An Empirical Comparison of Parsing Methods for Stanford Dependencies

Lingpeng Kong Noah A. Smith
Language Technologies Institute
Carnegie Mellon University
Pittsburgh, PA 15213, USA
{lingpenk,nasmith}@cs.cmu.edu

Abstract

Stanford typed dependencies are a widely desired representation of natural language sentences, but parsing is one of the major computational bottlenecks in text analysis systems. In light of the evolving definition of the Stanford dependencies and developments in statistical dependency parsing algorithms, this paper revisits the question of Cer et al. (2010): what is the tradeoff between accuracy and speed in obtaining Stanford dependencies in particular? We also explore the effects of input representations on this tradeoff: part-of-speech tags, the novel use of an alternative dependency representation as input, and distributional representaions of words. We find that direct dependency parsing is a more viable solution than it was found to be in the past. An accompanying software release can be found at: http://www.ark.cs.cmu.edu/TBSD

1 Introduction

The Stanford typed dependency (SD) representations of grammatical relations, introduced by de Marneffe and Manning (2008), have become a popular text analysis scheme for a variety of NLP applications such as event extraction (Björne et al., 2009), error correction (Tetreault et al., 2010), and machine translation (Stein et al., 2010). Relative to other dependency representations, such as those of Yamada and Matsumoto (YM; 2003), SD parses emphasize semantic relations (e.g., relative clauses are rooted in the verb rather than the complementizer, prepositional phrases in the nominal rather than the preposition). This may contribute to their attractiveness in downstream applications (Elming et al., 2013), but it also makes SD parsing more challenging than YM parsing.

Broadly speaking, there are two kinds of approaches to obtaining dependencies. One, which we call **c-parsing**, applies phrase-structure parsing algorithms to obtain constituents, then extracts dependencies by applying expert-crafted head rules and perhaps other transformations. This is the dominant approach for SD parsing; indeed, the rules[1] are considered definitive for the representation, and they are updated frequently.

The second approach, which we call **d-parsing**, applies dependency parsing algorithms, directly tackling the grammatical relations without representing constituents. These parsers tend to be faster, and for YM dependencies they achieve the best performance: Martins et al. (2013) report 93.1% unlabeled attachment score on PTB §23, while a state-of-the-art phrase-structure parser (Charniak and Johnson, 2005;

[1] http://nlp.stanford.edu/software/corenlp.shtml

McClosky et al., 2006) achieves 92.5%. Several recent dependency parsers (Rush and Petrov, 2012; Choi and McCallum, 2013; Zhang et al., 2013) further improve the speed of this approach while preserving accuracy.

The main contribution of this paper is an empirical comparison of a wide range of different ways to obtain SD parses. It follows on an important study by Cer et al. (2010), which found a 6.9% absolute unlabeled (8% absolute labeled) point gap in F_1 between c-parsing with the best available phrase-structure parser and d-parsing with the best available dependency parser in SD parsing for CCPROCESSED dependencies (the most linguistically rich representation). Çetinoğlu et al. (2010) explored similar strategies for parsing into lexical-functional grammar representations without phrase-structure parsing.

Since those studies, dependency parsing has advanced, and the definition of SDs has evolved considerably, so it is worth revisiting the viability of d-parsing for Stanford typed dependencies. For Chinese, Che et al. (2012) found greater success with a second-order dependency parser, Mate (Bohnet, 2010).

This paper's contributions are:

- We quantify the current tradeoff between accuracy and speed in SD parsing, notably closing the gap between c-parsing and d-parsing to 1.8% absolute unlabeled (2.0% absolute labeled) F_1 points (§3) for CCPROCESSED SD parsing. The current gap is 30% (25%) the size of the one found by Cer et al. (2010). An arc-factored d-parser is shown to perform a bit better than the Stanford CoreNLP pipeline, at twenty times the speed.
- We quantify the effect of part-of-speech tagging on SD parsing performance, isolating POS errors as a major cause of that gap (§4).
- We demonstrate the usefulness of the YM representation as a source of information for SD parsing, in a stacking framework (§5).
- Noting recently attested benefits of distributional word representations in parsing (Koo et al., 2008), we find that d-parsing augmented with Brown cluster features performs similarly to c-parsing with the Stanford recursive neural network parser (Socher et al., 2013), at three times the speed.

2 Background and Methods

A Stanford dependency graph consists of a set of ordered dependency tuples $\langle T, P, C\rangle$, where T is the type of the dependency and P and C are parent and child word tokens, respectively. These graphs were designed to be generated from the phrase-structure tree of a sentence (de Marneffe et al., 2006). This transformation happens in several stages. First, head rules are used to extract parent-child pairs from a phrase-structure parse. Second, each dependency is labeled with a grammatical relation type, using the most specific matching pattern from an expert-crafted set.

There are several SD conventions. The simplest, BASIC SD graphs, are always trees. Additional rules can be applied to a phrase-structure tree to identify EXTRA depenencies (e.g., *ref* arcs attaching a relativizer like *which* to the head of the NP modified by a relative clause), and then to collapse dependencies involving transitions and propagate conjunct dependencies, giving the richest convention, CCPROCESSED. In this paper we measure performance first on BASIC dependencies; in §3.2 we show that the quality of CCPROCESSED dependencies tends to improve as BASIC dependencies improve.

The procedures for c-parsing and d-parsing are well-established (Cer et al., 2010); we briefly review them. In c-parsing, a phrase-structure parser is applied, after which the Stanford CoreNLP rules are applied to obtain the SD graph. In this work, we use the latest version available at this writing, which is version 3.3.0. In d-parsing, a statistical dependency parsing model is applied to the sentence; these models are

trained on Penn Treebank trees (§02–21) transformed into BASIC dependency trees using the Stanford rules. To obtain CCPROCESSED graphs, EXTRA dependencies must be added using rules, then the collapsing and propagation transformations must be applied.

One important change in the Stanford dependencies since Cer et al. (2010) conducted their study is the introduction of rules to infer EXTRA dependencies from the phrase-structure tree. (Cer et al. used version 1.6.2; we use 3.3.0.) We found that, given *perfect* BASIC dependencies (but no phrase-structure tree), the inability to apply such inference rules accounts for a 0.6% absolute gap in unlabeled F_1 (0.5% labeled) between c-parsing and d-parsing for CCPROCESSED dependencies (version 1.6.2).[2]

3 Current Tradeoffs

We measure the performance of different c-parsing and d-parsing methods in terms of unlabeled and labeled attachment score (UAS and LAS, respectively) on Penn Treebank §22 and §23. We report parsing speeds on a Lenovo ThinkCentre desktop computer with Core i7-3770 3.4GHz 8M cache CPU and 32GB memory. All parsers were trained using Penn Treebank §02–21. We target version 3.3.0 of SDs (released November 12, 2013), and, where Stanford CoreNLP components are used, they are the same version.

We consider three c-parsing methods:

1. The Stanford "englishPCFG" parser, version 3.3.0 (Klein and Manning, 2003), which we believe is the most widely used pipeline for SD parsing. This model uses additional non-WSJ training data for their English parsing model.[3]

2. The Stanford "RNN" parser, version 3.3.0 (Socher et al., 2013), which combines PCFGs with a syntactically untied recursive neural network that learns syntactic/semantic compositional vector representations. Note this model uses distributional representations from external corpus; see section 5.1.

3. The Berkeley "Aug10(eng_sm6.gr)" parser, version 1.7 (Petrov et al., 2006).

4. Charniak and Johnson's "June06(CJ)" parser (Charniak and Johnson, 2005; McClosky et al., 2006). Note this is the self-trained model which uses 2 million unlabeled sentences from the North American News Text corpus, NANC (Graff, 1995). It is therefore technically semi-supervised.

Each of these parsers performs its own POS tagging. Runtime measurements for these parsers include POS tagging and also conversion to SD graphs.

We consider eight d-parsing methods:

4. MaltParser liblinear stackproj (Nivre et al., 2006) a transition-based dependency parser that uses the Stack-Projective algorithm. The transitions are essentially the same as in the "arc-standard" version of Nivre's algorithm and produce only projective dependency trees (Nivre, 2009; Nivre et al., 2009). In learning, it uses the LIBLINEAR package implemented by Fan et al. (2008). This is the same setting as the most popular pre-trained model provided by MaltParser.

[2] In version 3.3.0, inference rules have been added to the Stanford CoreNLP package to convert from BASIC to CCPROCESSED without a phrase-structure tree. Given perfect BASIC dependencies, there is still a 0.2% unlabeled (0.3% labeled) gap in F_1 in PTB §22 (0.4% and 0.5% for §23). We added some new rules to help close this gap by about 0.1 F_1 (unlabeled and labeled), but more can be done. The new rules are not fine-tuned to §22–23; they are given in Appendix A.

[3] See the Stanford Parser FAQ at `http://nlp.stanford.edu/software/parser-faq.shtml`.

5. MaltParser libsvm arc-eager (Nivre et al., 2006), a transition-based dependency parser that uses the "arc-eager" algorithm (Nivre, 2004). In learning, it uses LIBSVM implemented by Chang and Lin (2011). This is the default setting for the MaltParser.

6. MSTParser, a second-order "graph based" (i.e., global score optimizing) parser (McDonald et al., 2005; McDonald and Pereira, 2006).

7. Basic TurboParser (Martins et al., 2010), which is a first-order (arc-factored) model similar to the minimium spanning tree parser of McDonald et al. (2005).

8. Standard TurboParser (Martins et al., 2011), a second-order model that scores consecutive siblings and grandparents (McDonald and Pereira, 2006).

9. Full TurboParser (Martins et al., 2013), which adds grand-sibling and tri-sibling (third-order) features as proposed by Koo and Collins (2010) and implemented by Martins et al. (2013).

10. EasyFirst (Goldberg and Elhadad, 2010), a non-directional dependency parser which builds a dependency tree by iteratively selecting the best pair of neighbors to connect.[4]

11. Huang's linear-time parser (Huang and Sagae, 2010; Huang et al., 2012), a shift-reduce parser that applies a polynomial-time dynamic programming algorithm that achieves linear runtime in practice.[5]

POS tags for dependency parsers were produced using version 2.0 of the Stanford POS Tagger (MEMM tagging model "left3words-wsj-0-18"; Toutanova et al., 2003); this is identical to Cer et al. (2010). POS tagging time and rules to transform into CCPROCESSED graphs, where applied, are included in the runtime.

Our comparison includes most of the parsers explored by Cer et al. (2010), and all of the top-performing ones. They found the Charniak-Johnson parser to be more than one point ahead of the second best (Berkeley). MaltParser was the best among d-parsing alternatives considered.

3.1 BASIC Dependencies

Table 1 presents our results on BASIC dependencies. The most *accurate* approach is still to use the Charniak-Johnson parser (4), though Full TurboParser (10) is the best among d-parsing techniques, lagging Charniak-Johnson by 2–3 absolute points and with about twice the speed. If the Stanford englishPCFG model provides adequate accuracy for a downstream application, then we advise using MSTParser or any variant of TurboParser instead. In particular, without sacrificing the Stanford englishPCFG's level of performance, Basic TurboParser runs nearly 20 times faster.

Figure 1 plots the tradeoff between speed and accuracy for most of the approaches. For clarity, we exclude parsers at the extremely fast and slow ends (all with accuracy around the same or slightly below Stanford englishPCFG at the lower left of the plot).

3.2 CCPROCESSED Dependencies

Table 2 shows performance scores for CCPROCESSED dependencies extracted from the parses in §3.1. Because the number of attachments in a parser's CCPROCESSED output may not equal the number in the

[4]EasyFirst can only be trained to produce unlabeled dependencies. It provides a labeler for SD version 1.6.5, but it cannot be retrained. We therefore only report UAS for EasyFirst.

[5]Huang's parser only produces unlabeled dependencies, so we only report UAS.

	Type	Parser	PTB §22		PTB §23		Speed
			UAS	LAS	UAS	LAS	(tokens/s.)
1	c-parsing	Stanford englishPCFG	‡90.06	‡87.17	‡90.09	‡87.48	123.63
2		Stanford RNN	‡93.11	‡90.16	‡92.80	‡91.10	66.57
3		Berkeley	93.33	90.64	93.31	91.01	200.00
4		Charniak-Johnson	‡ **93.91**	‡ **91.25**	‡ **94.38**	‡ **92.07**	100.72
5	d-parsing	MaltParser (liblinear stackproj)	88.93	86.23	88.47	85.90	8799.91
6		MaltParser (libsvm arc-eager)	89.35	86.61	89.20	86.46	8.81
7		MSTParser	91.24	87.83	90.87	87.53	239.60
8		Basic TurboParser	90.25	87.93	90.12	87.89	2419.98
9		Standard TurboParser	92.16	89.50	91.95	89.40	413.67
10		Full TurboParser	**92.29**	**89.64**	**92.20**	**89.67**	209.98
11		EasyFirst	89.80	–	89.28	–	*2616.19
12		Huang	†90.90	–	†90.70	–	*616.55

Table 1: BASIC SD parsing performance and runtime. *EasyFirst and Huang do not include dependency labeling in the runtime. †When training the Huang parser, we provide §23 (§22) as development data when testing on §22 (§23), which gives a slight advantage. ‡Charniak-Johnson uses semi-supervised training; Stanford englishPCFG and RNN models use external resources; see text.

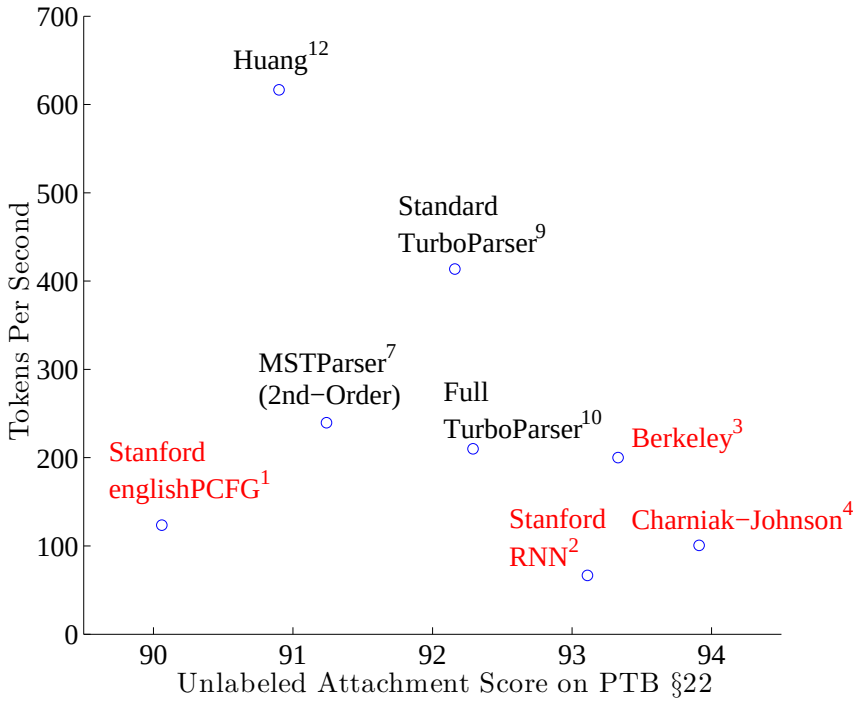

Figure 1: Speed-accuracy tradeoff. Huang, MSTParser, and all the three versions of TurboParser give better accuracy and speed than the "default" Stanford CoreNLP pipeline.

	Type	Parser	PTB §22		PTB §23	
			U. F_1	L. F_1	U. F_1	L. F_1
1	c-parsing	Stanford englishPCFG	†87.7	†84.6	†87.9	†85.0
2		Stanford RNN	†91.3	†88.1	†91.0	†88.1
3		Berkeley	91.4	88.5	91.4	88.9
4		Charniak-Johnson	†**92.2**	†**89.4**	†**92.7**	†**90.3**
5	d-parsing	MaltParser (liblinear stackproj)	86.3	83.2	85.4	82.5
6		MaltParser (libsvm arc-eager)	86.2	83.2	86.0	83.2
7		MSTParser	87.1	83.5	87.3	83.8
8		Basic TurboParser	87.0	84.1	86.7	84.0
9		Standard TurboParser	90.2	87.3	89.8	87.0
10		Full TurboParser	**90.4**	**87.4**	**90.1**	**87.3**

Table 2: CCPROCESSED SD parsing performance. Since converting to CCPROCESSED SD requires labeled dependencies, EasyFirst and Huang do not join in the comparison here. †Charniak-Johnson uses semi-supervised training; Stanford englishPCFG and RNN models use external resources; see text.

gold-standard tree, we follow the convention of reporting F_1 scores (unlabeled and labeled). The additional runtime for this transformation is negligible, so we do not report runtimes. The EasyFirst and Huang parsers cannot be evaluated this way, since labeled BASIC dependencies are required for the transformation to CCPROCESSED. The pattern is quite similar to the BASIC SD experiment, with the same top performers among c- and d-parsers. The gap between c-parsing and d-parsing is 2.6% unlabeled F_1 (3.0% labeled).

4 Part-of-Speech Effects

We next consider the effect of POS tagging quality on SD parsing performance. We focus on the Berkeley parser, which performed strongly among c-parsing techniques and is amenable to substituting its default POS tagger,[6] and the two strongest d-parsing models, Standard and Full TurboParser.

First, we consider how these parsers perform with gold-standard POS tags provided at test time. Results are shown in the top three rows of Table 3. As expected, all methods perform better with better POS tags. More interestingly, the gap between the Berkeley parser and Full TurboParser is essentially gone, with each showing a slight lead on one of the two datasets.

Next (second block in Table 3), we compared these three parsers, given the POS tags produced by the *Berkeley* parser. Both TurboParsers gain about one point in each score (compared to their performance with Stanford POS tags reported earlier and repeated in the third block of Table 3) and generally match the performance of the Berkeley parser with its own POS tags.

Further, we see that the Berkeley parser suffers a drop of performance—about one point on each score—when provided *Stanford* POS tags (the same tags provided to TurboParser). Given Stanford POS tags, the Berkeley parser and Full TurboParser again perform about the same.

Taken together, these results suggest that future work on improving part-of-speech representations (perhaps along the lines of latent annotation methods already optimized for phrase structure parsing in the

[6]We explored Berkeley POS tags rather than Charniak-Johnson because the Charniak-Johnson parser alters the Penn Treebank POS tag set slightly. (For example, it introduces tags AUX and AUXG.) A fair comparison would require extra steps to control for this important difference.

	POS Tags	Parser	BASIC				CCPROCESSED			
			PTB §22		PTB §23		PTB §22		PTB §23	
			UAS	LAS	UAS	LAS	U. F_1	L. F_1	U. F_1	L. F_1
	Gold	Berkeley	93.61	91.85	**93.65**	**92.05**	91.7	89.7	**91.8**	**90.0**
		Standard TurboParser	93.55	91.94	93.31	91.73	91.7	89.8	91.4	89.5
		Full TurboParser	**93.79**	**92.21**	93.56	91.99	**92.0**	**90.1**	91.6	89.8
3	Berkeley	Berkeley	93.33	90.64	**93.31**	**91.01**	91.4	88.5	**91.4**	**88.9**
		Standard TurboParser	93.25	90.67	92.74	90.42	91.4	88.6	90.8	88.2
		Full TurboParser	**93.46**	**90.88**	93.01	90.70	**91.6**	**88.8**	91.1	88.5
	Stanford	Berkeley	92.26	89.46	**92.41**	**89.87**	90.3	87.3	**90.3**	**87.6**
9		Standard TurboParser	92.16	89.50	91.95	89.40	90.2	87.3	89.8	87.0
10		Full TurboParser	**92.29**	**89.64**	92.20	89.67	**90.4**	**87.4**	90.1	87.3

Table 3: Effects of POS on SD parsing performance. Numerals in the leftmost column refer to rows in Table 1.

Berkeley parser; Petrov et al., 2006), specifically for Stanford dependency representations, might lead to further gains. Further, joint inference between part-of-speech tags and d-parsing might also offer improvements (Hatori et al., 2011; Li et al., 2011).

5 Yamada-Matsumoto Features

As noted in §1, dependency parsing algorithms have generally been successful for YM parsing, which emphasizes syntactic (and typically more local) relationships over semantic ones. Given that dependency parsing can be at least twice as fast as phrase-structure parsing, we consider exploiting YM dependencies within a SD parser. Simply put, a YM dependency parse might serve as a cheap substitute for a phrase-structure parse, if we can transform YM trees into SD trees.

Fortunately, the featurized, discriminative modeling families typically used in dependency parsing are ready consumers of new features. The idea of using a parse tree produced by one parser to generate features for a second was explored by Nivre and McDonald (2008) and Martins et al. (2008), and found effective. The technical approach is called "stacking," and has typically been found most effective when two different parsing models are applied in the two rounds. Martins et al. released a package for stacking with MSTParser as the second parser,[7] which we apply here. The descriptions of the second parser's features derived from the first parser are listed in Table 4; these were reported by to be the best-performing on §22 in more extensive experiments following from Martins et al. (2008).[8]

The method is as follows:

1. Sequentially partition the Penn Treebank §02–22 into three parts (P_1, P_2, and P_3).

2. Train three instances of the first parser g_1, g_2, g_3 using $P_2 \cup P_3$, $P_1 \cup P_3$, and $P_1 \cup P_2$, respectively. Then parse each P_i with g_i. These predictions are used to generate features for the second parser, h; the partitioning ensures that h is never trained on a first-round parse from a "cheating" parser.

[7] `http://www.ark.cs.cmu.edu/MSTParserStacked`

[8] Personal communication.

Name	Description
PredEdge	Indicates whether the candidate edge was present, and what was its label.
Sibling	Lemma, POS, link label, distance, and direction of attachment of the previous and next predicted siblings.
Grandparents	Lemma, POS, link label, distance, and direction of attachment of the previous and next predicted siblings.
PredHead	Predicted head of the candidate modifier (if PredEdge = 0).
AllChildren	Sequence of POS and link labels of all the predicted children of the candidate head.

Table 4: Features derived from the first parser, used in the second, in stacking.

Parser	PTB §22		PTB §23	
	UAS	LAS	UAS	LAS
MaltPaser-YM	89.60	85.80	89.37	85.95
MSTParser-YM	92.17	88.36	91.97	88.46

Table 5: Accuracies of MaltParser and MSTParser on YM dependencies.

3. Train the second parser h on §02–21, including the predictions from Step 2.
4. Train the first parser g on §02–21.
5. To parse the test set, apply g, then h.

In our experiments, we consider four different first parsers: MSTParser (second order, as before) and MaltParser (liblinear stackproj), each targeting YM and SD dependencies (2×2 combinations). The second parser is always MSTParser. These parsers were chosen because they are already integrated in to a publicly released implementation of stacked parsing by Martins et al. (2008). For reference, the performance of MaltParser and MSTParser on YM dependencies, on PTB §22–23, tagged by the Stanford POS Tagger are listed in Table 5.

Stacking results are shown in Table 6. First, we find that all four combinations outperform MSTParser on its own. The gains are usually smallest when the same parser (MSTParser) and representation (SD) are used at both levels. Changing either the first parser's representation (to YM) or algorithm (to MaltParser) gives higher performance, but varying the representation is more important, with YM features giving a 1.5% absolute gain on LAS over MSTParser. The runtime is roughly doubled; this is what we would expect, since stacking involves running two parsers in sequence.

These results suggest that in future work, Yamada-Matsumoto representations (or approximations to them) should be incorporated into the strongest d-parsers, and that other informative intermediate representations may be worth seeking out.

5.1 Incorporating Distributional Representations

Distributional information has recently been established as a useful aid in resolving some difficult parsing ambiguities. In phrase-structure parsing, for example, Socher et al. (2013) improves the Stanford parser

	Parser	PTB §22		PTB §23		Speed
		UAS	LAS	UAS	LAS	(tokens/s.)
5	MaltPaser-SD	88.93	86.23	88.47	85.90	8799.91
7	MSTParser-SD	91.24	87.83	90.87	87.53	239.60
	Stacked(MSTParser-SD, MSTParser-SD)	91.38	88.85	90.98	88.86	98.67
	Stacked(MSTParser-YM, MSTParser-SD)	**91.85**	**89.40**	**91.43**	**89.10**	95.94
	Stacked(MaltParser-SD, MSTParser-SD)	91.55	89.02	91.18	88.91	163.79
	Stacked(MaltParser-YM, MSTParser-SD)	91.61	89.09	91.13	88.83	164.07

Table 6: Integrating Yamada-Matsumoto dependencies into a BASIC SD parser. Numerals in the leftmost column refer to rows in Table 1. "MaltParser" refers to the liblinear stackproj version.

by 3.8% absolute F_1 score by injecting word vector representations and compositional operations on them (captured by recursive neural networks) into the parser's probabilistic context-free grammar. In dependency parsing, Koo et al. (2008) demonstrated that cluster features can effectively improve the performance of dependency parsers.

We employed two types of Brown clustering (Brown et al., 1992) features suggested by Koo et al.: 4–6 bit cluster representations used as replacements for POS tags and full bit strings used as replacements for word forms.[9] We incorporated these features into different variants of TurboParser, including its second and third order features. Because these cluster representations are learned from a large unannotated text corpus, the result is a semi-supervised d-parser.

Table 7 reports results on BASIC SD parsing. Both Full TurboParser and Standard TurboParser get improvement from the cluster-based features. We compare to the Stanford recursive neural network parser.[10] The Full TurboParser matches the performance of the Stanford RNN model with around 3 times the speed, and the Standard TurboParser is slightly behind the Stanford RNN model but may provides another reasonable accuracy/speed trade-off here.

Note that although both methods incorporating distributional representations, the methods and the unlabeled corpora used to construct these representations are different. Socher et al. (2013) uses the 25-dimensional vectors provided by Turian et al. (2010) trained on a cleaned version of the RCV1 (Lewis et al., 2004) corpus with roughly 37 million words (58% of the original size) using the algorithm of Collobert and Weston (2008). Koo et al. (2008) used the BLLIP corpus (Charniak et al., 2000), which contains roughly 43 million words of *Wall Street Journal* text with the sentences in the Penn Treebank removed. These differences imply that this comparison should be taken only as a practical one, not a controlled experiment comparing the methods.

6 Conclusion

We conducted an extensive empirical comparison of different methods for obtaining Stanford typed dependencies. While the most accurate method still requires phrase-structure parsing, we found that developments in dependency parsing have led to a much smaller gap between the best phrase-structure parsing (c-parsing) methods and the best direct dependency parsing (d-parsing) methods. Further experiments show that part-

[9] The cluster strings we use are the same as used by Koo et al. (2008); they are publicly available at `http://people.csail.mit.edu/maestro/papers/bllip-clusters.gz`

[10] We use the most recent model ("englishRNN.ser.gz"), shipped with Stanford CoreNLP Package (v. 3.3.0).

	Parser	PTB §22		PTB §23		Speed
		UAS	LAS	UAS	LAS	(tokens/s.)
1	Stanford basicPCFG	90.06	87.17	90.09	87.48	123.63
2	Stanford recursive neural network	**93.11**	90.16	**92.80**	90.10	66.57
9	Standard TurboParser	92.16	89.50	91.95	89.40	413.67
	Standard TurboParser with Brown cluster features	92.82	90.14	92.50	89.95	243.83
10	Full TurboParser	**92.29**	89.64	92.20	89.67	209.98
	Full TurboParser with Brown cluster features	92.96	**90.31**	92.75	**90.20**	179.26

Table 7: Incorporating distributional word representations into BASIC SD parsing. Numerals in the leftmost column refer to rows in Table 1.

of-speech tagging, which in the strongest phrase-structure parsers is carried out jointly with parsing, has a notable effect on this gap. This points the way forward toward targeted part-of-speech representations for dependencies, and improved joint part-of-speech/dependency analysis. We also found benefit from using an alternative, more syntax-focused dependency representation (Yamada and Matsumoto, 2003), to provide features for Stanford dependency parsing. Overall, we find that direct dependency parsing can achieve similar results to the Stanford CoreNLP pipeline (including the new recursive neural network-based Stanford parser) at much greater speeds. The TurboParser models trained in this work are available for download at: `http://www.ark.cs.cmu.edu/TBSD`

Acknowledgments

This research was partially supported by NSF grants IIS-1352440 and CAREER IIS-1054319. The authors thank several anonymous reviewers, André Martins, and Nathan Schneider for helpful feedback on drafts of the paper.

References

J. Björne, J. Heimonen, F. Ginera, A. Airola, T. Pahkkala, and T. Salakoski. Extracting complex biological events with rich graph-based feature sets. In *Proc. of BioNLP the Workshop on Current Trends in Biomedical Natural Language Processing: Shared Task*, 2009.

B. Bohnet. Very high accuracy and fast dependency parsing is not a contradiction. In *Proc. of COLING*, 2010.

P. Brown, P. Desouza, R. Mercer, V. Pietra, and J. Lai. Class-based n-gram models of natural language. *Computational linguistics*, 18(4):467–479, 1992.

D. Cer, M. C. de Marneffe, D. Jurafsky, and C. D Manning. Parsing to Stanford dependencies: Trade-offs between speed and accuracy. In *Proc. of LREC*, 2010.

Ö. Çetinoğlu, J. Foster, J. Nivre, D. Hogan, A. Cahill, and J. Genabith. LFG without C-structures. In *Proc. of TLT*, 2010.

C. Chang and C. Lin. Libsvm: a library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2(3):27, 2011.

E. Charniak and M. Johnson. Coarse-to-fine n-best parsing and maxent discriminative reranking. In *Proc. of ACL*, 2005.

E. Charniak, D. Blaheta, N. Ge, K. Hall, J. Hale, and M. Johnson. Bllip 1987-89 wsj corpus release 1. *Linguistic Data Consortium, Philadelphia*, 2000.

W. Che, V. I. Spitkovsky, and T. Liu. A comparison of Chinese parsers for Stanford dependencies. In *Proc. of ACL*, 2012.

J. Choi and A. McCallum. Transition-based dependency parsing with selectional branching. In *Proc. of ACL*, 2013.

R. Collobert and J. Weston. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proc. of ICML*, 2008.

M. C. de Marneffe and C. D Manning. The Stanford typed dependencies representation. In *Proc. of COLING workshop on Cross-Framework and Cross-Domain Parser Evaluation*, 2008.

M. C. de Marneffe, B. MacCartney, and C. D. Manning. Generating typed dependency parses from phrase structure parses. In *Proc. of LREC*, 2006.

J. Elming, A. Johannsen, S. Klerke, E. Lapponi, H. Martinez, and A. Søgaard. Down-stream effects of tree-to-dependency conversions. In *Proc. of NAACL-HLT*, 2013.

R. Fan, K. Chang, C. Hsieh, X. Wang, and C. Lin. Liblinear: A library for large linear classification. *The Journal of Machine Learning Research*, 9:1871–1874, 2008.

Y. Goldberg and M. Elhadad. An efficient algorithm for easy-first non-directional dependency parsing. In *Proc. of ACL*, 2010.

D. Graff. North american news text corpus, 1995.

J. Hatori, T. Matsuzaki, Y. Miyao, and J. Tsujii. Incremental joint pos tagging and dependency parsing in chinese. In *Proc. of IJCNLP*, 2011.

L. Huang and K. Sagae. Dynamic programming for linear-time incremental parsing. In *Proc. of ACL*, 2010.

L. Huang, S. Fayong, and Y. Guo. Structured perceptron with inexact search. In *Proc. of NAACL-HLT*, 2012.

D. Klein and C. D. Manning. Accurate unlexicalized parsing. In *Proc. of ACL*, 2003.

T. Koo and M. Collins. Efficient third-order dependency parsers. In *Proc. of ACL*, 2010.

T. Koo, X. Carreras, and M. Collins. Simple semi-supervised dependency parsing. In *Proc. of ACL*, 2008.

D. Lewis, Y. Yang, T. Rose, and F. Li. Rcv1: A new benchmark collection for text categorization research. *The Journal of Machine Learning Research*, 5:361–397, 2004.

Z. Li, M. Zhang, W. Che, T. Liu, W. Chen, and H. Li. Joint models for chinese pos tagging and dependency parsing. In *Proc. of EMNLP*, 2011.

A. F. T Martins, D. Das, N. A. Smith, and E. P. Xing. Stacking dependency parsers. In *Proc. of EMNLP*, 2008.

A. F. T. Martins, N. A. Smith, E. P. Xing, M. A. T. Figueiredo, and P. M. Q. Aguiar. Turbo parsers: Dependency parsing by approximate variational inference. In *Proc. of EMNLP*, 2010.

A. F. T. Martins, N. A. Smith, P. M. Q. Aguiar, and M. A. T. Figueiredo. Dual decomposition with many overlapping components. In *Proc. of EMNLP*, 2011.

A. F. T. Martins, M. Almeida, and N. A. Smith. Turning on the turbo: Fast third-order non-projective turbo parsers. In *Proc. of ACL*, 2013.

D. McClosky, E. Charniak, and M. Johnson. Effective self-training for parsing. In *Proc. of NAACL-HLT*, 2006.

R. McDonald and F. Pereira. Online learning of approximate dependency parsing algorithms. In *Proc. of EACL*, 2006.

R. McDonald, F. Pereira, K. Ribarov, and J. Hajič. Non-projective dependency parsing using spanning tree algorithms. In *Proc. of HLT-EMNLP*, 2005.

J. Nivre. Incrementality in deterministic dependency parsing. In *Proc. of ACL Workshop on Incremental Parsing: Bringing Engineering and Cognition Together*, 2004.
J. Nivre. Non-projective dependency parsing in expected linear time. In *Proc. of ACL-IJCNLP*, 2009.
J. Nivre and R. McDonald. Integrating graph-based and transition-based dependency parsers. In *Proc. of ACL-HLT*, 2008.
J. Nivre, J. Hall, and J. Nilsson. Maltparser: A data-driven parser-generator for dependency parsing. In *Proc. of LREC*, 2006.
J. Nivre, M. Kuhlmann, and J. Hall. An improved oracle for dependency parsing with online reordering. In *Proc. of IWPT*, 2009.
S. Petrov, L. Barrett, R. Thibaux, and D. Klein. Learning accurate, compact, and interpretable tree annotation. In *Proc. of COLING-ACL*, 2006.
A. M. Rush and S. Petrov. Vine pruning for efficient multi-pass dependency parsing. In *Proc. of NAACL*, 2012.
R. Socher, J. Bauer, C. D. Manning, and A. Ng. Parsing with compositional vector grammars. In *Proc. of ACL*, 2013.
D. Stein, S. Peitz, D. Vilar, and H. Ney. A cocktail of deep syntactic features for hierarchical machine translation. In *Proc. of AMTA*, 2010.
J. Tetreault, J. Foster, and M. Chodorow. Using parse features for preposition selection and error detection. In *Proc. of ACL*, 2010.
K. Toutanova, D. Klein, C. D. Manning, and Y. Singer. Feature-rich part-of-speech tagging with a cyclic dependency network. In *Proc. of COLING-ACL*, 2003.
J. Turian, L. Ratinov, and Y. Bengio. Word representations: a simple and general method for semi-supervised learning. In *Proc. of ACL*, 2010.
H. Yamada and Y. Matsumoto. Statistical dependency analysis with support vector machines. In *Proc. of IWPT*, 2003.
H. Zhang, L. Huang, K. Zhao, and R. McDonald. Online learning for inexact hypergraph search. In *Proc. of EMNLP*, 2013.

A Additional Inference Rules

We applied our additional inference rules after the collapsing transformation and propagation of conjuncts transformation (implemented by the Stanford CoreNLP Toolkit). The rules we use are as follows:

- Rule 1: Fixing the uncollapsed *cc* dependencies: For each *cc*$(A \rightarrow B)$, $T(A \rightarrow C)$, where C is the first right sibling of A, and A linearly precedes B, add *conj_B*$(A \rightarrow B)$ and remove $T(A \rightarrow C)$.
- Rule 2: Fixing the uncollapsed *prep* dependencies: For each *prep*$(A \rightarrow B)$, *pobj*(B, C) and A linearly precedes B linearly precedes C, add *prep_B*$(A \rightarrow C)$ and remove *prep*$(A \rightarrow B)$, *pobj*$(B \rightarrow C)$.

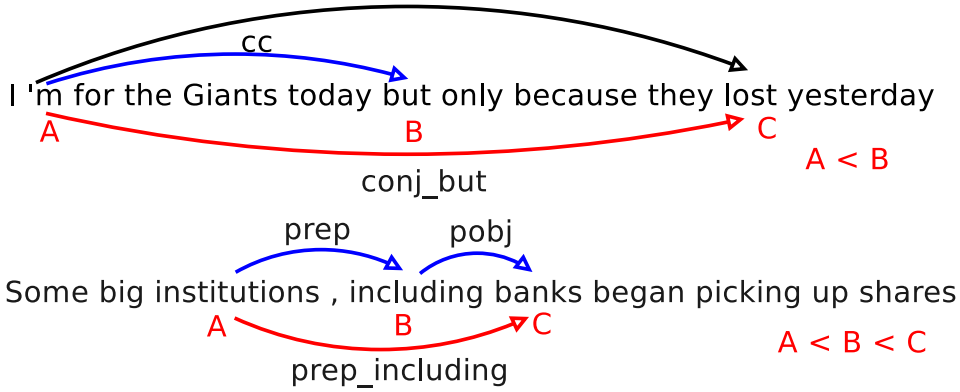

Figure 2: Examples of the application of rule 1 (above) and 2 (below), showing the patterns above the sentence and the added dependencies in red below the text. Dependencies in blue will be removed.