Parsing Dependencies

Abstract

1 Introduction

There are two dominant grammatical frameworks used for statistical syntactic parsing: phrase-structure and dependency parsing (). The two 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?

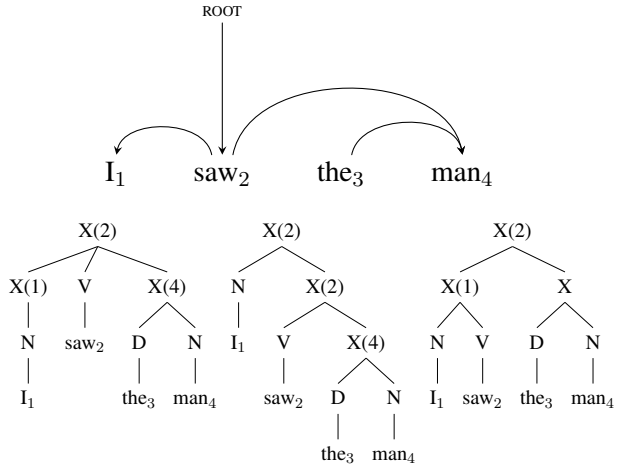

Figure 1: While an LCFG parse determines a unique dependency parse, the inverse problem is non-deterministic. The figure, adapted from (Collins et al., 1999), shows several LCFG parse structures that all correspond to the dependency structure shown. The parentheses $X(h)$ indicate the head h of each internal vertex.

This conversion is challenging in two ways: (1) 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. (2) 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 1.

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

name?

To approach this question we present a very simple statistical parser. 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. (Carreras et al., 2008) build a powerful dependency parsing-like model that predicts adjunctions of the elementary TAG spines. (Rush et al., 2010) 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. (Zhu et al., 2013) 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 determining the structure of the tree itself and only look at producing the nonterminals of the tree.

2 Background

We begin by developing notation for a lexicalized phrase-structure formalism and for dependency parsing. The notation aims to highlight the similarity between the two formalisms.

2.1 Lexicalized CFG Parsing

A binarized, lexicalized context-free grammar (LCFG) is an extension to a standard context-free grammar where each vertex also is annotated with a lexical head. Define an 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 either of the form $A \rightarrow \beta_1^* \ \beta_2$ or $A \rightarrow \beta_1 \ \beta_2^*$ consisting of a parent nonterminal $A \in \mathcal{N}$, a sequence of children $\beta_i \in \mathcal{N}$ for $i \in \{1 \dots 2\}$, and a distinguished head child annotated with $*$. The head child comes from the head rules associated with the grammar.
- r; a distinguished root symbol $r \in \mathcal{N}$.

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

For a parse of a sentence $y \in \mathcal{Y}(x)$, we further associate a triple $v = (\langle i, j \rangle, h, A)$ 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 sentence covered by the vertex.
- $h(v) \in \{1, \dots n\}$; 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 matches the head of the $*$'th child of the vertex where $A \rightarrow \beta$ is the rule production at this vertex.
- $A \in \mathcal{T} \cup \mathcal{N}$; the terminal or nonterminal symbol of the vertex.

Note that each word x_i is the head of a fringe vertex of the tree, and at some ancestor vertex ceases to play this role. Define the *spine* of word x_i to be the longest chain connected vertices $v_1, \dots, v_p$ where $h(v_j) = i$ for $j \in \{1, \dots, p\}$. Also if it exists, let vertex v_{p+1} be the parent of vertex v_p, where $h(v_{p+1}) \neq i$. The full notation is illustrated in Figure 2.

2.2 Dependency Parsing

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

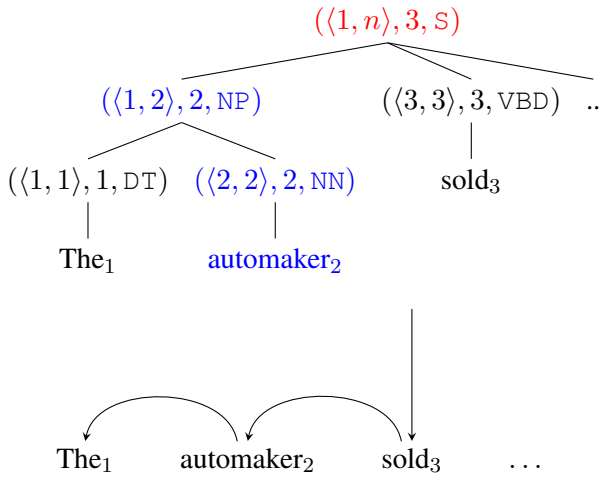

Figure 2: Figure illustrating an LCFG parse. The parse is an ordered tree with fringe $x_1, \ldots, x_n$. Each vertex is annotated with a span, head, and syntactic tag. The blue vertices represent the 3-vertex spine v_1, v_2, v_3 of the word $\texttt{automaker}_2$. The root vertex is v_4, which implies that $\texttt{automaker}_2$ modifies $\texttt{sold}_3$ in the induced dependency graph.

Given an input sentence $x_1 \ldots x_n$, we allow each word to modify another word, and define these dependency decisions as a sequence $d_1 \ldots d_n$ where for all i, $d_i \in \{0, \ldots, n\}$ 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 2). A dependency parse is valid if the corresponding directed graph is a directed tree rooted at vertex 0.

For a valid dependency parse, define the span of any word x_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.

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

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

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 $d_m = h(v_{p+1})$. 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.

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

3 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.

3.1 Constrained Lexicalized CKY

We will assume in this section that the LCFG is in lexicalized Chomsky normal form, with all non-preterminal rules of the form $A \rightarrow \beta_1\ \beta_2$. In the next section we show how to convert our original grammar to this form while preserving the head structure.

A standard parsing algorithm for a grammar in this form is the lexicalized variant CKY algorithm. This algorithm can be represented as a collection of inductive rule productions as shown in Figure **??**. These rules compactly encode the set of all valid parses for a given sentence under the grammar. For example consider the rule

$$\frac{(\langle i, k \rangle, m, \beta_1) \quad (\langle k+1, j \rangle, h, \beta_2)}{(\langle i, j \rangle, h, A)}$$

for all rules $A \rightarrow \beta_1^*\ \beta_2 \in \mathcal{R}$ and spans $i \leq k < j$. This rule indicates that grammatical rule $A \rightarrow \beta_1^*\ \beta_2$ was applied at a vertex covering $\langle i, j \rangle$ to produce two vertices covering $\langle i, k \rangle$ and $\langle k+1, j \rangle$.

Define this complete set of valid parses for a sentence as $\mathcal{Y}(x)$. The lexicalized parsing problem is to find the highest-scoring parse in this set, i.e.

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

where s is a scoring function that factors over each logical production.

The highest-scoring parser can found by bottom-up dynamic programming over this set. The running time of this algorithm is linear in the number of rules. Since five free indices i, j, k, h, m and $|\mathcal{R}|$

possible grammatical rules, the algorithm requires $O(n^5|\mathcal{R}|)$ time.

This running-time makes this algorithm highly impractical to run on real data without heavy pruning.

However we are interested in a highly-constrained version of this problem. Assume that we are additionally given a projective dependency parse $d_1, \ldots, d_n$. Define the set $\mathcal{Y}(x, d)$ as all valid LCFG parses that match this dependency parse

$$\mathcal{Y}(x, d) = \{y \in \mathcal{Y}(x) : \text{for all } (h, m) \in y, d_m = h\}$$

and our aim is to find

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

Now consider the following simple property of this new problem. For any word x_m with spine $v_1 \ldots v_p$ the LCFG span $\langle i, j \rangle$ of v_p is equal to the dependency span $\langle m_{\Leftarrow}, m_{\Rightarrow} \rangle$ of x_m. Furthermore these spans can be easily computed directly from d.

Using this property, we can greatly limit the search space of the original problem. Instead of searching over all possible spans $\langle i, j \rangle$ of each modifier, we can precompute $\langle m_{\Leftarrow}, m_{\Rightarrow} \rangle$.

Figure 3.1 shows the new inductive rules which are virtually identical to the original lexicalized algorithm. 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.

3.2 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 $A \rightarrow \beta_1\ \beta_2$.

3.3 Binarization

While the algorithm itself is not dependent on the LCFG binarization method used, the choice of binarization effects the run-time of the algorithm, through $\mathcal{R}$, as well as the modeling accuracy, through the factored scoring function s.

Premise:

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

Rules:

For all $i < m, h = d_m$ and rule $A \rightarrow \beta_1^*\ \beta_2$,

$$\frac{(\langle i, m_{\Leftarrow} - 1 \rangle, h, \beta_1) \quad (\langle m_{\Leftarrow}, m_{\Rightarrow} \rangle, m, \beta_2)}{(\langle i, m_{\Rightarrow} \rangle, h, A)}$$

For all $m < j, h = d_m$ and rule $A \rightarrow \beta_1\ \beta_2^*$,

$$\frac{(\langle m_{\Leftarrow}, m_{\Rightarrow} \rangle, m, \beta_1) \quad (\langle m_{\Rightarrow} + 1, j \rangle, h, \beta_2)}{(\langle m_{\Leftarrow}, j \rangle, h, A)}$$

Goal:

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

Figure 3: 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 $O(n^2|\mathcal{R}|)$

.

Since the parser traces a fixed dependency structure we select a binarization based around the head structure.

For simplicity, we consider binarizing rule $\langle A \rightarrow \beta_1 \dots \beta_m, k \rangle$ with $m > 2$. Relative to the head β_k the rule has left-side $\beta_1 \dots \beta_{k-1}$ and right-side $\beta_{k+1} \dots \beta_m$.

We replace this rule with binary rules that consume each side independently as a first-order Markov chain (horizontal Markovization). The main transformation is to introduce rules

- $A^{\Rightarrow}_{\beta_i} \rightarrow A^{\Rightarrow}_{\beta_{i-1}}{}^{*}\ \beta_i$ for $k > i > m$
- $A^{\Leftarrow}_{\beta_i} \rightarrow \beta_i\ A^{\Leftarrow}_{\beta_{i+1}}{}^{*}$ for $1 < i < k$

Additionally we introduce several additional rules to handle the boundary cases of starting a new rule, finishing the right side, and completing a rule. (These rules are slightly modified when $k = 1$ or $k = m$) .

- $A^{\Rightarrow}_{\beta_{k+1}} \rightarrow \beta^{*}_{k}\ \beta_{k+1}$
- $A^{\Rightarrow}_{\text{END}} \rightarrow A^{\Rightarrow}_{\beta_{m-1}}{}^{*}\ \beta_m$
- $A^{\Leftarrow}_{\beta_{k-1}} \rightarrow \beta_{k-1}\ A^{\Rightarrow}_{\text{END}}{}^{*}$
- $A \rightarrow \beta_1\ A^{\Leftarrow *}_{\beta_2}$

Each rule contains at most 3 nonterminals so the size of the new binarized rule set is bounded by $O(\mathcal{N}^3)$.

4 Structured Prediction

To learn the transformation from dependency trees to phrase-structure trees, we use a standard structured prediction setup. The goal is to predict the highest-scoring LCFG parse for sentence and dependency tree. 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 that will be trained and $f(x, d, y)$ is a feature function that maps parse production (as in Figure **??**) to feature vectors in $\{0, 1\}^D$. In this section we first discuss the features used and then how to estimate the parameter vector θ.

4.1 Features

We experimented with several settings of the part features. The final features are shown in Figure 4.1.

For a part $(i, j, k, h, m, A \rightarrow \beta_1\ \beta_2)$

Nonterm Features
(A, β_1)
(A, β_2)
$(A, \text{tag}(h), \text{tag}(m))$
$(A, B, \text{tag}(m))$
$(A, C, \text{tag}(h))$
$(A, \text{tag}(h))$
$(A, \text{word}(h))$
Rule Features
$(A \rightarrow \beta_1\ \beta_2)$
$(A \rightarrow \beta_1\ \beta_2, \text{tag}(h), \text{tag}(m))$
$(A \rightarrow \beta_1\ \beta_2, \text{word}(h), \text{tag}(m))$
$(A \rightarrow \beta_1\ \beta_2, \text{word}(h))$
$(A \rightarrow \beta_1\ \beta_2, \text{tag}(h))$
$(A \rightarrow \beta_1\ \beta_2, \text{word}(m))$
$(A \rightarrow \beta_1\ \beta_2, \text{tag}(m))$
shape

Figure 4: 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.

The first set of features look at the rule structure $A \rightarrow \beta_1\ \beta_2$. 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.

bug lp about this.

() () ()

4.2 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

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

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.

5 Data and Setup

5.1 Data

We used wsj..

We used ctb 5-1..

5.2 Implementation

We built...

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

5.3 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.

Model	fscore	speed
TURBOPARSER		
MALTPARSER		
EASYFIRST		

Table 2: This

5.4 Extension: Pruning

We also experiment with a simple pruning dictionary pruning technique. For each context-free rule $A \rightarrow \beta_1\ \beta_2$ and POS tag a we remove all rules that were not seen with that tag as a head in training.

5.5

6 Results

6.1

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

References

Xavier Carreras, Michael Collins, and Terry Koo. 2008. Tag, dynamic programming, and the perceptron for efficient, feature-rich parsing. In *Proceedings of the Twelfth Conference on Computational Natural Language Learning*, pages 9–16. Association for Computational Linguistics.

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*.

Alexander M Rush, David Sontag, Michael Collins, and Tommi Jaakkola. 2010. On dual decomposition and linear programming relaxations for natural language processing. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, pages 1–11. Association for Computational Linguistics.

Muhua Zhu, Yue Zhang, Wenliang Chen, Min Zhang, and Jingbo Zhu. 2013. Fast and accurate shift-reduce constituent parsing. In *ACL (1)*, pages 434–443.

Parsing Results		
	wsj	
	speed	fscore
Petrov		
Carreras		
	ctb	

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