Boosted Wrapper Induction

Dayne Freitag and Nicholas Kushmerick
AAAI 2000, 577-583

Presented by:

Dave Kauchak

Department of Computer Science

University of California, San Diego

dkauchak@cs.ucsd.edu

Information Extraction (IE)

Task: Convert text such as web pages into structured data objects suitable for automatic processing.

Example:

Given want ads for room rentals, extract the neighborhood, number of bedrooms and price.

Input:

Belltown CONCEPT ONE, 1 BDRM, $775-$895, Lake Union & Sound Views, Fplc, W/D, Gar Prkg Available 206-728-9515

Output:

Neighborhood:	Belltown
Bedroom:	1
Price:	895

Wrapper Induction

A wrapper is a contextual pattern that is simple but highly accurate.

Wrapper induction is the automated process of learning wrappers.

Once wrappers are learned, they are then used to extract appropriate portions from a text document.

Wrapper Development

Traditionally wrapper induction has been used on highly-structured text. However, this is changing.

Recent algorithms can discover high precision patterns in unstructured text.

- For example, “who:” followed by “dr .” could identify the start of the speaker name in seminar announcements.

Individual patterns typically have high precision, but low recall.

By using boosting, many high precision patterns can be combined to create a high recall pattern.

Definitions of Terms

Token - A token is one of three things (all other text is ignored, i.e. whitespace):

- An unbroken string of alphanumeric characters
- A single punctuation character
- A carriage return

Boundary - The space between two adjacent tokens

Field – The text to be extracted, which consists of one more tokens.

IE as Classification

The set of training examples is all of the boundaries in a document.

The goal is to approximate two extraction functions `Begin` and `End`:

`Begin(i)=` 1 if *i* begins a field
0 otherwise

Boundary Detectors

Boundary detectors are used to approximate the values of the two extraction function, `Begin(i)` and `End(i)`.

<u>Pattern</u> - a sequence of tokens, maybe including wildcards.

<u>Boundary detector</u> - a pair of patterns `d = ‹p,s›` (a prefix pattern `p` and a suffix pattern `s`) and a numeric confidence value `C(d)`.

Boundary Detector Matching

A boundary detectors d is functions from a boundary to {0,1}: d(i) = 1 if d matches i.

A boundary `‹p,s›` detector matches boundary `i` if `p` matches the tokens before `i` and `s` matches the tokens after `i`. For example,

Text:

```
text<b><a href=http://www.cs.ucsd.edu>
```

Boundary detectors:

```
‹[< a href = "], [http]›
‹[], [">]›
```

Matches:

```
text<b><a href="http://www.cs.ucsd.edu">
```

Wrappers

A wrapper W consists of three parts $\langle F, A, H \rangle$ where F and A are sets of boundary detectors.

- F is the set of "fore" boundary detectors, F_1 through F_f, which identify the start of a field, approximating $Begin(i)$.
- A is the set of "aft" boundary detectors, A_1 through A_a, which identify the end of a field approximating $End(i)$.
- $H(k)$ is a function that approximates the probability that a field will have length k.

Extraction with Wrappers

Every boundary i in the document is given a "fore" score $F(i)$ and an "aft" score $A(i)$. $F(i)$ and $A(i)$ are calculated as follows:

$$F(i) = \sum_{k=1}^{k=f} C(F_k)F_k(i) \qquad A(i) = \sum_{k=1}^{k=a} C(A_k)A_k(i)$$

The wrapper W classifies a text fragment between boundaries i and j as follows:

$$W(i,j) = \begin{cases} 1 & \text{If } F(i)A(j)H(j-i) > \tau \\ 0 & \text{otherwise} \end{cases}$$

where τ is a numeric threshold.

Boosted Wrapper Induction Algorithm

The input is two training sets S and E which were constructed using the same text. Boundaries in S are labeled with `Begin(i)`. Boundaries in E are labeled with `End(i)`.

```
Procedure BWI(training sets S and E)
  F = AdaBoost(LearnDetector, S)
  A = AdaBoost(LearnDetector, E)
  H = field length histogram from S and E
  return wrapper W = <F, A, H>
```

AdaBoost calls the LearnDetector algorithm *T* times to learn the set of "fore" detectors *F* and *T* more times to learn the "aft" detectors *A*.

***H* is constructed by simply recording the number of fields of length *k*.**

LearnDetector iteratively builds out from the empty detector.

Each iteration, the best prefix and suffix of length *L* or less is determined.

The current detector and best extensions are then compared. If a better one is found, the learning continues.

```
Procedure LearnDetector(training set Y(S or E))
   prefix pattern p = []
   prefix pattern s = []
   loop
      p' = BestPrefixExtension(<p,s>, Y)
      s' = BestSuffixExtension(<p,s>, Y)
      switch:
         <p', s> is best:
              p = the last |p| + 1 tokens of p'
         <p, s'> is best:
              s = the first |s| + 1 tokens of s'
         <p, s> is best:
              return the detector <p, s>
```

Look-ahead Parameter

BestPrefixExtension **and *BestSuffixExtension* both have an implicit look-ahead parameter *L*.**

Both these functions find the best extension *L* tokens or less.

Notice, however, that although the algorithm looks ahead *L* tokens, only the next token is included in the actual detector.

Some more difficult domains may require a larger *L* value to get good results. We will see this later in the experimental results.

Scoring Detectors

At each iteration, AdaBoost assigns a weight to each boundary.

The score of a detector is then the difference between the square root of the sum of the weights of the correctly classified boundaries and the square root of the sum of the weights of the negatively classified boundaries.

This score is then used to decide if the newly created detector is better (i.e. in the switch statement).

Wildcards

A wildcard is a special token that matches any single member of a set of tokens. For example,

- <alph> matches any token containing only alphabetical characters, e.g. wild, WILD, Wild or wILD.
- <Anum> matches any token containing only alphanumeric, e.g. Wild123.
- <Cap> matches any token that begins with an upper-case letter, e.g. Wild or WILD.
- <LC> matches any token that begins with a lower-case letter, e.g. wild or wILD.
- <Num> matches and token containing only digits, e.g. 1234.
- <Punc> matches any token that is a punctuation token, e.g. . , < >
- <*> matches any token

Experimental Results

What effect does the number of rounds of boosting T have on performance?

What effect does the look-ahead parameter L have on performance?

How important are wildcards and what is their effect?

How does BWI compare with other learning algorithms?

The performance is measured by precision, recall and F1 (the harmonic mean of precision and recall).

Data Sets

16 IE tasks were evaluated over eight distinct document collections

- ***SA*: 486 seminar announcements**
- ***Acq*: 600 Reuter articles detailing corporate acquisitions**
- ***Jobs*: 298 Usenet job announcements**
- ***CS*: 20 web pages containing CS departments listing the faculty**
- ***Zagats*: 91 web pages containing restaurant reviews**
- ***LATimes*: 20 web pages containing restaurant descriptions**
- ***IAF*: 10 web page containing results from a web email search engine**
- ***QS*: 10 pages from web stock quote service**

Data Sets cont.

The domains require a variety of fields to be extracted. For example, we want to extract four fields from the seminar announcements (SA):

- **speaker (speaker's name)**
- **location (seminar location)**
- **stime (starting time)**
- **etime (ending time)**

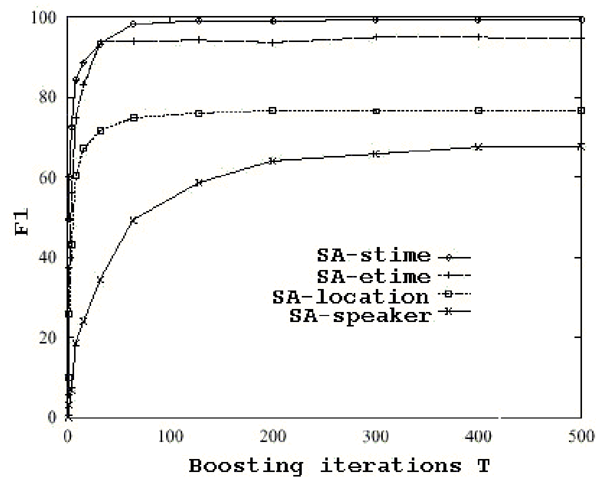
The first three domains have been used as test cases for traditional IE techniques.

The last five domains are typical of web wrapper induction techniques.

Performance was measured using cross validation.

Question 1: Benefit of Boosting?

Fix look-ahead L = 3 and use the default wildcard set. Let T vary from 0 (no boosting) to 500.



Question 1 Results

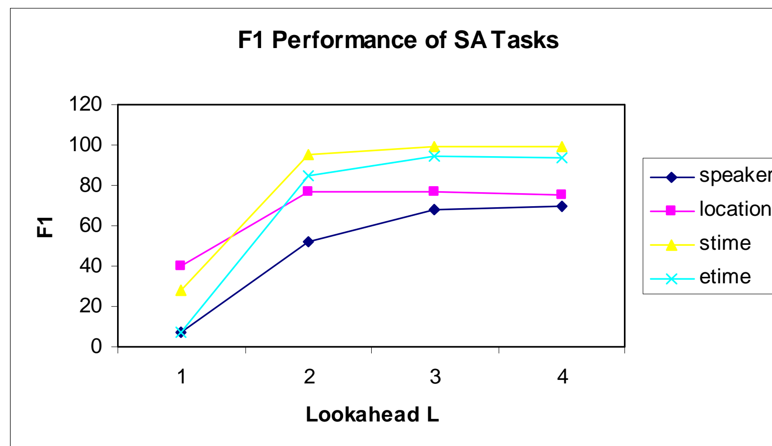
On easy tasks, such as seminar start time (SA-stime), BWI quickly achieves its peak performance.

On more difficult tasks, such as seminar speaker (SA-speaker), as many as 500 rounds may be beneficial.

On the highly-structured wrapper tasks fewer rounds of boosting are required.

Question 2: Benefit of Look-Ahead?

Fix boosting iterations T = 100 and use the default wildcard set. Let look-ahead L vary from 1 to 4.



Question 2 Results

Performance improves with increasing look-ahead.

Training time increases exponentially with L.

When training on the speaker task with L = 3 and T = 100, BWI took ~ 1 hour.

With L = 4, BWI took ~ 4 hours.

Unfortunately, some domains require deeper look-ahead.

- IAF-altname task: L = 3 results in F1 = 3%.
- IAF-altname task: L = 8 results in F1 = 58.8%.

Question 3: Wildcards

Fix look-ahead L = 3 and T = 500 rounds of boosting.
Vary the set of wildcards.

wildcards	speaker	location	stime	etime
none	15.1	69.2	95.7	83.4
just <*>	49.4	73.5	99.3	95.0
default	67.7	76.7	99.4	94.6
lexical	73.5	-	-	-

default: all eight wildcards discussed earlier

lexical: task specific lexical resources:

- **<FName>: common first names released by U.S. Census Bureau.**
- **<LName>: common last names**
- **<NEW>: tokens not found in /usr/dict/words on Unix**

Question 4: Comparison

16 different extraction tasks were compared.

Other algorithms tested:

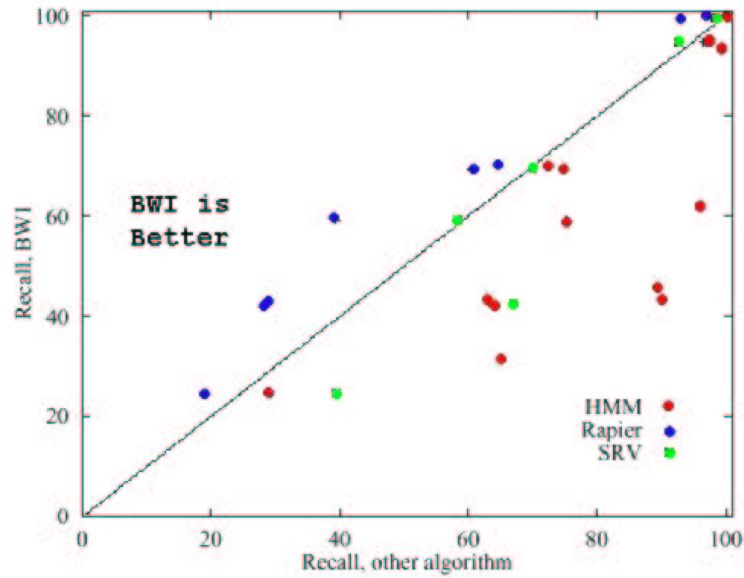
- **Two rule learners: SRV and Rapier**
- **One algorithm based on hidden Markov models**
- **One wrapper induction algorithm: Stalker**

For "traditional" domains, L = 3, T = 500 and the default wildcards were used.

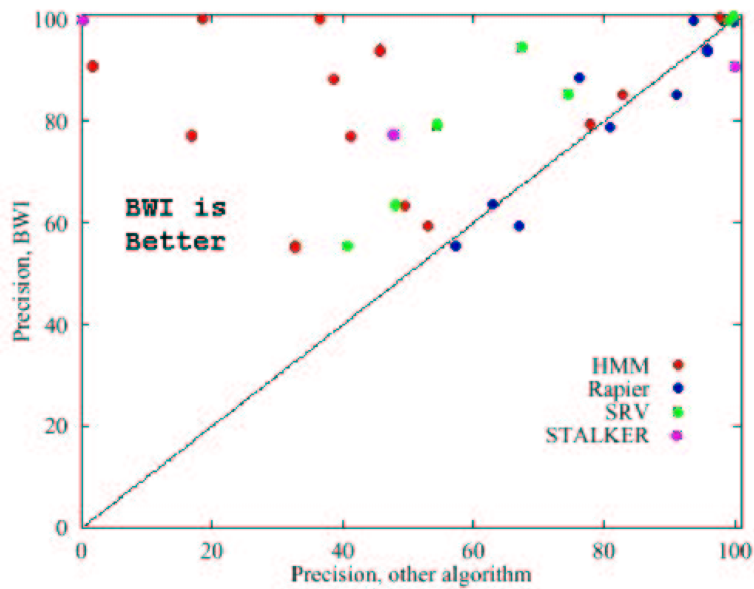
Two sets of wrapper domains:

- **For the easier fields (such as credit cards), L = 3, T = 50 and the lexical wildcards were used.**
- **For the more difficult fields (such as IAF-altname), L = 8, T = 50 and only the <*> wildcard, due to time constraints.**

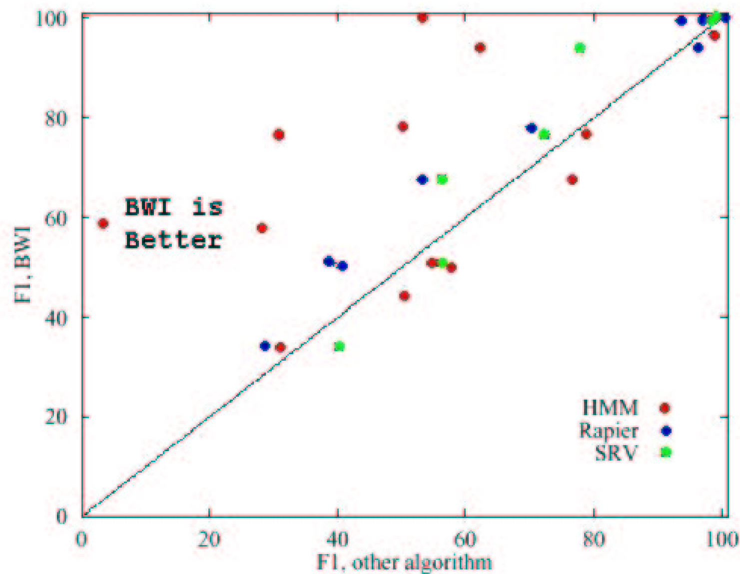
Question 4, Recall



Question 4, Precision



Question 4, F1



Conclusion

BWI is competitive with state-of-the-art IE methods in most domains, and superior in many (in both “natural” and “structured” domains).

BWI is biased toward high precision and has reasonable recall.