of Section 4.3. These curves indicate a clear benefit from using these special builders.

6. CONCLUSIONS

To summarize, we have argued that pure DOM- or token-based representations of web pages are inadequate for wrapper learning. We propose instead a wrapper-learning system called WL^2 that can exploit multiple document representations. WL^2 is part of an "industrial-strength" wrapper management system that is in active use at WhizBang Labs. Controlled experiments show that the learning component performs well. Lesion studies show that the more exotic builders do indeed improve performance on complex wrapper-learning tasks, and experiments on artificial data suggest that the system has broader coverage and a faster learning rate than two earlier wrapper-learning systems, WEIN [16] and STALKER [21, 22].

The system includes a single general-purpose master learning algorithm and a varying number of smaller, special-purpose "builders", which can exploit different views of a document. Implemented builders make use of both DOM-level and token-level views of a document. More interestingly, builders can also exploit other properties of documents. Special format-level builders exploit visual characteristics of text, like font size and font type, that are not immediately accessible from conventional views of the document. Special "table builders" exploit information about the two-dimensional geometry of tabular data in a rendered web page.

The learning system can exploit any of these views. It can also learn extractors that rely on multiple views (*e.g.*, "extract all table 'cut-in' cells that will be rendered in blue with a font size of 2"). Another advantage of the learning system's architecture is that since builders can be added and removed easily, the system is extensible and modular, and hence can be easily adapted to new wrapping tasks.

Acknowledgments

The authors thank Rich Hume, Rodney Riggs, Dallan Quass, and many of their other colleagues at WhizBang! for contributions to this work.

7. REFERENCES

[1] H. Adé, L. de Raedt, and M. Bruynooghe. Declarative bias for general-to-specific ILP systems. *Machine Learning*, 20(1/2):119–154, 1995.

[2] A. Blum. Empirical support for WINNOW and weighted majority algorithms: results on a calendar scheduling domain. In *Machine Learning: Proceedings of the Twelfth International Conference*, Lake Tahoe, California, 1995. Morgan Kaufmann.

[3] B. Chidlovskii. Wrapper generation by k-reversible grammar induction. In *Proceedings of the Workshop on Machine Learning and Information Extraction*, Berlin, Germany, 2000.

[4] XML path language (XPath) version 1.0. Available from http://www.w3.org/TR/1999/REC-xpath-19991116, 1999.

[5] W. W. Cohen. Grammatically biased learning: learning logic programs using an explicit antecedent description language. *Artificial Intelligence*, 68:303–366, 1994.

[6] W. W. Cohen. Recognizing structure in web pages using similarity queries. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99)*, Orlando, FL, 1999.

[7] W. W. Cohen and W. Fan. Learning page-independent heuristics for extracting data from web pages. In *Proceedings of the Eighth International World Wide Web Conference (WWW-99)*, Toronto, 1999.

[8] L. De Raedt, editor. *Advances in Inductive Logic Programming*. IOS Press, 1995.

[9] D. Freitag. Multistrategy learning for information extraction. In *Proceedings of the Fifteenth International Conference on Machine Learning*. Morgan Kaufmann, 1998.

[10] D. Freitag and N. Kushmeric. Boosted wrapper induction. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-2000)*, Austin, TX, 2000.

[11] C.-N. 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.

[12] HTML 4.01 specification. http://www.w3.org/TR/html4/, 1999.

[13] M. Hurst. *The Interpretation of Tables in Texts*. PhD thesis, University of Edinburgh, School of Cognitive Science, Informatics, University of Edinburgh, 2000.

[14] L. S. Jensen and W. W. Cohen. A structured wrapper induction system for extracting information from semi-structured documents. In *Proceedings of the IJCAI-2001 Workshop on Adaptive Text Extraction and Mining*, Seattle, WA, 2001.

[15] N. Kushmeric. Regression testing for wrapper maintenance. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99)*, Orlando, FL, 1999.

[16] N. Kushmeric. Wrapper induction: efficiency and expressiveness. *Artificial Intelligence*, 118:15–68, 2000.

[17] D. Lewis. Naive (bayes) at forty: The independence assumption in information retrieval. In *Proceedings of ECML-98, 10th European Conference on Machine Learning*, 1998.

[18] N. Littlestone. Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm. *Machine Learning*, 2(4), 1988.

[19] A. McCallum and K. Nigam. A comparison of event models for naive bayes text classification. In *AAAI-98 Workshop on Learning for Text Categorization*, 1998.

[20] S. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. *Journal of Logic Programming*, 19/20(7):629–679, 1994.

[21] I. Muslea, S. Minton, and C. Knoblock. A hierarchical approach to wrapper induction. In *Proceedings of the Third International Conference on Autonomous Agents (Agents'99)*, Seattle, WA, 1999.

[22] I. Muslea, S. Minton, and C. Knoblock. Wrapper induction for semistructured information sources. *Journal of Autonomous Agents and Multi-Agent Systems*, 16(12), 1999.

[23] K. Nigam, J. Lafferty, and A. McCallum. Using maximum entropy for text classification. In *Proceedings of Machine Learning for Information Filtering Workshop, IJCAI '99*, Stockholm, Sweden, 1999.

[24] J. R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5(3):239–266, 1990.

[25] J. R. Quinlan. *C4.5: programs for machine learning*. Morgan Kaufmann, 1994.

[26] J. R. Quinlan and R. M. Cameron-Jones. FOIL: A midterm report. In P. B. Brazdil, editor, *Machine Learning: ECML-93*, Vienna, Austria, 1993. Springer-Verlag. Lecture notes in Computer Science # 667.

[27] A. Sahuget and F. Azavant. Building light-weight wrappers for legacy web datasources using W4F. In *Proceedings of VLDB '99*, pages pages 738–741, 1999.

[28] Clean up your web pages with HTML TIDY. http://www.w3.org/People/Raggett/tidy/, 1999.

[29] X. Wang. *Tabular Abstraction, Editing, and Formatting*. PhD thesis, University of Waterloo, Waterloo, Ontario, Canada, 1996.

[30] J. M. Zelle and R. J. Mooney. Inducing deterministic Prolog parsers from treebanks: a machine learning approach. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, Seattle, Washington, 1994. MIT Press.