Fast Unsupervised Incremental Parsing

Yoav Seginer
Institute for Logic, Language and Computation
Universiteit van Amsterdam
Plantage Muidergracht 24
1018TV Amsterdam
The Netherlands
yseginer@science.uva.nl

Abstract

This paper describes an incremental parser and an unsupervised learning algorithm for inducing this parser from plain text. The parser uses a representation for syntactic structure similar to dependency links which is well-suited for incremental parsing. In contrast to previous unsupervised parsers, the parser does not use part-of-speech tags and both learning and parsing are local and fast, requiring no explicit clustering or global optimization. The parser is evaluated by converting its output into equivalent bracketing and improves on previously published results for unsupervised parsing from plain text.

1 Introduction

Grammar induction, the learning of the grammar of a language from unannotated example sentences, has long been of interest to linguists because of its relevance to language acquisition by children. In recent years, interest in unsupervised learning of grammar has also increased among computational linguists, as the difficulty and cost of constructing annotated corpora led researchers to look for ways to train parsers on unannotated text. This can either be semi-supervised parsing, using both annotated and unannotated data (McClosky et al., 2006) or unsupervised parsing, training entirely on unannotated text.

The past few years have seen considerable improvement in the performance of unsupervised parsers (Klein and Manning, 2002; Klein and Manning, 2004; Bod, 2006a; Bod, 2006b) and, for the first time, unsupervised parsers have been able to improve on the right-branching heuristic for parsing English. All these parsers learn and parse from sequences of part-of-speech tags and select, for each sentence, the binary parse tree which maximizes some objective function. Learning is based on global maximization of this objective function over the whole corpus.

In this paper I present an unsupervised parser from plain text which does not use parts-of-speech. Learning is local and parsing is (locally) greedy. As a result, both learning and parsing are fast. The parser is incremental, using a new link representation for syntactic structure. Incremental parsing was chosen because it considerably restricts the search space for both learning and parsing. The representation the parser uses is designed for incremental parsing and allows a prefix of an utterance to be parsed before the full utterance has been read (see section 3). The representation the parser outputs can be converted into bracketing, thus allowing evaluation of the parser on standard treebanks.

To achieve completely unsupervised parsing, standard unsupervised parsers, working from part-of-speech sequences, need first to induce the parts-of-speech for the plain text they need to parse. There are several algorithms for doing so (Schütze, 1995; Clark, 2000), which cluster words into classes based on the most frequent neighbors of each word. This step becomes superfluous in the algorithm I present here: the algorithm collects lists of labels for each word, based on neighboring words, and then directly

Proceedings of the 45th Annual Meeting of the Association of Computational Linguistics, pages 384–391,
Prague, Czech Republic, June 2007.

uses these labels to parse. No clustering is performed, but due to the Zipfian distribution of words, high frequency words dominate these lists and parsing decisions for words of similar distribution are guided by the same labels.

Section 2 describes the syntactic representation used, section 3 describes the general parser algorithm and sections 4 and 5 complete the details by describing the learning algorithm, the lexicon it constructs and the way the parser uses this lexicon. Section 6 gives experimental results.

2 Common Cover Links

The representation of syntactic structure which I introduce in this paper is based on links between pairs of words. Given an utterance and a bracketing of that utterance, *shortest common cover link sets* for the bracketing are defined. The original bracketing can be reconstructed from any of these link sets.

2.1 Basic Definitions

An *utterance* is a sequence of words $\langle x_1, \ldots, x_n \rangle$ and a *bracket* is any sub-sequence $\langle x_i, \ldots, x_j \rangle$ of consecutive words in the utterance. A set $\mathcal{B}$ of brackets over an utterance U is a *bracketing* of U if every word in U is in some bracket and for any $X, Y \in \mathcal{B}$ either $X \cap Y = \emptyset$, $X \subseteq Y$ or $Y \subseteq X$ (non-crossing brackets). The *depth* of a word $x \in U$ under a bracket $B \in \mathcal{B}$ ($x \in B$) is the maximal number of brackets $X_1, \ldots, X_n \in \mathcal{B}$ such that $x \in X_1 \subset \ldots \subset X_n \subset B$. A word x is a *generator* of depth d of B in $\mathcal{B}$ if x is of minimal depth under B (among all words in B) and that depth is d. A bracket may have more than one generator.

2.2 Common Cover Link Sets

A *common cover link* over an utterance U is a triple $x \xrightarrow{d} y$ where $x, y \in U$, $x \neq y$ and d is a non-negative integer. The word x is the *base* of the link, the word y is its *head* and d is the *depth* of the link. The common cover link set $R_{\mathcal{B}}$ associated with a bracketing $\mathcal{B}$ is the set of common cover links over U such that $x \xrightarrow{d} y \in R_{\mathcal{B}}$ iff the word x is a generator of depth d of the smallest bracket $B \in \mathcal{B}$ such that $x, y \in B$ (see figure 1(a)).

Given $R_{\mathcal{B}}$, a simple algorithm reconstructs the bracketing $\mathcal{B}$: for each word x and depth $0 \leq d$,

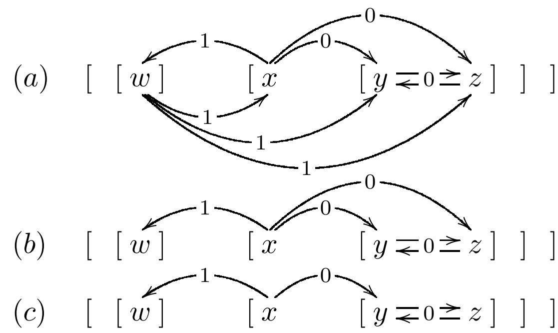


Figure 1: (a) The common cover link set $R_{\mathcal{B}}$ of a bracketing $\mathcal{B}$, (b) a representative subset R of $R_{\mathcal{B}}$, (c) the shortest common cover link set based on R.

create a bracket covering x and all y such that for some $d' \leq d$, $x \xrightarrow{d'} y \in R_{\mathcal{B}}$.

Some of the links in the common cover link set $R_{\mathcal{B}}$ are redundant. The first redundancy is the result of brackets having more than one generator. The bracketing reconstruction algorithm outlined above can construct a bracket from the links based at any of its generators. The bracketing $\mathcal{B}$ can therefore be reconstructed from a subset $R \subseteq R_{\mathcal{B}}$ if, for every bracket $B \in \mathcal{B}$, R contains the links based at least at one generator[1] of B. Such a set R is a *representative subset* of $R_{\mathcal{B}}$ (see figure 1(b)).

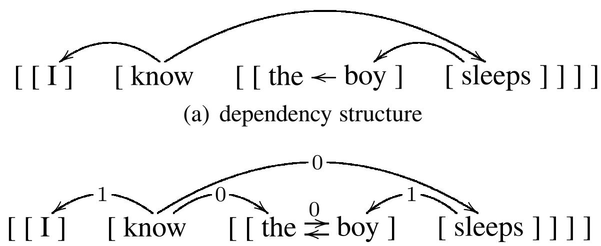
A second redundancy in the set $R_{\mathcal{B}}$ follows from the *linear transitivity* of $R_{\mathcal{B}}$:

Lemma 1 *If y is between x and z, $x \xrightarrow{d_1} y \in R_{\mathcal{B}}$ and $y \xrightarrow{d_2} z \in R_{\mathcal{B}}$ then $x \xrightarrow{d} z \in R_{\mathcal{B}}$ where if there is a link $y \xrightarrow{d'} x \in R_{\mathcal{B}}$ then $d = \max(d_1, d_2)$ and $d = d_1$ otherwise.*

This property implies that longer links can be deduced from shorter links. It is, therefore, sufficient to leave only the shortest necessary links in the set. Given a representative subset R of $R_{\mathcal{B}}$, a *shortest common cover link set* of $R_{\mathcal{B}}$ is constructed by removing any link which can be deduced from shorter links by linear transitivity. For each representative subset $R \subseteq R_{\mathcal{B}}$, this defines a unique shortest common cover link set (see figure 1(c)).

Given a shortest common cover link set S, the bracketing which it represents can be calculated by

[1] From the bracket reconstruction algorithm it can be seen that links of depth 0 may never be dropped.



(a) dependency structure

(b) shortest common cover link set

Figure 2: A dependency structure and shortest common cover link set of the same sentence.

first using linear transitivity to deduce missing links and then applying the bracket reconstruction algorithm outlined above for $R_{\mathcal{B}}$.

2.3 Comparison with Dependency Structures

Having defined a link-based representation of syntactic structure, it is natural to wonder what the relation is between this representation and standard dependency structures. The main differences between the two representations can all be seen in figure 2. The first difference is in the linking of the NP *the boy*. While the shortest common cover link set has an exocentric construction for this NP (that is, links going back and forth between the two words), the dependency structure forces us to decide which of the two words in the NP is its head. Considering that linguists have not been able to agree whether it is the determiner or the noun that is the head of an NP, it may be easier for a learning algorithm if it did not have to make such a choice.

The second difference between the structures can be seen in the link from *know* to *sleeps*. In the shortest common cover link set, there is a path of links connecting *know* to each of the words separating it from *sleeps*, while in the dependency structure no such links exist. This property, which I will refer to as *adjacency* plays an important role in incremental parsing, as explained in the next section.

The last main difference between the representations is the assignment of depth to the common cover links. In the present example, this allows us to distinguish between the attachment of the external (subject) and the internal (object) arguments of the verb. Dependencies cannot capture this difference without additional labeling of the links. In what follows, I will restrict common cover links to having depth 0 or 1. This restriction means that any tree represented by a shortest common cover link set will be skewed - every subtree must have a short branch. It seems that this is indeed a property of the syntax of natural languages. Building this restriction into the syntactic representation considerably reduces the search space for both parsing and learning.

3 Incremental Parsing

To calculate a shortest common cover link for an utterance, I will use an incremental parser. Incrementality means that the parser reads the words of the utterance one by one and, as each word is read, the parser is only allowed to add links which have one of their ends at that word. Words which have not yet been read are not available to the parser at this stage. This restriction is inspired by psycholinguistic research which suggests that humans process language incrementally (Crocker et al., 2000). If the incrementality of the parser roughly resembles that of human processing, the result is a significant restriction of parser search space which does not lead to too many parsing errors.

The adjacency property described in the previous section makes shortest common cover link sets especially suitable for incremental parsing. Consider the example given in figure 2. When the word *the* is read, the parser can already construct a link from *know* to *the* without worrying about the continuation of the sentence. This link is part of the correct parse whether the sentence turns out to be *I know the boy* or *I know the boy sleeps*. A dependency parser, on the other hand, cannot make such a decision before the end of the sentence is reached. If the sentence is *I know the boy* then a dependency link has to be created from *know* to *boy* while if the sentence is *I know the boy sleeps* then such a link is wrong. This problem is known in psycholinguistics as the problem of reanalysis (Sturt and Crocker, 1996).

Assume the incremental parser is processing a prefix $\langle x_1, \ldots, x_k \rangle$ of an utterance and has already deduced a set of links L for this prefix. It can now only add links which have one of their ends at x_k and it may never remove any links. From the definitions in section 2.2 it is possible to derive an exact characterization of the links which may be added at each step such that the resulting link set represents some bracketing. It can be shown that any shortest common cover link set can be constructed incrementally under these conditions. As the full specification of these conditions is beyond the scope of this paper, I will only give the main condition, which is based on adjacency. It states that a link may be added from x to y only if for every z between x and y there is a path of links (in L) from x to z but no link from z to y. In the example in figure 2 this means that when the word *sleeps* is first read, a link to *sleeps* can be created from *know*, *the* and *boy* but not from *I*.

Given these conditions, the parsing process is simple. At each step, the parser calculates a non-negative weight (section 5) for every link which may be added between the prefix $\langle x_1, \ldots, x_{k-1} \rangle$ and x_k. It then adds the link with the strongest positive weight and repeats the process (adding a link can change the set of links which may be added). When all possible links are assigned a zero weight by the parser, the parser reads the next word of the utterance and repeats the process. This is a greedy algorithm which optimizes every step separately.

4 Learning

The weight function which assigns a weight to a candidate link is lexicalized: the weight is calculated based on the lexical entries of the words which are to be connected by the link. It is the task of the learning algorithm to learn the lexicon.

4.1 The Lexicon

The lexicon stores for each word x a lexical entry. Each such lexical entry is a sequence of *adjacency points*, holding statistics relevant to the decision whether to link x to some other word. These statistics are given as weights assigned to labels and linking properties. Each adjacency point describes a different link based at x, similar to the specification of the arguments of a word in dependency parsing.

Let W be the set of words in the corpus. The set of *labels* $L(W) = W \times \{0, 1\}$ consists of two labels based on every word w: a *class label* $(w, 0)$ (denoted by $[w]$) and an *adjacency label* $(w, 1)$ (denoted by $[w\_]$ or $[\_w]$). The two labels $(w, 0)$ and $(w, 1)$ are said to be *opposite labels* and, for $l \in L(W)$, I write l^{-1} for the opposite of l. In addition to the labels, there is also a finite set $P = \{Stop, In^*, In, Out\}$ of *linking properties*. The $Stop$ specifies the strength of non-attachment, In and Out specify the strength of inbound and outbound links and In^* is an intermediate value in the induction of inbound and outbound strengths. A *lexicon* $\mathcal{L}$ is a function which assigns each word $w \in W$ a lexical entry $(\ldots, A^w_{-2}, A^w_{-1}, A^w_1, A^w_2, \ldots)$. Each of the A^w_i is an *adjacency point*.

Each A^w_i is a function $A^w_i : L(W) \cup P \to \mathbb{R}$ which assigns each label in $L(W)$ and each linking property in P a real valued *strength*. For each A^w_i, $\#(A^w_i)$ is the *count* of the adjacency point: the number of times the adjacency point was updated. Based on this count, I also define a normalized version of A^w_i: $\bar{A}^w_i(l) = A^w_i(l)/\#(A^w_i)$.

4.2 The Learning Process

Given a sequence of training utterances $(U_t)_{0 \leq t}$, the *learner* constructs a sequence of lexicons $(\mathcal{L}_s)_{0 \leq s}$ beginning with the zero lexicon $\mathcal{L}_0$ (which assigns a zero strength to all labels and linking properties). At each step, the learner uses the parsing function $\mathcal{P}_{\mathcal{L}_s}$ based on the previously learned lexicon $\mathcal{L}_s$ to extend the parse L of an utterance U_t. It then uses the result of this parse step (together with the lexicon $\mathcal{L}_s$) to create a new lexicon $\mathcal{L}_{s+1}$ (it may be that $\mathcal{L}_s = \mathcal{L}_{s+1}$). This operation is a *lexicon update*. The process then continues with the new lexicon $\mathcal{L}_{s+1}$. Any of the lexicons $\mathcal{L}_s$ constructed by the learner may be used for parsing any utterance U, but as s increases, parsing accuracy should improve. This learning process is open-ended: additional training text can always be added without having to re-run the learner on previous training data.

4.3 Lexicon Update

To define a lexicon update, I extend the definition of an utterance to be $U = \langle \emptyset_l, x_1, \ldots, x_n, \emptyset_r \rangle$ where $\emptyset_l$ and $\emptyset_r$ are boundary markers. The property of adjacency can now be extended to include the boundary markers. A symbol $\alpha \in U$ is *adjacent* to a word x relative to a set of links L over U if for every word z between x and α there is a path of links in L from x to z but there is no link from z to α. In the following example, the adjacencies of x_1 are $\emptyset_l$, x_2 and x_3:

$$x_1 \xrightarrow{0} x_2 \qquad x_3 \qquad x_4$$

If a link is added from x_2 to x_3, x_4 becomes adjacent to x_1 instead of x_3 (the adjacencies of x_1 are then $\emptyset_l$, x_2 and x_4):

$$x_1 \xrightarrow{0} x_2 \xrightarrow{0} x_3 \qquad x_4$$

The positions in the utterance adjacent to a word x are indexed by an index i such that $i < 0$ to the left of x, $i > 0$ to the right of x and $|i|$ increases with the distance from x.

The parser may only add a link from a word x to a word y adjacent to x (relative to the set of links already constructed). Therefore, the lexical entry of x should collect statistics about each of the adjacency positions of x. As seen above, adjacency positions may move, so the learner waits until the parser completes parsing the utterance and then updates each adjacency point A_i^x with the symbol α at the ith adjacency position of x (relative to the parse generated by the parser). It should be stressed that this update does not depend on whether a link was created from x to α. In particular, whatever links the parser assigns, $A_{(-1)}^x$ and A_1^x are always updated by the symbols which appear immediately before and after x.

The following example should clarify the picture. Consider the fragment:

$$\text{put} \xrightarrow{0} \text{the} \overset{0}{\rightleftarrows} \text{box} \qquad \text{on}$$

All the links in this example, including the absence of a link from *box* to *on*, depend on adjacency points of the form $A_{(-1)}^x$ and A_1^x which are updated independently of any links. Based on this alone and regardless of whether a link is created from *put* to *on*, A_2^{put} will be updated by the word *on*, which is indeed the second argument of the verb *put*.

4.4 Adjacency Point Update

The update of A_i^x by α is given by operations $A_i^x(p)$ += $f(A_{(-1)}^\alpha, A_1^\alpha)$ which make the value of $A_i^x(p)$ in the new lexicon $\mathcal{L}_{s+1}$ equal to the sum $A_i^x(p) + f(A_{(-1)}^\alpha, A_1^\alpha)$ in the old lexicon $\mathcal{L}_s$.

Let $Sign(i)$ be 1 if $0 < i$ and -1 otherwise. Let

$$\bullet A_i^\alpha = \begin{cases} true & \text{if } \nexists l \in L(W) : \\ & \qquad A_i^\alpha(l) > A_i^\alpha(Stop) \\ false & \text{otherwise} \end{cases}$$

The update of A_i^x by α begins by incrementing the count:

$$\#(A_i^x) \mathrel{+}= 1$$

If α is a boundary symbol ($\emptyset_l$ or $\emptyset_r$) or if x and α are words separated by stopping punctuation (full stop, question mark, exclamation mark, semicolon, comma or dash):

$$A_i^x(Stop) \mathrel{+}= 1$$

Otherwise, for every $l \in L(W)$:

$$A_i^x(l^{-1}) \mathrel{+}= \begin{cases} 1 & \text{if } l = [\alpha] \\ \bar{A}_{Sign(-i)}^\alpha(l) & \text{otherwise} \end{cases}$$

(In practice, only $l = [\alpha]$ and the 10 strongest labels in $A_{Sign(-i)}^\alpha$ are updated. Because of the exponential decay in the strength of labels in $A_{Sign(-i)}^\alpha$, this is a good approximation.)

If $i = -1, 1$ and α is not a boundary or blocked by punctuation, simple bootstrapping takes place by updating the following properties:

$$A_i^x(In^*) \mathrel{+}= \begin{cases} -1 & \text{if } \bullet A_{Sign(-i)}^\alpha \\ +1 & \text{if } \neg \bullet A_{Sign(-i)}^\alpha \wedge \bullet A_{Sign(i)}^\alpha \\ 0 & \text{otherwise} \end{cases}$$

$$A_i^x(Out) \mathrel{+}= \bar{A}_{Sign(-i)}^\alpha(In^*)$$

$$A_i^x(In) \mathrel{+}= \bar{A}_{Sign(-i)}^\alpha(Out)$$

4.5 Discussion

To understand the way the labels and properties are calculated, it is best to look at an example. The following table gives the linking properties and strongest labels for the determiner *the* as learned from the complete Wall Street Journal corpus (only $A_{(-1)}^{the}$ and A_1^{the} are shown):

the

A_{-1}		A_1	
$Stop$	12897	$Stop$	8
In^*	14898	In^*	18914
In	8625	In	4764
Out	-13184	Out	21922
[the]	10673	[the]	16461
[of_]	6871	[a]	3107
[in_]	5520	[_the]	2787
[a]	3407	[of]	2347
[for_]	2572	[_company]	2094
[to_]	2094	['s]	1686

A strong class label $[w]$ indicates that the word w frequently appears in contexts which are similar to *the*. A strong adjacency label $[w\_]$ (or $[\_w]$) indicates that w either frequently appears next to *the* or that w frequently appears in the same contexts as words which appear next to *the*.

The property $Stop$ counts the number of times a boundary appeared next to *the*. Because *the* can often appear at the beginning of an utterance but must be followed by a noun or an adjective, it is not surprising that $Stop$ is stronger than any label on the left but weaker than all labels on the right. In general, it is unlikely that a word has an outbound link on the side on which its $Stop$ strength is stronger than that of any label. The opposite is not true: a label stronger than $Stop$ indicates an attachment but this may also be the result of an inbound link, as in the following entry for *to*, where the strong labels on the left are a result of an inbound link:

to

A_{-1}		A_1	
$Stop$	822	$Stop$	48
In^*	-4250	In^*	-981
In	-57	In	-1791
Out	-3053	Out	4010
[to]	5912	[to]	7009
[%_]	848	[_the]	3851
[in]	844	[_be]	2208
[the]	813	[will]	1414
[of]	624	[_a]	1158
[a]	599	[the]	954

For this reason, the learning process is based on the property ${}_\bullet A_i^x$ which indicates where a link is *not possible*. Since an outbound link on one word is inbound on the other, the inbound/outbound properties of each word are then calculated by a simple bootstrapping process as an average of the opposite properties of the neighboring words.

5 The Weight Function

At each step, the parser must assign a non-negative weight to every candidate link $x \overset{d}{\to} y$ which may be added to an utterance prefix $\langle x_1, \ldots, x_k \rangle$, and the link with the largest (non-zero) weight (with a preference for links between x_{k-1} and x_k) is added to the parse. The weight could be assigned directly based on the In and Out properties of either x or y but this method is not satisfactory for three reasons: first, the values of these properties on low frequency words are not reliable; second, the values of the properties on x and y may conflict; third, some words are ambiguous and require different linking in different contexts. To solve these problems, the weight of the link is taken from the values of In and Out on the best matching label between x and y. This label depends on both words and is usually a frequent word with reliable statistics. It serves as a prototype for the relation between x and y.

5.1 Best Matching Label

A label l is a *matching label* between A_i^x and $A_{Sign(-i)}^y$ if $A_i^x(l) > A_i^x(Stop)$ and either $l = (y, 1)$ or $A_{Sign(-i)}^y(l^{-1}) > 0$. The *best matching label* at A_i^x is the matching label l such that the *match strength* $\min(\bar{A}_i^x(l), \bar{A}_{Sign(-i)}^y(l^{-1}))$ is maximal (if $l = (y, 1)$ then $\bar{A}_{Sign(-i)}^y(l^{-1})$ is defined to be 1). In practice, as before, only the top 10 labels in A_i^x and $A_{Sign(-i)}^y$ are considered.

The *best matching label from x to y* is calculated between A_i^x and $A_{Sign(-i)}^y$ such that A_i^x is on the same side of x as y and was either already used to create a link or is the first adjacency point on that side of x which was not yet used. This means that the adjacency points on each side have to be used one by one, but may be used more than once. The reason is that optional arguments of x usually do not have an adjacency point of their own but have the same labels as obligatory arguments of x and can share their adjacency point. The A_i^x with the strongest matching label is selected, with a preference for the unused adjacency point.

As in the learning process, label matching is blocked between words which are separated by stopping punctuation.

5.2 Calculating the Link Weight

The best matching label $l = (w, \delta)$ from x to y can be either a class ($\delta = 0$) or an adjacency ($\delta = 1$) label at A_i^x. If it is a class label, w can be seen as taking the place of x and all words separating it from y (which are already linked to x). If l is an adjacency label, w can be seen to take the place of y. The calculation of the weight $Wt(x \overset{d}{\to} y)$ of the link from x to y is therefore based on the strengths of the In and Out properties of A_σ^w where $\sigma = Sign(i)$ if $l = (w, 0)$ and $\sigma = Sign(-i)$ if $l = (w, 1)$. In addition, the weight is bounded from above by the best label match strength, $s(l)$:

- If $l = (w, 0)$ and $A_\sigma^w(Out) > 0$:

$$Wt(x \overset{0}{\to} y) = \min(s(l), \bar{A}_\sigma^w(Out))$$

Model	WSJ10			WSJ40			Negra10			Negra40		
	UP	UR	UF_1	UP	UR	UF_1	UP	UR	UF_1	UP	UR	UF_1
Right-branching	55.1	70.0	61.7	35.4	47.4	40.5	33.9	60.1	43.3	17.6	35.0	23.4
Right-branching+punct.	59.1	74.4	65.8	44.5	57.7	50.2	35.4	62.5	45.2	20.9	40.4	27.6
Parsing from POS												
CCM	64.2	81.6	71.9				48.1	85.5	61.6			
DMV+CCM(POS)	69.3	88.0	77.6				49.6	89.7	63.9			
U-DOP	70.8	88.2	78.5			63.9	51.2	90.5	65.4			
UML-DOP			82.9			66.4			67.0			
Parsing from plain text												
DMV+CCM(DISTR.)	65.2	82.8	72.9									
Incremental	75.6	76.2	75.9	58.9	55.9	57.4	51.0	69.8	59.0	34.8	48.9	40.6
Incremental (right to left)	75.9	72.5	74.2	59.3	52.2	55.6	50.4	68.3	58.0	32.9	45.5	38.2

Table 1: Parsing results on WSJ10, WSJ40, Negra10 and Negra40.

- If $l = (w, 1)$:
  - If $A^w_\sigma(In) > 0$:
  $$Wt(x \overset{d}{\rightarrow} y) = \min(s(l), \bar{A}^w_\sigma(In))$$
  - Otherwise, if $A^w_\sigma(In^*) \geq |A^w_\sigma(In)|$:
  $$Wt(x \overset{d}{\rightarrow} y) = \min(s(l), \bar{A}^w_\sigma(In^*))$$

  where if $A^w_\sigma(In^*) < 0$ and $A^w_\sigma(Out) \leq 0$ then $d = 1$ and otherwise $d = 0$.
- If $A^w_\sigma(Out) \leq 0$ and $A^w_\sigma(In) \leq 0$ and either $l = (w, 1)$ or $A^w_\sigma(Out) = 0$:
  $$Wt(x \overset{0}{\rightarrow} y) = s(l)$$
- In all other cases, $Wt(x \overset{d}{\rightarrow} y) = 0$.

A link $x \overset{1}{\rightarrow} y$ attaches x to y but does not place y inside the smallest bracket covering x. Such links are therefore created in the second case above, when the attachment indication is mixed.

To explain the third case, recall that $s(l) > 0$ means that the label l is stronger than $Stop$ on A^x_i. This implies a link unless the properties of w block it. One way in which w can block the link is to have a positive strength for the link in the opposite direction. Another way in which the properties of w can block the link is if $l = (w, 0)$ and $A^w_\sigma(Out) < 0$, that is, if the learning process has explicitly determined that no outbound link from w (which represents x in this case) is possible. The same conclusion cannot be drawn from a negative value for the In property when $l = (w, 1)$ because, as with standard dependencies, a word determines its outbound links much more strongly than its inbound links.

6 Experiments

The incremental parser was tested on the Wall Street Journal and Negra Corpora.[2] Parsing accuracy was evaluated on the subsets WSJX and Negra$X$ of these corpora containing sentences of length at most X (excluding punctuation). Some of these subsets were used for scoring in (Klein and Manning, 2004; Bod, 2006a; Bod, 2006b). I also use the same precision and recall measures used in those papers: multiple brackets and brackets covering a single word were not counted, but the top bracket was.

The incremental parser learns while parsing, and it could, in principle, simply be evaluated for a single pass of the data. But, because the quality of the parses of the first sentences would be low, I first trained on the full corpus and then measured parsing accuracy on the corpus subset. By training on the full corpus, the procedure differs from that of Klein, Manning and Bod who only train on the subset of bounded length sentences. However, this excludes the induction of parts-of-speech for parsing from plain text. When Klein and Manning induce the parts-of-speech, they do so from a much larger corpus containing the full WSJ treebank together with additional WSJ newswire (Klein and Manning, 2002). The comparison between the algorithms remains, therefore, valid.

Table 1 gives two baselines and the parsing results for WSJ10, WSJ40, Negra10 and Negra40 for recent unsupervised parsing algorithms: CCM

[2] I also tested the incremental parser on the Chinese Treebank version 5.0, achieving an F_1 score of 54.6 on CTB10 and 38.0 on CTB40. Because this version of the treebank is newer and clearly different from that used by previous papers, the results are not comparable and only given here for completeness.

and DMV+CCM (Klein and Manning, 2004), U-DOP (Bod, 2006b) and UML-DOP (Bod, 2006a). The middle part of the table gives results for parsing from part-of-speech sequences extracted from the treebank while the bottom part of the table given results for parsing from plain text. Results for the incremental parser are given for learning and parsing from left to right and from right to left.

The first baseline is the standard right-branching baseline. The second baseline modifies right-branching by using punctuation in the same way as the incremental parser: brackets (except the top one) are not allowed to contain stopping punctuation. It can be seen that punctuation accounts for merely a small part of the incremental parser's improvement over the right-branching heuristic.

Comparing the two algorithms parsing from plain text (of WSJ10), it can be seen that the incremental parser has a somewhat higher combined F_1 score, with better precision but worse recall. This is because Klein and Manning's algorithms (as well as Bod's) always generate binary parse trees, while here no such condition is imposed. The small difference between the recall (76.2) and precision (75.6) of the incremental parser shows that the number of brackets induced by the parser is very close to that of the corpus[3] and that the parser captures the same depth of syntactic structure as that which was used by the corpus annotators.

Incremental parsing from right to left achieves results close to those of parsing from left to right. This shows that the incremental parser has no built-in bias for right branching structures.[4] The slight degradation in performance may suggest that language should not, after all, be processed backwards.

While achieving state of the art accuracy, the algorithm also proved to be fast, parsing (on a 1.86GHz Centrino laptop) at a rate of around 4000 words/sec. and learning (including parsing) at a rate of 3200 – 3600 words/sec. The effect of sentence length on parsing speed is small: the full WSJ corpus was parsed at 3900 words/sec. while WSJ10 was parsed at 4300 words/sec.

[3] The algorithm produced 35588 brackets compared with 35302 brackets in the corpus.

[4] I would like to thank Alexander Clark for suggesting this test.

7 Conclusions

The unsupervised parser I presented here attempts to make use of several universal properties of natural languages: it captures the skewness of syntactic trees in its syntactic representation, restricts the search space by processing utterances incrementally (as humans do) and relies on the Zipfian distribution of words to guide its parsing decisions. It uses an elementary bootstrapping process to deduce the basic properties of the language being parsed. The algorithm seems to successfully capture some of these basic properties, but can be further refined to achieve high quality parsing. The current algorithm is a good starting point for such refinement because it is so very simple.

Acknowledgments I would like to thank Dick de Jongh for many hours of discussion, and Remko Scha, Reut Tsarfaty and Jelle Zuidema for reading and commenting on various versions of this paper.

References

Rens Bod. 2006a. An all-subtrees approach to unsupervised parsing. In *Proceedings of COLING-ACL 2006*.

Rens Bod. 2006b. Unsupervised parsing with U-DOP. In *Proceedings of CoNLL 10*.

Alexander Clark. 2000. Inducing syntactic categories by context distribution clustering. In *Proceedings of CoNLL 4*.

Matthew W. Crocker, Martin Pickering, and Charles Clifton. 2000. *Architectures and Mechanisms for Language Processing*. Cambridge University Press.

Dan Klein and Christopher D. Manning. 2002. A generative constituent-context model for improved grammar induction. In *Proceedings of ACL 40*, pages 128–135.

Dan Klein and Christopher D. Manning. 2004. Corpus-based induction of syntactic structure: Models of dependency and constituency. In *Proceedings of ACL 42*.

David McClosky, Eugene Charniak, and Mark Johnson. 2006. Effective self-training for parsing. In *Proceedings of HLT-NAACL 2006*.

Hinrich Schütze. 1995. Distributional part-of-speech tagging. In *Proceedings of EACL 7*.

Patrick Sturt and Matthew W. Crocker. 1996. Monotonic syntactic processing: A cross-linguistic study of attachment and reanalysis. *Language and Cognitive Processes*, 11(5):449–492.