Joint Incremental Disfluency Detection and Dependency Parsing

Matthew Honnibal
Department of Computing
Macquarie University
Sydney, Australia
matthew.honnibal@mq.edu.edu.au

Mark Johnson
Department of Computing
Macquarie University
Sydney, Australia
mark.johnson@mq.edu.edu.au

Abstract

We present an incremental dependency parsing model that jointly performs disfluency detection. The model handles speech repairs using a novel non-monotonic transition system, and includes several novel classes of features. For comparison, we evaluated two pipeline systems, using state-of-the-art disfluency detectors. The joint model performed better on both tasks, with a parse accuracy of 90.5% and 84.0% accuracy at disfluency detection. The model runs in expected linear time, and processes over 550 tokens a second.

1 Introduction

Most unscripted speech contains filled pauses (*um*s and *uh*s), and errors that are usually edited on-the-fly by the speaker. Disfluency detection is the task of detecting these infelicities in spoken language transcripts. The task has some immediate value, as disfluencies have been shown to make speech recognition output much more difficult to read (Jones et al., 2003), but has also been motivated as a module in a natural language understanding pipeline, because disfluencies have proven problematic for PCFG parsing models.

Instead of a pipeline approach, we build on recent work in transition-based dependency parsing, to perform the two tasks jointly. There have been two small studies of dependency parsing on unscripted speech, both using entirely greedy parsing strategies, without a direct comparison against a pipeline architecture (Jorgensen, 2007; Rasooli and Tetreault, 2013). We go substantially beyond these pilot studies, and present a system that compares favourably to a pipeline consisting of state-of-the-art components. Our parser largely follows the design of Zhang and Clark (2011). We use a structured averaged perceptron model with beam-search decoding (Collins, 2002). Our feature set is based on Zhang and Clark (2011), and our transition-system is based on the arc-eager system of Nivre (2003).

We extend the transition system with a novel non-monotonic transition, Edit. It allows sentences like '*Pass the pepper uh salt*' to be parsed incrementally, without the need to guess early that *pepper* is disfluent. This is achieved by re-processing the leftward children of the word Edit marks as disfluent. For instance, if the parser attaches *the* to *pepper*, but subsequently marks *pepper* as disfluent, *the* will be returned to the stack. We also exploit the ease with which the model can incorporate arbitrary features, and design a set of features that capture the 'rough copy' structure of some speech repairs, which motivated the Johnson and Charniak (2004) noisy channel model.

Our main comparison is against two pipeline systems, which use the two current state-of-the-art disfluency detection systems as pre-processors to our parser, minus the custom disfluency features and transition. The joint model compared favourably to the pipeline parsers at both tasks, with an unlabelled attachment score of 90.5%, and 84.0% accuracy at detecting speech repairs. An efficient implementation is available under an open-source license.[1] The future prospects of the system are also quite promising. Because the parser is incremental, it should be well suited to unsegmented text such as the output of a speech-recognition system. We consider our main contributions to be:

- a novel non-monotonic transition system, for speech repairs and restarts,

[1]http://github.com/syllog1sm/redshift

Transactions of the Association for Computational Linguistics, 2 (2014) 131–142. Action Editor: Joakim Nivre.
Submitted 11/2013; Revised 2/2014; Published 4/2014. ©2014 Association for Computational Linguistics.

$$\text{A flight to } \underbrace{\text{um}}_{\text{FP}}\ \underbrace{\text{Boston}}_{\text{RM}}\ \underbrace{\text{I mean}}_{\text{IM}}\ \underbrace{\text{Denver}}_{\text{RP}} \text{ Tuesday}$$

Figure 1: A sentence with disfluencies annotated in the style of Shriberg (1994) and the Switchboard corpus. FP=Filled Pause, RM=Reparandum, IM=Interregnum, RP=Repair. We follow previous work in evaluating the system on the accuracy with which it identifies speech-repairs, marked *reparandum* above.

- several novel feature classes,
- direct comparison against the two best disfluency pre-processors, and
- state-of-the-art accuracy for both speech parsing and disfluency detection.

2 Switchboard Disfluency Annotations

The Switchboard portion of the Penn Treebank (Marcus et al., 1993) consists of telephone conversations between strangers about an assigned topic. Two annotation layers are provided: one for syntactic bracketing (MRG files), and one for disfluencies (DPS files). The disfluency layer marks elements with little or no syntactic function, such as filled pauses and discourse markers, and annotates speech repairs using the Shriberg (1994) system of reparandum/interregnum/repair. An example is shown in Figure 1.

In the syntactic annotation, edited words are covered by a special node labelled EDITED. The idea is to mark text which, if excised, would result in a grammatical sentence. The MRG files do not mark other types of disfluencies. We follow the evaluation defined by Charniak and Johnson (2001), which evaluates the accuracy of identifying speech repairs and restarts. This definition of the task is the standard in recent work. The reason for this is that filled pauses can be detected using a simple rule-based approach, and parentheticals have less impact on readability and down-stream processing accuracy.

The MRG and DPS layers have high but imperfect agreement over what tokens they mark as speech repairs: of the text annotated with both layers, 33,720 tokens are marked as disfluent in at least one layer, 32,310 are only marked as disfluent by the DPS files, and 32,742 are only marked as disfluent by the MRG layer.

The Switchboard annotation project was not fully completed. Because disfluency annotation is cheaper to produce, many of the DPS training files do not have matching MRG files. Only 619,236 of the 1,482,845 tokens in the DPS disfluency-detection training data have gold-standard syntactic parses. Our system requires the more expensive syntactic annotation, but we find that it outperforms the previous state-of-the-art (Qian and Liu, 2013), despite training on less than half the data.

2.1 Dependency Conversion

As is standard in statistical dependency parsing of English, we acquire our gold-standard dependencies from phrase-structure trees. We used the 2013-04-05 version of the Stanford dependency converter (de Marneffe et al., 2006). As is standard for English dependency parsing, we use the Basic Dependencies scheme, which produces strictly projective representations.

At first we feared that the filled pauses, disfluencies and meta-data tokens in the Switchboard corpus might disrupt the conversion process, by making it more difficult for the converter to recognise the underlying production rules.

To test this, we performed a small experiment. We prepared two versions of the corpus: one where EDITED nodes, filled pauses and meta-data were removed before the trees were transformed by the Stanford converter, and one where the disfluency removal was performed after the dependency conversion. The resulting corpora were largely identical: 99.54% of unlabelled and 98.7% of labelled dependencies were the same. The fact that the Stanford converter is quite robust to disfluencies was useful for our baseline joint model, which is trained on dependency trees that also included governors for disfluent words.

We follow previous work on disfluency detection by lower-casing the text and removing punctuation and partial words (words tagged XX and words ending in ‘-’). We also remove one-token sentences, as their syntactic analyses are trivial. We found that two additional simple pre-processes improved our results: discarding all ‘um’ and ‘uh’ tokens; and merging ‘you know’ and ‘i mean’ into single tokens.

These pre-processes can be completed on the input string without losing information: none of the ‘um’ or ‘uh’ tokens are semantically significant, and the bigrams *you know* and *i mean* have a dependency between the two tokens over 99.9% of the times they occur in the treebank, with *you* and *I* never having any children. This makes it easy to unmerge the tokens deterministically after parsing: all incoming and outgoing arcs will point to *know* or *mean*. The same pre-processing was performed for all our parsing systems.

3 Transition-based Dependency Parsing

A transition-based parser predicts the syntactic structure of a sentence incrementally, by making a sequence of classification decisions. We follow the architecture of Zhang and Clark (2011), who use beam-search for decoding, and a structured averaged perceptron for training. Despite its simplicity, this type of parser has produced highly competitive results on the Wall Street Journal: with the extended feature set described by Zhang and Nivre (2011), it achieves 93.5% unlabelled accuracy on Stanford basic dependencies (de Marneffe et al., 2006). Converting the constituency trees produced by the Charniak and Johnson (2005) reranking parser results in similar accuracy.

Briefly, the transition-based parser consists of a configuration (or 'state') which is sequentially manipulated by a set of possible transitions. For us, a state is a 4-tuple $c = (\sigma, \beta, A, D)$, where σ and β are disjoint sets of word indices termed the *stack* and *buffer* respectively, A is the set of dependency arcs, and D is the set of word indices marked disfluent. There are no arcs to or from members of D, so the dependencies and disfluencies can be implemented as a single vector (in our parser, a token is marked as disfluent by setting it as its own head).

We use the arc-eager transition system (Nivre, 2003, 2008), which consists of four parsing actions: **S**hift, **L**eft-Arc, **R**ight-Arc and Re**d**uce. We denote the stack with its topmost element to the right, and the buffer with its first element to the left. A vertical bar is used to indicate concatenation to the stack or buffer, e.g. $\sigma|i$ indicates a stack with the topmost element i and remaining elements σ. A dependency from a governor i to a child j is denoted $i \rightarrow j$. The four arc-eager transitions are shown in Figure 2.

The Shift action moves the first item of the buffer onto the stack. The Right-Arc does the same, but also adds an arc, so that the top two items on the stack are connected. The Reduce move and the Left-Arc both pop the stack, but the Left-Arc first adds an arc from the first word of the buffer to the word on top of the stack. Constraints on the Reduce and Left-Arc moves ensure that every word is assigned exactly one head in the final configuration. We follow the suggestion

$(\sigma, i|\beta, A, D) \vdash (\sigma|i, \beta, A, D)$ S

$(\sigma|i, j|\beta, A, D) \vdash (\sigma, j|\beta, A \cup \{j \rightarrow i\}, D)$ L

Only if i does not have an incoming arc.

$(\sigma|i, j|\beta, A, D) \vdash (\sigma|i|j, \beta, A \cup \{i \rightarrow j\}, D)$ R

$(\sigma|i, \beta, A, D) \vdash (\sigma, \beta, A, D)$ D

Only if i has an incoming arc.

$(\sigma|i, j|\beta, A, D) \vdash (\sigma|[x_1, x_n], j|\beta, A', D')$ E

Where

$A' = A \setminus \{x \rightarrow y \text{ or } y \rightarrow x : \forall x \in [i, j), \forall y \in \mathbb{N}\}$

$D' = D \cup [i, j)$

$x_1 ... x_n$ are the former left children of i

Figure 2: Our parser's transition system. The first four transitions are the standard arc-eager system; the fifth is our novel Edit transition.

of Ballesteros and Nivre (2013) and add a dummy token that governs root dependencies to the end of the sentence. Parsing terminates when this token is at the start of the buffer, and the stack is empty. Disfluencies are added to D via the Edit transition, E, which we now define.

4 A Non-Monotonic Edit Transition

One of the reasons disfluent sentences are hard to parse is that there often appear to be syntactic relationships between words in the reparandum and the fluent sentence. When these relations are considered in addition to the dependencies between fluent words, the resulting structure is not necessarily a projective tree.

Figure 3 shows a simple example, where the repair *square* replaces the reparandum *rectangle*. An incremental parser could easily become 'garden-pathed' and attach the repair *square* to the preceding words, constructing the dependencies shown dotted in Figure 3. Rather than attempting to devise an incremental model that avoids constructing such dependencies, we allow the parser to construct these dependencies and later delete them if the governor or child are marked disfluent.

Psycholinguistic models of human sentence processing have long posited *repair* mechanisms (Frazier and Rayner, 1982). Recently, Honnibal et al. (2013) showed that a limited amount of 'non-monotonic' behaviour can improve an incremental parser's accuracy. We here introduce a non-monotonic transition, Edit, for speech repairs.

The Edit transition marks the word i on top of the stack $\sigma|i$ as disfluent, along with its rightward descendents — i.e., all words in the sequence $i...j - 1$, where j is the word at the start of the buffer. It then restores the words both preceding and formerly governed by i to the stack.

In other words, the word on top of the stack and

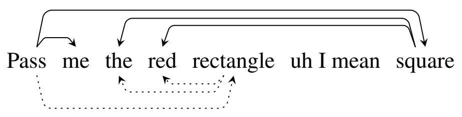

Figure 3: Example where apparent dependencies between the reparandum and the fluent sentence complicate parsing. The dotted edges are difficult for an incremental parser to avoid, but cannot be part of the final parse if it is to be a projective tree. Our solution is to make the transition system non-monotonic: the parser is able to delete edges.

its *rightward descendents* are all marked as disfluent, and the stack is popped. We then restore its leftward children to the stack, and all dependencies to and from words marked disfluent are deleted. The transition is *non-monotonic* in the sense that it can delete dependencies created by a previous transition, and replace tokens onto the stack that had been popped.

Why revisit the leftward children, but not the right? We are concerned about dependencies which might be mirrored between the reparandum and the repair. The rightward subtree of the disfluency might well be incorrect, but if it is, it would still be incorrect if the word on top of the stack were actually fluent. We therefore regard these as parsing errors that we will train our model to avoid. In contrast, avoiding the Left-Arc transitions would require the parser to predict that the head is disfluent when it has not necessarily seen any evidence indicating that.

4.1 Worked Example

Figure 4 shows a gold-standard derivation for a disfluent sentence from the development data. Line 1 shows the state resulting from the initial Shift action. In the next three states, *His* is Left-Arced to *company*, which is then Shifted onto the stack, and Left-Arced to *went* in Line 4.

The dependency between *went* and *company* is not part of the gold-standard, because *went* is disfluent. The correct governor of *company* is the second *went* in the sentence. The Left-Arc move in Line 4 can still be considered correct, however, because the gold-standard analysis is still derivable from the resulting configuration, via the Edit transition. Another non-gold dependency is created in Line 6, between *broke* and *went*, before *broke* is Reduced from the stack in Line 7.

Lines 9 and 10 show the states before and after the Edit transition. The word on top of the stack in Line 9, *went*, has one leftward child, and one right-

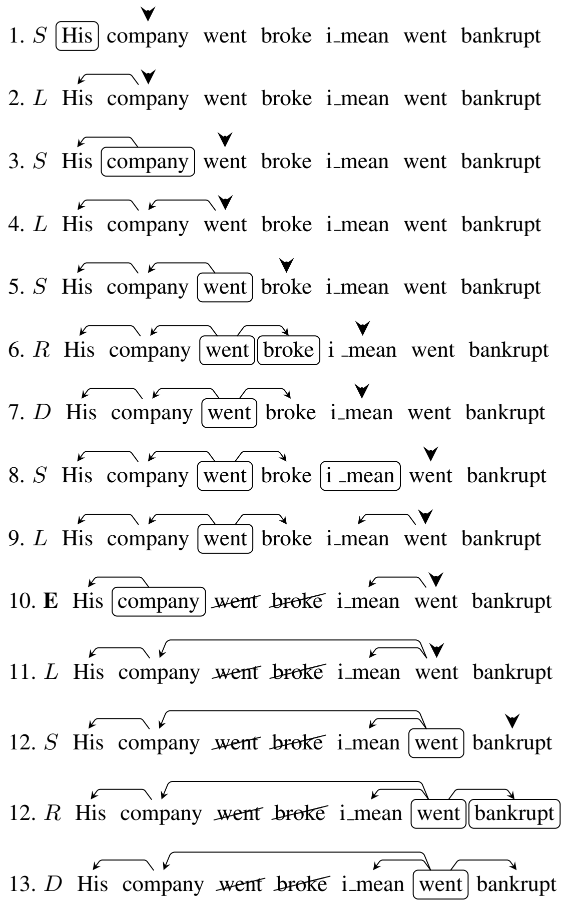
1. S His company went broke i_mean went bankrupt
2. L His company went broke i_mean went bankrupt
3. S His company went broke i_mean went bankrupt
4. L His company went broke i_mean went bankrupt
5. S His company went broke i_mean went bankrupt
6. R His company went broke i_mean went bankrupt
7. D His company went broke i_mean went bankrupt
8. S His company went broke i_mean went bankrupt
9. L His company went broke i_mean went bankrupt
10. E His company ~~went~~ ~~broke~~ i_mean went bankrupt
11. L His company ~~went~~ ~~broke~~ i_mean went bankrupt
12. S His company ~~went~~ ~~broke~~ i_mean went bankrupt
12. R His company ~~went~~ ~~broke~~ i_mean went bankrupt
13. D His company ~~went~~ ~~broke~~ i_mean went bankrupt

Figure 4: A gold-standard transition sequence using our EDIT transition. Each line specifies an action and shows the state resulting from it. Words on the stack are circled, and the arrow indicates the start of the buffer. Disfluent words are struck-through.

ward child. After the Edit transition is applied, *went* and its rightward child *broke* are both marked disfluent, and *company* is returned to the stack. All of the previous dependencies to and from *went* and *broke* are deleted.

Parsing then proceeds as normal, with the correct governor of *company* being assigned by the Left-Arc in Line 11, and *bankrupt* being Right-Arced to *went* in Line 12. To conserve space, we have omitted the dummy ROOT token, which is placed at the end of the sentence, following the suggestion of Ballesteros and Nivre (2013). The final action will be a Left-Arc from the ROOT token to *went*.

4.2 Dynamic Oracle Training Algorithm

Our non-monotonic transition system introduces substantial *spurious ambiguity*: the gold-standard parse can be derived via many different transition sequences. Recent work has shown that this can be advantageous (Sartorio et al., 2013; Honnibal et al., 2013; Goldberg and Nivre, 2012), because difficult decisions can sometimes be delayed until more information is available.

Line 5 of Figure 4 shows a state that introduces spurious ambiguity. From this configuration, there are multiple actions that could be considered 'correct', in the sense that the gold-standard analysis can be derived from them. The Edit transition is correct because *went* is disfluent, but the Left-Arc and even the Right-Arc are also correct, in that there are continuations from them that lead to the gold-standard analysis.

We regard all transition sequences that can result in the correct analysis as equally valid, and want to avoid stipulating one of them during training. We achieve this by following Goldberg and Nivre (2012) in using a *dynamic oracle* to create partially labelled training data.[2] A dynamic oracle is a function that determines the cost of applying an action to a state, in terms of gold-standard arcs that are newly unreachable.

We follow Collins (2002) in training an averaged perceptron model to predict transition *sequences*, rather than individual transitions. This type of model is often referred to as a structured perceptron, or sometimes a global perceptron. During training, if the model does not predict the correct sequence, an update is performed, based on the gold-standard sequence and part of the sequence predicted by the current weights. Only part of the sequence is used to calculate the weight update, in order to account for search errors. We use the maximum violation strategy described by Huang et al. (2012) to select the subsequence to update from.

To train our model using the dynamic oracle, we use the latent-variable structured perceptron algorithm described by Sun et al. (2009). Beam-search is performed to find the highest-scoring gold-standard sequence, as well as the highest-scoring prediction. We use the same beam-width for both search procedures.

4.3 Path Length Normalisation

One problem introduced by the Edit transition is that the number of actions applied to a sentence is no longer constant — it is no longer guaranteed to be $2n - 1$, for a sentence of length n. When the Edit transition is applied to a word with leftward children, those children are returned to the stack, and processed again. This has little to no impact on the algorithm's empirical efficiency, although worst-case complexity is no longer linear, but it does pose a problem for decoding.

The perceptron model tends to assign large positive scores to its top prediction. We thus observed a problem when comparing paths of different lengths, at the end of the sentence. Paths that included Edit transitions were longer, so the sum of their scores tended to be higher.

The same problem has been observed during incremental PCFG parsing, by Zhu et al. (2013). They introduce an additional transition, IDLE, to ensure that paths are the same length. So long as one candidate in the beam is still being processed, all other candidates apply the IDLE transition.

We adopt a simpler solution. We normalise the figure-of-merit for a candidate state, which is used to rank it in the beam, by the length of its transition history. The new figure-of-merit is the arithmetic mean of the candidate's transition scores, where previously the figure-of-merit was the sum of the candidate's transition scores.

Interestingly, Zhu et al. (2013) report that they tried exactly this, and that it was less effective than their solution. We found that the features associated with the IDLE transition were uninformative (the state is at termination, so the stack and buffer are empty), and had nothing to do with how many edit transitions were earlier applied.

5 Features for the Joint Parser

Our baseline parser uses the feature set described by Zhang and Nivre (2011). The feature set contains 73 templates that mostly refer to the properties of 12 *context tokens*: the top of the stack (S0), its two leftmost and rightmost children ($S0_L$, $S0_{L2}$, $S0_R$, $S0_{R2}$), its parent and grand-parent ($S0_h$, $S0_{h2}$), the first word of the buffer and its two leftmost children (N0, $N0_L$, $N0_{LL}$), and the next two words of the buffer (N1, N2).

Atomic features consist of the word, part-of-speech tag, or dependency label for these tokens; and multiple feature atoms are often combined for feature templates. There are also features for the string-distance between S0 and N0, and the left and right valencies (total number of children) of

[2] The training data is partially labelled in the sense that instances can have multiple true labels. Equivalently, one might say that the transitions are latent variables, which generate the dependencies.

S0 and N0, as well as the set of their children's dependency labels. We restrict these to the first and last 2 children for implementation efficiency, as we found this had no effect on accuracy. Numeric features (for distance and valency) are binned with the function $\lambda x : \min(x, 5)$. There is only one bilexical feature template, which pairs the words of S0 and N0. There are also ten tri-tag templates.

Our feature set includes additional dependency label features not used by Zhang and Nivre (2011), as we found that disfluency detection errors often resulted in ungrammatical dependency label combinations. The additional templates combine the POS tag of S0 with two or three dependency labels from its left and right subtrees. Details can be found in the supplementary material.

5.1 Brown Cluster Features

The Brown clustering algorithm (Brown et al., 1992) is a well known source of semi-supervised features. The clustering algorithm is run over a large sample of unlabelled data, to generate a type-to-cluster map. This mapping is then used to generate features that sometimes generalise better than lexical features, and are helpful for out-of-vocabulary words (Turian et al., 2010).

Koo and Collins (2010) found that Brown cluster features greatly improved the performance of a graph-based dependency parser. On our transition-based parser, Brown cluster features bring a small but statistically significant improvement on the WSJ task (0.1-0.3% UAS). Other developers of transition-based parsers seem to have found similar results (personal communication). Since a Brown cluster mapping computed by Liang (2005) is easily available,[3] the features are simple to implement and cheap to compute.

Our templates follow Koo and Collins (2010) in including features that refer to cluster prefix strings, as well as the full clusters. We adapt their templates to transition-based parsing by replacing 'head' with 'item on top of the stack' and 'child' with 'first word of the buffer'. The exact templates can be found in the supplementary material.

The Brown cluster features are used in our 'baseline' parser, and in the parsers we use as part of our pipeline systems. They improved development set accuracy by 0.4%. We experimented with the other feature sets in these parsers, but found that they did not improve accuracy on fluent text.

[3] `http://www.metaoptimize.com/projects/wordreps`

5.2 Rough Copy Features

Johnson and Charniak (2004) point out that in speech repairs, the repair is often a 'rough copy' of the reparandum. The simplest case of this is where the repair is a single word repetition. It is common for the repair to differ from the reparandum by insertion, deletion or substitution of one or more words.

To capture this regularity, we first extend the feature-set with three new context tokens:[4]

1. S0_{re}: The rightmost edge of S0 descendants;
2. S0_{le}: The leftmost edge of S0 descendants;
3. N0_{le}: The leftmost edge of N0 descendants.

If a word has no leftward children, it will be its own left-edge, and similarly it will be its own rightward edge if it has no rightward children. Note that the token S0_{re} is necessarily immediately before N0_{le}, unless some of the tokens between them are disfluent. We use the S0_{le} and N0_{le} to compute the following rough-copy features:

1. How long is the *prefix word match* between S0_{le}...S0 and N0_{le}...N0?

   If the parser were analysing *the red the blue square*, with *red* on the stack and *square* at N0, its value would be 1.
2. How long is the *prefix POS tag match* between S0_{le}...S0 and N0_{le}...N0?
3. Do the words in S0_{le}...S0 and N0_{le}...N0 match exactly?
4. Do the POS tags in S0_{le}...S0 and N0_{le}...N0 match exactly?

   If the parser were analysing *the red square the blue rectangle*, with *square* on the stack and *rectangle* at N0, its value would be *true*.

The prefix-length features are binned using the function $\lambda x : \min(x, 5)$.

5.3 Match Features

This class of features ask which pairs of the *context tokens* match, in word or POS tag. The context tokens in the Zhang and Nivre (2011) feature set are the top of the stack (S0), its head and

[4] As is common in this type of parser, our implementation has a number of vectors for properties that are defined before parsing, such as word forms, POS tags, Brown clusters, etc. A context token is an index into these vectors, allowing features considering these properties to be computed.

grandparent (S0_h, S0_{h2}), its two left- and rightmost children (S0_L, S0_{L2}, S0_R, S0_{R2}), the first three words of the buffer (N0, N1, N2), and the two leftmost children of N0 (N0_L, N0_{LL}). We extend this set with the S0_{le}, S0_{re} and N0_{le} tokens described above, and also the first left and right child of S0 and N0 (S0_{L0}, S0_{R0}, N0_{L0}).

All up, there are 18 context tokens, so $\binom{18}{2} = 153$ token pairs. For each pair of these tokens, we add two binary features, indicating whether the two tokens match in word form or POS tag. We also have two further classes of features: if the words do match, a feature is added indicating the word form; if the tags match, a feature is added indicating the tag. These finer grained versions help the model adjust for the fact that some words can be duplicated in grammatical sentences (e.g. 'that that'), while most rare words cannot.

5.4 Edited Neighbour Features

Disfluencies are usually string contiguous, even if they do not form a single constituent. In these situations, our model has to make multiple transitions to mark a single disfluency. For instance, if an utterance begins *and the and a*, the stack will contain two entries, for *and* and *the*, and two Edit transitions will be required.

To mitigate this disadvantage of our model, we add four binary features. Two fire when the word or pair of words immediately preceding N0 have been marked disfluent; the other two fire when the word or pair of words immediately following S0 have been marked disfluent. These features provide an additional string-based view that the parser would otherwise be missing. Speakers tend be disfluent in bursts: if the previous word is disfluent, the next word is more likely to be disfluent. These four features are therefore all associated with positive weights for the Edit transition. Without these features, we would miss an aspect of disfluency processing that sequence models naturally capture.

6 Part-of-Speech Tagging

We adopt the standard strategy of using a POS tagger as a pre-process before parsing. Most transition-based parsers use a structured averaged perceptron model with beam-search for tagging, as this model achieves competitive accuracy and matches the standard dependency parsing architecture. Our tagger also uses this architecture.

We performed some additional feature engineering for the tagger, in order to improve its accuracy given the lack of case distinctions and punctuation in the data. Our additional features use two sources of unsupervised information. First, we follow the suggestion of Manning (2011) by using Brown cluster features to improve the tagger's accuracy on unknown words. Second, we compensate for the lack of case distinctions by including features that ask what percentage of the time a word form was seen title-cased, upper-cased and lower-cased in the Google Web1T corpus.

Where most previous work uses cross-fold training for the tagger, to ensure that the parser is trained on tags that reflect run-time accuracies, we do online training of the tagger alongside the parser, using the current tagger model to produce tags during parser training. This had no impact on parse accuracy, and made it slightly easier to develop our tagger alongside the parser.

The tagger achieved 96.5% accuracy on the development data, but when we ran our final test experiments, we found its accuracy dropped to 96.0%, indicating some over-fitting during our feature engineering. On the development data, our parser accuracy improves by about 1% when gold-standard tags are used.

7 Experiments

We use the Switchboard portion of the Penn Treebank (Marcus et al., 1993), as described in Section 2, to train our joint models and evaluate them on dependency parsing and disfluency detection. The pre-processing and dependency conversion are described in Section 2.1. We use the standard train/dev/test split from Charniak and Johnson (2001): Sections 2 and 3 for training, and Section 4 divided into three held-out sections, the first of which is used for final evaluation.

Our parser evaluation uses the SPARSEVAL (Roark et al., 2006) metric. However, we wanted to use the Stanford dependency converter, for the reasons described in Section 2.1, so we used our own implementation. Because we do not need to deal with recognition errors, we do not need to report our parsing results using $P/R/F$-measures. Instead, we report an unlabelled accuracy score, which refers to the percentage of fluent words whose governors were assigned correctly. Note that words marked as disfluent cannot have any incoming or out-going dependencies, so if a word is incorrectly marked as disfluent, all of its dependencies will be incorrect.

We follow Johnson and Charniak (2004) and others in restricting our disfluency evaluation to speech repairs, which we identify as words that have a node labelled EDITED as an ancestor. Unlike most other disfluency detection research, we train only on the MRG files, giving us 619,236 words of training data instead of the 1,482,845 used by the pipeline systems. It may be possible to improve our system's disfluency detection by leveraging the additional data that does not have syntactic annotation in some way.

All parsing models were trained for 15 iterations. We found that optimising the number of iterations on a development set led to small improvements that did not transfer to a second development set (part of Section 4, which Charniak and Johnson (2001) reserved for 'future use').

We test for statistical significance in our results by training 20 models for each experimental configuration, using different random seeds. The random seeds control how the sentences are shuffled during training, which the perceptron model is quite sensitive to. We use the Wilcoxon rank-sums non-parametric test. The standard deviation in UAS for a sample was typically around 0.05%, and 0.5% for disfluency F-measure.

All of our models use beam-search decoding, with a beam width of 32. We found that a beam width of 64 brought a very small accuracy improvement (about 0.1%), at the cost of 50% slower run-time. Wider beams than this brought no accuracy improvement. Accuracy seems to plateau with slightly narrower beams than on newswire text. This is probably due to the shorter sentences in Switchboard.

The baseline and pipeline systems are configured in the same way, except that the baseline parser is modified slightly to allow it to predict disfluencies, using a special dependency label, ERASED. All descendants of a word attached to its head by this label are marked as disfluent. Both the baseline and pipeline/oracle parsers use the same feature set: the Zhang and Nivre (2011) features, plus our Brown cluster features.

The baseline system is a standard arc-eager transition-based parser with a structured averaged perceptron model and beam-search decoding. The model is trained in the standard way, with a 'static' oracle and maximum-violation update, following (Huang et al., 2012).

7.1 Comparison with Pipeline Approaches

The accuracy of incremental dependency parsers is well established on the Wall Street Journal, but there are no dependency parsing results in the literature that make it easy to put our joint model's parsing accuracy into context. We therefore compare our joint model to two pipeline systems, which consist of a disfluency detector, followed by our dependency parser. We also evaluate parse accuracies after oracle pre-processing, to gauge the net effect of disfluencies on our parser's accuracy.

The dependency parser for the pipeline systems was trained on text with all disfluencies removed, following Charniak and Johnson (2001). The two disfluency detection systems we used were the Qian and Liu (2013) sequence-tagging model, and a version of the Johnson and Charniak (2004) noisy channel model, using the Charniak (2001) syntactic language model and the reranking features of Zwarts and Johnson (2011). They are the two best published disfluency detection systems.

8 Results

Table 1 shows the development set accuracies for our joint parser. Both the disfluency features and the Edit transition make statistically significant improvements, in both disfluency F-measure, unlabelled attachment score (UAS), and labelled attachment score (LAS).

The **Oracle pipeline** system, which uses the gold-standard to clean disfluencies prior to parsing, shows the total impact of speech-errors on the parser. The baseline parser, which uses the Zhang and Nivre (2011) feature set plus the Brown cluster features, scores 1.8% UAS lower than the oracle.

When we add the features described in Sections 5.2, 5.3 and 5.4, the gap is reduced to 1.2% (**+Features**). Finally, the improved transition system reduces the gap further still, to 0.8% UAS (**+Edit transition**). We also tested these features in the Oracle parser, but found they were ineffective on fluent text.

The **w/s** column shows the tokens analysed per second for each system, including disfluencies, with a single thread on a 2.4GHz Intel Xeon. The additional features reduce efficiency, but the non-monotonic Edit transition does not. The system is easily efficient enough for real-time use.

	P	R	F	UAS	LAS	w/s
Baseline joint	79.4	70.1	74.5	89.9	86.9	711
+Features	86.0	77.2	81.3	90.5	87.5	539
+Edit transition	92.2	80.2	85.8	90.9	87.9	555
Oracle pipeline	100	100	100	91.7	88.6	782

Table 1: Development results for the joint models. For the baseline model, disfluencies reduce parse accuracy by 1.7% Unlabelled Attachment Score (UAS). Our features and Edit transition reduce the gap to 0.7%, and improve disfluency detection by 11.3% F-measure.

	Disfl. F	UAS
Johnson et al pipeline	82.1	90.3
Qian and Liu pipeline	83.9	90.1
Baseline joint parser	73.9	89.4
Final joint parser	84.1	**90.5**

Table 2: Test-set parse and disfluency accuracies. The joint parser is improved by the features and Edit transition, and is better than pre-processing the text with state-of-the-art disfluency detectors.

Table 2 shows the final evaluation. Our main comparison is with the two pipeline systems, described in Section 7.1. The Johnson and Charniak (2004) system was 1.8% less accurate at disfluency detection than the other disfluency detector we evaluated, the state-of-the-art Qian and Liu (2013) system. However, when we evaluated the two systems as pre-processors before our parser, we found that the **Johnson et al pipeline** achieved 0.2% better unlabelled attachment score than the **Qian and Liu pipeline**. We attribute this to the use of the Charniak and Johnson (2001) syntactic language model in the Johnson et al pipeline, which would help the system produce more syntactically consistent output.

Our joint model achieved an unlabelled attachment score of 90.5%, out-performing both pipeline systems. The **Baseline joint parser**, which did not include the Edit transition or disfluency features, scores 1.1% below the **Final joint parser**. All of the parse accuracy differences were found to be statistically significant ($p < 0.001$).

The Edit transition and disfluency features together brought a 10.1% improvement in disfluency F-measure, which was also found to be statistically significant. The final joint parser achieved 0.2% higher disfluency detection accuracy than the previous state-of-the-art, the Qian and Liu (2013) system,[5] despite having approximately half as much training data (we require syntactic annotation, for which there is less data).

[5] Our scores refer to an updated version of the system that corrects minor pre-processing problems. We thank Qian Xian for making his code available.

Our significance testing regime involved using 20 different random seeds when training each of our models, which the perceptron algorithm is sensitive to. This could not be applied to the other two disfluency detectors, so we cannot test those differences for significance. However, we note that the 20 samples for our disfluency detector ranged in accuracy from 83.3-84.6, so we doubt that 0.2% mean improvement over the Qian and Liu (2013) result is meaningful.

Although we did not systematically optimise on the development set, our test scores are lower than our development accuracies. Much of the over-fitting seems to be in the POS tagger, which dropped in accuracy by 0.5%.

9 Analysis of Edit Behaviour

In order to understand how the parser applies the Edit transition, we collected some additional statistics over the development data. The parser predicted 2,558 Edit transitions, which together marked 2,706 words disfluent (2,495 correctly). The Edit transition can mark multiple words disfluent when S0 has one or more rightward descendants. It turns out this case is uncommon; the parser largely assigns disfluency labels word-by-word, only sometimes marking words with rightward descendents as disfluent.

Of the 2,558 Edit transitions, there were 682 cases were at least one leftward child was returned to the stack, and the total number of leftward children returned was 1,132. The most common type of construction that caused the parser to return words to the stack were disfluent predicates, which often have subjects and discourse conjunctions as leftward children. An example of a disfluent predicate with a fluent subject is shown in Figure 4.

There were only 48 cases of the same word being returned to the stack twice. The possibility of words being returned to the stack multiple times is what gives our system worse than linear worst-case complexity. In the worst case, the ith word of a sentence of length n could be returned to the stack $n - (i + 1)$ times. Empirically, the Edit transition made no difference to run-time.

Once a word has been returned to the stack by the Edit transition, how does the parser end up analysing it? If it turned out that almost all of the former leftward children of disfluent words are subsequently marked as disfluent, there would be little point in returning them to the stack — we could just mark them as disfluent in the original Edit transition. On the other hand, if they are almost all marked as fluent, perhaps they can just be attached as children to the first word of the buffer.

In fact the two cases are almost equally common. Of the 1,132 words returned to the stack, 547 were subsequently marked disfluent, and 584 were not. The parser was also quite accurate in its decisions over these tokens. Of the 547 tokens marked disfluent, 500 were correct — similar to the overall development set precision, 92.2%.

Accuracy over the words returned to the stack might be improved in future by features referring to their former heads. For instance, in *He went broke uh became bankrupt*, we do not currently have features that record the deleted dependency became *he* and *went*. We thank one of the anonymous reviewers for this suggestion.

10 Related Work

The most similar system to ours was published very recently. Rasooli and Tetreault (2013) describe a joint model of dependency parsing and disfluency detection. They introduce a second classification step, where they first decide whether to apply a disfluency transition, or a regular parsing move. Disfluency transitions operate either over a sequence of words before the start of the buffer, or a sequence of words from the start of the buffer forward. Instead of the dynamic oracle training method that we employ, they use a two-stage bootstrap-style process.

Direct comparison between our model and theirs is difficult, as they use the Penn2MALT scheme, and their parser uses greedy decoding, where we use beam search. They also use gold-standard part-of-speech tags, which would improve our scores by around 1%. The use of beam-search may explain much of our performance advantage: they report an unlabelled attachment score of 88.6, and a disfluency detection F-measure of 81.4%. Our training algorithm would be applicable to a beam-search version of their parser, as their transition-system also introduces substantial spurious ambiguity, and some non-monotonic behaviour.

A hybrid transition system would also be possible, as the two types of Edit transition seem to be complementary. The Rasooli and Tetreault system offers a token-based view of disfluencies, which is useful for examples such as, *and the and the*, which would require two applications of our transition. On the other hand, our Edit transition may have the advantage for more syntactically complicated examples, particularly for disfluent verbs.

The importance of syntactic features for disfluency detection was demonstrated by Johnson and Charniak (2004). Despite this, most subsequent work has used sequence models, rather than syntactic parsers. The other disfluency system that we compare our model to, developed by Qian and Liu (2013), uses a cascade of Maximum Margin Markov Models to perform disfluency detection with minimal syntactic information.

One motivation for sequential approaches is that most applications of these models will be over unsegmented text, as segmenting unpunctuated text is a difficult task that benefits from syntactic features (Zhang et al., 2013).

We consider the most promising aspect of our system to be that it is naturally incremental, so it should be straightforward to extend the system to operate on unsegmented text in subsequent work. Due to its use of syntactic features, from the joint model, the system is substantially more accurate than the previous state-of-the-art in incremental disfluency detection, 77% (Zwarts et al., 2010).

11 Conclusion

We have presented an efficient and accurate joint model of dependency parsing and disfluency detection. The model out-performs pipeline approaches using state-of-the-art disfluency detectors, and is highly efficient, processing over 550 tokens a second. Because the system is incremental, it should be straight-forward to apply it to unsegmented text. The success of an incremental, non-monotonic parser at disfluent speech parsing may also be of some psycholinguistic interest.

Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable comments. This research was supported under the Australian Research Council's Discovery Projects funding scheme (project numbers DP110102506 and DP110102593).

References

Miguel Ballesteros and Joakim Nivre. 2013. Going to the roots of dependency parsing. *Computational Linguistics. 39:1.*

Peter F. Brown, Peter V. deSouza, Robert L. Mercer, Vincent J. Della Pietra, and Jenifer C. Lai. 1992. Class-based n-gram models of natural language. *Computational Linguistics*, 18:467–479.

Eugene Charniak. 2001. Immediate-head parsing for language models. In *Proceedings of 39th Annual Meeting of the Association for Computational Linguistics*, pages 124–131. Association for Computational Linguistics, Toulouse, France.

Eugene Charniak and Mark Johnson. 2001. Edit detection and parsing for transcribed speech. In *Proceedings of the 2nd Meeting of the North American Chapter of the Association for Computational Linguistics*, pages 118–126. The Association for Computational Linguistics.

Eugene Charniak and Mark Johnson. 2005. Coarse-to-fine n-best parsing and MaxEnt discriminative reranking. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics*, pages 173–180. Association for Computational Linguistics, Ann Arbor, Michigan.

Michael Collins. 2002. Discriminative training methods for hidden Markov models: Theory and experiments with perceptron algorithms. In *Proceedings of the 2002 Conference on Empirical Methods in Natural Language Processing*, pages 1–8. Association for Computational Linguistics.

Marie-Catherine de Marneffe, Bill MacCartney, and Christopher D. Manning. 2006. Generating typed dependency parses from phrase structure parses. In *Proceedings of the 5th International Conference on Language Resources and Evaluation (LREC)*.

Lyn Frazier and Keith Rayner. 1982. Making and correcting errors during sentence comprehension: Eye movements in the analysis of structurally ambiguous sentences. *Cognitive Psychology*, 14(2):178–210.

Yoav Goldberg and Joakim Nivre. 2012. A dynamic oracle for arc-eager dependency parsing. In *Proceedings of the 24th International Conference on Computational Linguistics (Coling 2012)*. Association for Computational Linguistics, Mumbai, India.

Matthew Honnibal, Yoav Goldberg, and Mark Johnson. 2013. A non-monotonic arc-eager transition system for dependency parsing. In *Proceedings of the Seventeenth Conference on Computational Natural Language Learning*, pages 163–172. Association for Computational Linguistics, Sofia, Bulgaria.

Liang Huang, Suphan Fayong, and Yang Guo. 2012. Structured perceptron with inexact search. In *Proceedings of the 2012 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 142–151. Association for Computational Linguistics, Montréal, Canada.

Mark Johnson and Eugene Charniak. 2004. A TAG-based noisy channel model of speech repairs. In *Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics*, pages 33–39.

Douglas A. Jones, Florian Wolf, Edward Gibson, Elliott Williams, Evelina Fedorenko, Douglas A. Reynolds, and Marc A. Zissman. 2003. Measuring the readability of automatic speech-to-text transcripts. In *INTERSPEECH*. ISCA.

Fredrik Jorgensen. 2007. The effects of disfluency detection in parsing spoken language. In Joakim Nivre, Heiki-Jaan Kaalep, Kadri Muischnek, and Mare Koit, editors, *Proceedings of the 16th Nordic Conference of Computational Linguistics NODALIDA-2007*, pages 240–244.

Terry Koo and Michael Collins. 2010. Efficient third-order dependency parsers. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 1–11.

Percy Liang. 2005. *Semi-supervised learning for natural language*. Ph.D. thesis, MIT.

Christopher D. Manning. 2011. Part-of-speech tagging from 97linguistics? In *Proceedings of the 12th international conference on Computational linguistics and intelligent text processing - Volume Part I*, CICLing'11, pages 171–189. Springer-Verlag, Berlin, Heidelberg.

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

Joakim Nivre. 2003. An efficient algorithm for projective dependency parsing. In *Proceedings* *of the 8th International Workshop on Parsing Technologies (IWPT)*, pages 149–160.

Joakim Nivre. 2008. Algorithms for deterministic incremental dependency parsing. *Computational Linguistics*, 34:513–553.

Xian Qian and Yang Liu. 2013. Disfluency detection using multi-step stacked learning. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 820–825. Association for Computational Linguistics, Atlanta, Georgia.

Mohammad Sadegh Rasooli and Joel Tetreault. 2013. Joint parsing and disfluency detection in linear time. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 124–129. Association for Computational Linguistics, Seattle, Washington, USA.

Brian Roark, Mary Harper, Eugene Charniak, Bonnie Dorr, Mark Johnson, Jeremy Kahn, Yang Liu, Mary Ostendorf, John Hale, Anna Krasnyanskaya, Matthew Lease, Izhak Shafran, Matthew Snover, Robin Stewart, and LisaYung. 2006. Sparseval: Evaluation metrics for parsing speech. In *Proceedings of Language Resource and Evaluation Conference*, pages 333–338. European Language Resources Association (ELRA), Genoa, Italy.

Francesco Sartorio, Giorgio Satta, and Joakim Nivre. 2013. A transition-based dependency parser using a dynamic parsing strategy. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics*, pages 135–144. Association for Computational Linguistics, Sofia, Bulgaria.

Elizabeth Shriberg. 1994. *Preliminaries to a Theory of Speech Disfluencies*. Ph.D. thesis, University of California, Berkeley.

Xu Sun, Takuya Matsuzaki, Daisuke Okanohara, and Jun'ichi Tsujii. 2009. Latent variable perceptron algorithm for structured classification. In *IJCAI*, pages 1236–1242.

Joseph Turian, Lev-Arie Ratinov, and Yoshua Bengio. 2010. Word representations: A simple and general method for semi-supervised learning. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, pages 384–394. Association for Computational Linguistics, Uppsala, Sweden.

Dongdong Zhang, Shuangzhi Wu, Nan Yang, and Mu Li. 2013. Punctuation prediction with transition-based parsing. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics*, pages 752–760. Association for Computational Linguistics, Sofia, Bulgaria.

Yue Zhang and Stephen Clark. 2011. Syntactic processing using the generalized perceptron and beam search. *Computational Linguistics*, 37(1):105–151.

Yue Zhang and Joakim Nivre. 2011. Transition-based dependency parsing with rich non-local features. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 188–193. Association for Computational Linguistics, Portland, USA.

Muhua Zhu, Yue Zhang, Wenliang Chen, Min Zhang, and Jingbo Zhu. 2013. Fast and accurate shift-reduce constituent parsing. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics*, pages 434–443. Association for Computational Linguistics, Sofia, Bulgaria.

Simon Zwarts and Mark Johnson. 2011. The impact of language models and loss functions on repair disfluency detection. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*, pages 703–711. Association for Computational Linguistics, Portland, USA.

Simon Zwarts, Mark Johnson, and Robert Dale. 2010. Detecting speech repairs incrementally using a noisy channel approach. In *Proceedings of the 23rd International Conference on Computational Linguistics (Coling 2010)*, pages 1371–1378. Coling 2010 Organizing Committee, Beijing, China.