Simulating Dependencies to Improve Parse Error Detection

Markus Dickinson and Amber Smith

Department of Linguistics
Indiana University
E-mail: md7@indiana.edu, smithamj@indiana.edu

Abstract

We improve parse error detection, weighting dependency information on the basis of simulated parses. Such simulations extend the training grammar, and, although the simulations are not wholly correct or incorrect—as observed from the results with different weightings for small treebanks—they help to determine whether a new parse fits the training grammar.

1 Introduction and motivation

Obtaining high-quality annotation is a bottleneck for corpus and parser development, leading to methods of leveraging existing resources to improve the parameters of a lesser-resourced language—through the use of annotated corpora of higher-resourced languages, parallel language data, universal treebanks, and so on [5, 7, 11, 13, 18, 19, 22]. Building on work that envisions bootstrapping corpora via parse error detection and human correction [3, 24]—from a larger paradigm of exploiting a small amount of annotated data in a language [6, 8, 14]—we focus on extracting more resources from just the treebank itself. Specifically, we explore the use of *parse simulations* within a (small) treebank to supplement the parameters learned from the treebank itself, in order to improve parse error detection.

This bears much in common with contrastive estimation [e.g., 25, 26] and negative sampling [e.g., 9, 20], where learning is the process of distinguishing a positive example from similar negative examples in the data. With small data and the task of error detection, however, it is not clear whether the simulated information is truly incorrect or not: for small data sets, one may uncover structures which, although incorrect in a particular sentence, can be correct. Thus, instead of maximizing the differences between actually-occurring and simulated data, we employ "negative" information heuristically, to modify the scores of syntactic dependency structures.

Employing simulations has the effect of generalizing the grammar implicit in a treebank. This allows one to explore a large amount of information in the annotation space (e.g., 100x more structures). Our specific contributions are to:

- Posit annotation simulations during training;
- Replace ad hoc weighting by a weighting scheme using the simulations, leading to improved error detection precision; and
- Experiment with different weighting schemes, testing the relationship between simulations and (un)seen structures.

2 Parse error detection

We start with the DAPS (Detecting Anomalous Parse Structures) method [3, 4],[1] effective across a variety of settings. N-gram sequences of dependency structures are used to identify anomalous (likely erroneous) dependencies. Referring to the suggested simulation (dotted line) in Figure 1 as an example, the process is:

1. Extract rules from dependency trees (rule = a head (`H`) + its dependents). e.g., one rule (headed by *board*) is `START det:DT NN-H prep:IN END.`
2. Extract rule n-grams, e.g., `det:DT NN-H prep:IN` or `prep:IN END.`
3. Score an element by adding up relevant training n-gram counts. e.g., to score `prep:IN` here, one adds up training counts of the n-grams with `prep:IN`.[2]

Using information from complete trees, the DAPS method serves as a sanity check on more complicated parse models. While such checks may be possible inside a parser, this method optimizes for unlikelihood, whereas parsers more directly maximize likelihood. In the context of low-resource languages, DAPS has the benefit of using very general representations—as opposed to integrating information from, e.g., word clusters—that do not require a lot of (un)annotated data.

Insights Our previous DAPS work [3, 24] has noted that not all n-grams are equal: a parsed rule lacking any training bigrams indicates an error, and yet, without more context, bigrams often upweight erroneous rules because they generally occur in training. Previous variations ignored bigrams (*high* method: $n \geq 3$) or downweighted them (weighted all-grams [*wall*]: $n \geq 2$, bigram counts multiplied by 0.01). Problematic is setting the weight arbitrarily and equally across bigrams.

A second insight comes from a stage of **revision checking** [3], where revisions (alternative attachments and labelings) are posited for a parse. Errors are *flagged* when the parser did not choose better-scoring possible revisions. The crucial insight is that dependency trees provide information not only about what they are, but what they could have been, an insight we will apply during training

[1] `http://cl.indiana.edu/~md7/papers/dickinson-smith11.html`

[2] See [3] for more details.

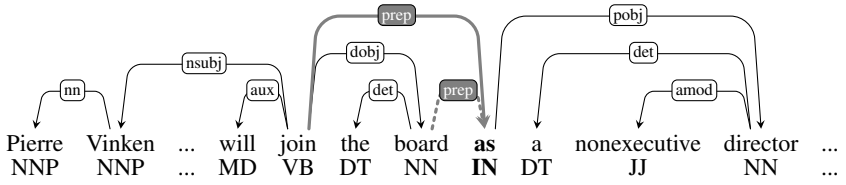

Figure 1: A gold tree from the WSJ [15], converted to dependencies [2]; Gray dotted line indicates a suggested simulation

3 Annotation simulations

3.1 Positing simulations

We focus on alternate plausible structures within the *training* data and use these syntactic **parse simulations** to gauge the utility of each n-gram. Consider, for example, the attachment of the preposition (IN) *as* in Figure 1. It may help here to see that in the gold tree *as* doesn't attach to the immediately preceding noun (NN) *board*—a potential attachment, as indicated by the gray dotted line.

We view the process of generating alternative structures, i.e., simulations, as involving the reattachment of each node (i.e., word/POS) in the dependency tree. This means that: a) a full set of alternatives is generated for every node in the tree (e.g., for the attachment of *Pierre*, of *Vinken*, etc.), and b) no more than one node is reattached for a given simulation (e.g., if *as* is re-attached, then no other word is). Each simulation is thus only "one off" from the original tree, preventing highly implausible generalizations from the source trees. While each simulation results in a specific incorrect tree (token), to be useful for error detection simulations should ideally correspond to valid structures (types) used elsewhere in the corpus, i.e., plausible structures a parser would likely consider for the sentence at hand.

We use the reattachment procedure as implemented for revision checking (section 2).[3] From this, we keep a frequency database of n-grams that actually occurred (A) and a separate one for simulated n-grams (S); we call each database a **grammar**. The n-grams are a sequence of pairs of POS and dependency labels.

To build plausible simulations, we currently stipulate [cf., e.g., 1, 10] that: a) a posited reattachment must maintain projectivity; and b) its posited dependency label must have been seen with its POS in training.[4] We only consider reattachments, and not simple relabelings, because: a) we already posit multiple labels for each reattached dependency, and b) the grammar size is more manageable. One could explore creating new POS, label pairs or filtering implausible simulations; we leave these for future work, but will see the effects of not filtering in section 4.

[3]The interpretation is different for revision checking, as one hopes to go from an incorrect parse to a correct one, and here the situation is reversed, but the process is the same.

[4]We do not enforce acyclicity, as in [3], as this unduly restricts the search space for revisions [12].

3.2 Weighting simulations

With a definition of simulations, the next step is to integrate such information with actually-occurring rules. As mentioned, we do not simply view every simulation as incorrect, so our methods reflect a gradable nature of the simulations.

During training, we use the actual count of an n-gram (a) and its simulated count (s) to calculate an adjusted score (a^*). Previous (**0**) weighting used an ad hoc weight (*wall*) on bigrams or no weight (*all/high*: $a^* = a$).

The formulas in Figure 2 attempt to downweight n-grams which are too confusable with other structures to be good indicators of quality (i.e., occur a great deal in simulated fashion [$s > a$]) and to upweight helpful n-grams (i.e., ones where actual occurrences outweigh simulated ones [$a > s$]), as can be seen in the ratio $\frac{a+1}{s+1}$. Starting with formula **B**, the training grammar is expanded by giving non-zero scores to what we term `SimOnly` n-grams, where $a = 0$ and $s > 0$.

Formulas **B**–**E** differ in how they handle the interplay between these `SimOnly` n-grams, the actually-occurring n-grams ($a > 0$, `Actual`), and the never-occurring n-grams ($a = 0$, $s = 0$), i.e., the ones `Novel` to both grammars. The general principles are characterized at the bottom of each section of Figure 2: less important than the specifics is what these indicate about how to use simulations for new data.

3.3 Combining simulations and revisions

As an additional evaluation, we run revision checking during the error detection stage, in order to see the effect of combining parse simulations with parse revisions. Note what this means: parse simulations posit additional structures to expand grammars *during training*, while revisions are additional structures posited *after parsing*. (For revision checking we try both reattachments and relabelings as revisions.) These posited revisions are scored with the same training grammar as the original parse is scored with—i.e., the grammar including simulations—and thus no scoring proceeds in the same way for the parse and any suggested revision.

4 Evaluation

4.1 Experimental conditions

We use the Wall Street Journal (WSJ) corpus from the Penn Treebank [15], converted to Stanford dependencies [2], to be comparable to previous work [24]. Based on the recommendations we outline in [24], we consider two complementary parsers (graph-based MSTParser [17]; transition-based MaltParser [21]) for the various methods, as well as a more recent one (TurboParser [16]). We use a small training corpus (section 02 [48k tokens]) [see 24], as this matches our focus, and test on a slightly larger corpus (sections 00+01+22 [134k tokens]).

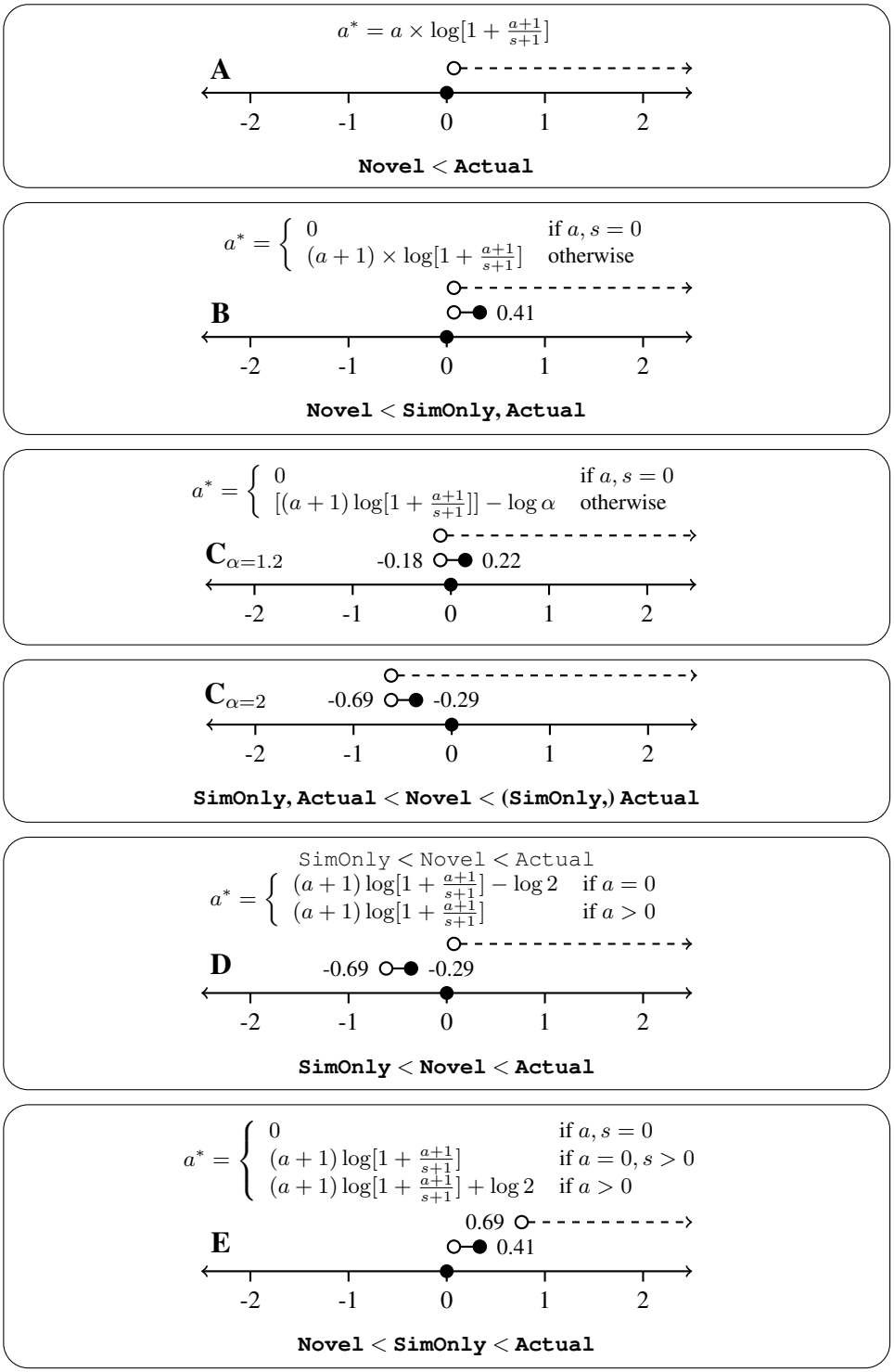

Figure 2: Different formulas & visualizations (`Actual`: dashed line, `SimOnly`: regular line, `Novel`: on the number line)

Grammar sizes Before evaluating the effectiveness on error detection, we can see the impact of parse simulations by examining training grammar sizes, as in Table 1. In moving from Actual (A) to Simulated (S) grammars, we see a 88x increase in type grammar size (94k to 8.3m) and a 104x increase in token size.[5] We also see a gain in type coverage and a drop in token coverage, indicating simulations of infrequent n-grams; given the difficulty of predicting rare events, positing such rules may benefit many applications.[6]

Grammar (All)	Types		Tokens	
	All	Over.	All	Over.
02_A	94,102	17.8%	272,703	72.7%
02_S	8,349,644	23.7%	28,363,605	68.2%
$02_{A \cup S}$	8,411,547	30.8%	28,636,308	78.5%
$00{+}01{+}22_A$	201,499	n/a	757,556	n/a

Table 1: Relevant grammar sizes, in terms of number of dependency n-grams (A=Actual, S=Simulated), for *All* n-grams; *Over(lap)* is calculated w.r.t. testing data ($00{+}01{+}22_A$), i.e., percentages of covered testing types/tokens.

When the entire Actual and Simulated grammars are combined ($A \cup S$), the overlap with the testing data increases by an absolute 13% (types) or 6% (tokens). Creating simulations thus shows promise for expanding the grammar in ways that extend to new data—with a caveat of a huge leap in grammar size for a smaller gain in grammar coverage, thus indicating a need for some grammar filtering.

Evaluation metric For evaluating error detection, we report precision (P) of detected errors.[7] We rely on equivalently-sized *segments* across different error detection methods, accounting for recall. Using segment size—e.g., 5% of corpus tokens—approximates the idea of annotators having a set amount of time to correct parser errors [see 24, for discussion]. When it comes to segment size, we assume that correction works from the lowest scores up to the highest.

For methods employing revision checking (sections 2 and 3.3), the methodology is slightly different: first, all *flagged* positions are examined, from lowest to highest scores, and then all unflagged positions. We focus on flagged positions.

4.2 Results

We report results for MST and Malt in Table 2 and for Turbo in Table 3, with the best formula results in bold, and the best overall results underlined. To summarize: 1) Weighting seems to work, as formula **0** is generally among the worst

[5]There are 32,199 types (188,698 Actual tokens) in common between the A and S grammars.

[6]We omit numbers broken down by n-gram size: there are many more types for higher n values.

[7]In [24], we also recommend reporting a revised labeled attachement score (LAS_r), to see the impact of correcting errors on resulting corpus quality. As we are more interested in method development and as the LAS_r trends precisely mirror the precision ones, we do not report LAS_r here.

formulas; 2) The best overall settings seem to be a mixture of hand- (`wall`) and auto-weighting (**A**, **B**, **E**); and 3) A better parser (Turbo) mitigates the improvements to some extent, but basically follows the same trends.

While the `wall` method is generally best, for MST and Malt the `all` method curiously outperforms it for formulas **B**–**E** at the 5% size, i.e., bigram weighting seems unhelpful. For Turbo, the `all` method is almost always lowest, with `high` and `wall` being fairly comparable across the different segments; unweighted bigrams (`all`) are apparently more misleading with a better parser. Additionally, formula **E** (`Novel` < `SimOnly` < `Actual`) is most often the best setting, but **A** (ignoring the `Novel`/`SimOnly` distinction) can be better. Formulas employing a `Novel`/`SimOnly` distinction are perhaps more unstable than formula **A**.

If forced to pick a best setting, `wall.E` is consistently strong, aside from its dip at 5% for MST and Malt. Simulations seem to be both positive and negative: formula **E** equals or outperforms **B** (`SimOnly` ≈ `Actual`), while also outperforming **D** (`SimOnly` < `Novel`) and often **A** (`SimOnly` ≈ `Novel`). This pattern, along with the 5% dip, may in part be due to not having sorted the simulations: some `SimOnly` cases behave like `Novel` and some like `Actual` n-grams.

Model	score ≤0	5%	10%	15%
Positions		6710	13420	20130
all.0	83.7% (2,313)	72.6%	58.7%	50.8%
all.A	83.7% (2,313)	75.2%	62.4%	54.6%
all.B	**85.9%** (467)	75.3%	63.2%	55.2%
all.$C_{1.2}$	80.4% (2,253)	75.5%	63.1%	55.1%
all.$C_{1.5}$	77.8% (5,303)	75.3%	62.9%	55.0%
all.C_2	74.4% (6,680)	74.2%	62.6%	54.9%
all.D	79.7% (4,609)	75.5%	63.3%	55.4%
all.E	**85.9%** (467)	**76.1%**	**63.6%**	**55.5%**
high.0	73.6% (8,959)	**73.6%**	63.9%	55.3%
high.A	73.6% (8,959)	**73.6%**	**66.1%**	56.9%
high.B	**75.4%** (2,480)	70.8%	65.4%	57.0%
high.$C_{1.2}$	69.9% (6,102)	69.7%	65.8%	57.1%
high.$C_{1.5}$	67.3% (12,429)	63.0%	65.9%	56.9%
high.C_2	63.0% (14,902)	59.2%	63.0%	57.0%
high.D	68.5% (11,583)	66.1%	65.6%	56.9%
high.E	**75.4%** (2,480)	73.3%	**66.1%**	**57.2%**
wall.0	83.7% (2,313)	76.0%	64.1%	54.9%
wall.A	83.7% (2,313)	**77.4%**	65.6%	56.4%
wall.B	**85.9%** (467)	72.8%	65.4%	56.7%
wall.$C_{1.2}$	76.4% (2,766)	73.7%	65.6%	56.6%
wall.$C_{1.5}$	71.6% (8,031)	70.8%	64.9%	56.6%
wall.C_2	65.6% (10,979)	63.5%	64.4%	56.5%
wall.D	71.2% (7,750)	70.6%	65.2%	57.0%
wall.E	**85.9%** (467)	74.9%	**66.0%**	**57.1%**

Model	score ≤0	5%	10%	15%
Positions		6710	13420	20130
all.0	91.2% (3,604)	79.5%	63.4%	54.0%
all.A	91.2% (3,604)	81.7%	65.5%	55.9%
all.B	**96.1%** (1,700)	81.3%	65.7%	56.3%
all.$C_{1.2}$	88.4% (3,598)	81.3%	65.6%	56.2%
all.$C_{1.5}$	81.3% (6,498)	80.8%	65.4%	56.2%
all.C_2	77.3% (7,758)	77.3%	65.2%	55.9%
all.D	84.0% (5,654)	81.3%	65.9%	56.3%
all.E	**96.1%** (1,700)	**82.3%**	**66.2%**	**56.6%**
high.0	77.0% (8,713)	77.0%	65.8%	56.3%
high.A	77.0% (8,713)	77.0%	68.6%	**58.9%**
high.B	**83.5%** (3,881)	77.2%	68.0%	58.8%
high.$C_{1.2}$	76.3% (6,873)	76.3%	68.3%	58.8%
high.$C_{1.5}$	69.2% (12,409)	60.0%	68.1%	58.7%
high.C_2	65.2% (14,902)	56.4%	65.2%	58.6%
high.D	70.7% (11,537)	70.7%	68.0%	58.8%
high.E	**83.5%** (3,881)	**79.0%**	**68.7%**	58.8%
wall.0	91.2% (3,604)	80.4%	67.0%	56.7%
wall.A	91.2% (3,604)	**81.3%**	68.5%	58.2%
wall.B	**96.1%** (1,700)	79.0%	68.3%	58.4%
wall.$C_{1.2}$	84.8% (3,874)	78.4%	68.0%	58.3%
wall.$C_{1.5}$	75.2% (8,289)	68.2%	67.3%	58.2%
wall.C_2	69.1% (10,997)	61.0%	66.6%	57.9%
wall.D	76.0% (7,857)	76.0%	67.6%	58.4%
wall.E	**96.1%** (1,700)	79.9%	**68.9%**	**58.6%**

Table 2: Error detection precision for MST (left, LAS = 80.5%) and Malt (right, LAS = 81.1%)

Model	score $\leq$0	5%	10%	15%
Positions		6710	13420	20130
all.0	77.3% (719)	55.0%	45.2%	39.1%
all.A	77.3% (719)	58.6%	48.1%	42.7%
all.B	**79.8%** (104)	58.9%	48.9%	42.8%
all.$C_{1.2}$	71.7% (1,010)	58.5%	48.7%	42.7%
all.$C_{1.5}$	68.2% (2,820)	58.2%	48.6%	42.8%
all.C_2	64.0% (3,934)	58.0%	48.6%	**42.9%**
all.D	70.5% (2,207)	58.9%	49.0%	**42.9%**
all.E	**79.8%** (104)	**59.4%**	**49.3%**	42.8%
high.0	65.1% (4,864)	**60.7%**	48.2%	41.4%
high.A	65.1% (4,864)	60.6%	49.2%	42.5%
high.B	**67.5%** (1,381)	58.7%	48.9%	42.9%
high.$C_{1.2}$	60.9% (3,504)	58.8%	49.0%	42.8%
high.$C_{1.5}$	58.1% (7,540)	58.1%	49.1%	42.9%
high.C_2	54.0% (9,664)	49.6%	49.3%	42.7%
high.D	59.7% (6,845)	59.7%	**49.5%**	43.0%
high.E	**67.5%** (1,381)	60.0%	49.1%	**43.2%**
wall.0	77.3% (719)	60.2%	48.0%	41.4%
wall.A	77.3% (719)	61.1%	49.2%	43.0%
wall.B	**79.8%** (104)	60.5%	49.4%	**43.5%**
wall.$C_{1.2}$	67.2% (1,380)	61.0%	49.6%	**43.5%**
wall.$C_{1.5}$	62.6% (4,542)	60.7%	49.4%	43.3%
wall.C_2	57.1% (6,829)	57.1%	49.4%	43.1%
wall.D	62.7% (4,192)	60.8%	**49.7%**	**43.5%**
wall.E	**79.8%** (104)	**61.3%**	49.5%	**43.5%**

Table 3: Error detection precision for Turbo (LAS = 84.1%)

4.3 Revision checking

Employing revision checking presents a slightly different—and more consistent—picture, shown in Tables 4 and 5, where we focus on evaluation cutoffs that highlight the effect of revisions.[8] Most noticeably, formula **E** almost always has the highest precision for both all the flagged positions (*AF*) and the flagged positions with the lowest scores (*0F*), and the `wall.E` method is clearly best. Although some of the differences are small (especially for the MST results), we see 4–10% increases in precision for Malt for the 5% segment, comparing **E** to formulas **0/A**.

We suspect that this improvement comes because simulations and revisions rely upon the same procedure of positing alternate structures, and formula **E** provides better scoring for the unseen structures involved in the revisions. Because it assigns positive scores to simulated training n-grams (`SimOnly`), formula **E** more often positively scores revisions. Thus, it is easier to find a revision that scores better than the original parse (compare, e.g., 86% [=$\frac{1,460}{1,700}$] of zero-scoring dependencies being flagged [*0F*] for `Malt.all` with formula **E** vs. 59% [=$\frac{2,129}{3,604}$] with formula **A**). The extended grammar thus helps identify when there is a potentially better

[8] We also ignore the **C** and **D** models here, as they continue to be the worst-performing ones.

Model	0F	AF	5%
all.0	86.2% (1,598)	50.2% (14,424)	70.2%
all.A	86.0% (1,603)	54.3% (13,430)	73.5%
all.B	**86.4%** (397)	55.1% (13,415)	74.5%
all.E	**86.4%** (397)	**55.7%** (13,484)	**75.7%**
high.0	82.6% (4,718)	50.1% (14,676)	71.4%
high.A	**82.6%** (4,729)	54.2% (13,524)	73.3%
high.B	77.2% (1,665)	55.4% (14,936)	72.6%
high.E	77.2% (1,663)	**55.9%** (15,030)	**74.4%**
wall.0	86.2% (1,603)	52.5% (15,595)	74.7%
wall.A	86.0% (1,608)	56.3% (14,169)	76.1%
wall.B	**86.4%** (396)	56.5% (14,612)	74.6%
wall.E	**86.4%** (396)	**56.8%** (14,764)	**76.2%**

Model	0F	AF	5%
all.0	91.3% (2,127)	47.5% (13,460)	70.8%
all.A	91.4% (2,129)	51.3% (12,218)	71.8%
all.B	**96.6%** (1,460)	53.9% (12,671)	76.2%
all.E	**96.6%** (1,460)	**54.3%** (12,662)	**77.1%**
high.0	78.6% (3,369)	45.5% (13,050)	63.9%
high.A	78.5% (3,363)	49.5% (11,632)	67.0%
high.B	**84.8%** (2,273)	52.9% (13,367)	73.2%
high.E	**84.8%** (2,271)	**53.2%** (13,377)	**74.2%**
wall.0	91.4% (2,129)	49.4% (14,179)	72.4%
wall.A	91.4% (2,130)	53.1% (12,566)	74.4%
wall.B	**96.6%** (1,461)	55.2% (13,535)	77.1%
wall.E	**96.6%** (1,461)	**55.5%** (13,636)	**78.0%**

Table 4: Error detection precision with revision checking for MST (left, LAS = 80.5%) and for Malt (right, LAS = 81.1%) (*0F* = zero-scoring (or lower) flagged positions, *AF* = all flagged positions)

Model	0F	AF	5%
all.0	76.0% (442)	36.2% (11,147)	47.7%
all.A	76.1% (443)	39.9% (9,967)	49.3%
all.B	**77.0%** (74)	40.8% (9,921)	50.5%
all.E	**77.0%** (74)	**41.5%** (9,940)	**51.6%**
high.0	67.2% (2,305)	37.5% (11,628)	49.3%
high.A	67.3% (2,315)	41.3% (10,316)	51.1%
high.B	**68.3%** (942)	43.3% (11,311)	55.1%
high.E	**68.3%** (941)	**43.5%** (11,342)	**56.1%**
wall.0	76.0% (442)	39.0% (11,937)	51.9%
wall.A	76.0% (442)	42.6% (10,424)	53.1%
wall.B	**76.7%** (73)	43.8% (10,903)	55.6%
wall.E	**76.7%** (73)	**44.0%** (11,003)	**56.4%**

Table 5: Error detection precision for Turbo with revision checking (LAS = 84.1%)

revision that the parser ignored. This seems to be an encouraging direction to pursue if one wishes to extend error detection into automatic correction.

Discussion The best formula, **E**, cleanly separates `Novel`, `SimOnly`, and `Actual` cases. What does this mean, then, in terms of whether simulated information is incorrect or not, i.e., whether the grammar is appropriately being generalized? To address this question, we first note that formula **E** equals or outperforms **B**,[9] indicating that the `SimOnly` cases are less helpful than the `Actual` cases. There is thus an indication that simulated n-grams often contain negative information.

But the picture is more complicated: we also need to note that **E** almost always has higher precision than formula **D**, the formula that cleanly ranks `SimOnly` lower than `Novel` cases. Additionally, formula **E** generally—though, not always—

[9] **B** and **E** are equivalent when treating zero cases the same, i.e., as `Novels` only; see Figure 2.

outperforms formula **A**, the formula where `Novel` and `SimOnly` are grouped together as equally bad. Taken together, the performances indicate that simulations as a whole are better than cases which have never been seen.

We conjecture that this behavior is due, at least in part, to not having sorted out the simulations: some of the `SimOnly` cases seem to behave like `Novel` cases and some more like `Actual` n-grams. As mentioned, a crucial next step is thus to sort the `SimOnly` cases, so that only the valid cases receive higher scores.

5 Summary and outlook

We have improved a method for parse error detection, weighting dependency n-grams on the basis of parse simulations in training. As important as the improvements in error detection is the idea of extending the training grammar for small treebanks by such simulations. As we observed from the results with different formulas, the simulations cannot be treated as wholly correct or incorrect—especially given the small size of the training data—but are nonetheless useful for helping determine whether a new parse fits into the (extended) grammar.

There are several next steps for this work. There is a need to try different (sub)sets of simulations, to see which constraints or filters can be best employed, in particular to sort out the good and bad `SimOnly` cases. Adding filters will increase the amount of time spent in building a grammar during training, but result in smaller—as well as more accurate—grammars, thus improving the speed for post-parsing analysis. We have experimented only with a small training corpus, resulting in a huge extension of n-grams; one could investigate larger and more diverse training corpora, to observe both the effectiveness of the techniques and the efficiency with such large grammars.

One could also test the methods on out-of-domain data, to see how well the grammar extensions cover different kinds of data, as well as comparing to alternative methods of tree exploration, such as parse reranking models employing n-grams or subtree features over a set of k-best parses [e.g., 23]. In that light, one also needs to explore more extrinsic evaluation, to see whether the methods are actually identifying errors that truly impact parsing models (i.e., have a real-world impact) vs. ones which are more ad hoc.

Acknowledgements

We wish to thank the three anonymous reviewers for very helpful feedback, as well as the CL discussion group at Indiana University.

References

[1] Giuseppe Attardi and Felice Dell'Orletta. Reverse revision and linear tree combination for dependency parsing. In *Proceedings of HLT-NAACL-09, Short Papers*, pages 261–264, Boulder, CO, June 2009.

[2] Marie-Catherine de Marneffe, Bill MacCartney, and Christopher D. Manning. Generating typed dependency parses from phrase structure parses. In *LREC 2006*, 2006.

[3] Markus Dickinson and Amber Smith. Detecting dependency parse errors with minimal resources. In *Proceedings of IWPT-11*, pages 241–252, Dublin, October 2011.

[4] Markus Dickinson and Amber Smith. Finding parse errors in the midst of parse errors. In *Proceedings of the 13th International Workshop on Treebanks and Linguistic Theories (TLT13)*, Tübingen, Germany, 2014.

[5] Long Duong, Trevor Cohn, Steven Bird, and Paul Cook. Low resource dependency parsing: Cross-lingual parameter sharing in a neural network parser. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, pages 845–850, Beijing, China, July 2015.

[6] Long Duong, Trevor Cohn, Steven Bird, and Paul Cook. A neural network model for low-resource universal dependency parsing. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 339–348, Lisbon, Portugal, September 2015. Association for Computational Linguistics.

[7] Anna Feldman and Jirka Hana. *A resource-light approach to morpho-syntactic tagging*. Rodopi, Amsterdam/New York, NY, 2010.

[8] Dan Garrette and Jason Baldridge. Learning a part-of-speech tagger from two hours of annotation. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 138–147, Atlanta, Georgia, June 2013. Association for Computational Linguistics.

[9] Yoav Goldberg and Omer Levy. `word2vec` explained: Deriving Mikolov et al.'s negative-sampling word-embedding method. Technical report, Ben-Gurion University of the Negev, 2014.

[10] Enrique Henestroza Anguiano and Marie Candito. Parse correction with specialized models for difficult attachment types. In *Proceedings of EMNLP-11*, pages 1222–1233, Edinburgh, 2011.

[11] Rebecca Hwa, Philip Resnik, Amy Weinberg, Clara Cabezas, and Okan Kolak. Bootstrapping parsers via syntactic projection across parallel texts. *Natural Language Engineering*, 11:311–325, 2005.

[12] Mohammad Khan, Markus Dickinson, and Sandra Kübler. Does size matter? text and grammar revision for parsing social media data. In *Proceedings of the Workshop on Language Analysis in Social Media*, Atlanta, GA USA, 2013.

[13] Teresa Lynn, Jennifer Foster, Mark Dras, and Lamia Tounsi. Cross-lingual transfer parsing for low-resourced languages: An Irish case study. In *Proceedings of CLTW 2014*, Dublin, 2014.

[14] Teresa Lynn, Jennifer Foster, Mark Dras, and Josef van Genabith. Working with a small dataset — semi-supervised dependency parsing for Irish. In *Proceedings of SPMRL 2013*, Seattle, 2013.

[15] Mitch Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. Building a large annotated corpus of English: The Penn Treebank. *Computational Linguistics*, 19(2):313–330, 1993.

[16] Andre Martins, Miguel Almeida, and Noah A. Smith. Turning on the turbo: Fast third-order non-projective turbo parsers. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 617–622, Sofia, Bulgaria, August 2013.

[17] Ryan McDonald, Kevin Lerman, and Fernando Pereira. Multilingual dependency analysis with a two-stage discriminative parser. In *Proceedings of CoNLL-X*, pages 216–220, New York City, June 2006.

[18] Ryan McDonald, Joakim Nivre, Yvonne Quirmbach-Brundage, Yoav Goldberg, Dipanjan Das, Kuzman Ganchev, Keith Hall, Slav Petrov, Hao Zhang, Oscar Täckström, Claudia Bedini, Núria Bertomeu Castelló, and Jungmee Lee. Universal dependency annotation for multilingual parsing. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 92–97, Sofia, Bulgaria, August 2013. Association for Computational Linguistics.

[19] Ryan McDonald, Slav Petrov, and Keith Hall. Multi-source transfer of delexicalized dependency parsers. In *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing*, pages 62–72, Edinburgh, Scotland, July 2011. Association for Computational Linguistics.

[20] Tomas Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. In *Advances in Neural Information Processing Systems 26*, pages 3111–3119, Lake Tahoe, NV, 2013.

[21] Joakim Nivre, Johan Hall, Jens Nilsson, Atanas Chanev, Gulsen Eryigit, Sandra Kübler, Svetoslav Marinov, and Erwin Marsi. MaltParser: A language-independent system for data-driven dependency parsing. *Natural Language Engineering*, 13(2):95–135, 2007.

[22] Mohammad Sadegh Rasooli and Michael Collins. Density-driven cross-lingual transfer of dependency parsers. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 328–338, Lisbon, Portugal, September 2015.

[23] Mo Shen, Daisuke Kawahara, and Sadao Kurohashi. Dependency parse reranking with rich subtree features. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 22(7):1208–1218, 2014.

[24] Amber Smith and Markus Dickinson. Evaluating parse error detection across varied conditions. In *Proceedings of the 13th International Workshop on Treebanks and Linguistic Theories (TLT13)*, Tübingen, Germany, 2014.

[25] Noah A. Smith and Jason Eisner. Contrastive estimation: Training log-linear models on unlabeled data. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL'05)*, pages 354–362, Ann Arbor, MI, June 2005.

[26] Noah A. Smith and Jason Eisner. Guiding unsupervised grammar induction using contrastive estimation. In *International Joint Conference on Artificial Intelligence (IJCAI) Workshop on Grammatical Inference Applications*, pages 73–82, Edinburgh, 2005.