

A Flexible Learning System for “Wrapping” Tables and Lists  
or  
How to Write a *Really Complicated* Learning Algorithm  
Without Driving Yourself Mad

William W. Cohen (me)

Matthew Hurst

Lee S. Jensen

WhizBang Labs – Research

## Learning “Wrappers”

- A “wrapper” is a program that makes (part of) a [web site](#) look like (part of) a [database](#).

For instance, [job postings on microsoft.com](#) might be converted to tuples from a relation:

<i>Job title</i>	<i>Location</i>	<i>Employer</i>
C# software developer	Seattle, WA	Microsoft
Receptionist	Seattle, WA	Microsoft
Research Scientist	Beijing, China	Microsoft–Asia
...	...	...

## Learning “Wrappers”

- Reasons for wanting wrappers:
  - Collect training data for an IE system from lots of websites.
  - IE from not-too-many websites  $O(10^2-10^3)$
  - Boost performance of IE on “important” sites.
- Ways of creating wrappers:
  - Code them up (in Perl, Java, WebL, . . . , )
  - Learn them from examples

## What's Hard About Learning Wrappers

- A good wrapper induction system should generalize across future pages as well as current pages.

### **WheezeBong.com:**

#### **Contact info**

Currently we have offices in two locations:

- Pittsburgh, PA
- Provo, UT

## What's Hard About Learning Wrappers

- A good wrapper induction system should generalize across **future pages** as well as **current pages**.
- Many generalizations of the first two examples are **possible**, but only a few will generalize.
- **Prior solutions:** hand-crafted learning algorithms and carefully chosen heuristics.

### **WheezeBong.com:**

#### **Contact info**

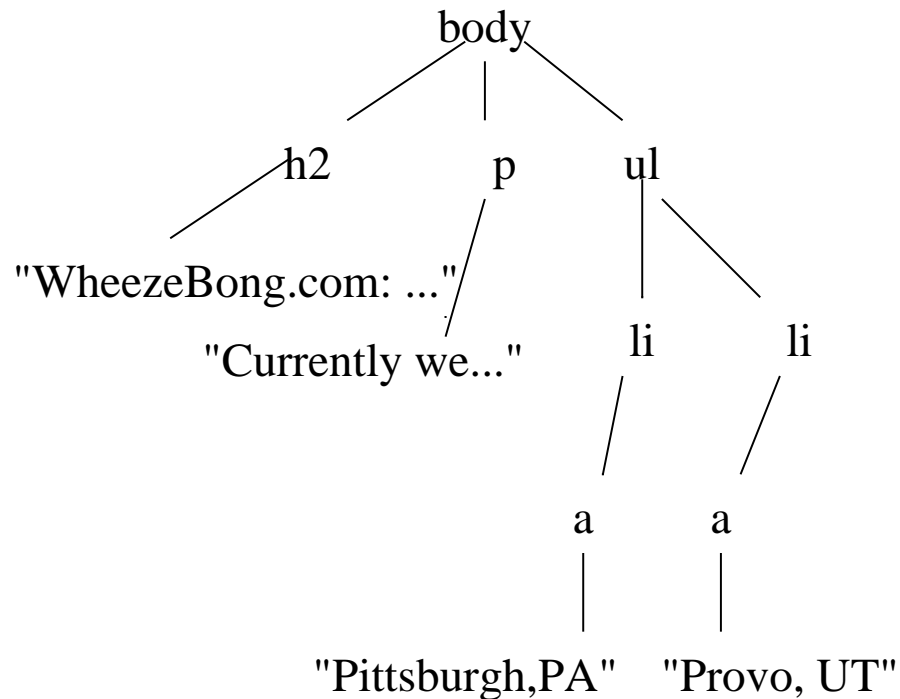
Currently we have offices in three locations:

- Pittsburgh, PA
- Provo, UT
- Honolulu, HI

## Our Approach to Wrapper Induction

- **Premise:** A wrapper learning system needs careful engineering (and possibly re-engineering).
  - 6 hand-crafted languages in WIEN (Kushmeric AIJ2000)
  - 13 ordering heuristics in STALKER (Muslea et al AA1999)
- **Approach:** architecture that facilitates hand-tuning the “bias” of the learner.
  - Bias is an ordered set of “builders”.
  - Builders are **simple** “micro-learners”.
  - A single master algorithm co-ordinates learning.

## Our Approach: Document Representation\*

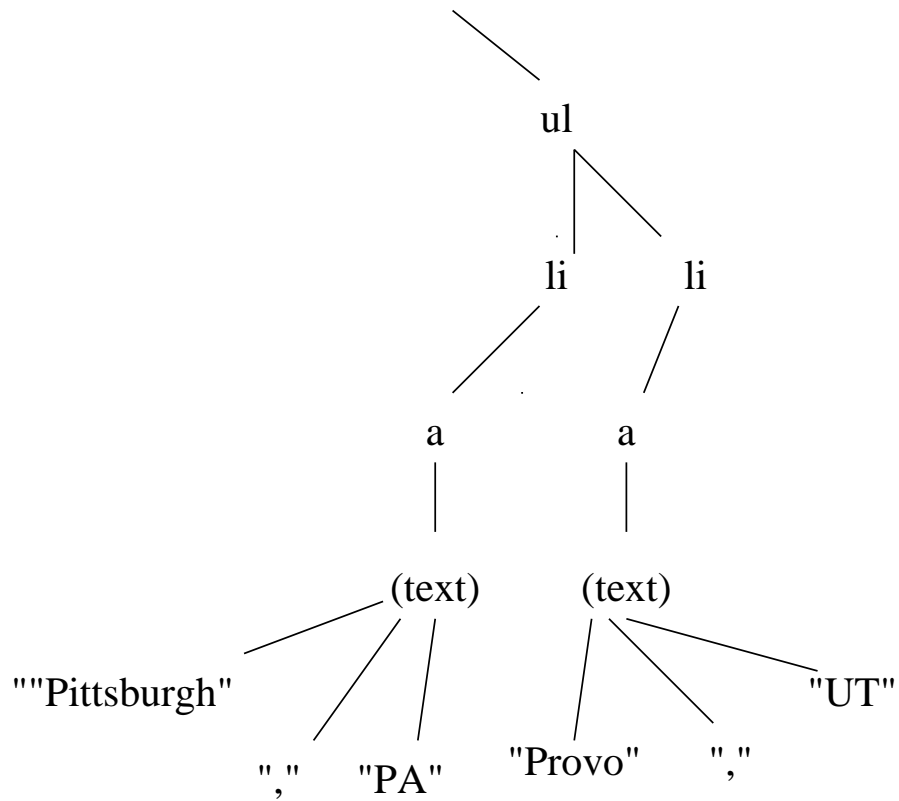


Structured documents (e.g. HTML) are labeled trees (DOMs).

---

\*Slightly over-simplified...

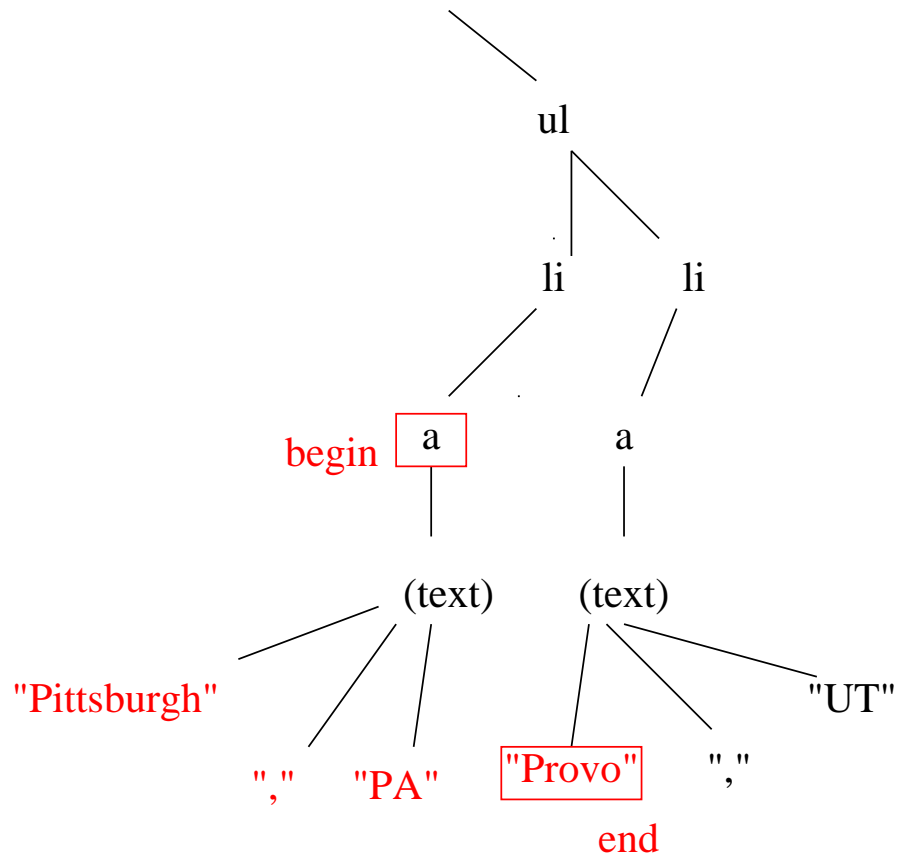
## Our Approach: Document Representation



Imagine the DOM extended with a new node for each token of text...

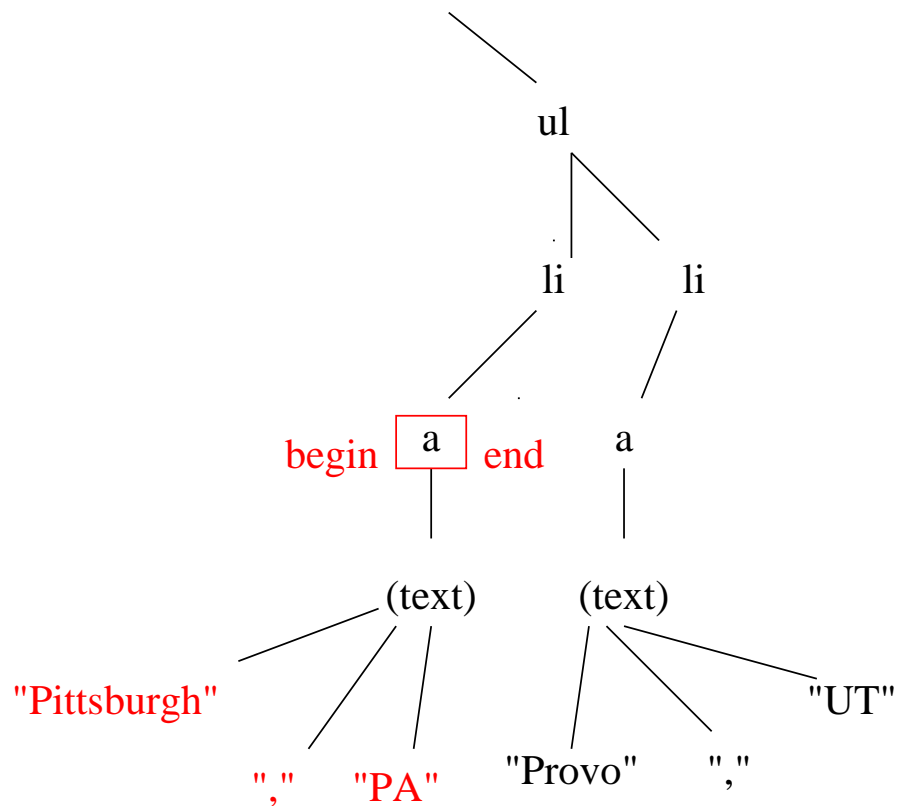


## Our Approach: Document Representation



A “span” is defined by a **start node** and an **end node**...

## Our Approach: Document Representation



...and the **start node** and **end node** might be identical (a “node span”).

## Our Approach: Representing Extractors

- A **predicate** is a binary relation on spans:  $p(s_1, s_2)$  means that  $s_2$  is extracted from  $s_1$ .
- Membership in a predicate can be tested:
  - Given  $(s_1, s_2)$ , is  $p(s_1, s_2)$  true?
- Predicates can be **executed**:
  - **EXECUTE** $(p, s_1)$  is the set of  $s_2$  for which  $p(s_1, s_2)$  is true.

## Example Predicate

Example:

- $p(s_1, s_2)$  iff  $s_2$  are the tokens below an `li` node inside  $s_1$ .
- `EXECUTE( $p, s_1$ )` extracts
  - “Pittsburgh, PA”
  - “Provo, UT”

**WheezeBong.com:**

**Contact info**

Currently we have offices in two locations:

- Pittsburgh, PA
- Provo, UT

## Our Approach: Representing Bias

- The hypothesis space of the learner is built up from **simple sublanguages**.
- $L_{\text{bracket}}$ :  $p$  is defined by a pair of strings  $(\ell, r)$ , and  $p_{\ell,r}(s_1, s_2)$ , is true iff  $s_2$  is preceded by  $\ell$  and followed by  $r$ .

$\text{EXECUTE}(p_{\text{in,locations},s_1}) = \{ \text{“two”} \}$

- $L_{\text{tagpath}}$ :  $p$  is defined by  $\text{tag}_1, \dots, \text{tag}_k$ , and  $p_{\text{tag}_1, \dots, \text{tag}_k}(s_1, s_2)$  is true iff  $s_1$  and  $s_2$  correspond to DOM nodes and  $s_2$  is reached from  $s_1$  by following a path ending in  $\text{tag}_1, \dots, \text{tag}_k$ .

$\text{EXECUTE}(p_{\text{ul,li},s_1}) = \{ \text{“Pittsburgh, PA”}, \text{“Provo, UT”} \}$

## Our Approach: Representing Bias

For each sublanguage  $L$  there is a builder  $\mathcal{B}_L$  which implements a few simple operations:

- LGG( positive examples of  $p(s_1, s_2)$  ): least general  $p$  in  $L$  that covers all the positive examples.

For  $L_{\text{bracket}}$ , longest common prefix and suffix of the examples.

- REFINE(  $p$ , examples ): a set of  $p$ 's that cover some but not all of the examples.

For  $L_{\text{tagpath}}$ , extend the path with one additional tag that appears in the examples.

## Our Approach: Representing Bias

Builders can be **composed**: given  $\mathcal{B}_{L_1}$  and  $\mathcal{B}_{L_2}$  one can automatically construct

- a builder for the **conjunction** of the two languages,  $L_1 \wedge L_2$
- a builder for the **composition** of the two languages,  $L_1 \circ L_2$

Requires an additional input: how to decompose an example  $(s_1, s_2)$  of  $p_1 \circ p_2$  into an example  $(s_1, s')$  of  $p_1$  and an example  $(s', s_2)$  of  $p_2$ .

So, **complex builders** can be constructed by **combining** simple ones.

## Example of combining builders

- Consider composing builders for  $L_{\text{tagpath}}$  and  $L_{\text{bracket}}$ .
- The LGG of the **locations** would be  $p_{\text{tags}} \circ p_{\ell,r}$   
where
  - $\text{tags} = \text{ul,li}$
  - $\ell = \text{"("}$
  - $r = \text{")"}$

### **Jobs at WheezeBong:**

To apply, call:

1-(800)-555-9999

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



## Combining “local” and “global” extractors

- Consider composing builders for  $L_{\text{tagpath}}$  and  $L_{\text{city}}$ .

where

$p_{\text{city}}(s_1, s_2)$ : true iff  $s_2$  is a city (state, town, ...) name inside  $s_1$ .

- The LGG of the locations would be  $p_{\text{tags}} \circ p_{\text{city}}$ .

### **Jobs at WheezeBong:**

To apply, call Alberta Hill  
at 1-(800)-555-9999

- Webmaster, **New York** city area. Perl, servlets a plus.
- Librarian, **Pittsburgh**. MLS required.
- Ditch Digger needed in **Palo Alto** office. No experience needed.

## Limitations of DOMs

- The “real” regularities are at the level of the visual appearance of the document.
- What if the underlying DOM doesn't show the same regularities?

`<b><i>Provo</i></b>` versus `<i><b>Pittsburgh</b></i>`

## Limitations of DOMs

“Actresses”			
Lucy	Lawless	<u>images</u>	<u>links</u>
Angelina	Jolie	<u>images</u>	<u>links</u>
...	...	...	...
“Singers”			
Madonna		<u>images</u>	<u>links</u>
Brittany	Spears	<u>images</u>	<u>links</u>
...	...	...	...

How can you easily express “links to pages about singers”?

## Fancy Builders: Understanding Table Rendering

1. **Classify** HTML tables nodes as “data tables” or “non-data tables”.

On 339 examples, precision/recall of 1.00/0.92 with Winnow and features ...

2. **Render** each data table.
3. Find the **logical cells** of the table.
4. Construct **geometric model** of table: an integer grid, with each logical cell having co-ordinates on the grid.
5. Tag each cell with (some aspects) of its **role** in the table.
  - Currently, “cut-in cells”.

## Fancy Builders: Understanding Table Rendering

“Actresses” <i>cutin</i> ,1.1-1.1			
Lucy 2.1-2.1	Lawless 2.2-2.2	<u>images</u> 2.3-2.3	<u>links</u> 2.4-2.4
Angelina 3.1-3.1	Jolie 3.2-3.2	<u>images</u> 3.3-3.3	<u>links</u> 3.4-3.4
“Singers” <i>cutin</i> ,4.1-4.1			
Madonna 5.1-5.2		<u>images</u> 5.3-5.3	<u>links</u> 5.4-5.4
Brittany 6.1-6.1	Spears 6.2-6.2	<u>images</u> 6.3-6.3	<u>links</u> 6.4-6.4

Table builders:

Element name + words  
in last cut-in (e.g.,  
“table cells where  
the last cut-in  
contains ‘singers’”)

“Tagpath” builder  
extended to condition  
on (x,y) co-ordinates  
(e.g., “table cells  
with y-coordinates  
‘3-3’ inside ... )

## The Learning Algorithm

Inputs:

- an **ordered** list of builders  $\mathcal{B}_1, \mathcal{B}_k$ .
- **positive examples**  $(s_1, s_2)$  of the predicate to be learned
- information about what parts of each page have been **completely** labeled (implicit negative examples)

## The Learning Algorithm

Algorithm:

- Compute LGG of positive examples with each builder  $\mathcal{B}_i$ .
- If any LGG is **consistent** with the (implicit) negative data, then return it\*.
- Otherwise, execute the best\* LGG to get **explicit** negative examples, then apply a FOIL-like learning algorithm, using LGG and REFINE to create “features\*”.

---

\* Break **ties** in favor of earlier builders. With few positive examples there are lots of ties.

## Experimental results

Problem#	WIEN(=)	STALKER( $\approx$ )	WL <sup>2</sup> (=)
S1	46	1	1
S2	274	8	6
S3	$\infty$	$\infty$	1
S4	$\infty$	$\infty$	4

Examples needed to learn accurate extraction rules for all parts of a wrapper for WIEN (Kushmerick '00), STALKER (Muslea, Minton, Knoblock '99), and the WhizBang Labs Wrapper Learner (WL<sup>2</sup>).

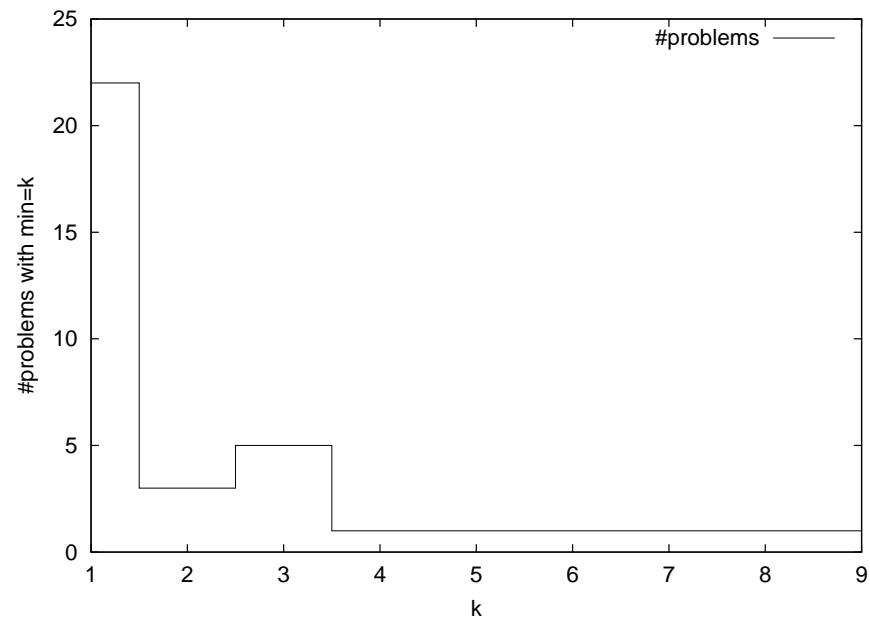


## Experimental results

Problem	WL <sup>2</sup>	Problem	WL <sup>2</sup>
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

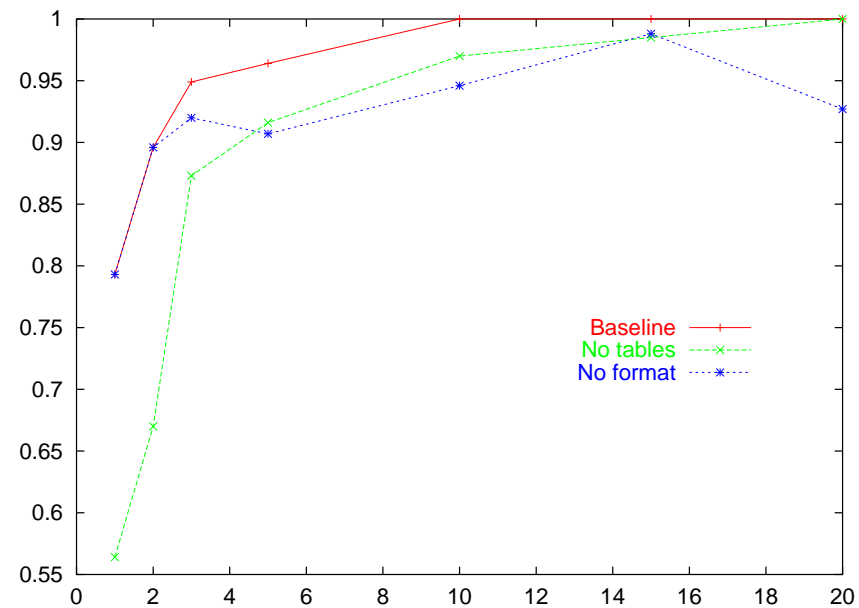
WL<sup>2</sup> on representative real-world wrapping problems.

## Experimental results



$WL^2$  on representative real-world wrapping problems.

## Experimental results



Variants of  $WL^2$  on real-world wrapping problems:  
average accuracy versus number of training examples.

## Conclusions/Summary

- Wrapper learners need **tuning**. Structuring the **bias space** provides a principled approach to tuning.
- “Builders” let one **mix** generalization strategies based on **different views** of the document:
  - as DOM
  - as sequence of tokens
  - as sequence of rendered fragments of text
  - as geometric model of table
  - ...
- Performance seems to be **better** than previous systems.