Parsing Dependencies

Author 1
XYZ Company
111 Anywhere Street
Mytown, NY 10000, USA
author1@xyz.org

Author 2
ABC University
900 Main Street
Ourcity, PQ, Canada A1A 1T2
author2@abc.ca

Abstract

1 Introduction

There has been a surge of recent work on dep parsing () () ()

This work has brought the accuracy of dependency parsing much closer to the level of powerful constituency parser. For instance the parser of () score % on dependency UAS whereas the has UAS etc.

In this paper we ask a simple question: can we invert this process? Given the raw skeleton of a dependency parse, is it plausible to obtain an accurate phrase-structure parse for a sentence.

To approach this question we build a very simple parser, that takes in the fully specified

There has been a series of work that looks at the relationship between dependency and constituency parsing. (**?**) build a powerful dependency parsing-like model that predicts adjunctions of the elementary TAG spines. (**?**) use dual decomposition to combine a powerful dependency parser with a simple constituency model.

There have also been several papers that take the idea of shift-reduce parsing, popular from dependency parsing, and apply it constituency parsing. (**?**) produces a state-of-the-art shift-reduce system. (**?**)

Our work is differs in that it takes on a much more stripped down problem. We give up entirely on determining the structure of the tree itself and only look at producing the nonterminals of the tree.

2 Background

2.1 Lexicalized CFG Parsing

Define a lexicalized context-free grammar as a 4-tuple $(\mathcal{N}, \mathcal{R}, \mathcal{T}, r)$:

- $\mathcal{N}$; a set of nonterminal symbols.
- $\mathcal{T}$; a set of terminal symbols.
- $\mathcal{R}$; a set of lexicalized rule productions of the form $\langle A \rightarrow \beta_1 \dots \beta_m, h \rangle$ consisting of a parent nonterminal $A \in \mathcal{N}$, a sequence of children $\beta_i \in \mathcal{N} \cup \mathcal{T}$ for $i \in \{1 \dots m\}$, and a distinguished head child β_h for $h \in \{1 \dots m\}$.
- r; a distinguished root symbol $r \in \mathcal{N}$.

Given fixed sequence of terminal symbols $x_1 \in \mathcal{T}, \dots, x_n \in \mathcal{T}$, for instance a sentence of words, a context-free parse consists of any ordered, finite-size tree where the interior labels come $\mathcal{N}$, the fringe labels correspond to the initial sequence, and all tree productions $A \rightarrow \beta$ are members of $\mathcal{R}$.

Furthermore, we can associate a triple $(\langle i, j \rangle, h, A)$ with each vertex in the graph.

- $\langle i, j \rangle$; the *span* of the vertex, i.e. the contiguous sequence $\{i, j\}$ of the original input covered by the vertex.
- $h \in \{1, \dots n\}$; the *head* of the vertex, defined recursively by the following rules: (a) if the vertex is a leaf, then $h = i$ where i is the position in the input sequence. (b) Otherwise, h is the head of the k'th child of the vertex where $\langle A \rightarrow \beta_1 \dots \beta_m, k \rangle$ is the rule production at

Figure 1: Figure illustrating the notation.

this vertex. We use the notation $h(v)$ to indicate the head of vertex v.

- $A \in \mathcal{T} \cup \mathcal{N}$; the terminal or nonterminal symbol of the vertex.

Finally for a given parse and any input index $i \in \{1 \dots n\}$ define the *spine* of that index as a chain of vertices $v_1 \dots v_p$ such that for all $k \in 1 \dots p$, $h(v_k) = i$ and v_i is the parent of v_{i-1}.

2.2 Dependency Parsing

Dependency trees provide an alternative, and in some sense simpler, representation of grammatical structure. Given an input sequence $x_1 \dots x_n$, we allow each position i to modify another position or to be the head of the sentence, i.e. it modifies $g_i \in \{0, n\}$ where 0 is a special pseudo-root symbol. If each word and the pseudo-root is represented as a vertex, than these relations correspond to arcs (g_i, i) in a directed graph (as shown in Figure **??**). In a valid dependency parse, the corresponding directed graph is a directed tree rooted at vertex 0.

A dependency parse is called *projective* if the descendants of every word in the tree form a contiguous span of the original sentence (). For a given position $0 \leq m \leq n$, we use the notation $m_{\Leftarrow}$ and $m_{\Rightarrow}$ to represent the left- and right-boundaries of this span.

Any lexicalized context-free parse, as defined above, can be converted deterministical into a projective dependency tree. For an input symbol x_m with spine $v_1, \dots, v_p$, if v_p is the root of the tree, then $g_m = 0$, otherwise let v_{p+1} be the parent vertex of v_p and $g_m = h(v_{p+1})$. The span $\langle i, j \rangle$ of v_p in the lexicalized parse is equivalent to $\langle m_{\Leftarrow}, m_{\Rightarrow} \rangle$ in the induced dependency parse.

Figure 2: Several distinct context-free structures that correspond to the same dependency structure.

3 Overview

In recent years there has been a steady progression in the speed and accuracy of dependency parsers (); however there are still many tasks that rely heavily on the full context-free parses (). It is therefore desirable to produce context-free parses directly from dependency structures.

However, while this conversion is deterministic from lexicalized context-free grammars to dependency parses, the inverse problem is more difficult. For a given dependency graph there may be many different valid context-free parses. This problem is the focus of this work.

This conversion is challenging in two ways:

- The original dependency parse does not contain any information about the symbols at each vertex. From the context we need to determine the nonterminals A and the rules used.
- The dependency structure does not specify the shape of the context-free parse. Even without syntactic symbols many context-free trees may be valid for a single dependency parse. This is shown in Figure **??**

Because of these issues is not obvious how to directly perform this conversion without doing search over possible structures.

4 Parsing

Our contribution to this problem will

For any parse produced in the new grammar, we can convert it to a tree in the original grammar with the same induced dependency tree.

check this

4.1 Setup

To begin we consider parsing a standard lexicalized context-free grammar with a factored scoring function.

Recall that the each vertex in the tree has the form $(\langle i, j \rangle, h, A)$. After binarization, each tree production is roughly of the form $(\langle i, j \rangle, h, A) \rightarrow$

$(\langle i, k\rangle, m, B)$ $(\langle k+1, j\rangle, h, C)$ for $A \in \mathcal{N}, B, C \in \mathcal{N} \cup \mathcal{T}, i < k < j$, and . For notational simplicity, we write this production as tuple $(\langle i, j, k\rangle, h, m, r)$ where $r = \langle A \to B\ C, 1\rangle$ is the rule applied.

Given a sentence $x_1 \dots x_n$, let $\mathcal{Y}$ be all possible valid parses under the binarized grammar. And let each parse $y \in \mathcal{Y}$ be given an a indicator vector of possible productions over where $y(\langle i, j, k\rangle, h, m, r) = 1$ if the production appears in the parse and 0 otherwise.

The standard lexicalized parsing problem is to find the highest-scoring parse

$$\hat{y} \leftarrow \arg\max_{y \in \mathcal{Y}} s(y; x)$$

where s is a linear scoring function that scores each production (described in more detail in section **??**).

This combinatorial optimization problem can be solved dynamic programming by using a simple lexicalized extension to the standard CKY parsing algorithm, shown in Figure **??**.

Since there are five free indices i, j, k, h, m, the algorithm in the worst case requires $O(n^5\mathcal{R})$ running time.

The standard algorithm for binarized context-free grammar is known as the CKY algorithm. For review the inductive form of the CKY algorithm is shown in Figure 4.2.

4.2 Parsing Dependencies

Now we make a simple extension to this algorithm. In the last section x was just the sentence, now we assume that we also are given the dependency parse. Define $\mathcal{Y}(x)$ to the be the set of parse tree that agree with the dependency structure.

Formally let

$$\mathcal{Y}(x) = \{y \in \mathcal{Y} : (i, j, k, h, m, r) \in y \implies (h, m) \in x\}$$

That is if a rule in the parse $y \in \mathcal{Y}$ then the implied dependency is (h, m) is in the dependency parse. The new problem is to solve

$$\arg\max_{y \in \mathcal{Y}(x)} s(y; x)$$

Figure **??** shows the new inductive rules. Recall that $m_{\Leftarrow}$ and $m_{\Rightarrow}$ are the precomputed left- and right- boundary words of the cover of word m.

Premise:

$$(\langle i, i\rangle, i, A) \quad \forall i \in \{1 \dots n\}, A \in \mathcal{N}$$

Rules:
For all $i < m < j < h < k$, and rules $\langle A \to B\ C, 1\rangle$,

$$\frac{(\langle i, k\rangle, h, B) \quad (\langle k, j\rangle, m, C)}{(\langle i, j\rangle, h, A)}$$

For all $i < h < j < m < k$, rules $\langle A \to B\ C, 2\rangle$,

$$\frac{(\langle i, k\rangle, m, B) \quad (\langle k, j\rangle, h, C)}{(\langle i, j\rangle, h, A)}$$

Goal:

$$(\langle 1, n\rangle, \text{root})$$

Figure 3: Standard lexicalized CKY algorithm.

While the algorithm is virtually identical to the standard lexicalized CKY algorithm, we are now able to fix many of the free indices. In fact, since there are n dependency links (h, m) and n indices i the new algorithm has $O(n^2|\mathcal{R}|)$ running time.

Here we assume there can

4.3 Extension: Labels

Standard dependency parsers also predict labels from a set $\mathcal{L}$ on each dependency link. In a labeled dependency parser a would be of the form (i, j, l).

This label information can be used to encode further information about the parse structure. For instance if we use the label set $\mathcal{L} = \mathcal{N} \times \mathcal{N} \times \mathcal{N}$, encoding the binary rule decisions $A \to BC$.

5 Model

5.1 Features

We model this problem using a standard structured prediction set-up.

The goal is to predict the best possible constituency parse for this sentence. We define the scoring function s as

$$s(y; x) = \theta^\top f(x, y)$$

Premise:

$$(\langle i, i\rangle, i, X) \quad \forall i \in \{1 \dots n\}, A \in \mathcal{N}$$

Rules:
For all i, deps. (h, m) and rules $\langle A \rightarrow B\ C, 1\rangle$,

$$\frac{(\langle i, m_{\Leftarrow} - 1\rangle, h, B) \quad (\langle m_{\Leftarrow}, m_{\Rightarrow}\rangle, m, C)}{(\langle i, m_{\Rightarrow}\rangle, h, A)}$$

For all j, deps. (h, m), and rules $\langle A \rightarrow B\ C, 2\rangle$,

$$\frac{(\langle m_{\Leftarrow}, m_{\Rightarrow}\rangle, m, B) \quad (\langle m_{\Rightarrow} + 1, j\rangle, h, C)}{(\langle m_{\Leftarrow}, j\rangle, h, A)}$$

Goal:

$$(\langle 1, n\rangle, h, r)$$

Figure 4: The main parsing algorithm.

where $\theta \in \mathbb{R}^d$ is a weight vector and $f(x, y)$ is a linear feature function that maps parse production to feature vectors in $\mathbb{R}^d$.

We experimented with several settings of the part features. The final features are shown in Figure 5.1.
() () ()

5.2 Training

Define a loss function $\Delta(\hat{y}, y)$ to determine the difference between two parses. We assume that the loss function is linear in its first argument.

At training time we run 10-fold jack-knifing to produce dependency parses at training.

One issue with learning the weight vector for this problem is the role that parsing issues play. If there is a dependency parsing mistake makes it often impossible for the system to recover the correct parse.

Define the oracle parse to be the parse minimizes the loss function

$$y^o \leftarrow \arg\max_{y' \in \mathcal{Y}} -\Delta(y', y)$$

Since this functions is linear we can use our

To learn the weight vector θ we optimize a standard structured svm objective

$$\min_{\theta} \sum_{i=1}^{m} [\theta^\top y_i^o + \max_{y'}(\theta^\top f(x, y') + \Delta(y_i^o, y'))]_+ + \frac{\lambda}{2}||\theta||_2^2$$

For a part $(i, j, k, h, m, A \rightarrow BC)$
A, h.tag, m.tag
A, B, m.tag
A, C, h.tag
$A \rightarrow BC$, h.tag, m.tag
$A \rightarrow BC$, h.word, m.tag
A, h.tag
A, h.word
A, B
A, C
unary, h.tag
unary, A B C
$A \rightarrow BC$, h.word
$A \rightarrow BC$, h.tag
$A \rightarrow BC$, m.word
$A \rightarrow BC$, m.tag

Figure 5: The features used in our experiments.

We optimize using stochastic gradient descent. ()
Note that we are optimizing towards the oracle parse.

6 Data and Setup

6.1 Data

We used wsj..
We used ctb 5-1..

6.2 Implementation

We built...
Parser is in C++, publicly available, 500 lines of code..

6.3 Binarization

Before describing our parsing algorithm we first describe a binarization approach to make efficient parsing possible and highlight the relationship between the LCFG and the dependency parse.

Consider a rule $\langle A \rightarrow \beta_1 \dots \beta_m, k\rangle$. We instead introduce the following binarized context-free rules.

fix me

- $\langle A \rightarrow \beta_1\ A_{l,1}, 2\rangle$
- $\langle A_{l,i-1} \rightarrow \beta_i\ A_{l,i}, 1\rangle$ for $1 < i < c$
- $\langle A_{r,i-1} \rightarrow A_{r,i}\ \beta_i, 1\rangle$ for $i > c$

Model	fscore	speed
TurboParser		
MaltParser		
EasyFirst		

Table 2: This

6.4 Extension: Pruning

We also experiment with a simple pruning dictionary pruning technique. For each context-free rule $A \rightarrow BC$ and POS tag a we remove all rules that were not seen with that tag as a head in training.

6.5

7 Results

7.1

Acknowledgment sections should go as a last (unnumbered) section immediately before the references.

Parsing Results		
	wsj	
	speed	fscore
Petrov Carreras		
	ctb	

Table 1: This is the big monster result table that should tower above all comers.