

---

# Effect of Boosting in BWI

---

**David Kauchak**

Department of Computer Science  
University of California San Diego  
La Jolla, CA 92093-0114  
*dkauchak@cs.ucsd.edu*

## Abstract

Recent work in information extraction has brought about a new method for text extraction using wrappers. A wrapper is a simple, but highly accurate extraction procedure. Unfortunately, these wrappers tend to have low recall. To remedy this problem, boosted wrapper induction (BWI) was proposed. This method combines a weak wrapper learner with AdaBoost to generate a more general extraction rule. The result is an algorithm with a bias towards precision, but with reasonable recall in both traditional extraction domains. The exact benefit of boosting over more traditional approaches is not always apparent. In this paper, we examine the benefits of boosting by comparing BWI to two different sequential covering algorithms with wrappers for text extraction in the framework of both highly structured and natural text. Sequential covering is a simple, straightforward algorithm which tries to cover as many possible positive examples with a single rule, removes the covered examples from the training set and continues until all of the positive examples have been covered. We show results from a broad range of information extraction tasks and show that the basic benefit of boosting in this domain is to allow BWI to continue learning new and helpful rules without over fitting the training data even after all of the positive examples are covered. This result is consistent with previous theoretical and experimental results.

## 1 Introduction

Information extraction (IE) is by no means a new problem. However, as the Internet has become more popular and as more and more text is made available to the public in digital form, IE has become important. This large quantity of text has both benefits and drawbacks. On the positive side, users now have a huge amount of data at their fingertips. The drawback is that the sheer amount of data makes finding particular information burdensome. Information extraction is commonly used to extract key fields from unstructured text to be processed later in some automated process such as a user query.

A variety of systems and techniques have been developed to try and attack the information extraction problem. Early successful techniques used statistical models

such as n-gram models, hidden Markov Models and probabilistic context free grammars [1]. Recently, though, rule based systems that employ some form of machine learning have become increasingly popular and successful. These rule-based systems have taken a variety of different approaches, but have all recognized a number of key facts. First, creating rules by hand is extremely difficult and time consuming. For this reason, most of the systems generate the rules given raw unlabeled data or partially labeled data. Second, people have recognized that trying to generate a single, general rule for extracting a given field is difficult, if not impossible. Instead, most of the systems attempt to learn a number of rules that cover the training set and then combine these rules in some way.

One recent technique for generating rules in the realm of text extraction is wrapper induction. Wrapper induction techniques have proved to be fairly successful [7] [5]. A wrapper is a simple procedure for extracting a field from a data set. When a wrapper rule fires, the field that the wrapper identifies is extracted. These wrapper rules are then combined in some manner to create a general extraction function.

Recent research in improving weak classification rules using boosting [10] has led to a method for generating weak rules that tends to produce highly accurate rules from the weak rules without over fitting. Boosted wrapper induction (BWI) is an IE technique that uses AdaBoost to generate a more general extraction rule from a set of wrappers [4]. Boosting adjusts the weights of the training examples to increase the importance of misclassified examples. BWI has been shown to do well on a wide variety of tasks including natural language tasks and traditional, highly structured wrapper tasks.

Boosting has been shown theoretically to perform well and performs well in practice. However, there has been little analysis of the benefit of boosting over other weak rule combining algorithms. A simple and common approach to combining weak learners is sequential covering. With sequential covering, the rules are ordered in some way according to "quality." The best rule is chosen, and all of the examples that the rule correctly classifies are removed from the training set. The process is then repeated until the entire training set has been covered. For a more detailed description of sequential covering see [2]. Sequential covering has been used in a number of systems ([1][3][6][7][8]) because it is fairly simple to implement, tends to generate understandable and intuitive rules and has achieved good results.

Besides these nice properties, a key motivation for using sequential covering for comparison with BWI can be found in an interesting result in [4]. In an attempt to impose a tradeoff between precision and recall, the authors of [4] proposed a threshold value to decide whether the example should be classified or not. If the test example score is above this threshold then the test case is classified positively. The idea was that by varying this value, the algorithm could realize a tradeoff between precision and recall. Unfortunately, in practice, the value that worked best for this threshold was simply zero. When a zero threshold is used, anything with any score other than zero is classified as a positive case. So, even though a score is calculated for each test case, the only information from this score that is actually used is whether it is non-zero or not. This is the same property that sequential covering has since rules are not given weights; examples are simply identified as either positive or negative.

This paper is broken down into a number of sections. In section 2, we give a quick review of some of the terms and the problem setup that are described in [4] and are useful here. In section 3, we explain in more detail the sequential covering wrapper induction algorithm (SWI), which is a modification to the BWI algorithm. In section 4, we present experimental results on a variety of information extraction

tasks comparing BWI and SWI, including natural and highly structured texts. Finally, in section 5 we conclude.

## 2 Problem Statement

Most of the material in this section can be found in [4]. We present an abridged version here for clarity. Before the details of the problem are described, a bit of vocabulary will be introduced for convenience. Each document can be broken up into tokens. A token is one of three things: an unbroken string of alphanumeric characters, a punctuation character or a carriage return. The problem of information extraction is to extract some number of tokens from a test document. To do this, we reformulate the IE problem as a classification problem. Instead of thinking of the problem as a string of tokens, we look at the problem as a function of boundaries. A boundary is the space between two tokens. Notice that a boundary is not something that is actually in the text (such as white space), but just comes about from the parsing of the text into tokens. We then want to approximate two functions from a boundary to the binary set  $\{0,1\}$ : one function that is 1 if the boundary is the beginning of a field to be extracted and one function that is 1 if the boundary is the end of a field.

These approximation functions are represented as sets of boundary detectors, or just detectors. A detector is a pair of strings of tokens,  $\langle p, s \rangle$ . A detector matches a boundary if the prefix string of tokens,  $p$ , matches the tokens before the boundary and if the suffix string of tokens,  $s$ , matches the tokens after the boundary. For example, the detector  $\langle \text{Who:}, \text{Dr.} \rangle$  would match “Who: Dr. John Smith” between the ‘:’ and the ‘D’. Once the beginning and ending functions are approximated, extraction is performed by identifying the beginning and end of a field and extracting the text between the two points

## 3 Sequential Covering Wrapper Induction (SWI)

The details for sequential covering wrapper induction (SWI) rely strongly on some knowledge of BWI [4]. SWI uses the same basic framework as BWI with some slight modifications to the selection of rules. The basic algorithm for SWI can be seen in Figure 1. SWI simply generates two sets of detectors,  $F$  and  $A$ , which identify the start and end of a field respectively and a histogram of the lengths of the fields to be extracted,  $H$ .

SWI generates  $F$  and  $A$  by calling the **GenerateDetectors** procedure. The **GenerateDetectors** procedure is a basic sequential covering rule learner. At each iteration through the while loop, a new rule is learned. This rule consists of a prefix part and a suffix part. The prefix and suffix parts of the rule are generated by exhaustively searching all of the possible extensions up to a given length  $L$ , the look ahead value. The extensions are then scored based on some evaluation function, which we will discuss later.

Once the best extensions of length  $L$  are found, this pair (i.e. the rule) is added to the set of rules to be returned. Before continuing, all the positive examples that were covered by the new rule are removed, leaving only uncovered examples. When this set of rules covers all the positive training examples, then the loop ends and the set of rules that now covers all the positive training examples is returned.

A few things should be clarified at this point. There is a distinction between the actual text, which is simply a set of tokens, and the two training functions, which indicate whether a boundary is the start or end of a field to be extracted. When the algorithm removes the positive examples that are covered by a rule, the values in

```

wrapper SWI(training sets S and E)
  F = GenerateDetectors(S)
  A = GenerateDetectors(E)
  H = field length histogram from S and E
  return <F,A,H>

list of detectors GenerateDetectors(training set Y(S or E))
  prefix pattern p = []
  prefix pattern s = []
  list of detectors d

  while(positive examples are still uncovered){
    FindTheBestExtensions()
    add best extensions, <p, s>, to d
    remove examples covered by <p, s> from Y
  }
  return d

```

Figure 1: The SWI and GenerateDetectors algorithms.

training function are set so that they are no longer positive. The actual sequence of tokens is not being changed. Changing the tokens would alter the data inappropriately and diverge from the idea of sequential covering and the problem setup as described above.

There are a couple of key differences between BWI and SWI. First, as mentioned above, with sequential covering the training examples are altered by changing the set of positive examples. In BWI, the examples are reweighted but are never removed. Second, the scoring functions for the extensions and detector are slightly different. We propose two different versions of SWI. The basic SWI algorithm uses a simple greedy model. An extension or detector is scored simply by the number of positive examples that it covers without covering any negative examples. So, if a detector misclassifies one example (i.e. covers a negative example), it is given a score of zero. We call this version Greedy\_SWI. A second version of SWI, called Root\_SWI, is also implemented and uses the same evaluation function as BWI. Given the sum of the weights of the positive examples covered by the extension or detector,  $W_+$ , and the sum of the weights for the negative examples covered,  $W_-$ , the score is calculated as follows:

$$score = \sqrt{W_+} - \sqrt{W_-}$$

Notice that for Root\_SWI these sums will just be the number of examples covered because the examples are all given the same weight.

Finally, one last difference between BWI and SWI is that BWI terminates after some fixed number of boosting iterations has been reached. Instead, SWI terminates when all of the positive examples have been covered. This turns out to be a key difference between the two different methods. With BWI we can continue learning rules even after all of the positive examples are covered because we simply change the weights of the examples. However, with SWI there is some upper bound on the number of iterations for which the algorithm can iterate. At each iteration, at least one example will be covered correctly. Therefore, the upper bound on the number of iterations for **GenerateDetectors** to terminate is simply the number of positive examples. In practice, as we will see below, the algorithms tend to

terminate much earlier than this depending on the difficulty of the problem and diversity of the training examples.

## 4 Experiments

In order to examine the differences between BWI, Greedy\_SWI and Root\_SWI we examined the three algorithms on 10 different information tasks from 5 different text domains. These tasks are fairly standard and have been used in testing a variety of IE techniques. The tasks can be broken into two sets: traditional information extraction tasks and highly structured, traditional wrapper tasks. The traditional information extraction tasks closely resemble natural text, but still contain some high level structure. The wrapper tasks are highly structured and are often automatically generated texts. The data sets used were obtained from [9].

For the traditional information extraction tasks, we examined two different document collections, 486 speaker announcements (SA) and 298 Usenet job announcements (Jobs). In the speaker announcement document collection, four different fields were extracted: the speaker's name (SA-speaker), the location of the seminar (SA-location), the starting time of the seminar (SA-stime) and the ending time of the seminar (SA-etime). In the job announcement document collection, three fields were extracted: the message identifier (Jobs-id), the name of the company (Jobs-company) and the title of the available position (Jobs-title).

The wrapper tasks were designed from three different document collections: 20 web pages containing restaurant descriptions from the Los Angeles Times (LATimes), 91 web pages containing restaurant reviews from Zagat's Guide to Los Angeles restaurants (Zagat) and 10 web pages containing responses from a stock quote service (QS). Note that although these domains tend to have a small number of documents, each document consists of multiple examples (for example, multiple restaurant reviews).

For the experiments, we used cross validation, where the documents are partitioned into training and testing sets a number of times. Figure 2 contains a summary of the results from the experiments. For the full set of results, see Figure 4 at the end of the paper. The systems were evaluated by three different metrics: precision, recall and F1, the harmonic mean between precision and recall. The goal, however, is to get the highest F1 value since this is a good general metric for the performance of the systems. The documents were split 10 different ways, so each value presented in the results is the average of these 10 runs. For all algorithms, a look ahead value,  $L$ , of 3 was used. For the traditional domains (SA and Jobs) the default wildcard set was used and for the wrapper domains (LATimes, Zagat and QS) the lexical and default wildcard sets were used.

To attempt to get an understanding of the effects of boosting and also of the evaluation functions, we performed four different tests. First, we ran BWI with the same number of rounds of boosting as was used in [4]. For the SA and Jobs document collections, 500 rounds of boosting were used. For the LATimes, Zagats and QS document collections, 50 rounds were used. Next, we ran Greedy\_SWI and Root\_SWI on the same data sets until all of the examples were covered. The actual number of iterations for which these algorithms ran varied across the different IE tasks. The number of iterations are presented in Figure 5. Finally, we ran BWI for the same number of iterations that it took for Root\_SWI to complete (for example, 1 round for Jobs-id and QS-date).

First, to see if BWI is actually better than the naive greedy approach, we compare Greedy\_SWI and BWI. The first thing that we notice is that BWI appears to perform better than SWI. The F1 values for BWI are better for almost all the tasks.

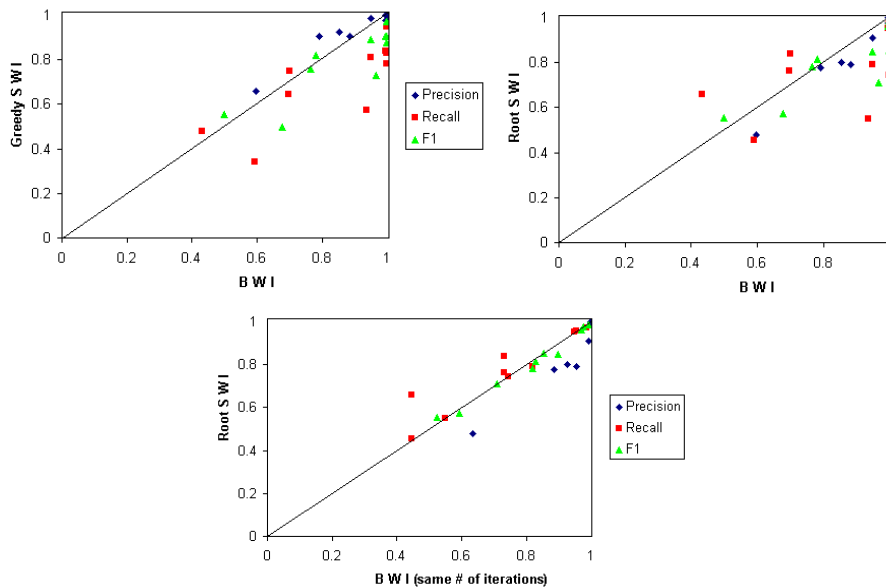


Figure 2: Each point in the graphs represents a comparison between the algorithm labeled on the x-axis and the algorithm labeled on the y-axis for the 10 different tasks. The algorithms are compared by the three metrics: precision, recall and F1.

Also, Greedy\_SWI tends to have higher precision, while BWI tends to have higher recall. This could be explained by a number of factors. Notice that BWI runs for more iterations than Greedy\_SWI and therefore learns more rules. Since BWI has more rules, it will tend to have better recall. Another contributing factor to the difference in precision likely has to do with the evaluation functions. BWI allows rules to cover some negative examples, while Greedy\_SWI will only take a rule that covers only positive examples. This type of rule will tend to be more specific and will, therefore, have a higher precision but lower recall.

Given the results above, one might then wonder what part of BWI's success comes from the extra rules that are learned and what part comes from the difference in evaluation functions. To explore the benefit of a more general evaluation function, we compare Greedy\_SWI and Root\_SWI. The only difference between these two algorithms is their evaluation function. As we hypothesized earlier, Greedy\_SWI has a higher precision while Root\_SWI has a higher recall. The rationale behind this is that by allowing rules to cover some negative examples, the rules can be made more general and will then cover more test examples. The drawback of this is that the rules do actually cover negative examples. This will have an adverse effect on the precision. An interesting thing to notice, however, is that even though the two algorithms tend to have fairly different precision and recall values, they have similar F1 values. This implies that although changing the evaluation function from the greedy to the same one used by BWI does increase our recall, this is not the actual reason why BWI performs better than Greedy\_SWI. This result can also be seen by comparing Root\_SWI and BWI. Both these algorithms use the difference of square roots evaluation function. As can be seen in Figure 2, BWI still outperforms Root\_SWI.

There are only two differences between BWI and Root\_SWI. First, boosting reweights the examples at each iteration. This causes BWI to focus on slightly

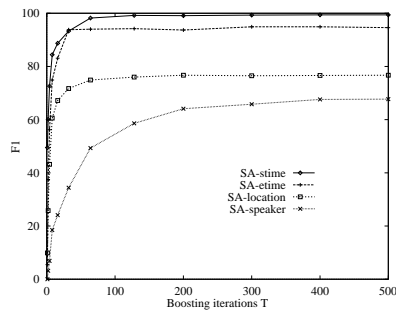


Figure 3: F1 performance on the SA tasks as a function of the number of rounds of boosting. The figure is taken from [4].

different training examples. Second, boosting allows BWI to continue to learn meaningful rules even after all the positive examples have been covered. Examining Figure 5 shows that BWI runs for many more iterations than Root\_SWI. In [4], they examined the benefit of boosting as the number of iterations increased. The results from these tests are presented in Figure 3. When running BWI, many of the tasks require on the order of 500 iterations to achieve the best result.

To investigate this idea further, we ran BWI for the same number of iterations as it took Root\_SWI to complete (Fixed\_BWI). Notice that this value will vary depending on the data set. For example, for the SA-speaker task, we ran BWI for 111 rounds of boosting. The most important thing that can be seen from these results is that Root\_SWI and Fixed\_BWI have similar F1 performance. When we combine this fact with previous results, we conclude that the real benefit that boosting gives BWI is to increase the number of iterations for which BWI can run without overfitting the training data. This increase in iterations results in more rules learned, which consequently results in better recall.

A few other interesting things can be seen from the comparison of Root\_SWI and Fixed\_BWI. Fixed\_BWI tends to have better precision, while Root\_SWI tends to have higher recall. These differences are a result of the boosting, which is the only difference between these two algorithms. Boosting increases the weight of the difficult examples and forces BWI to focus on these hard examples. This focusing causes BWI to learn more specific rules, which results in a higher precision and lower recall. However, we should notice that as we run BWI for more iterations, we make up for this disadvantage by learning more rules, thereby increasing the recall.

## 5 Conclusion

Information extraction is becoming an increasingly important problem. Highly structured texts offer a great deal of information that allow algorithms to easily acquire patterns to extract information. Natural text, on the other hand, offers much less in the form of structure. Many of the domains that are particularly interesting, such as news articles, e-mails and web pages, consist of natural text. Sophisticated algorithms are required to tackle the information extraction task in these natural domains. BWI is an algorithm that boosts a weak learning algorithm that has traditionally been used in highly structured domains to create an algorithm that performs well in natural text domains. The benefits from boosting have been seen empirically and theoretically. We have taken another approach to exploring the benefits of boosting by comparing a boosted algorithm to an algorithm that uses

sequential covering. We have shown that the key benefit of boosting is to allow the BWI algorithm to continue to learn relevant rules even after all of the positive examples are already covered. Another effect of boosting is to actually increase the precision of the general wrapper learned. This is offset by learning a larger number of rules, which increases the recall.

### **Acknowledgments**

I would like to thank Charles Elkan for all his help and suggestions and also Dayne Freitag for his input and for helping to make the BWI code available.

### **References**

- [1] Califf, M. (1998). Rational Learning Techniques for Natural Language Information Extraction, *Artificial Intelligence* 1998, pg. 276.
- [2] Cardie, Claire (2001). Rule Induction and Natural Language Applications of Rule Induction, <http://www.cs.cornell.edu/Info/People/cardie/tutorial/tutorial.html>.
- [3] Clark, P. and Niblett, T. (1989). The CN2 Induction Algorithm. *Machine Learning* 3, pg. 261-283.
- [4] Freitag, D. & Kushmerick, N. (2000). Boosted Wrapper Induction, *American Association for Artificial Intelligence* 2000, pg. 577-583.
- [5] Kushmerick, N. (2000). Wrapper Induction: Efficiency and Expressiveness, *Artificial Intelligence* 2000, pg. 118.
- [6] Michalski, S. (1980). Pattern recognition as rule-guided inductive inference. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2, 349-361.
- [7] Muslea, I, Minton, S. & Knoblock, C. (1999). A Hierarchical Approach to Wrapper Induction.
- [8] Quinlan, J.R. Learning logical definitions from Relations. *Machine Learning*, 5:239--266, 1990.
- [9] RISE (1998). A repository of online information sources used in information extraction tasks [<http://www.isi.edu/~muslea/RISE/index.html>]. University of Southern California, Information Sciences Institute.
- [10] Shapire, Robert E. (1999). A Brief Introduction to Boosting. *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*.

	SA-speaker			SA-location			SA-stime			SA-etime		
	Prec.	Rec.	F1	Prec.	Rec.	F1	Prec.	Rec.	F1	Prec.	Rec.	F1
BWI	79.1	59.2	67.7	85.4	69.6	76.7	99.6	99.6	99.6	94.9	94.9	94.9
Fixed BWI	88.7	44.6	59.4	92.7	73.3	81.9	99.1	94.9	96.9	99.3	81.8	89.7
SWI	90.4	34.2	49.4	92.4	64.7	75.9	97.9	84.2	90.2	98.7	81.3	88.5
Root_SWI	77.7	45.7	57.6	80.2	76.6	78.2	97.5	95.2	96.4	91.2	79.3	84.8

	Jobs-id			Jobs-company			Jobs-title		
	Prec.	Rec.	F1	Prec.	Rec.	F1	Prec.	Rec.	F1
BWI	100	100	100	88.4	70.1	78.2	59.6	43.2	50.1
Fixed BWI	100	95.6	97.8	95.5	73.3	83.0	66.1	47.9	55.6
SWI	99.6	82.9	90.2	90.4	75.1	80.2	70.0	47.7	54.9
Root_SWI	100	95.6	97.8	79.4	83.8	81.6	48.0	65.8	55.5

	LATimes-cc			Zagats-Addr			QS-date		
	Prec.	Rec.	F1	Prec.	Rec.	F1	Prec.	Rec.	F1
BWI	99.6	100	99.8	100	93.7	96.7	100	100	100
Fixed BWI	100	98.5	99.3	100	54.9	70.9	100	74.4	85.3
SWI	100	94.8	97.3	100	45.1	62.1	100	78.3	87.5
Root_SWI	99.9	97.5	98.7	100	54.9	70.9	100	74.3	85.3

Figure 4: A complete list of all tests performed. BWI is the boosted wrapper induction algorithm as described in [4]. Values in the tables are percentages ranging between 0 and 100.

	speaker	location	stime	etime	id	company	title	cc	addr	date
BWI	500	500	500	500	500	500	500	50	50	50
SWI	139.4	75.6	72.1	25.0	15.2	46.9	132.1	5.1	1.3	1.0
Root_SWI	110.7	66.8	62.0	17.9	1.0	46.3	119.6	3.6	1.3	1.0

Figure 5: The number of iterations for the three different algorithms on the 10 different IE tasks.