A Dependency-based Method for Evaluating Broad-Coverage Parsers

Dekang Lin"
Department of Computer Science
University of Manitoba
Winnipeg, Manitoba, Canada R3T 2N2
lindek'OJcs umamtoba ca

Abstract

With tht emergence of broad-coverage parsers, quantitative evaluation of parsers becomes increasingly more important We propose a dependency-based method for evaluating broad-coverage parsers The method offers several advantages over previous methods that are based on phrase boundaries The error count score WL propose here is not only more mtuitivtlv meaningful than other scores but also more relevant lo semantic interpretation We will also present an algorithm for transforming constituency trees into dependency trees so thai the (.valuation method is applicable to both dependency and constituency grammars Finally we discuss a set of operations for modifying dependency trees that can be used lo eliminate inconsequential differences among different parse trees and allow us to selectively evaluate different aspects of a parser

1 Introduction

With the emergence of broad-coverage parsers, quantitative evaluation of parsers becomes increasingly more *important* It is generally accepted thai such evaluation should bt conducted b\ comparing the parser-generated parse trees (we call them answers) with manually constructed parse trees (we call them keys) However, how such comparison should be performed is still subject to debate Several proposals have been put forward [Black *ct al* 1991 1992, Magerman, 1994], all of which are based on Lhe comparison between phrase boundaries in answers and kev«, Wt propose a dependency-based evaluation scheme m which tht dependency relations rather than phrase boundaries, are the focus in the comparison between answers and keys We then show that the dependency based scheme offers several advantages over previous proposals Note that the use of dependency relations here does not mean that the scheme is onlv applicable to dependency grammars It only means that constituency trees have Lo be transformed into dependency trees before answers and keys are compared A transformation procedure will be presented in Section 4 3

2 Previous Approaches

*G*iven a node in a parse tret, the sequence of words dominated bv the node form a phrase and the boundary of the phrase can be denoted by an integer interval *[i,j]* where ; is the index *of* the first word *m* the phrasr and *j* is the index of the last word in the phrase For Lxample the parse tree in (1) contains three phrase boundaries [0,2], [1,1], and [1,2]

(1) [They [[came] yesterday]]

Previous evaluation schemes can be classified as phrase-level or sentence level In a phrase-level (valuation, the following goodness scores are computed

Precision and recall The phrase boundaries in the answer and the key art treated as two sets (A and K respectively) The recall is defined as the percentage of phrase boundaries in the kev that are also found in the answer (kfSf[1]) The precision is defined as the percentage of phrase boundaries in the answer that are also found in the key (UnAI / MI '

Number of crossing-brackets A pair of phrase boundaries *[i j]* and [i' *j']* are said to be crossing brackets if *i* < *i'* < *j* < / Parsers can be evaluated by the dverage pairs of crossing brackets per sentence

For example, suppose, (1) is the key and (2) is the answer

(2) [[They [came]] [yesterday]]

The phrase boundaries in (2) are [0 2], [1,1], [0,1], and [2,2] Thus, the scores of (2) are precision=^=50%, recall=|=66 7% and there is one pair of crossing brackets [0,1] in the key and [1,2] in the answer These scores have to be considered together to be meaningful For example, treating the sentence "they came yesterday as a flat list of words [they came yesterday] would achieve 0-crossing-brackets and 100% precision However, the recall is quite low (1/3=33 3%)

'The aulhor wishes to Lhank 1JCAI reviewers for pointing out several errors in the draft and Mr Wei Xiao for Implementing the algorithms presented in the paper The author is a member of the Institute for Robotics and Intelligent Systems (IRIS) and wishes to acknowledge Lhe support of the Networks of Centres of Excellence Program of the Goveminent of Canada the Natural Sciences and Engineering Research Council (NSERC), and the participation of PRECARN Associates Inc Thi6 research has also been partially supported by NSERC Research Grant OGP121338

In a sentence-level evaluation, a sentence is considered to be correctly parsed if certain criteria have been met. In [Black *et al.*, 1992], the correctness criterion is that the number of crossing-brackets is 0. In [Magerman, 1994], a sentence is correctly parsed if both precision and recall are 100%. The scores assigned to parsers are their percentages of correctly parsed sentences in the test corpus.

A problem with the phrase-level scores is that they do not necessarily reflect the quality of parse trees because a single error may be counted multiple times. For example, suppose (3a) is the key and (3b) (3c) are answers returned by two parsers.

(3)

a. [I [saw [[a man] [with [[a dog] and [a cat]]]] [in [the park]]]]

b. [I [saw [[a man] [with [[a dog] and [[a cat] [in [the park]]]]]]]]

c. [I [saw [a man] with [a dog] and [a cat] [in [the park]]]]

The only difference between (3a) and (3b) is that the prepositional phrase [in [the park]] is the sister of "saw" in (3a) but the sister of "a cat" in (3b). However, because of this single attachment error, there are 3 pairs of crossing brackets:

1. [a dog and a cat] vs. [a cat in the park]
2. [with a dog and a cat] vs. [a dog and a cat in the park]
3. [a man with a dog and a cat] vs. [with a dog and a cat in the park]

The recall of (3b) is $\frac{6}{10}$=60% and the precision is $\frac{7}{11}$=63.6%. In contrast, (3c) is a very shallow parse of the sentence. It has no crossing brackets, perfect precision (100%) and a better recall ($\frac{7}{10}$=70%) than (3b). Intuitively, the structure (3b) has a lot more in common with (3a) than (3c). Yet, (3b) scored much poorly than (3c) according to the precision, recall and the number of crossing-brackets.

In sentence-level evaluations, an error will not be counted more than once. However, the other extreme has to be adopted: no matter how many mistakes a parser makes in a parse tree, they are only counted as one error. Thus an answer with a single error is treated the same as an answer with many serious errors. Since how much a parse tree deviates from the correct one greatly influence the chance of the sentence being interpreted correctly, the evaluation scheme should take the degree of the deviation into account.

3 Desiderata

In this section, we motivate the dependency-based evaluation by pointing out several desirable properties that are missing from previous evaluation schemes.

3.1 Ignoring the inconsequential differences

An objective of an evaluation scheme is to identify the differences between answers and keys. However, certain types of differences are of no consequence to the interpretation of a sentence. An ideal evaluation scheme should make provisions to ignore such differences.

For example, consider sentence structures in (4).

(4)

a. [[Bellows [made [the request]]] [while [[the all-woman jury] [was [out [of [the courtroom]]]]]]]

b. [Bellows [[made [the request]] [while [[the [[all-woman] jury]] [was [out of [the courtroom]]]]]]]

In (4a), the clause [while ...] is the sister of [Bellows made the request]. In (4b), it is the sister of [made the request]. Differences such as this are typically of no consequence to the interpretation of the sentence. However, if (4b) is evaluated against (4a), its scores are as follows:

recall=$\frac{9}{11}$=81.8%; precision=$\frac{9}{12}$=75.0%; crossings=1

If a sentence-level evaluation is used, (4b) would be classified as incorrect, even though, from linguistic point of view, it may well be as reasonable as (4a).

3.2 Selective evaluation

Previous evaluation schemes only assess the overall performance of parsers. A more flexible scheme should be able to selectively evaluate parsers with respect to any given types of syntactic phenomena. It would be interesting to know, for example, how well a parser handles conjunctions or how well a parser would perform if prepositional attachments were ignored. Answers to these questions would help to determine the suitability of a parser for a particular purpose.

3.3 Facilitate the diagnosis of incorrect parses

Besides measuring the performance of parsers, another service that should be provided by a parser evaluation scheme is to help developers to improve their parsers by pin-pointing exactly where the errors are. As we have discussed earlier, a single attachment error may cause several crossing brackets and several spurious/missing phrase boundaries. Given the list of crossing brackets or the sets of spurious/missing phrase boundaries, it is not obvious what caused them to occur.

4 Dependency-based Evaluation

In this section, we propose a dependency-based parser evaluation scheme that offers the desirable properties discussed in the previous section. Instead of phrase boundaries, the scheme is based on the comparison between the dependency relations in answers and keys.

4.1 Dependency trees

In a dependency tree, every word in the sentence is a modifier of exactly one other word (called its head), except the head word of the sentence, which does not have a head [Mel'čuk, 1987]. We use a list of tuples to specify a dependency tree. A tuple has the following format:

`(modifier cat position head [relationship])`

where, **modifier** is a word in the sentence, **cat** is its lexical category, **head** is the word that **modifier** modifies, **relationship** is an optional specification of the

(5)

a [I [saw [[a man] [with [[a dog] and [a cat]]]] [in [the park]]]]

b [I [saw [[a man] [with [[a dog] and [[a cat] [in [the park]]]]]]]]

c

Key (5a)					Answer (5b)					error
I	N	<	saw	subj	I	N	<	saw	subj	
saw	V	*			saw	V	*			
a	Det	<	man	spec	a	Det	<	man	spec	
man	N	>	saw	cmpl	man	N	>	saw	cmpl	
with	P	>	man	adjn	with	P	>	man	adjn	
a	Det	<	dog	spec	a	Det	<	dog	spec	
dog	N	<	and		dog	N	<	and		
and	Conj	>	with	cmpl	and	Conj	>	with	cmpl	
a	Det	<	cat	spec	a	Det	<	cat	spec	
cat	N	>	and		cat	N	>	and		
in	P	>	saw	adjn	in	P	>	cat	adjn	yes
the	Det	<	park	spec	the	Det	<	park	spec	
park	N	>	in	cmpl	park	N	>	in	cmpl	

(6)

a [I [saw [a man] with [a dog] and [a cat] [in [the park]]]]

b

Key (5a)					Answer (6a)					error
I	N	<	saw	subj	I	N	<	saw	subj	
saw	V	*			saw	V	*			
a	Det	<	man	spec	a	Det	<	man	spec	
man	N	>	saw	cmpl	man	N	>	saw	cmpl	
with	P	>	man	adjn	with	P	?			yes
a	Det	<	dog	spec	a	Det	<	dog	spec	
dog	N	<	and		dog	N	?			yes
and	Conj	>	with	cmpl	and	Conj	?			yes
a	Det	<	cat	spec	a	Det	<	cat	spec	
cat	N	>	and		cat	N	?			yes
in	P	>	saw	adjn	in	P	?			yes
the	Det	<	park	spec	the	Det	<	park	spec	
park	N	>	in	cmpl	park	N	>	in	cmpl	

type of the dependency relationship between **head** and **modifier**, such as subj (subject), **adjn** (adjunct) **cmpl** (complement), **spec** (specifier), *etc*, **position** indicates the position of the head relative to the modifier. It can take one of the following values {<, >, <<, >>, <<<, , *, ?}, where < (or >) means that the head of **modifier** is the first occurrence of the word **head** to the *left* (or right) of the modifier, << (or >>) means head is the second occurrence of the word **head** to the left (or right) of the modifier. If **position** is '*', then the word is the head of the sentence. If **position** is ?, then the word's head is unknown (either to the parser or the human analyst who created the parse tree)

The dependency trees of (3a) and (3b) (re-written as (5a) and (5b)) are shown in the first and second column in (5c) respectively

4 2 Evaluation

Once the answer and the key are both represented as dependency trees, the answer can be scored on a word-by-word basis. Since both the answer and the key assign a head to every word in the sentence we define the **error count** of the answer to be the number of words that are assigned different heads in the answer than in the key. For example, there is one error in (5b). In contrast, the dependency tree corresponding to the shallow parse tree (3c) (re-written as (6a)) contains 5 unknown heads each of which is counted as an *error* (see (6b))

Given two dependency trees **key** and **answer**, the function **evaluate** (7) returns the error count of the answer

The error count is a Hamming distance between the answer and the key, because it is the number of dependency relationships that must be altered in order to make the answer identical to the key. Compared with scores such as precision, recall, and the number of crossing-brackets, the error-count score is intuitively more meaningful. Since the phrase boundaries themselves are not used directly in semantic interpretation, it is hard to predict how missing or spurious phrase boundaries affect semantic interpretation. On the other hand, since the semantic dependencies are embedded in the syntactic dependencies, the semantic interpretation process should be more sensitive to the number of missing or spurious syntactic dependencies than the number of missing, spurious or crossed phrase boundaries

4 3 Converting constituency trees into dependency trees

Since the procedure **evaluate** takes the dependency trees as inputs, whereas almost all the broad-coverage

```
lilt evaluate(DepTree key  DepTree anseer)

  errorCount
  for each word in the sentence
    if (the position of the key is not equal to   ' and the position
        or the head of the key is not equal to that at the answer)
      errorCount - error-Count

  return errorCount
```

parsers and treebanks use constituency grammars a crucial issue that must be resolved is how to apply the method to constituency grammars

In this section we preterit an algorithm lu transform the constituency trees into dependency trees IF one or both of the key and the answer arc represented as *constituency* trees, we first transform them into dependency trees and then evaluatetc the parser with the resulting dependency trees

The transformation algorithm is based on Magerman's method for determining heads (lexical representatives) in (FG parse trees [Magerman, 1994 p 64-66] following Magerrman thf[4] transformation is driven bv a Tree Head Table which contains an entry Tor every non-terminal symbol in the grammar Given a node in a constituency tree the corresponding entry in the Tret. Head Table can be used to determine the head child of the node (the head child of a node is either its lexical head or a child that dominates its lexical head)

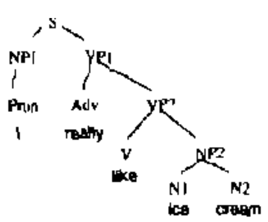
Untries in a tree head table are triples (parent direction head-list) where parent is a grammatical catagory, direction is either right-to-left or lett-to-right and head-list is a list of grammatical categories Three sample entries are shown in (8)

(8) (S right-to-left (Aux VP HP AP Pp)>
    (VP lelt-to-nght (V VP))
    (HP right-to-left (Pron N HP))

The firtst entry means that the head hild of an S node is the first A.ux node from right to left or if the S node does not have an Aux child the first VP node from right to left, For example given the tree head table in (8) and the constituency tree in (9a) the lexical heads and the head children of the nodes in (9a) are listed in (9b)

(9)

a



b

node	lexical head	head child
S	V	VP_1
NP_1	Pron	Pron
VP_1	V	VP_2
VP_2	V	V
NP_2	N_2	N_2

The function findHeadChild (10) returns the head child of any given node in a constituency tree using the tree head table

Unlike [Magerman, 1994*]*, where lexical heads *of* phrase are identified from bottom up wc use a top dovvn recursm procedure makeDeps to construct dep> ndency *trert, according to* parse trees 7 lie procedure returns the lexical head of the tree

```
(II) Tree ■&keDepB(Tree root, DepTree depe)
  {
    if (root is a leaf node) return root
    Tree headChild - findHeadChild(root)
    Tree lexHead " MakeDepsGieadChild deps)
    for each non-head child ol root {
      lexHeadOfChild - BakeDepa(child, deps)
      addDepRelQejcHead lexHeadOfChild, deps),
    >
    return leiHead
  >
```

The function addDepReKhead, modifier, depTree) inserts the dependency between head and modifier into the dependency tree depTree The main idea of the algorithm is *as* follows

- find the head child of the root
- make a recursive call to construct the dependency tree according to the subtree rooted at the head child and return the lexical li<°ad of thf head child (which ih also the lexical head of the root node)
- for all other children of the root
  - recursively construct a dependency tree according to the subtree rooted at that child and return the lexical head *of the child*
  - add the dependency relationship between the lexical head of the root and the lexical head of the child

5 Modifying dependency trees

In [Black *cl al*, 199l], certain nodes in the answers and keys arc F ased before they are compared Ihe erased elements include for instance auviharies 'not and preninriitival ' to Ihe reason for the removal is that there are many possible ways to analyse structures involving ihest elements, all of which are correct m their own way A evaluation scheme should not prefer any one of the theories and penalize the others

There are many other kinds of allowable differences that may not be eliminated by simply removing elements from parse trees In this stction, we propose a set of operations for modifying dependency trees in a more flexible and principled fashion We then demonstrate, by

```
(10) Tree findHead Child(Tree node)    node    is assumed to be interior

       TreeHeadEntry entry    search_entry(label(node)    treeHeadTable)
       for each h in headList(entry)
         enumerate children of node according to direction(entry)
           if (label(currentChild)~h) return currentChild,

       if (direction(entry)-'left-to-nght') return flrstChild (node)
       else return last Child (node)
```

means of examples, how these operations can be used to eliminate inconsequential differences and to allow selective evaluation

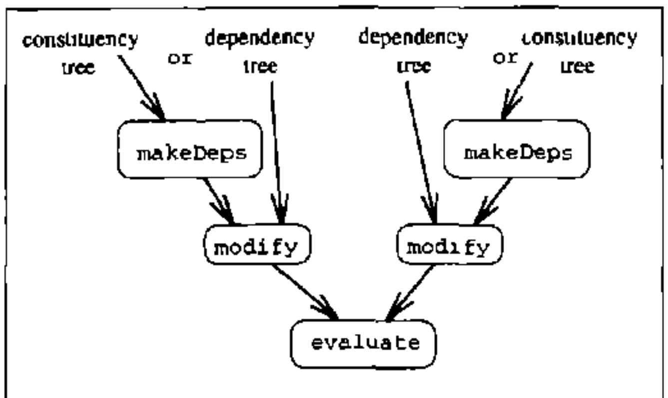


Figure 1 The process of parser evaluation

The process of dependency-based parser evaluation is depicted in Figure 1 The modify module normalize the dependency trees before they are evaluated The modify module consists of a sequence of operations Each operation specifies a possible alternation to a dependency relationship It consists of a condition part and an action part If a dependency relationship satisfy the condition, the corresponding action will be performed on the dependency The algorithm for modify is shown in (12)

A condition is a triple

(head modifier [relationship])

where head and modifier are restrictions on the head and the modifier of a dependency relationship The optional relationship component is a restriction on the type of the dependency relationship The first column in Table 1 contains several example conditions The second column contains the dependency relationships that satisfy the conditions

The action part specifies the modifications to the dependency relationship We have implemented three types of actions {deletion, inversion and transfer]

Deletion delets(head, modifier, depTree)
: removes the dependency relationship between head and modifier from the dependency tree depTree

Inversion invert(head modifier, depTree)
: reverses the direction of the dependency relationship between head and modifier In the mean time, if head also has a head (called head of Head), then the dependeniy between the head Of Head and head is replaced with the dependency between headOfHead and modifier

Transfer tranefer (head, modifier, depTree)
: transfers modifiers of modifier to head In other words, all the modifiers of modifier now become modifiers of head

Figure 2 shows an example of each of these actions

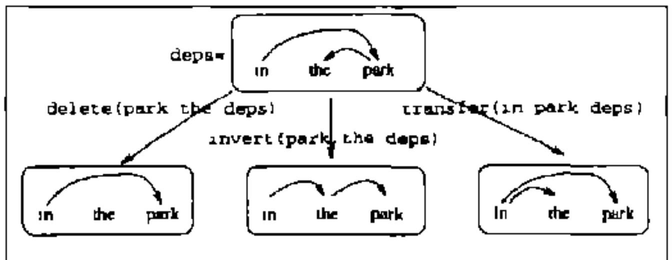


Figure 2 Modifying dependency relationships

In the remainder of this section, we demonstrate how these modifications can be used to eliminate inconsequential differences and to allow selective evaluation

5 1 Eliminating inconsequential differences

Different grammars often treat adverbs differently For example, in 'she will leave soon", the adverb 'soon' can either be analyzed as the modifier of 'will' (Figure 3a) or "leave' (Figure 3b) If the operation

(if ((cat Aux) (cat V)) (invert transfer))

is applied to both trees, they become identical (Figure 3r) In Figure 3a the dependency link from wilT to ' leave' is first inverted, so that "will' becomes a modifier of "leave" Then, the modifiers of "wiir ("she and 'soon') are transferred to 'leave', resulting in Figure 3c

Conjunction is another syntactic phenomenon that tends to be treated differently in different theories Figure 4 shows three alternative analyses of the dependency tree of "saw A and B " They can be transformed into an identical form by the operations shown in the figure Note that such variations in the analyses of conjunctions cannot be normalized by simply removing elements from parse trees

5 2 Selective evaluation

The modification to the dependency tree also allows us to selectively evaluate the performance of parsers with

(12)

```
void modify (operations DepTree depTree)

  for each operation (condition, action) in operations
    for each dependency relation dep in depTree
      if (dep satisfies condition) perform action on dep
```

Table 1 Example conditions

Condition	Dependency relationship between
((cat N) (cat Det))	a noun and its determiner
((cat N) (or (string "a") (string "the')))	a noun and 'a or the
((cat Conj) t)	a connective (e g and or') and any other word
(t (cat P) (type adjn))	any word and its prepositional adjunct

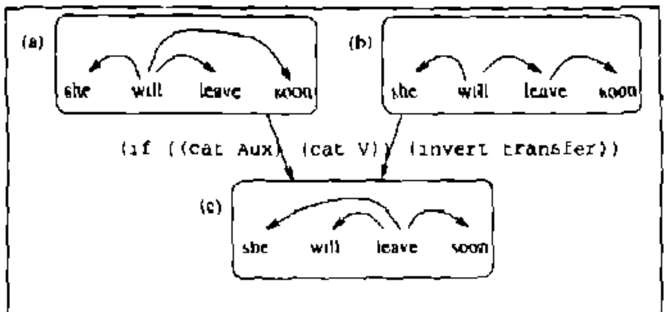


Figure 3 Normalization of auxiliaries

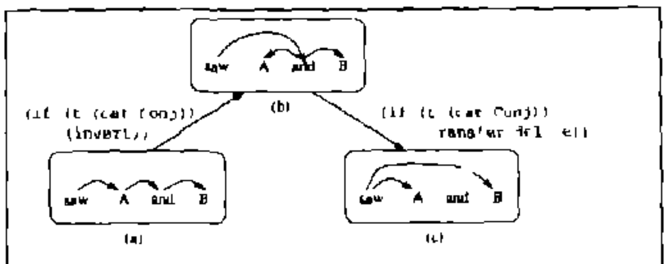


Figure 4 Normalization of conjunctions

regard to various syntactic phenomena. Vor example if we want Lo find out how successfuly a parser deals with prepositional phrase attachments wo can use the following operation to delete all the other dependencies except those in which the modifier is *A* preposition

(if (t (not (cat P))) (delete))

On the other hand evaluating the result of applying

(if (t (cat P)) (delete))

to dependency trees would tell us how a parser would fare if attachments of prepositional phrases are ignored

6 Conclusion

We have presented a dependency-based method for eval uating broad-cover age parsers The method offers several advantages over previous methods that relied on the comparison of phrase boundaries The error count score is not only more intuitively meaningful than other scores but also more relevant lo semantic inUrprcti-lion We also presented an algorithm that transforms constituent trees into dependency tree so that the valuation method is applicable lo both dependency -ind constituency grammars Finally w< proposed a set of operations for modifying dependency treeb thd(can ht used to eliminate inconsi quentnl difference? among dif ferenl parse trees and allow us to sthctiulv cvaluite different, aspects of a parstr

References

[Black *ti al* 1901] L Black, S Ahne> D Thckenger (Gdaniec, R (irishman P Harrison D llintlle, R Ingni t Jflinek J KldVans M Libt rm in M Marcus S Roukos B Sanlonni and I Str/a-lkowski A proc(dur(for quantitatively comparing lhe syntactic coverage of enghsh grammars In *Proceedings of Speech and \atvral Language \\ tukshop^* pages 306-311 DARPA February 19<)1

[Black *el al* 1992] E?ra Black John Laffertv and Sihin Rouko<; Developrmnl *d,nd i* \?Ju<ilie>u <f *i* broad-coveragi probabilistic grammar of ! nglish-language computer manuals In Proceedings *of AC L-9J* pa^es IB') 192 Newark Delaware *WJ2*

[Magermau, 1994] Davtd M Magirman \atuial *Language Parsing a& Statistical Pattern Recognition* PhD thesis Stanford University, I994

[MelVuk 1987] Igor A Melcuk *Dependent y syntex theory and practice state* University of New. york Press Albany 1987