NAIST-IS-MT1551119

Master's Thesis

Joint Transition-based Dependency Parsing and Disfluency Detection for Automatic Speech Recognition Texts

Masashi Yoshikawa

January 31, 2016

Department of Information Processing
Graduate School of Information Science
Nara Institute of Science and Technology

A Master's Thesis
submitted to Graduate School of Information Science,
Nara Institute of Science and Technology
in partial fulfillment of the requirements for the degree of
Master of ENGINEERING

Masashi Yoshikawa

Thesis Committee:

Professor Yuji Matsumoto	(Supervisor)
Professor Satoshi Nakamura	(Co-supervisor)
Associate Professor Masashi Shimbo	(Co-supervisor)
Assistant Professor Hiroyuki Shindo	(Co-supervisor)
Assistant Professor Hiroshi Noji	(Co-supervisor)

Joint Transition-based Dependency Parsing and Disfluency Detection for Automatic Speech Recognition Texts*

Masashi Yoshikawa

Abstract

Joint dependency parsing with disfluency detection is an important task in speech language processing. Recent methods show high performance for this task, although most authors make the unrealistic assumption that input texts are transcribed by human annotators. In real-world applications, the input text is typically the output of an automatic speech recognition (ASR) system, which implies that the text contains not only disfluency noises but also recognition errors from the ASR system. In this work, we propose a parsing method that handles both disfluency and ASR errors using an incremental shift-reduce algorithm with several novel features suited to ASR output texts. Because the gold dependency information is usually annotated only on transcribed texts, we also introduce an alignment-based method for transferring the gold dependency annotation to the ASR output texts to construct training data for our parser. We conducted an experiment on the Switchboard corpus and show that our method outperforms conventional methods in terms of dependency parsing and disfluency detection.

Keywords:

Dependency Parsing, Disfluency Detection, Automatic Speech Recognition

*Master's Thesis, Department of Information Processing, Graduate School of Information Science, Nara Institute of Science and Technology, NAIST-IS-MT1551119, January 31, 2016.

音声認識結果文に対する係り受け構造と言い淀み箇所の同時推定*

吉川将司

内容梗概

音声言語処理において, 言い淀み検出を行うことは重要である．近年の言い淀み検出の手法では, 係り受け解析と同時に解くものが高い精度を示しているが、それらの研究は, 人手で書き起こされたコーパス上で実験を行っており, 実際の応用場面では音声認識プログラムの出力に対して言い淀み検出を行う必要性を考えると, 非現実的な仮定を有している．実際には, 音声認識結果には言い淀みに加え音声認識の誤りが加わるということを考慮しなければならない．本研究では, 音声認識結果に対しても対処することのできる shift-reduce 法に基づく係り受け構造, 言い淀み箇所の同時推定手法を提案する．また, 本手法を評価するためには, 音声認識結果に対して係り受け, 言い淀み箇所の情報が付与されたコーパスが必要であるが、そのようなコーパスは存在しない．そのため、音声認識結果と書き起こし文のアライメントに基づく音声認識結果を用いたデータ作成手法も提案する．Switchboard コーパス上での実験において, 提案手法は従来手法に対して係り受け, 言い淀み検出の両方で精度を上回った．

キーワード

係り受け解析, 言い淀み検出, 音声認識

*奈良先端科学技術大学院大学 情報科学研究科 情報処理学専攻 修士論文, NAIST-IS-MT1551119, 2016 年 1 月 31 日.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

One of characteristics of spontaneous speech that makes it different from written text, even including informal text, is the presence of disfluencies. Disfluencies are prevalent in all forms of spontaneous speech, both in casual discussions and formal arguments.

According to Shriberg [29] and the other work on disfluency detection, disfluencies are classified into three categories:

Filler parts of speech that are usually recognized as containing no formal meaning, such as "uh", "um".

Discourse marker parts of speech that are used to manage the flow and structure of discourse, such as "I mean", "You know".

Reparandum parts of speech that are repeated, discarded or corrected by the following phrase (repair). In the Figure 1.1, "to Boston" is a reparandum and corrected by the subsequent repair phrase "to Denver".

Disfluency is mainly studied in the psycholinguistic domains, but their treatment in the Natural Language Processing (NLP) is also important when building an NLP system that processes human conversations or speech.

Figure 1.1: Examples of disfluencies. FL, DM, RP and RPM stand for a filler, a discourse marker, a repair and a reparandum.

$$\text{I want a flight } \underbrace{\text{to Boston}}_{RPM} \underbrace{\text{ uh}}_{FL} \underbrace{\text{ I mean}}_{DM} \underbrace{\text{to Denver}}_{RP}$$

Disfluency detection, the task of detecting which part of input sentence is disfluent, helps improve the readability of an utterance, and make it easy for downstream NLP modules. Especially the work of Cho [4] has shown that removing disfluencies helps improve speech translation system performance.

Detection of filler and discourse marker is fairly simple; As these types of disfluencies consist of closed set of vocabulary, they can be processed by just pattern matching on the word form.

However, the task of reparandum detection is quite challenging, as the expressions involved in reparandum is not limited, and a reparandum can span over several words (as exemplified in the Figure 1.1 and 2.2).

There are a number of studies that address the problem of detecting disfluencies. Some of these studies include dependency parsing [11, 26, 27, 33], whereas others are dedicated systems [8, 12, 13, 20, 25]. Among these studies, Honnibal [11] and Wu [33] address this problem by adding a new action to transition-based dependency parsing that removes the disfluent parts of the input sentence from the stack. Using this approach, they achieved high performance in terms of both dependency parsing and disfluency detection on the Switchboard corpus.

However, the authors assume that the input texts to parse are transcribed by human annotators, which, in practice, is unrealistic. In real-world applications, in addition to disfluencies, the input texts contain ASR errors; these issues might degrade the parsing performance. As in the following example, proper nouns that are not contained in the ASR system vocabulary may break up into smaller pieces, yielding a difficult problem for the parsing unit [3]:

> REF: what can we get at **Litanfeeth**
> HYP: what can we get *it leaks on feet*

In this work, we propose a method for joint dependency parsing and disfluency detection that can robustly parse ASR output texts. Our parser handles both disfluencies and ASR errors using an incremental shift-reduce algorithm, with novel features that consider recognition errors of the ASR system.

Furthermore, to evaluate dependency parsing performance on real human utterances, we create a tree-annotated corpus that contains ASR errors.

We conducted an experiment on the Switchboard corpus and show that our method outperforms conventional methods in terms of both dependency parsing and disfluency detection.

Chapter 2

Related Work

2.1 Transition-based Dependency Parsing

Transition-based dependency parsing [15, 23, 34, 37] utilizes a deterministic shift-reduce process to predict dependency structure for input sentence. Compared to graph-based dependency parsing, it offers linear time complexity and is amenable to the situation where the entire NLP system needs to process the speech data (e.g. speech translation).

A transition-based parser consists of a configuration (or state) that is sequentially manipulated by a set of possible actions. A state is a 3-tuple of (σ, β, A), where σ and β are disjoint sets of word indices, referred to as stack and buffer, respectively, and A is the set of constructed dependency arcs. Each dependency arc is of the form of $i \rightarrow j$, where i and j are word indices; this notation indicates a dependency relation from a head i to a child j. The deductive system for the parser is shown in Figure 2.1. There, we illustrate the stack using the topmost element to the right, and the buffer with the topmost element to the left, with vertical bar notation (e.g., $\sigma|i$ represents the stack with a first element i), following the convention in the literature. Figure 2.1 shows a running example of arc-eager parsing method.

In transition-based parsing, as a sequence of parsing actions maps to a dependency tree, the problem of finding the most probable dependency tree $\mathbf{y}$ for the input sentence $\mathbf{x}$ can be factorized as that of finding the most probable action a in the every parsing state s:

Action		**Precondition**	
Shift	$(\sigma, i\|\beta, A, D) \vdash (\sigma\|i, \beta, A, D)$		
Reduce	$(\sigma\|i, \beta, A, D) \vdash (\sigma, \beta, A, D)$	$\exists j'[\ j' \rightarrow i \in A\ ]$	(2.1)
LeftArc	$(\sigma\|i, j\|\beta, A, D) \vdash (\sigma, j\|\beta, A \cup \{j \rightarrow i\}, D)$	$\neg\exists j'[\ j' \rightarrow i \in A\ ]$	
RightArc	$(\sigma\|i, j\|\beta, A, D) \vdash (\sigma\|i\|j, \beta, A \cup \{i \rightarrow j\}, D)$		

Figure 2.1: Deductive system for arc-eager transition-based parser. σ indicates a stack, β a buffer, and A a set of constructed arcs.

$$\begin{aligned} \mathbf{y} &= \arg\max_{\mathbf{y}' \in \mathbf{Y}(\mathbf{x})} Score(\mathbf{y}'|\mathbf{x}) \\ &= \arg\max_{a_1,\ldots,a_n \in Actions} \sum_{i=1}^{2n} Score(a_i|s_i), \end{aligned} \tag{2.2}$$

where $Y(\mathbf{x})$ is a set of possible dependency trees of sentence $\mathbf{x}$. $Score(a|s)$ is a scoring function which scores how plausible to take action a in state s and is defined as:

$$Score(a|s) = \mathbf{w}_a \cdot \mathbf{f}(s), \tag{2.3}$$

where $\mathbf{w}_a$ is the weight vector for action a and $\mathbf{f}(s)$ is the feature representation for parsing state s, respectively. In order to construct $\mathbf{f}(s)$ one can use template-based feature [38] or neural network [2].

Arc-eager parser is guaranteed to terminate with $2n$ transitions when the input sentence consists of n words. In greedy search, a parser takes the one-best action in every parsing state. As this might lead to subsequent error propagation, one can instead adopt beam search, in which the parser retains the best N states in every time step.

Some extensions to transition-based parsing [11], and transition-based parsing of constituency trees [39] involve the action sequence of the length that is not constant of sentence length n. Especially, in the disfluency detection that is done jointly with dependency parsing, a parser removes a word token it classifies as disfluent, which

Step	Action	σ	β	A
1	(initial state)	[R]	[his company went broke]	$\emptyset$
2	Shift	[R his]	[company went broke]	
3	LeftArc	[R]	[company went broke]	$A \cup \{company \rightarrow his\}$
4	Shift	[R company]	[went broke]	
5	LeftArc	[R]	[went broke]	$A \cup \{went \rightarrow company\}$
6	RightArc	[R went]	[broke]	$A \cup \{R \rightarrow went\}$
7	RightArc	[R went broke]	[]	$A \cup \{went \rightarrow broke\}$
8	Reduce	[R went]	[]	
9	Reduce	[R]	[]	

Table 2.1: Parsing example of sentence "his company went broke", R stands for dummy root token. In every time step, parser performs one out of four actions.

apparently leads to action sequences with the uneven lengths. In these methods, in order to prevent the parser from preferring to lengthen the action sequence (which maximizes the score in Eq. 2.2), it is necessary to take some countermesures, e.g. normalizing scores by action sequence length[11] or making the parser do dummy transition when other candidates in beam has not reached the goal state [39].

2.2 Disfluency Detection

2.2.1 Joint Methods with Dependency Parsing

The task of disfluency detection has been tackled from syntactic point of view. The work of Johnson and Charniak[17] is the first work that addresses this task in terms of syntactic processing. They show the use of syntactic parser-based language model instead of bi- or tri-gram language model improves the accuracy of repair detection. They also show that Tree Adjoining Grammar [18], one of mildly context-sensitive grammar formalisms, can precisely describe, thus model the fact that the repair is "rough copy" of the reparandum.

The advantage of solving the problem of disfleuncy detection jointly with dependency parsing is that one can make use of the partially constructed tree to extract useful features to classify which part of the input sentence is disfluent. Many studies adopt transition-based parser and propose actions that remove disfluencies from parsing state when the parser cliassifies as such; Those parsers are trained to construct the dependency tree that is clean of any disfluencies (Figure 2.2).

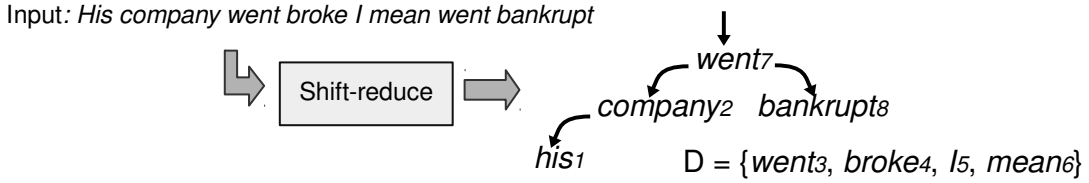

Figure 2.2: Joint dependency parsing and disfluency detection. The task is to construct the dependency tree that is clean of any disfluencies. In the figure, the set D is the set of detected disfluencies.

Rasooli and Tetreault [26] is the first study that proposes to add new actions to transition-based dependency parser in order to handle disfluencies. In every parsing state, their method first classfies whether there is a disfluency and needs to remove it from the state, or perform the standard arc-eager action to construct dependency tree. When the former is chosen, the parser removes the detected disfluency deterministically. In the latter case, the second classifier decides which one to perform among standard Shift, Reduce, LeftArc and RightArc actions. In their following work, they extended the original architecture by adding four more classfiers [27]. By adopting the cascaded architecture, their method showed the reduced memory usage and speed up gain.

Honnibal and Johnson [11] is the work on which this thesis is mainly based. They propose "Edit" action, which removes the first element of the stack and its right descendants, while pushing all the left descendants back to the stack. By using this action, their method needs not to guess early that the input contains disfluency, unlike Rasooli and Tetreault [26]'s method. They also propose the template-based features that capture the "rough copy"-ness of reparandum. We give the details of their method in Section 4.1.

Wu [33] shows that inverting the input sentence to the transition-based parser improves the performance of both dependency parsing and disfluency detection performance. However their method has the disadvantage that the inversion requires a com-

plete sentence as input, thus does not match such a situation that the output from the ASR system needs to be processed incrementally to reduce the latency of the whole system (e.g. SMT).

2.2.2 Other Methods

The other studies tackle this problem as sequence labeling problem [8, 12, 13, 20, 25, 32, 35].

Ferguson et al. [8] proposes that the semi-Markov Conditional Random Field [28] is suitable to this task, as many examples of disfluencies such as repair are not made up of single word token, rather they span over several words (thus form chunks). They also propose the use of prosodic features, i.e. the features that are extracted from raw wave form. These can be used as cues for pauses and hesitation and proven to be useful for disfluency detection [30].

There are several studies that apply the neural network methods to this task [13, 32, 35]. At the time of writing, the system proposed by Zayats et al. [35] achieves the state-of-the-art performance in the disfluency detection by using Bi-directional LSTM [16].

There are also studies of disfluency detection in the context of automatic speech recognition [20, 21]. While the most studies introduced above evaluate their method on the Switchboard corpus [9], a corpus annotated by human annotators, these studies seek for a method that detects disfluencies on ASR output texts and share the same objective as ours. The novelty of our work is that our aim is to extend the joint method of disfluency detection with dependency parsing so that it can be applicable to the output of ASR system.

Chapter 3

Data Preparation

To evaluate dependency parsing and disfluency detection performance on real speech texts, we need a corpus of ASR output texts with these annotations. However, to our knowledge, there is no corpus of this sort available at the time of writing. In the following, in order to overcome this situation, we propose to create a tree-annotated corpus of ASR output texts.

3.1 Process

Given a corpus that consists of speech data, transcription text and its syntactic annotation (e.g., the Switchboard corpus), we first apply the ASR system to the speech data. Next, we perform alignment between the ASR output texts and the transcription. Then, we transfer the gold syntactic annotation to the ASR output texts based on this alignment (Figure 3.1).

The alignment is performed by minimizing the edit distance between the two sentences. We include "NULL" tokens in this alignment to allow for some tokens not having an aligned conterpart ("*N*" tokens in the Figure 3.1).

In the constructed trees, there are three problematic cases based on how an ASR output text and its transcription are aligned with each other: (1) a word in the ASR output text aligns with a NULL token in the transcription (**ASR-to-NULL**), (2) a word in the gold transcription aligns with a NULL in the ASR output (**Trans-to-NULL**), and (3) two words align, but do not match exactly in terms of characters (**NOT MATCH**). To create a consistent dependency tree that spans the entire sentence, we must address each of these cases.

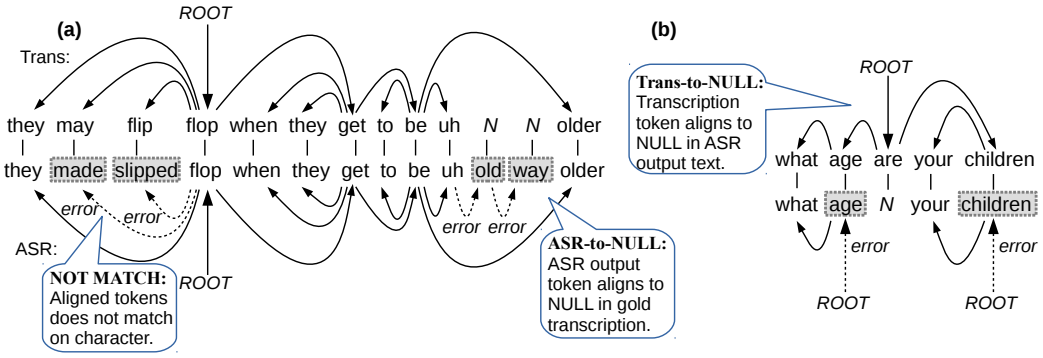

Figure 3.1: Examples of three problematic cases. Above shows the gold transcription and its tree, below shows the aligned ASR output and its newly transferred tree, where the dotted edges are ASR error edges.

3.1.1 ASR-to-NULL

In the case of ASR-to-NULL, a token from the ASR system has no corresponding token in the gold transcription. In this case, we automatically annotate a dependency relation with an "error" label such that the token's head becomes the previous word token.

Figure 3.1(a) shows an example of this case. In the figure, the words "old" and "way" have no corresponding words in the gold transcription. Thus, we automatically annotate the dependency relations between ("old", "uh") and ("way", "old"), respectively, with the "error" label.

3.1.2 Trans-to-NULL

Although NULL tokens are introduced to facilitate alignment, as these tokens in the ASR output are not actual words, we must remove them in the final tree. Without any treatment, the gold transcription tokens aligned to these tokens are also deleted along with them. This causes the child tokens in the sentence not to have heads; consequently, these child tokens are not included in the syntactic tree. To avoid this problem, we instead attach them to the head of the deleted token.

For example, in Figure 3.1(b), the word "are" is missing in the ASR hypothesis. Then, this token's children lose their head in the transfer process. Thus, we rescue these children by attaching them to the head of "are", which, in this case, is ROOT token.

If the head of the removed token is also of the Trans-to-NULL type, then we look for an alternative head by climbing the tree in a recursive manner, until reaching ROOT. We also label the newly created edges in this process as “error”.

3.1.3 NOT MATCH

In cases in which two aligned tokens do not match exactly on the character level, the mismatch is regarded as an instance of a substitution type of ASR error. Therefore, we encode this fact in the label of the arc from the token to its head.

In Figure 3.1(a), the words “made” and “slipped” in the ASR hypothesis do not match the gold transcription tokens, “may” and “flip”, respectively. Therefore, we automatically re-label the arc from each token to its head as “error”.

Chapter 4

Proposed Parsing Method

To parse texts that contain ASR errors and disfluencies, we propose the extension of transition-based dependency parsing method that jointly detects them.

4.1 Honnibal and Johnson, TACL 2014 [11]

As our method is based on that of Honnibal and Johnson [11], we describe their work in detail.

In joint parsing and disfluency detection, the configuration (or state) introduced in Section 2.1 is extended with a new set D, that is a set of detected (thus removed) disfluency tokens, and define parsing state as four-tuple (σ, β, A, D). In the work of Honnibal and Johnson, they extend the arc-eager parsing method with new *Edit* action:

$$Edit : (\sigma|i, \beta, A, D) \vdash (\sigma|[x_1, \ldots, x_n], \beta, A', D \cup [i, j))$$
$$where A' = A \setminus \{x \rightarrow y \; or \; y \rightarrow x | \forall x \in [i, j), y \in \mathbb{N}\}.$$

The runnning example of this action is shown in Figure 4.1. When this action is performed, the first element of the stack and its right descendants are classified as disfluent, and removed from the state (added to the set D). All dependency relation to and from disfluent tokens are also removed, while the tokens in the left descendants are pushed back to the stack again.

Unlike previous work such as Rasooli and Tetreault [26], their method needs not to guess early that the input contains disfluency and does not require complicated multi-classifier architecture, but simply one additional action to standard arc-eager parser.

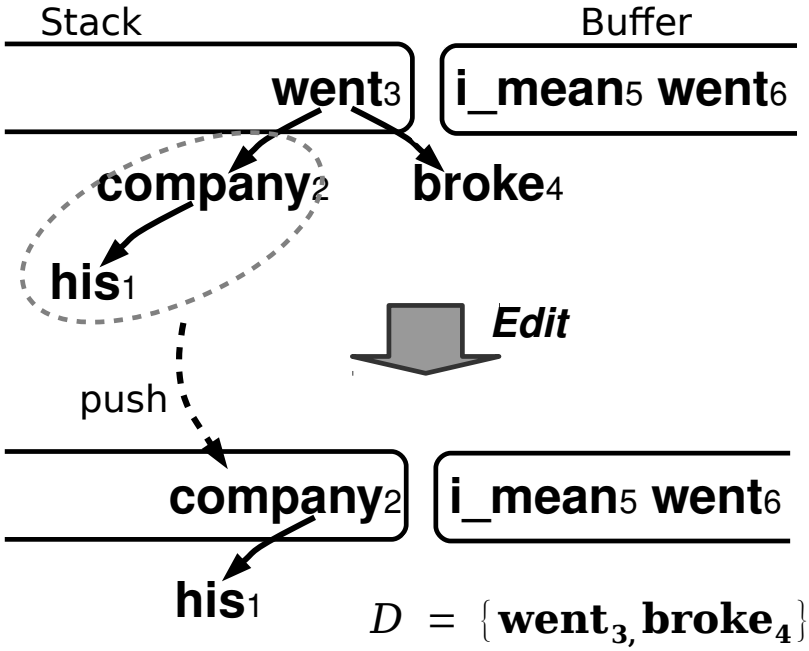

Figure 4.1: Illustrative example of performing *Edit* action during parsing sentence "his company went broke I mean went bankrupt". When performing *Edit* action, the first element of the stack is removed. Its root token and tokens in the right descendants are added to D, while its left child tokens are pushed back to the stack.

They also proposed template-based features which capture the nature of disfluencies (e.g. "rough copy") that is useful for the parser to predict when to perform the *Edit* action. We will introduce them in § 4.3.1.

4.2 Proposed Method

We also propose our own extension of the arc-eager parsing method, that is tailored to detect both disfluencies and ASR errors.

Firstly, we augment it with three new actions, *SimpleEdit* action that detect disfluencies, and $LeftArcError$ and *RightArcError* actions, each of which is introduced to cope with ASR errors. We give the details of each action below. The deductive system of newly proposed parsing actions are shown in Figure 4.3.

As with previous work, our *SimpleEdit* action is also introduced to remove a disfluent token. However in our method, it is used only when the first element of the stack does not have any children. We give the illustrative image of performing this action in Figure 4.3. This process is different from that of *Edit* proposed by Honnibal and Johnson [11]: theirs accumulates consecutive disfluent tokens on the top of the stack and removes them all at once, whereas our method removes this kind of tokens one-by-one. Thus, the parser does not use a dynamic oracle [10], rather, it uses a static oracle and is forced to perform *Shift* whenever a disfluent token is on the top of the buffer, and then, performs an *SimpleEdit* on it. Use of *SimpleEdit* action guarantees that the length of the action sequence is always $2n$. This property is advantageous because the parser can use the standard beam search and does not require normalization methods, such as those used in Honnibal and Johnson [11] or Zhu et al. [39].

LeftArcError and *RightArcError* are introduced to cope with ASR errors. They perform in the same way as *LeftArc* and *RightArc*, except that we train the parser to perform these actions only on ASR error tokens, whereas the original *LeftArc* and *RightArc* are reserved for non ASR error tokens; *LeftArcError* is expected to be performed when the first element of the stack is an ASR error token, whereas *RightArcError* should be used when the first element of the buffer is an ASR error. By having two different kinds of *Arc* actions for the two types of tokens (ASR error token or non ASR error token), one can keep the weights $\mathbf{w}_{LeftArc}$ and $\mathbf{w}_{RightArc}$ in the parsing model separate from $\mathbf{w}_{LeftArcError}$ and $\mathbf{w}_{RightArcError}$, that are used to process noisy ASR error tokens, and is expected to bring improved performance.

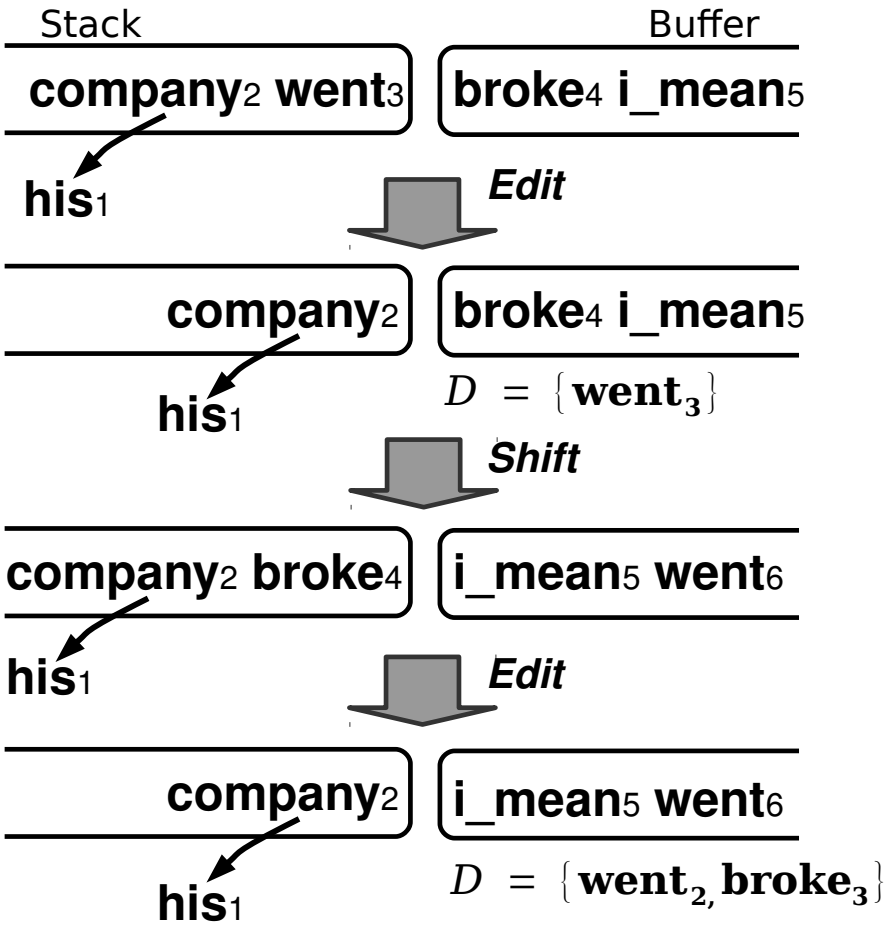

Figure 4.2: Proposed *SimpleEdit* action. This action removes disfluent tokens one-by-one and guarantees that the length of the action sequence is always two times the length of the input setence.

Action		**Precondition**
SimpleEdit	$(\sigma\|i,\beta,A,D) \vdash (\sigma,\beta,A',D\cup\{i\})$	$\neg\exists j'[\ i \to j' \in A\ ]$
	$A' = A \setminus \{j' \to i\}$ if such j'exists otherwise $A' = A$	
LeftArcError	$(\sigma\|i,j\|\beta,A,D) \vdash (\sigma,j\|\beta,A\cup\{j \to i\},D)$	$\neg\exists j'[\ j' \to i \in A\ ]$
RightArcError	$(\sigma\|i,j\|\beta,A,D) \vdash (\sigma\|i\|j,\beta,A\cup\{i \to j\},D)$	

Figure 4.3: Deductive system for newly proposed actions. D is a set of removed tokens. The *SimpleEdit* action removes the top element of the stack and *LeftArcError* and *RightArcError* perform in the same way as *LeftArc* and *RightArc*, but are used to classify ASR error tokens.

4.3 Features

We describe template-based features used in our parsing model that capture the nature of both disfluencies and ASR errors. In the following, we use notation $\langle\cdot\rangle$ for one feature template example, and use $\circ$ to indicate feature concatenation. $i.w$ and $i.p$ each stands for i'th token's word form and part-of-speech tag (e.g. $\langle i.w \circ i.p\rangle$ stands for the feature concatenation of i'th word form and part-of-speech tag.). σ_i, β_i are used to indicate the token index of i'th top element of the stack (the i'th rightmost element) and the buffer (the i' th leftmost element). Other subscripts used with σ and β are: ih, il, ir (i'th token's head, left child, and right child), i$h2$, i$l2$, i$r2$ (i'th token's head's head, 2nd left child, and 2nd right child), and ire, ile ri'th token's leftmost descendant and rightmost descendant).

4.3.1 Features for Disfluencies and ASR errors

Firstly in order to capture rich context features, we use features proposed by Zhang and Nivre [38], which we show in Table 4.1. They show higher order features and unigram features that inspect deep under the trees (Unigram, Bigram and Trigram in the Table) and features inspired by graph-based parsing (Distance and Valency) are useful for transition-based parsing. In Distance and Valency type features, d stands for the distance between σ_0 token and β_0 token (that is, β_0 - σ_0). v_r and v_l stands for the number of left and right child tokens.

We also include features that capture the nature of disfluencies proposed in Honnibal [11], which are summarized in Table 4.3. We modify some of these features

Type	Token	Template
Unigram	$\forall p \in \{\sigma_0, \beta_0, \beta_1, \beta_2\}$	$\langle p.w \rangle, \langle p.p \rangle, \langle p.wp \rangle$
	$\forall p \in \left\{ \begin{array}{l} \sigma_{0h},\ \sigma_{0l},\ \sigma_{0r}, \beta_{0l}, \\ \sigma_{0h2}, \sigma_{0l2}, \sigma_{0r2}, \beta_{0l2}, \\ \sigma_{0re}, \sigma_{0le},\ \beta_{0le} \end{array} \right\}$	$\langle p.w \rangle, \langle p.p \rangle$
	$\forall p \in \{\sigma_{0re}, \sigma_{0le}, \beta_{0le}\}$	$\langle p.w, p.p \rangle$
Bigram	$(p,q) = (\sigma_0, \beta_0)$	$\langle p.wp \circ q.wp \rangle, \langle p.wp \circ q.w \rangle,$ $\langle p.w \circ q.wp \rangle, \langle p.wp \circ q.p \rangle,$ $\langle p.p \circ q.wp \rangle, \langle p.w \circ q.w \rangle,$ $\langle p.p \circ q.p \rangle$
	$\forall (p,q) \in \left\{ \begin{array}{l} (\beta_0, \beta_1), (\sigma_{0re}, \beta_0), \\ (\sigma_0, \beta_{0le}) \end{array} \right\}$	$\langle p.p, q.p \rangle$
Trigram	$\forall (p,q,r) \in \left\{ \begin{array}{ll} (\beta_0, \beta_1, \beta_2), & (\sigma_0, \beta_0, \beta_1), \\ (\sigma_{0h}, \sigma_0, \beta_0), & (\sigma_0, \sigma_{0l}, \beta_0), \\ (\sigma_0, \sigma_{0r}, \beta_0), & (\sigma_0, \beta_0, \beta_{0l}), \\ (\sigma_0, \sigma_{0l}, \sigma_{0l2}), & (\sigma_0, \sigma_{0r}, \sigma_{0r2}), \\ (\sigma_0, \sigma_{0h}, \sigma_{0h2}), & (\beta_0, \beta_{0l}, \beta_{0l2}) \end{array} \right\}$	$\langle p.p, q.p, r.p \rangle$
Distance	$\forall p \in \{\sigma_0, \beta_0\}$	$\langle p.w, d \rangle, \langle p.p, d \rangle$
	$(p,q) = (\sigma_0, \beta_0)$	$\langle p.w, q.w, d \rangle, \langle p.p, q.p, d \rangle$
Valency	$p = \sigma_0$	$\langle p.wv_r \rangle, \langle p.pv_r \rangle$
	$\forall p \in \{\sigma_0, \beta_0\}$	$\langle p.wv_l \rangle, \langle p.pv_l \rangle$

Table 4.1: Rich context features based on the work [38]. In each template, $.w$, and $.p$ is word form and part-of-speech tag, $.d$ distance between σ_0 and β_0, and $.v_r$, $.v_l$ the number of left and right children. Details are in § 4.3.1.

Type	Token	Template
Edited Neighbor	$p = \beta_0$	$\langle disfl(p-1)\rangle$, $\langle disfl(p-1) \circ disfl(p-2)\rangle$
	$p = \sigma_0$	$\langle disfl(p+1)\rangle$, $\langle disfl(p+1) \circ disfl(p+2)\rangle$
ASR Error Neighbor	$p = \beta_0$	$\langle ASRerror(p-1)\rangle$, $\langle ASRerror(p-1) \circ ASRerror(p-2)\rangle$
	$p = \sigma_0$	$\langle ASRerror(p+1)\rangle$, $\langle ASRerror(p+1) \circ ASRerror(p+2)\rangle$
Rough Copy	$\forall(p,q,r,s) \in \left\{ \begin{array}{l} (\sigma_0,\beta_0,\beta_1,\beta_2), \\ (\sigma_0,\beta_1,\beta_2,\beta_4), \\ (\sigma_0,\beta_0,\beta_2,\beta_3), \\ (\sigma_0,\beta_1,\beta_3,\beta_5), \\ (\sigma_1,\beta_0,\beta_0,\beta_1), \\ (\sigma_1,\beta_0,\beta_1,\beta_3) \end{array} \right\}$	$\langle prefix\_match([p,q],[r,s])\rangle$, $\langle prefix\_match([p,q],[r,s]) \circ disfl(\beta_0-1)\rangle$, $\langle prefix\_match([p,q],[r,s]) \circ disfl(\beta_0+1)\rangle$
Brown	$\forall p \in \left\{ \begin{array}{l} \sigma_0,\ \sigma_{0l},\sigma_{0r},\sigma_{0l2}, \\ \sigma_{0r2},\sigma_{0le},\sigma_{0re},\beta_0, \\ \beta_{0l},\ \beta_{0l2},\ \beta_{0le},\beta_1, \\ \beta_2,\ \ \beta_3 \end{array} \right\}$	$\langle brown(p.w)\rangle$
	$(p,q) = (\beta_0,\sigma_0)$	$\langle brown(p.w) \circ brown(q.w)\rangle$
Match	$\forall(p,q) \in Comb\left(\left\{ \begin{array}{l} \sigma_0,\ \sigma_{0l},\ \sigma_{0l}, \\ \sigma_{0le},\sigma_{0l2},\beta_0, \\ \beta_{0l},\beta_{0l2},\beta_{0le}, \\ \beta_1,\ \beta_2,\ \beta_3, \\ \beta_4,\ \ \beta_5 \end{array} \right\} \right)$	$\langle match(p.w,q.w)\rangle, \langle match(p.p,q.p)\rangle$

Table 4.2: Features that capture the nature of disfluencies and ASR errors. $disfl(i)$ and $ASRerror(i)$ is the binary functions that return if i'th token has been classified as disfluent or ASR error. $prefix\_match(a,b)$ is a function that check how long the prefixes of spans a and b match. $brown(\cdot)$ returns brown cluster and $match(\cdot,\cdot)$ is a binary function that checks the two given arguments are identical. $Comb(\{\})$ stands for all combinations in the set. Details are in § 4.3.1.

because of the different timing of our parser to perform the *SimpleEdit* action from that of the original *Edit* action. Besides, we extend some features to take cues of ASR errors.

Edited Neighbor features check if the previous word of β_0 and the next word of σ_0 have been classified as disfluent. These features are motivated from the fact that once a disfluency occurs in a speech, the subsequent words or phrases also have higher possibility to be disfluent, as can be seen in the examples of the Figure 1.1 and 2.2. Based on the observation that ASR errors too, have the similar tendency, we add new features to check if the tokens in question have been classified as ASR error by our parser (ASR Error Neighbor).

Rough Copy features use $prefix\_match$ function to see how long the prefixes of the given spans match on word form and part-of-speech tag. These features consider the occurance of the pairs of reparandum and repair, as these types of phrases resemble in their beginning in the most cases (repair is "rough copy" of reparandum.).

Brown features use Brown clustering algorithm [1], which is well-known source of semi-supervised features. The clustering algorithm is run over a large size of unlabeled data to give a mapping of word type to their classes. We use the pre-computed version distributed by Liang [19].[1]

Match features examine which pair from all combinations of tokens in parsing context (all σ_i and β_i tokens) match on word form or part-of-speech tag. This also gives the parser cues for disfluencies.

4.3.2 Features based on Word Confusion Network

To better capture which parts of the texts are likely to be ASR errors, we use additional features extracted from a word confusion network (WCN) generated by ASR models. WCN is a compact representation of recognition results in which competing word hypotheses and their posterior probabilities are represented as arcs in time-ordered sets (slots). Marin [22] reports his observation that WCN slots with more arcs tend to correspond to erroneous region. Following Marin we use WCN-based features such as the mean and standard deviation of the slot arc posteriors and the highest posterior in the slot (Table 4.3).

[1] http://www.metaoptimize.com/projects/wordreps

Type	Token	Template
WCN Features	$\forall p \in \{\ \sigma_0, \beta_0, \beta_1, \beta_2, \beta_3, \beta_4\ \}$	mean(p.w), std(p.w), highpos(p.w)

Table 4.3: Word Confusion Network Features

4.3.3 Backoff Action Feature

With the newly proposed *LeftArcError* and *RightArcError* actions, we fear that the relatively low frequency of "error" tokens may cause the weights for these actions to be updated too infrequently to be accurately generalized.

We resort to using the "backoff action feature" to avoid this situation. This means that, for each action $a \in \{LeftArc, LeftArcError\}$, the score of performing it in a state s is calculated as follow:

$$SCORE(a,s) = \mathbf{w}_a \cdot \mathbf{f}(s) + \mathbf{w}_{a'} \cdot \mathbf{f}(s) \tag{4.1}$$

where $a' = LeftArcBackoff$, $\mathbf{w}_a$ is the weight vector for the action a and $\mathbf{f}(\cdot)$ is the feature representation, respectively. *LeftArcBackoff* is not actual action performed by our parser, rather it is used to provide the common feature representation which both *LeftArc* and *LeftArcError* can "back off" to. *RightArc* and *RightArcError* actions also calculate their scores as in Eq.(4.1), with $a' = RightArcBackoff$. The scores for all the other actions are calculated in the normal way: $SCORE(a,s) = \mathbf{w} \cdot \mathbf{f}(a,s)$.

4.4 Training

We train the parsing model using averaged structured perceptron [5]. By using structured perceptron, we train the model to predict the correct sequence of actions, rather than a correct action at some state. In order to make the model learn efficiently despite relatively many feature templates, We use max violation criterion update [14]. In max violation, we update the the model parameters with the only subsequence of the predicted actions, that involves "violation", that is defined as the sequence of actions with higher model score than that of the correct one. In [14], they show that this update strategy helps learn better model and reduce the learning time.

Chapter 5

Experiments

5.1 the Switchboard corpus

Our experiments were performed using the Switchboard corpus [9] following previous work in the literature. As shown in Figure 5.1, this corpus consists of speech data and its transcription texts (A), and contains 2438 two-sided conversations. The subset of the corpus is annotated with part-of-speech tags and disfluency information using the notation of Shriberg [29], which contains 1126 conversations (B). A further subset of this part is annotated with syntactic information, constituting 650 conversations (C). Following the practice in the literature, We converted the annotated phrase structure trees in the corpus to dependency trees using the Stanford converter [7].

5.2 Experimental Settings

5.2.1 ASR Settings

To obtain the ASR output texts of the corpus, we used the off-the-shelf NeuralNet recipe (p-norm Network [36]) presented by Kaldi project [24], with the additional use of the Fisher corpus [6] for language modeling.

We used the jackknife method to obtain the ASR output texts throughout the syntactically annotated part of the corpus. Following Figure 5.1, in the adopted jackknife method, we first split the syntactically annotated part (C) into three (Jack1, 2, and 3). Second, we chose two of them (Jack1 and Jack2, for example), and using these and the part of the corpus that has no syntactic annotation ($A \setminus C$ in set theory notation),

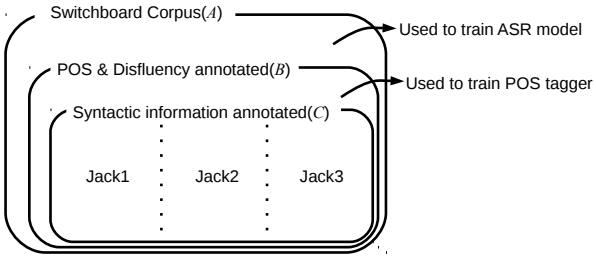

Figure 5.1: Overview of Switchboard corpus. The Switchboard corpus contains speech data and its transcription texts (*A*). A subset of the corpus is annotated with part-of-speech and disfluency information (*B*). A further subset is annotated with syntactic information (*C*). The syntactically annotated part is divided into three (Jack1, 2, and 3) in data creation using the jackknife method (details are provided in the text).

we train a NeuralNet ASR model. Finally, we decode over the remaining syntactically annotated part (in this case, Jack3). We repeated this process three times, until we obtained the ASR output texts for the entire syntactically annotated part of the corpus. With this strategy, the amount of training data lost for each of the ASR models due to the jackknife method is guaranteed to be less than 10% of the whole Switchboard corpus with only three iterations of the train-decode processes. The average word error rate of the three resulting models were 13.9% on the Switchboard part of the HUB5 evaluation dataset [1].

From these ASR output texts, we created the tree-annotated corpus by applying the data creation method introduced in Chapter 3. Out of all 857,493 word tokens, there are 32,606 ASR-to-NULL, 34,952 Trans-to-NULL, and 93,138 NOT MATCH cases, meaning 15.6% of all word tokens had "error" labeled arcs.

5.2.2 Parsing Settings

We assigned part-of-speech tags (POS tags) to the corpus created in § 5.2.1 using the Stanford POS tagger [31] trained on a part of the corpus that is annotated with POS information but not syntactic information ($B \setminus C$, which contains 56,001 sentences.). The performance of the tagger is evaluated on the syntactically annotated part (*C*); the tagger has an accuracy score of 95.0%.

[1] https://catalog.ldc.upenn.edu/LDC2002S09

We adopt the same train/dev/test split as in Honnibal [11], although the data size reduces slightly in the process of the data creation. We report the unlabeled attachment score (UAS), which indicates how many heads of fluent tokens are correctly predicted. The "error" edge label is used to distinguish the ASR error parts from non error parts of sentences and to report the UAS scores separately on each type of tokens.

Disfluency detection is a simple binary classification task to predict whether each token is disfluent or not; therefore, we report the precision/recall/F1-score values following the previous studies.

As a baseline, we use a setting in which we train a model of the parser in § 4.2 without *Left/RightArcError* actions on the train part of the gold Switchboard corpus, then test its performance on the test part of the ASR output version of the corpus. We exclude *Left/RightArcError* actions in this case, as there are no ASR error tokens in the training data. This setting can be seen as reproducing the typical situation, in which a parser is trained on ASR-error-free texts, but nevertheless needs to parse the ASR output texts. In the following section, we refer to this parser configuration as *Disfl*.

5.3 Results and Analysis

Table 5.1 shows the parser's performance for different parsing configurations. Based on the baseline *Disfl* parser, we report scores with the additional (and additive) use of *Left/RightArcError* actions (*ErrorAct*), the WCN feature (*WCN*), and the backoff action feature (*Backoff*). Using *Left/RightArcError* actions resulted in 3.6% and 1.2% improvement in UAS and disfluency detection accuracy, respectively. The backoff action feature led to further improved UAS, whereas WCN features cause an increase in disfluency detection accuracy. We observe that a reversion in precision and recall occurs when using *Left/RightArcError* actions.

Table 5.2 reports parsing performance on various train and test data settings. In Table 5.2, the Train and Test columns represent which data to use in training and testing; *Trans* refers to the gold transcription text of the Switchboard corpus, and *ASR* indicates the text created through the data creation process in § 5.2.1.

Our parser's performance on the gold Switchboard corpus ((Train, Test) = (*Trans*, *Trans*)) can be seen as an upper bound of the rest of the parsing experiments. As this experiment uses the same train/test data settings, the result can be compared to those of existing studies of joint dependency parsing and disfluency detection [11, 33, 27]. Although our parser does not achieve the state-of-the-art result (92.2% and 85.1% in

Model	Dependency (UAS)		Disfluency		
	ALL	ASR error / other	Prec.	Rec.	F1
Disfl	72.7	37.8 / 79.2	58.6	**62.2**	60.3
Disfl + ErrorAct	76.3	40.8 / 82.9	66.0	57.6	61.5
Disfl + ErrorAct + Backoff	**76.4**	40.6 / **83.0**	65.6	57.3	61.1
Disfl + ErrorAct + WCN + Backoff	76.2	**41.1** / 82.7	**67.9**	57.9	**62.5**

Table 5.1: Performance of the proposed methods. *Disfl* refers to the joint parser with disfluency detection without *Left/RightArcError* actions. *ErrorAct* introduces these actions. *Backoff* and *WCN*, respectively, refer to the backoff action feature in § 4.3.3 and the WCN feature in § 4.3.2. UAS reports parsing performance on all parts of the sentence (ALL) and the ASR error part and the non ASR error part of the sentence (ASR error / other), separately.

UAS and disfluency F1 score in [33]), it still shows a relatively good result with a simpler parser configuration.

When evaluated on the ASR texts, the parser trained on the ASR output texts showed degraded performance compared to the parser trained on the gold transcription ((Train, Test) = (*ASR*, *ASR*)). This result is surprising, given that in this setting, both the train and test data are ASR output texts and share characteristics; therefore, we expect that there will be some domain adaptation effect. We hypothesized that the drop in the performance is due to the relatively noisy nature of our corpus, which is created from the ASR output texts with recognition errors. Having ASR-error-specific actions, *Left/RightArcError* mitigates this problem by separately treating the ASR error tokens and non ASR error tokens. Finally, with backoff action and WCN features, the performance increase was 1.5% and 0.7% for UAS and disfluency detection, respectively, compared to the baseline. With the newly proposed features, the parser trained on ASR output texts outperforms the parser trained on the transcription texts.

However, when compared with the case of (Train, Test) = (*Trans*, *Trans*), we observe significant decreases in performance in both dependency parsing and disfluency detection in the experiments conducted on ASR output texts. This result clearly poses a new challenge for the disfluency detection community.

Train	Test	Model	Dependency (UAS)		Disfluency		
			ALL	ASR error / other	Prec.	Rec.	F1
Trans	*Trans*	*Disfl*	89.7	-	90.4	76.8	83.1
Trans	*ASR*	*Disfl*	74.7	36.1 / 82.0	58.5	**65.6**	61.8
ASR	*ASR*	*Disfl*	72.7	37.8 / 79.2	58.6	62.2	60.3
ASR	*ASR*	*Disfl + ErrorAct + Backoff*	**76.4**	40.6 / **83.0**	65.6	57.3	61.1
ASR	*ASR*	*Disfl + ErrorAct + WCN + Backoff*	76.2	**41.1** / 82.7	**67.9**	57.9	**62.5**

Table 5.2: Parsing result on different train-test settings. Each *Trans* and *ASR* refers to the gold transcription texts of the Switchboard corpus and the ASR output version.

Chapter 6

Conclusion

In this work, we have proposed a novel joint transition-based dependency parsing method with disfluency detection. This method is capable of robustly parsing ASR output texts. With the introduction of new actions, backoff action features, and WCN features, the parser shows reasonably good results and outperforms conventional parsers for ASR output texts. We also introduce an alignment-based data construction method and propose evaluation of both of parsing and disfluency detection performance for real speech data. As the experimental performance for ASR texts is significantly lower than the performance achieved for the gold transcription texts, we have clarified the need to develop a method of disfluency detection that is robust to recognition errors in the ASR system. In future work, we will further improve the parsing performance, and develop a method to correct erroneous parts of sentences.

Bibliography

[1] P. F. Brown, P. V. deSouza, R. L. Mercer, V. J. Della Pietra, and J. C. Lai. Class-Based N-Gram Models of Natural Language. pp. 467–479. Computational Linguistics, 1992.

[2] D. Chen and C. Manning. A Fast and Accurate Dependency Parser using Neural Networks. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 740–750, Doha, Qatar, October 2014. Association for Computational Linguistics.

[3] H. Cheng, H. Fang, and M. Ostendorf. Open-Domain Name Error Detection using a Multi-Task RNN. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pp. 737–746. Association for Computational Linguistics, 2015.

[4] E. Cho, J. Niehues, and A. Waibel. Tight Integration of Speech Disfluency Removal into SMT. In *Proceedings of the 14th Conference of the European Chapter of the Association for Computational Linguistics, volume 2: Short Papers (EACL)*, pp. 43–47. Association for Computational Linguistics, 2014.

[5] M. Collins. Discriminative Training Methods for Hidden Markov Models: Theory and Experiments with Perceptron Algorithms. In *Proceedings of the ACL-02 conference on Empirical methods in natural language processing-Volume 10*, pp. 1–8. Association for Computational Linguistics, 2002.

[6] C. C. David, D. Miller, and K. Walker. The Fisher Corpus: a Resource for the Next Generations of Speech-to-Text. In *in Proceedings 4th International Conference on Language Resources and Evaluation*, pp. 69–71, 2004.

[7] M.-C. de Marneffe, B. MacCartney, and C. D. Manning. Generating Typed Dependency Parses from Phrase Structure Parses. In *In Proceedings of the 5th International Conference on Language Resources and Evaluation (LREC)*, 2006.

[8] J. Ferguson, G. Durrett, and D. Klein. Disfluency Detection with a Semi-Markov Model and Prosodic Features. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL)*, pp. 257–262. Association for Computational Linguistics, 2015.

[9] J. J. Godfrey, E. C. Holliman, and J. McDaniel. “switchboard: Telephone speech corpus for research and development”. In *Acoustics, Speech, and Signal Processing, 1992. ICASSP-92., 1992 IEEE International Conference on (Volume:1)*. Proc. IEEE Int. Conf. Acoust. Speech Sig. Proc, 1992.

[10] Y. Goldberg and J. Nivre. A Dynamic Oracle for Arc-Eager Dependency Parsing. In *In Proceedings of the 24th International Conference on Computational Linguistics (Coling)*, pp. 959–976. The COLING 2012 Organizing Committee, 2012.

[11] M. Honnibal and M. Johnson. Joint Incremental Disfluency Detection and Dependency Parsing. In *Transactions of the Association of Computational Linguistics Volume 2, Issue 1 (TACL)*, pp. 131–142. Association for Computational Linguistics, 2014.

[12] J. Hough and M. Purver. Strongly Incremental Repair Detection. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 78–89. Association for Computational Linguistics, 2014.

[13] J. Hough and D. Schlangen. Recurrent Neural Networks for Incremental Disfluency Detection. Interspeech 2015, 2015.

[14] L. Huang, S. Fayong, and Y. Guo. 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*. Association for Computational Linguistics, 2012.

[15] L. Huang and K. Sagae. Dynamic Programming for Linear-Time Incremental Parsing. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, pp. 1077–1086, Uppsala, Sweden, July 2010. Association for Computational Linguistics.

[16] Z. Huang, W. Xu, and K. Yu. Bidirectional LSTM-CRF Models for Sequence Tagging. *CoRR*, abs/1508.01991, 2015.

[17] M. Johnson and E. Charniak. A TAG-based Noisy-Channel Model of Speech Repairs. In *Proceedings of the 42nd Meeting of the Association for Computational Linguistics (ACL'04), Main Volume*, pp. 33–39, Barcelona, Spain, July 2004.

[18] A. K. Joshi, L. S. Levy, and M. Takahashi. Tree Adjunct Grammars. *Journal of computer and system sciences*, 10(1):136–163, 1975.

[19] P. Liang. In *Semi-Supervised Learning for Natural Language*. Ph.D. thesis, MIT, 2005.

[20] Y. Liu, E. Shriberg, and A. Stolcke. Automatic disfluency identification in conversational speech using multiple knowledge sources. In *In Proceedings of the 8th Eurospeech Conference*, 2003.

[21] Y. Liu, E. Shriberg, A. Stolcke, D. Hillard, M. Ostendorf, and M. Harper. Enriching Speech Recognition with Automatic Detection of Sentence Boundaries and Disfluencies. *IEEE Transactions on Audio, Speech, and Language Processing*, 14(5):1526–1540, Sept 2006.

[22] M. A. Marin. In *Effective Use of Cross-Domain Parsing in Automatic Speech Recognition and Error Detection. Ph.D. thesis*. University of Washington, 2015.

[23] J. Nivre, J. Hall, J. Nilsson, G. Eryiğit, and S. Marinov. Labeled Pseudo-Projective Dependency Parsing with Support Vector Machines. In *Proceedings of the Tenth Conference on Computational Natural Language Learning (CoNLL-X)*, pp. 221–225, New York City, June 2006. Association for Computational Linguistics.

[24] D. Povey, A. Ghoshal, G. Boulianne, L. Burget, O. Glembek, N. Goel, M. Hannemann, P. Motlicek, Y. Qian, P. Schwarz, J. Silovsky, G. Stemmer, and K. Vesely. The Kaldi Speech Recognition Toolkit. In *IEEE 2011 Workshop on Automatic Speech Recognition and Understanding*, 2011.

[25] X. Qian and Y. Liu. 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*. Association for Computational Linguistics, 2013.

[26] M. S. Rasooli and J. Tetreault. Joint Parsing and Disfluency Detection in Linear Time. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pp. 124–129, Seattle, Washington, USA, October 2013. Association for Computational Linguistics.

[27] M. S. Rasooli and J. Tetreault. Non-Monotonic Parsing of Fluent Umm I mean Disfluent Sentences. In *Proceedings of the 14th Conference of the European Chapter of the Association for Computational Linguistics, volume 2: Short Papers (EACL)*, pp. 48–53. Association for Computational Linguistics, 2014.

[28] S. Sarawagi and W. W. Cohen. Semi-Markov Conditional Random Fields for Information Extraction. In *In Proceedings of Advances in Neural Information Processing Systems*, 2004.

[29] E. Shriberg. In *Preliminaries to a Theory of Speech Disfluencies. Ph.D. thesis*. University of California, Berkeley, 1994.

[30] E. Shriberg, R. Bates, and A. Stolcke. A prosody-only decision-tree model for disfluency detection. In *In Proceedings of Eurospeech*, 1997.

[31] K. Toutanova, D. Klein, C. Manning, and S. Yoram. Feature-Rich Part-of-Speech Tagging with a Cyclic Dependency Network. In *In Proceedings of the 2003 Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics*, pp. 173–180. Association for Computational Linguistics, 2003.

[32] S. Wang, W. Che, Y. Liu, and T. Liu. *Enhancing Neural Disfluency Detection with Hand-Crafted Features*, pp. 336–347. Springer International Publishing, Cham, 2016.

[33] S. Wu, D. Zhang, M. Zhou, and T. Zhao. Efficient Disfluency Detection with Transition-based Parsing. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers) (ACL)*, pp. 495–503. Association for Computational Linguistics, 2015.

[34] H. Yamada and Y. Matsumoto. Statistical Dependency Analysis With Support Vector Machines. In *In proceedings of 8th International Workshop on Parsing Technologies*, pp. 195–206, 2003.

[35] V. Zayats, M. Ostendorf, and H. Hajishirzi. Disfluency Detection using a Bidirectional LSTM. *CoRR*, abs/1604.03209, 2016.

[36] X. Zhang, J. Trmal, D. Povey, and S. Khudanpur. Improving Deep Neural Network Acoustic Models using Generalized Maxout Networks. In *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2014.

[37] Y. Zhang and S. Clark. A Tale of Two Parsers: Investigating and Combining Graph-based and Transition-based Dependency Parsing. In *Proceedings of the 2008 Conference on Empirical Methods in Natural Language Processing*, pp. 562–571, Honolulu, Hawaii, October 2008. Association for Computational Linguistics.

[38] Y. Zhang and J. Nivre. 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 (ACL)*, pp. 188–193. Association for Computational Linguistics, 2011.

[39] M. Zhu, Y. Zhang, W. Chen, M. Zhang, and J. Zhu. Fast and Accurate Shift-Reduce Constituent Parsing. In *In Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (ACL)*, pp. 434–443. Association for Computational Linguistics, 2013.