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

Over the last two decades there have been two dominant grammatical frameworks for statistical syntactic parsing: phrase-structure and dependency parsing (). The two frameworks often offer a practical trade-off. Phrase-structure parsing is very accurate and provides a full context-free grammatical representation; while dependency parsing is much faster, both asymptotically and empirically, while still predicting much of the important structural relationships in a sentence.

Since statistical phrase-structure parsers often use an internal lexicalized representation, it is possible to directly compare the performance of the two models in terms of dependency accuracy. (Kong and Smith, 2014) run this comparison across a wide-range of popular freely available parsing models and find that while this trade-off is still true, the gap in accuracy has significantly narrowed over the last several years. State-of-art dependency parsers such as TurboParser (), and (**?**; **?**) perform only slightly worse in terms of accuracy than large-scale reranking dependency parsers, while still maintaining efficient run-time.

Given these improvements, we ask the natural reverse question. How much of the phrase-structure is recoverable from the dependency representation? While the transformation from phrase-structure to dependencies is deterministic, the inverse is not (see Figure **??**). Can we learn to invert this process and recover the richer phrase-structure trees from the dependency representation?

name?

To approach this question we present a very simple statistical parser, MINIPARSER. The parser is a complete exact lexicalized parser; except that unlike standard parsers, it takes both a sentence and a dependency tree as input. Using this non-standard setup has several advantages:

- The parser is asymptotically three-orders of magnitude faster than standard phrase-structure parsers and practically as efficient as the fastest high-accuracy dependency parsers.
- Despite being constrained to pre-selected dependency decisions, the parser is comparably accurate to non-reranked phrase-structure parsers.
- The framework can be used...

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 look at a more stripped-down problem. We give up entirely on de-

termining the structure of the tree itself and only look at producing the nonterminals of the tree.

2 Preliminaries

We begin by introducing notation for two standard grammatical structures: lexicalized context-free parses and dependency parses. The notation emphasizes the similarity between the two formalisms.

2.1 Lexicalized CFG Parsing

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

- $\mathcal{N}$; a set of nonterminal symbols, e.g. `NP`, `VP`.
- $\mathcal{T}$; a set of terminal symbols, often consisting of the words in the language.
- $\mathcal{R}$; a set of lexicalized rule productions of the form $\langle A \rightarrow \beta_1 \dots \beta_m, k \rangle$ 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_k.
- $root$; a distinguished root symbol $root \in \mathcal{N}$.

For a given input sequence $x_1, \dots, x_n$ consisting of terminal symbols from $\mathcal{T}$, define $\mathcal{Y}(x)$ as the set of valid lexicalized parses for the sentence. Informally this set consists of all ordered trees with fringe x, internal nodes labeled from $\mathcal{N}$, all tree productions $A \rightarrow \beta$ consisting of members of $\mathcal{R}$, and root label $root$.

For a given parse $y \in \mathcal{Y}(x)$, we further associate a triple $(\langle i, j \rangle, h, A)$ with each vertex in the tree, where

- $\langle i, j \rangle$; the *span* of the vertex, i.e. the contiguous sequence $\{x_i, \dots, x_j\}$ of the original input covered by the vertex.
- $h \in \{0, \dots n\}$; index indicating that x_h is the *head* of the vertex, defined recursively by the following rules:
  1. If the vertex is leaf x_i, then $h = i$.
  2. 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 this vertex.

  Let $h(v)$ indicate the head of vertex v.
- $A \in \mathcal{T} \cup \mathcal{N}$; the terminal or nonterminal symbol of the vertex.

In a parse, each word x_i begins as a head at the fringe of the tree, and at some ancestor vertex ceases to play this role. To capture this notion, define the *spine* of index i to be the longest chain of vertices $v_1 \dots v_p$, where v_1 is at the fringe and for all $j \in \{2, \dots, p\}$, v_j is the parent of v_{j-1} and $h(v_j) = i$. Define the vertex v_{p+1} to be the parent of the top vertex v_p, where $h(v_{p+1}) \neq i$.

2.2 Binarization

Any LCFG is weakly-equivalent to a binarized LCFG where all rules are either binary or unary. In this work we use the following binarization.

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

$\vdots$

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 $i > c$

From here on we assume that all tree productions are binary. The benefit of using a binarized LCFG is that the set of tree $\mathcal{Y}(x)$ can be compactly enumerated using the inductive productions of lexicalized CKY. These production are shown in Figure **??**. The main production used is

$$\frac{(\langle i, k \rangle, m, B) \quad (\langle k+1, j \rangle, h, C)}{(\langle i, j \rangle, h, A)}$$

for $A \in \mathcal{N}, B, C \in \mathcal{N} \cup \mathcal{T}, i < k < j$. This rule indicates that rule $\langle A \rightarrow B \; C, 1 \rangle$ was applied at a vertex covering $\langle i, j \rangle$ to produce two veritices covering $\langle i, k \rangle$ and $\langle k+1, j \rangle$.

We abbreviate this decision as a tuple $(\langle i, k, j \rangle, h, m, r)$, and for $y \in \mathcal{Y}$, define y as an indicator vector over decisions i.e. $(\langle i, k, j \rangle, h, m, r) = 1$ to indicate that the rule above was used.

The lexicalized parsing problem is to find the highest-scoring parse in this set, i.e.

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

where $s : \mathcal{Y} \rightarrow \mathbb{R}$ is a linear scoring function that scores each production.

This combinatorial optimization problem can be solved by bottom-up dynamic programming over the set of rule productions shown in figure **??**. Since there are five free indices i, j, k, h, m and $|\mathcal{R}|$ possible grammartical, in the worst-case this algorithm $O(n^5\mathcal{R})$ running time.

2.3 Dependency Parsing

Dependency trees provide an alternative, and in some sense simpler, representation of grammatical structure.

Given an input sentence $x_1 \dots x_n$, we allow each word to modify another word, and define these dependency decisions as a sequence $d_1 \dots d_n$ where for all i, $d_i \in \{0, \dots, n\}$ and 0 is a pseudo-root position. These dependency relations can be seen as arcs (d_i, i) in a directed graph. (as shown in Figure **??**). A dependency parse is valid if the corresponding directed graph is a directed tree rooted at vertex 0.

Given a valid dependency parse, we can define the span of any word m as the set of indices reachable from vertex m in the directed tree. A dependency parse is *projective* if the descendants of every word in the tree form a contiguous span of the original sentence (). For we use the notation $m_{\Leftarrow}$ and $m_{\Rightarrow}$ to represent the left- and right-boundaries of this span.

The highest-scoring projective dependency parse under an arc-factored scoring function can be found in time $O(n^3)$.

We now give the important property for this work.

Lemma 2.1 *Any lexicalized context-free parse, as defined above, can be converted deterministically into a projective dependency tree.*

Conversion.

For an input symbol x_m with spine $v_1, \dots, v_p$,

1. If v_p is the root of the tree, then $d_m = 0$.
2. 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.

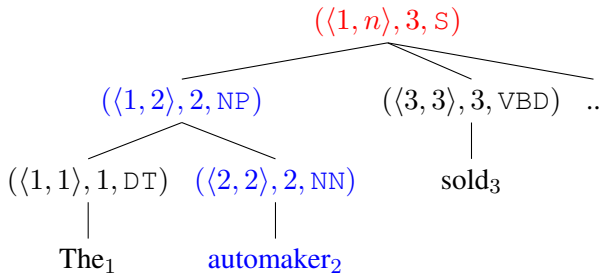

Figure 1: Figure illustrating a LCFG parse. The parse is an ordered tree with fringe $x_1, \dots, x_n$. Each vertex is annotated with a span, head, and syntactic tag. The blue vertices represent the 3-vertex spine v_1, v_2, v_3 of the word $\texttt{automaker}_2$. The root vertex is v_4, which implies that $\texttt{automaker}_2$ modifies $\texttt{sold}_3$ in the induced dependency graph.

The conversion produces a directed tree rooted at 0, by preserving the tree structure of the original LCFG parse.

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.

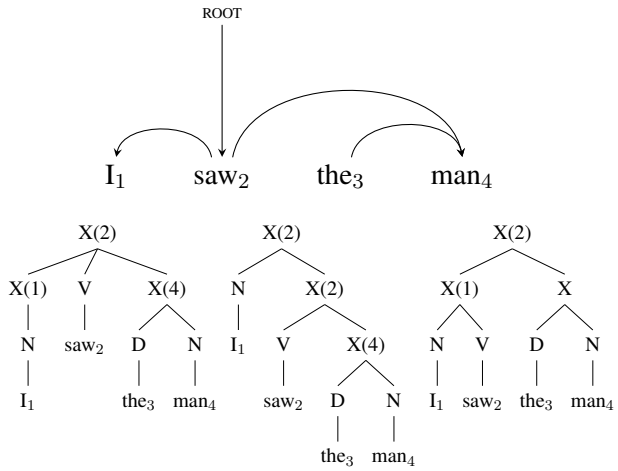

Figure 2: While an LCFG parse determines a unique dependency parse, the inverse problem is non-deterministic. The figure, adapted from (Collins et al., 1999), shows several LCFG parse structures that all correspond to the dependency structure shown.

4 Parsing Dependencies

To solve this inverse problem we set up as simple variant of the original lexicalized parsing algorithm. While this algorithm has the same form of the original, we show that it can be solved asymptotically faster.

4.1 Algorithm

Recall that $\mathcal{Y}(x)$ is the set of valid LCFG parses for sentence x. We now assume that we now additionally are given a projective dependency parse $d_1, \ldots, d_n$. Define the set $\mathcal{Y}(x, d)$ as LCFG parses that induce this dependency structure, i.e.

$$\begin{aligned}\mathcal{Y}(x,d) \quad = \quad & \{y \in \mathcal{Y} : \\ & \text{for all } d_m = h \text{ there exists } i,j,k,r \\ & \text{s.t. } y(i,j,k,h,m,r) = 1\}\end{aligned}$$

and our aim is to find

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

We make the following simple observation about this new problem. For word m with spine $v_1 \ldots v_p$ the span $\langle i, j \rangle$ of v_p is equal to $\langle m_{\Leftarrow}, m_{\Rightarrow} \rangle$. These spans can be computed directly from d.

Premise:

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

Rules:

For $i \leq m \leq j < h \leq k$, and rule $\langle A \rightarrow B\ C, 1 \rangle$,

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

For $i \leq h \leq j < m \leq k$, rule $\langle A \rightarrow B\ C, 2 \rangle$,

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

Goal:

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

Figure 3: Standard CKY algorithm for LCFG parsing stated as inductive rules. Starting from the *premise*, any valid application of *rules* that leads to a *goal* is a valid parse. Finding the optimal parse with dynamic programming is linear in the number of rules. For this algorithm there are $O(n^5|\mathcal{R}|)$ rules where n is the length of the sentence.

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.

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.

4.2 Extension: Labels

Finish this section

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 $A \rightarrow B\ C$.

5 Structured Prediction

We model this problem using a standard structured prediction setup.

Premise:

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

Rules:
For all $i < m, h = d_m$ and rule $\langle A \rightarrow B\ C, 1 \rangle$,

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

For all $m < j, h = d_m$ and rule $\langle A \rightarrow B\ C, 2 \rangle$,

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

Goal:

$$(\langle 1, n \rangle, m, root) \text{ for any } m \text{ s.t. } d_m = 0$$

Figure 4: The constrained CKY parsing algorithm for $\mathcal{Y}(x, d)$. The algorithm is nearly identical to Figure **??** except that many of the free indices are now fixed to the dependency parse. Finding the optimal parse is now $O(n^2|\mathcal{R}|)$
.

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

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

where $\theta \in \mathbb{R}^d$ is a learned weight vector and $f(x, d, y)$ is a linear feature function that maps parse production to feature vectors. In this section we discuss the feature function f and estimating the parameter vector θ.

5.1 Features

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

The first set of features look at the rule structure $A \rightarrow B\ C$. We have features on several combinations of rule and non-terminals.

The next set of features look at properties of the lexicalization. These look at combinations of the head word and tags.

The final set of features looks at more structural properties.

For a part $(i, j, k, h, m, A \rightarrow B\ C)$

Nonterm Features
(A, B)
(A, C)
$(A, \text{tag}(h), \text{tag}(m))$
$(A, B, \text{tag}(m))$
$(A, C, \text{tag}(h))$
$(A, \text{tag}(h))$
$(A, \text{word}(h))$
Rule Features
$(A \rightarrow B\ C)$
$(A \rightarrow B\ C, \text{tag}(h), \text{tag}(m))$
$(A \rightarrow B\ C, \text{word}(h), \text{tag}(m))$
$(A \rightarrow B\ C, \text{word}(h))$
$(A \rightarrow B\ C, \text{tag}(h))$
$(A \rightarrow B\ C, \text{word}(m))$
$(A \rightarrow B\ C, \text{tag}(m))$
shape

Figure 5: The templates used in the structured model. The functions $\text{tag}(()i)$ and $\text{word}(()i)$ give the part-of-speech tag and word at position i.

bug lp about this.

() () ()

5.2 Training

We train the parameters θ using a variant of standard structured SVM training ().

We assume that we are given a set of gold-annotated parse examples: $(x^1, y^1), \dots, (x^D, y^D)$. We also define $d^{(1)} \dots d^{(D)}$ as the dependency structures induced from $y^1 \dots y^D$

Our aim to find parameters that satisfy the following regularized risk minimization

$$\min_\theta \sum_{i=1}^{D} \max\{0, \ell(x^i, d^i, y^i, \theta)\} + \frac{\lambda}{2}||\theta||_2^2$$

and define

$$\ell(x, d, y, \theta) = s(y) + \max_{y' \in \mathcal{Y}(x,d)} \left(s(y') + \Delta(y, y')\right)$$

where Δ is a problem specific cost-function that we assume is linear in either arguments.

We consider two extensions to the standard setup.

First is that at training time we do not have access to the true dependency structure d, but instead only predicted dependencies $\hat{d}$. Past work on parsing () has shown that it can be important to train using these predicted structures.

However, using $\hat{d}$ may make it impossible to recover the gold LCFG parse y, e.g. $y \notin \mathcal{Y}(x, \hat{d})$. To handle this issue, we also replace y with the oracle parse $\bar{y}$ defined at the projection of y onto $\mathcal{Y}(x, \hat{d})$

$$\bar{y} \leftarrow \arg\min_{y' \in \mathcal{Y}(x,\hat{d})} \Delta(y', y)$$

If Δ is linear in its first argument, this projection can be found by running the same algorithm as in Figure **??**.

We replace the loss with $\ell(x^{(i)}, \hat{d}^{(i)}, \bar{y}^{(i)})$

In our experiments, we use a simple hamming loss $\Delta(y, \bar{y}) = ||y - \bar{y}||$.

At training time, we run 10-fold jack-knifing to produce dependency parses $\hat{d}$. We then run a single pass to calculate $\bar{y}$ for each training example.

We then optimize using stochastic gradient descent (). The gradient requires calculating a loss-augmented argmax for each training example.

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.

6.4 Extension: Pruning

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

Model	fscore	speed
TurboParser		
MaltParser		
EasyFirst		

Table 2: This

6.5

7 Results

7.1

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

References

Michael Collins, Lance Ramshaw, Jan Hajič, and Christoph Tillmann. 1999. A statistical parser for czech. In *Proceedings of the 37th annual meeting of the Association for Computational Linguistics on Computational Linguistics*, pages 505–512. Association for Computational Linguistics.

Lingpeng Kong and Noah A Smith. 2014. An empirical comparison of parsing methods for stanford dependencies. *arXiv preprint arXiv:1404.4314*.

Parsing Results		
	wsj	
	speed	fscore
Petrov		
Carreras		
	ctb	

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