Dependency Parsing with Feature Engineering
NLP1 Final Project

Daan van Stigt
ILLC, UvA
10255141

Mathijs Mul
ILLC, UvA
6267939

Selina Blijleven
UvA
10574689

May Lee
UvA
11436581

Abstract

This paper describes a transition-based dependency parser. Linguistic features are engineered manually and represented as 50-dimensional vectors. A Multilayer Perceptron is trained to predict the correct transition given an input of this format. This results in an unlabeled attachment score of 85.6% and a labeled attachment score of 82.9% on the Stanford CoNLL corpus. Closer investigation of the network weights shows that the POS tags of the top words on the stack are critical in making correct predictions, whereas more advanced features have little impact on the parser's performance. The research is largely based on (Chen and Manning, 2014).

1 Introduction

A dependency parser is an algorithm that determines the relation between words in a sentence. To be more specific, it determines which words are so-called 'head'-words and which words are modified by those. The main verb is the structural centre and is the unique word within the sentence that gets the root as head. All other words are directly or indirectly connected to the main verb. In addition to a direction, these attachments can be labelled to indicate the nature of the dependency. An example is shown in Figure 1. A dependency graph can can give insight into the grammatical structure of a phrase. As such a dependency graph can be used to interpret a sentence, which is useful in the case of ambiguity.

There are several ways to form a dependency graph: transition-based (shift-reduce) parsing, spanning trees, and cascaded chunking. Transition-based parsing is a fast method that parses the sentence from left to right. The spanning tree method calculates the whole tree at once and is (generally) slightly more accurate, but also slower. The cascaded chunking method chunks words into phrases and attempts to find the heads, while discarding non-heads. This research will focus on transition-based parsing, building on the research of Chen & Manning (Chen and Manning, 2014).

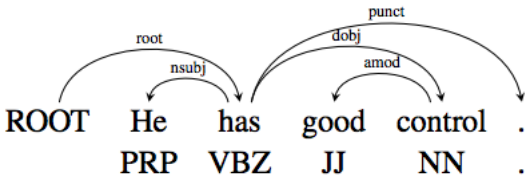


Figure 1: Dependency graph of the sentence 'He has good control.'

2 Problem

Transition-based parsing builds a dependency tree by loading the words of a sentence on a buffer and traversing it word by word, choosing between available transitions. Admissible operations are shifting, left-reducing and right-reducing. The shift operation shifts one word from the buffer onto the stack. The reduction operations apply a grammar rule to place an arc between two words on the stack. The parser outputs a collection of arcs indicating the relations between words and their dependents. Consider the sentence 'He has good control'. Then Figure 2 shows the series of configurations and transitions that a perfect parser performs in order to construct the correct dependency graph displayed in Figure 1. The underlying assumption is that English sentences are structured in such a way that any sentence can be correctly parsed using a sequence of admissible shift-reduce transitions.

This research takes a symbolic approach to implementing a transition-based parser. A set of fea-

transition	stack	buffer	A
	[ROOT]	[He has good control .]	$\emptyset$
SHIFT	[ROOT He]	[has good control .]	
SHIFT	[ROOT He has]	[good control .]	
LEFT-REDUCE(nsubj)	[ROOT has]	[good control .]	$A \cup$ nsubj(has, He)
SHIFT	[ROOT has good]	[control .]	
SHIFT	[ROOT has good control]	[.]	
LEFT-REDUCE	[ROOT has control]	[.]	$A \cup$ amod(control, good)
...	...	...	...

Figure 2: Initial fragment of the shift-reduce parse of the sentence 'He has good control.'

tures is defined as described in Chen & Manning. A neural network uses these features to determine the correct parsing action. The paper is meant to evaluate the approach of Chen & Manning by considering the effects of modifying the chosen set of features. Moreover, it aims to provide a better understanding of what actually happens inside the network by studying the learned weight vectors, identifying which features are most crucial in predicting correct transitions and which ones only improve performance marginally.

3 Relevant Literature

This research is mostly based on the approach proposed by Chen & Manning (Chen and Manning, 2014), who use manually engineered dense feature representations as input to an MLP.

This method differs from traditional parsing algorithms, which tend to use indicator features, e.g. (Huang et al., 2009) and (Zhang and Nivre, 2011). Indicator features do not provide dense representations, but encode whether the word at a given position equals an element in the vocabulary by means of a large, sparse vector. Such features soon become extremely numerous, especially if a parser does not only consider information of single words, but also of pairs or even triples. This leads to serious sparsity issues, as well as incompleteness and high computational costs.

(Chen and Manning, 2014) use vectorial embeddings to convert words, POS tags and arc labels into a dense, numerical representation. In this research, the pretrained GloVe word embedding is used that is described in (Pennington et al., 2014). The baseline of the current paper uses one-hot encodings for labels and tags. Additional results are obtained using trained embeddings for labels and tags, using the method described in (Mikolov, 2013).

Features are extracted from configurations by a fixed scheme with a clear linguistic interpretation, so the presented approach is a symbolic one. Automated feature learning has also been applied in dependency parsing, e.g. by (Kiperwasser and Goldberg, 2016) and (Cross and Huang, 2016) who use a bidirectional LSTM to learn features in an automated way.

The main alternative to transition-based parsers are graph-based parsers, such as (Hall, 2007) and (Eisner, 1996), and parser that use cascaded chunking, such as (Kudo and Matsumoto, 2002). Hybrid methods, combining graph- and transition-based approaches, have also been advanced, as in (Nivre and McDonald, 2008) and (Zhang and Clark, 2008). Within transition-based parsing, algorithms include arc-standard, which is used by (Nivre, 2004), arc-eager (Nivre, 2003) and arc-hybrid (Kuhlmann et al., 2011). Arc-standard is used in the current research, following (Chen and Manning, 2014). Arc-eager performs greedy parsing and arc-hybrid is a variation on arc-standard that also considers the top word on the buffer in performing reductions.

4 Implementation

In this section we describe how we implemented the (Chen and Manning, 2014) model.

4.1 Processing data

The first task is to process the available training data. The Stanford CoNLL corpus is used, which represents the dependency structure of almost 40,000 English sentences. All words are indexed according to their position in the sentence, along with their POS tag, their head word and the arc label that represents the nature of the relation to their head word. Figure 3 shows how the parsed sentence 'He has good control' is included in the CoNLL training set.

```
1       He      _       PRP     PRP     _       2       nsubj
2       has     _       VBZ     VBZ     _       0       root
3       good    _       JJ      JJ      _       4       amod
4       control _       NN      NN      _       2       dobj
5       .       _       .       .       _       2       punct
```

Figure 3: Example of the CoNLL format.

First, these data have to be processed in such a way that they can be used to train a transition-based parser. This is far from trivial. As explained in Section 2, transition-based parsers do not simply output a dependency structure upon receiving some sentence as input. Rather, they traverse the sentence word by word, and decide between the available actions as they go along.

In the case of transition-based parsing, the input consists of a particular representation of a configuration of words (a stack, a buffer, and a set of arcs). The output is the transition that the parser decides to take, which may either be to shift the top word of the buffer to the stack, to left-reduce the second word on the stack, or to right-reduce the first word on the stack. Reductions are subdivided according to the collection of arc labels that are distinguished. Hence, the challenge is to transform the CoNLL training data for a sentence into a series of configurations, which serve as inputs to the parser. The corresponding targets are the transitions that an ideal parser would make upon witnessing these particular configurations. 'Ideal' here means that the parser performs exactly those actions that eventually result in a correct dependency structure. Only after the data have been transformed into this format, a parser can be trained.

More formally, the dependency structure of any sentence S can be represented by a set of arcs A^S. The set A^S consists of triples of the form $(w_{head}, w_{dependent}, l)$, if we let w_{head} denote a head word, $w_{dependent}$ a dependent and l the arc label that characterizes the dependency. This is also the form in which complete dependency structures can readily be extracted from the CoNLL format. However, because a transition-based dependency parser cannot be trained by simply feeding it S as input and A^S as target, the set A^S has to be rewritten as a set $C^S = \{C_1^S, \ldots, C_n^S\}$ of configurations and a set $T^S = \{T_1^S, \ldots, T_n^S\}$ of transitions. C^S and T^S are such that the application of T_i^S to C_i^S for $i = 1, \cdots, n$ will produce a final configuration C_{n+1}^S whose arcs equal A^S. Every configuration C_i^S consists of a stack *Stack*(C_i^S), a buffer *Buffer*(C_i^S) and an (intermediate) set of labeled arcs *Arcs*(C_i^S) that follows from the reductions that have preceded C_i^S.

A function is implemented that performs the transformation of the set of arcs A^S of some sentence S into the required sets C^S of configurations and T^S of transitions. This is done by initialising a default configuration C_1^S for S whose stack only contains the root, while the words are loaded onto the buffer in the order in which they also occur in the sentence. Note that here, for the words it suffices to take just their index in the sentence, because dependency structures only specify relations between words. Information about the exact words at sentence indices and their POS tags can be stored elsewhere.

Following the initialisation, the target transitions are determined on the basis of the information in A^S. Let $h(A^S)$ denote the head words in A^S, i.e. $h(A^S) = \{w_h \mid \exists w_d, \exists l : (w_h, w_d, l) \in A^S\}$. Applying a transition T_i^S to a configuration C_i^S can only have one new configuration C_{i+1}^S as a result, which can be regarded as the result of applying a transition function t to C_i^S and T_i^S: $t(C_i^S, T_i^S) = C_{i+1}^S$. This new configuration is then added to C^S until all words in the sentence have been reduced and only the root remains at the stack. This procedure is captured by the pseudocode in Algorithm 1.

It is hardcoded in the algorithm that the first transition must be a shift, because no reductions are possible with an empty stack. Next, the algorithm calculates the second configuration with the transition function: $t(C_1^S, T_1^S) = C_2^S$. It checks the size of the stack, which is necessarily two after the initial shift. So it continues to check the top element on the stack, which it stores as $stack_1$, and the second element, which it stores as $stack_2$. After the first transition, $stack_2$ is the root node. The root node cannot be reduced itself, and has a single dependent that should only be right-reduced at the very end of the parse. So the algorithm first performs another shift, but only if there are still words on the buffer. This check is necessary, because the corpus contains sentences of length 1. In the next iteration, the new configuration is computed. Now the two top words on the stack do not include the root, so the algorithm checks if A^S includes a dependency between them with some label l. Suppose there exists such a dependency. Then, depending on whether the head word is the first or the second word on the stack, the algorithm

checks if it can left- or right-reduce the dependent. Transition-based parsing works bottom-up, so once a word has been reduced, no more words can be added as its dependents, implying that reduction can only take place if the dependent does not have any other dependents itself in A^S, i.e. if it is not in $h(A^S)$. If it is in $h(A^S)$, then a shift must be performed, but only if there are still words on the buffer. Assuming correctness of A^S, if none of these conditions are met, it can only mean that the buffer is empty and the last remaining dependency on the stack is the one from the root. This can then be safely accommodated with a final right-reduction.

4.2 Feature extraction

Before the resulting C^S and T^S can, respectively, be used as input to train the parser and as set of targets, features have to be extracted from the configurations. Configurations contain a stack, a buffer and a set of labeled arcs, all of which include a range of information that may not all be required for a decent parser. Following (Chen and Manning, 2014) and (Zhang and Nivre, 2011), we use a rich set of manually selected features. For the baseline results, we use 48 features per configuration. They are listed below:

- *Words*
  - s_i for $i \in \{1, 2, 3\}$: the top 3 words on the stack
  - b_i for $i \in \{1, 2, 3\}$: the top 3 words on the buffer
  - $lc_1(s_i), lc_2(s_i)$ for $i \in \{1, 2\}$: the first and second leftmost children of the top two words on the stack
  - $rc_1(s_i), rc_2(s_i)$ for $i \in \{1, 2\}$: the first and second rightmost children of the top two words on the stack
  - $lc_1(lc_1(s_i))$ for $i \in \{1, 2\}$: the leftmost of leftmost children of the top two words on the stack
  - $rc_1(rc_1(s_i))$ for $i \in \{1, 2\}$: the rightmost of rightmost children of the top two words on the stack
- *POS tags*
  - $s_i.t$ for $i \in \{1, 2, 3\}$: the POS tags of the top 3 words on the stack
  - $b_i.t$ for $i \in \{1, 2, 3\}$: the POS tags top 3 words on the buffer
  - $lc_1(s_i).t, lc_2(s_i).t$ for $i \in \{1, 2\}$: the POS tags of the first and second leftmost children of the top two words on the stack
  - $rc_1(s_i).t, rc_2(s_i).t$ for $i \in \{1, 2\}$: the POS tags of the first and second rightmost children of the top two words on the stack
  - $lc_1(lc_1(s_i)).t$ for $i \in \{1, 2\}$: the POS tags of the leftmost of leftmost children of the top two words on the stack
  - $rc_1(rc_1(s_i)).t$ for $i \in \{1, 2\}$: the POS tags of the rightmost of rightmost children of the top two words on the stack
- *Arc labels*
  - $lc_1(s_i).l, lc_2(s_i).l$ for $i \in \{1, 2\}$: the labels of the arcs between the first and second leftmost children of the top two words on the stack and their head words
  - $rc_1(s_i).l, rc_2(s_i).l$ for $i \in \{1, 2\}$: the labels of the arcs between the first and second rightmost children of the top two words on the stack and their head words

Algorithm 1: Preprocess CoNLL data

Input: labeled arcs A^S for sentence S
Output: corresponding configurations and transitions (C^S, T^S)

$i \leftarrow 1$
initialise C_1^S
$T_1^S \leftarrow$ `SHIFT`
$i \leftarrow 2$
while $length(Stack(t(C_{i-1}^S, T_{i-1}^S))) > 1$ **do**
  $C_i^S \leftarrow t(C_{i-1}^S, T_{i-1}^S)$
  $stack_1 \leftarrow Stack(C_i^S)[-1]$
  $stack_2 \leftarrow Stack(C_i^S)[-2]$
  if $not\ stack_2 = ROOT$ **then**
    if $(stack_1, stack_2, l) \in A^S$ **then**
      if $not\ stack_2 \in h(A^S)$ **then**
        $T_i^S \leftarrow$ `LEFT-REDUCE`(l)
        $A^S \leftarrow A^S \backslash \{(stack_1, stack_2, l)\}$
      else if $length(\textit{Buffer}(C_i^S)) > 0$ **then**
        $T_i^S \leftarrow$ `SHIFT`
    else if $(stack_2, stack_1, l) \in A^S$ **then**
      if $not\ stack_1 \in h(A^S)$ **then**
        $T_i^S \leftarrow$ `RIGHT-REDUCE`(l)
        $A^S \leftarrow A^S \backslash \{(stack_2, stack_1, l)\}$
      else if $length(\textit{Buffer}(C_i^S)) > 0$ **then**
        $T_i^S \leftarrow$ `SHIFT`
    else if $length(\textit{Buffer}(C_i^S)) > 0$ **then**
      $T_i^S \leftarrow$ `SHIFT`
  else if $length(\textit{Buffer}(C_i^S)) > 0$ **then**
    $T_i^S \leftarrow$ `SHIFT`
  else
    $T_i^S \leftarrow$ `RIGHT-REDUCE`$(ROOT)$
  $i \leftarrow i + 1$
end
return (C^S, T^S)

- $lc_1(lc_1(s_i)).l$ for $i \in \{1, 2\}$: the labels of the arcs between the leftmost of leftmost children of the top two words on the stack and their head words
- $rc_1(rc_1(s_i)).l$ for $i \in \{1, 2\}$: the labels of the arcs between the rightmost of rightmost children of the top two words on the stack and their head words

Obviously, not all of these 48 features always contain information. There may be fewer than three words on the stack or the buffer, the top two words on the stack may not (yet) have any children, and if they do have children, these children may not have children themselves yet. Hence, in practice, it does not occur very often that all 48 features are occupied.

Figure 4 illustrates how features are extracted from a configuration. For the given situation, we see that s_1 = 'good', $b_1.t$ = NN, $lc_1(s_2).t$ = PRP, $lc_1(s_2).l$ = nsubj and $rc_1(s_2)$ = NONE because s_2 has no child nodes on the right.

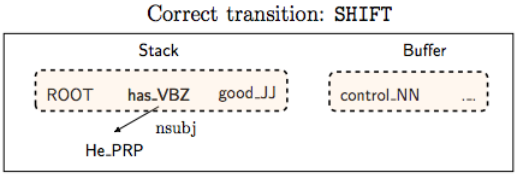


Figure 4: Feature extraction from a configuration.

4.3 Vector embeddings

4.3.1 Baseline

The features that are extracted from a configuration have to be represented numerically before they can be used to train the model. This is why we convert all words, POS tags and arc labels into 50-dimensional vectors. For the words, we use the pretrained GloVe embedding of (Pennington et al., 2014). For the baseline, POS tags and arc labels are represented as one-hot vectors e_j for $1 \leq j \leq n$.[1]

4.3.2 Trained POS and arc embeddings

Additionally, the vector representations of POS tags and arc labels are trained.

[1]Here we follow standard linear algebra notation. In $\mathbb{R}^n$ the vector e_j, for $1 \leq j \leq n$, is the j-th standard basis vector: 1 in j-th entry, and 0 everywhere else. However, as the total number of observed arc labels is 62, we cannot assign a unique e_j to each of them. For this reason we map the twelve least frequent labels to the vectors e_{50}/i for $i = 1, \ldots, 12$.

There are two main algorithms to produce embeddings. Both provide that words occuring often together in a corpus are represented by similar valued vectors. The skip gram method predicts context, the surrounding words, from a given word, while the continuous bag of words method (cbow) predicts the word given its surrounding context. According to Mikolov, Skip gram provides a higher accuracy for large corpora, a higher dimension of embeddings, and rare words, while Cbow is less computationally expensive and faster to train. Skip gram was selected due to its accuracy.

The skip gram algorithm uses a neural network with a single hidden layer and an output activation function. The input consists of one hot vectors with dimension the same size as the size of the vocabulary. The hidden layer consists of 50 dimensional vectors (the same as the desired embedding length), which, when fully trained, provide the final embeddings for the words. Similarity of vectors is measured by taking their normalized dot products, or cosine similarity. Error at each step is measured by noise contrastive estimation (NCE), by calculating the logarithmic ratio of the probability of predicting a noise sample and of predicting correct words. The network backpropagates this error and updates weights using stochastic gradient descent.

Implementation

The skip gram algorithm was implemented using the tensorflow library and word2vec basic example code.

Experiments on each of the following variables (with the exception of dimension) were carried out to find the most accurate settings. The results for each were evaluated using t-distributed stochastic neighbour embedding (t-SNE). t-SNE is a 2 dimensional reduced dimensional representation of embeddings, such that similar valued embeddings are mapped close together. The t-SNE plots were evaluated on the basis of clusters such as noun and verb POS tag clusters and the distance between certain marker arc labels, for example that amod should be closer to num than nsubj (Chen and Manning).

- Dimension of embeddings=50 A higher dimension is suitable for a larger corpus, as the greater accuracy obtained with more data can be matched with that obtained with higher di-

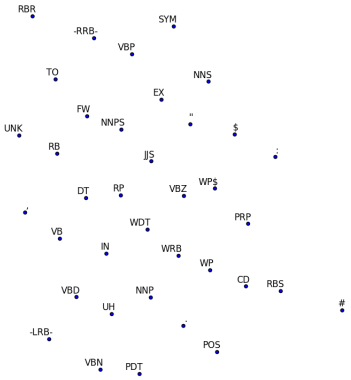


Figure 5: A visualization of the POS tag embeddings.

Figure 6: A visualization of the mixed tag embeddings.

mensions. The corpus is relatively large as, POS tags and arc labels, are few in number so occur very often. A dimension of 50 was chosen to balance accuracy with time to train the dependency parsing model.

- Batch size=50 This is the size of the sample of words from which the neural network trains in one iteration and from which error is measured. A larger size, therefore increases accuracy but takes longer to converge.
- Skip window=2 This is the number of words on either side of the input word that the neural network will attempt to predict. A larger window is suitable for sentences in which dependencies extend further. However, syntactic structures, such as POS tags are likely to be strongly dependent on the 1-2 tags on either side of it.
- Number sampled=10 This is the size of the sample of words used to calculate NCE loss.
- Development parameters: validation set=10, valid window=20, These control the size of the validation set selected from a window of the most common words.

The inbuilt t-SNE function allowed customisation of perplexity and number of iterations, which both lead to a more sensitive representation. Following experimentation on these parameters, values of 60 and 30000 were selected, respectively.

Further possibilities

Training words, POS tags and arc labels with the same model could provide a higher degree of context during training and encode the relation between the three categories of embedding. In the latter stages of the project, vectors for all three categories were jointly trained. Although, there was insufficient time remaining to utilize these in the project, an interesting comparison of results could be made in future work. The advanced word2vec implementation was also considered.

4.4 Training the neural network

A Multilayer Perceptron (MLP) is trained for the task of predicting correct transitions from configurations. The network is trained on configuration-transition pairs $\{(C_i, T_i)\}_{i=1}^{m}$ that are extracted from the training corpus as described in Section 4.1, and embedded in $\mathbb{R}^n$ as described in section 4.3. For the implementation of the neural network the MLP of the Python library scikit-learn[2] is used (Pedregosa et al., 2011). In the choice of hyper-parameters, (Chen and Manning, 2014) are followed wherever possible, although the choice for scikit-learn has forced the choice for alternatives over some of the settings adopted by Chen and Manning. It will be noted below whenever this occurs.

We use a standard MLP architecture with one hidden layer of 200 nodes, and use the tanh function as non-linear transformation (see Eq.1). Chen and Manning use the cubic function $x \mapsto x^3$, but this is choice of non-linear transformation is not available in scikit-learn—the tanh function is the closest available option. The input layer accepts vectors of size 2,400. These are the vector representations of the configurations (48 features $\cdot\, 50$ dimensions). The hidden layer is followed by a soft-max output-layer with $2|\mathcal{T}| + 1$ nodes, where $\mathcal{T}$ is the collection of arc labels used: $|\mathcal{T}|$ nodes for a right-arc with corresponding label, $|\mathcal{T}|$ for a

[2]http://scikit-learn.org/stable/modules/neural_networks_supervised.htmlmulti-layer-perceptron

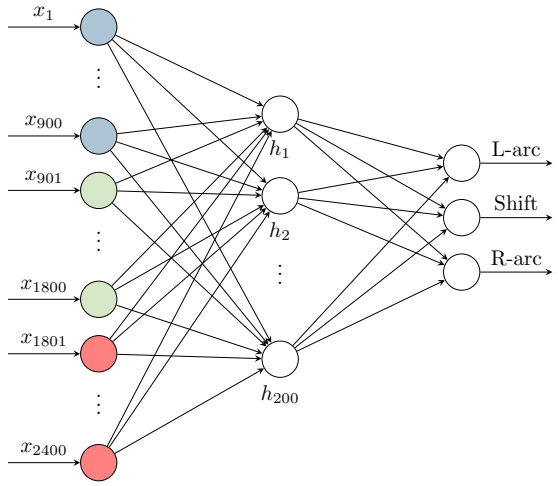


Figure 7: The MLP architecture. The colours indicate the feature input corresponding to the node: blue for word features, green for POS features, red for arc label features. Note that the last layer is simplified: the L-arc and R-arc node represent $|\mathcal{T}|$ distinct nodes; one for each label.

left-arc with corresponding label, and 1 node for a shift.

$$\tanh : \mathbb{R} \to \mathbb{R} : x \mapsto \frac{e^x - e^{-x}}{e^x + e^{-x}} \qquad (1)$$

As loss function we use cross-entropy with an additional l_2 regularization term $\lambda = 10^{-8}$. We experimented on the development set with two optimization methods: the 'adam' solver (Kingma and Ba, 2014), which is closest to the adagrad method used by (Chen and Manning, 2014), and stochastic gradient descent (SGD). Due to better performance on the development set SGD was chosen over adam. For SGD a learning rate of 0.001 was used, following (Chen and Manning, 2014).

For the SGD we let sentence-length dictate batch-size: if a training sentence S has length n, we extract $2n$ pairs (C_i^s, T_i^s) in the preprocessing stage[3]. This gives $\{(C_i^s, T_i^s)\}_{i=1}^{2n}$. This is the batch input into our network.

We train the MLP on the full Stanford CoNLL training set[4] for multiple epochs, and evaluate the performance at intermediate stages (see next section). We do this twice: ones with one-hot POS tag and arc label embeddings; one time for the trained ones. The training is repeated for another epoch as long as improvement is made. When the improvement of accuracy on the development set stagnates the training is halted.

4.5 Prediction and evaluation

The trained parser is used to generate parses of sentences by successively predicting transitions from configurations. In turn, the predicted parses of these sentences are evaluated against their gold-standard parses. This is done on the development set[5] to determine some final hyperparameter-settings[6], and on the test set to generate the final results[7].

To produce a predicted parse the trained parser reads in a sentences from the CoNLL file and disregards all the information pertaining to the dependency structure: only the words, their order, and their POS tags are considered. During each step of the parsing the model extracts all specified feature-information from the current configuration, inputs this to the trained neural network in the embedded, vectorized form, and picks the transition with the highest soft-max probability. Herein we follow (Chen and Manning, 2014) by only considering the soft-max of the feasible transitions[8]. This transition is executed, and the resulting configuration is determined. When finished, the predicted dependency parses are recorded in the CoNLL-format.

To evaluate the performance of the model, we evaluate the accuracy of the predicted dependency parses against their gold-standard parses. For this purpose we use a perl script that is written for (Kiperwasser and Goldberg, 2016)[9]. The script compares a predicted parse in the CoNLL-file against its gold-standard CoNLL, and records the results as measured by two metrics: the labeled accuracy score (LAS, see Eq. 2) and the the unlabeled accuracy score (UAS, see Eq. 3). The UAS records the fraction of correctly predicted arcs, and the LAS records the fraction of correctly predicted arcs that in addition also carry the correct label $l \in \mathcal{T}$:

[3] For each word in the sentence we perform exactly two transitions: one shift and one reduction

[4] train-stanford-raw.conll ($\sim 40,000$ sentences)

[5] dev-stanford-raw.conll ($\sim 1,800$ sentences)

[6] The choice for SGD over adam as optimization algorithms was made this way.

[7] test-stanford-raw.conll ($\sim 2,500$ sentences)

[8] E.g. if the buffer of the current configuration is empty, a shift transitions is not feasible.

[9] This script can be found at https://github.com/elikip/bist-parser/blob/master/barchybrid/src/utils/eval.pl

$$\text{LAS} = \frac{\#\{\text{correct arcs with correct labels}\}}{\text{total number of arcs}} \quad (2)$$

$$\text{UAS} = \frac{\#\{\text{correct arcs}\}}{\text{total number of arcs}} \quad (3)$$

5 Results

This section discusses the results obtained by the parsers trained as described above: one using one-hot vector embeddings for POS tags and arc labels and the other using trained embeddings for these features. Moreover, we attempt to account for the performance of the parsers by interpreting the trained weights. Based on these observations we formulate a hypothesis that is tested in an experiment, the results of which are discussed at the end of the section.

5.1 Results

The Stanford CoNLL corpus is used to train the two parsers. The training set contains $\sim$ 40,000 sentences with their gold-standard dependency parse. The two parsers are trained for 12 epochs, after which stagnation of improvement of accuracy on the development set sets in.

Both parsers are evaluated on the test-set at every epoch, the results of which are shown in figure 9. A steady learning curve is visible with stagnation around the twelfth epoch.[10] Figure 8 compares the accuracy of both parsers, and (Chen and Manning, 2014).

We can make two observations on the basis of these results. First, the two parsers perform almost identically: the parser with trained embeddings does slightly better during the first 10 epochs of training, but is matched and finally surpassed by the parser with one-hot embeddings. The difference however never exceeds 0.5%, and most of the time it is much less than that. Secondly, both parsers perform around 6-8% worse than the parser of (Chen and Manning, 2014). Some of the possible reasons for this deficiency are mentioned in the Discussion.

[10] Actually, as we can see from the curve, some very small improvement might have been made on the test set if training had continued 1-3 epochs longer: in Figure 9 we still see some tendency of improvement. This was not the case on the development set, however, which is where our decision to stop training is based on. But, based on our experience with the development set, this improvement would probably have been only in the order of 0.1%.

		Dev		Test	
Parser		UAS	LAS	UAS	LAS
Our parser	One-hot	85.9	83.0	85.8	83.1
	Trained	86.0	83.3	85.6	82.9
C & M		92.2	91.0	92.0	90.7

Figure 8: Accuracy percentages for the described parsers with POS-tags and arc labels represented as one-hot vectors and as trained embeddings, and performance of (Chen and Manning, 2014).

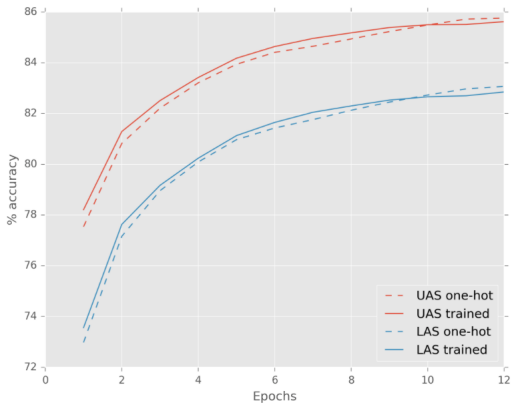


Figure 9: The UAS and LAS scores of the trained model on the test-set, evaluated after each epoch of training. Dashed lines show one-hot embeddings, continuous lines show trained embeddings.

5.2 Interpreting the trained weights

This section investigates what aspects of the model account for its performance. We focus on what the trained weights of the MLP capture, and we will see that they can inform us on the usefulness of the features we used. We focus primarily on the weights from the parser with trained embeddings, and briefly address the one-hot embeddings at the end.

5.2.1 Trained embeddings

We recall the architecture of the MLP: the network takes input vectors $x \in \mathbb{R}^{2400}$ that are transformed by a weight-matrix $W \in \mathbb{R}^{200\times2400}$ to give $h = Wx_i$, the vector which in turn is transformed, entry-wise, by the tanh function. We focus on the action of one hidden node, say node i, such that $h_i = w_i^T x_i$, where $w_i \in \mathbb{R}^{2400}$ is the i-th row of W. This vector w_i can be partitioned horizontally into 3 sections: entries 1 to 900 hold the 900

weights belonging to the embeddings of the 18 word features ($18 \cdot 50 = 900$); entries 901 to 1,800 hold the 900 weights belonging to the embeddings of the 18 POS features ($18 \cdot 50$); and entries 1,801 to 2,400 hold the 600 weights belonging to the embeddings of the 12 arc features ($12 \cdot 50 = 600$).

With this in mind, we reshape the vector w_i to the matrix $W_i \in \mathbb{R}^{50 \times 48}$ by repeatedly peeling off 50 entries from the top of w_i and letting these form the columns of W_i. See Figure 10. In the resulting matrix W_i each column j contains the 50 weights corresponding to the 50-dimensional embedding of exactly one feature: feature j.

The reshaped matrices are plotted using a heatmap, see Figure 11. In all four plots we observe a pronounced vertical bar in the columns $18, 19, 20$ and, to a somewhat lesser extent, in columns $0, 1, 2$: these weights are weights with larger absolute value than the weights in other columns. Additionally, we note that some of the columns hardly receive any weight whatsoever; these columns are very light. We can interpret this as an indication of the relative importance of features in the classification: more weight means more importance, less weight means less importance.

However, these plot are informative only to a limited extent: there are 200 hidden nodes, and here we consider only six of them. For a more global view of the action of the weight matrices, we plot a sum of all the matrices W_i, where we take the absolute values of the entries as we are interested in both large negative and large positive values. The resulting heatmap—now in grey-scale as all values are positive—can be seen in Figure 12.

We can clearly discriminate three different regions corresponding to the three types of features used: 18 columns for word features, followed by 18 columns for POS features, followed by 12 columns for arc label features. For each of these regions we can see a clear color gradience going from left to right; from dark in the beginning, to almost white at the end. We recall that in each region the first, darker-colored, columns correspond to features extracted from nearby words: $s_1, s_2, s_1.t, s_2.t$ etc., while more distant features such as $lc_1(lc_1(s_1))$ and $rc_1(rc_1(s_2))$ that carry information about already reduced words are all found in the lighter-colored end of the regions of the heatmap. This finding corresponds well with the intuitive expectation that features of words high-up on the stack are more important in making a correct classification than features of more distant words, such as already reduced words, which are often assigned the value NONE in absence of already reduced words.

The most visible pattern, however, is perhaps the following: columns 18 and 19, corresponding to features $s_1.t, s_2.t$, receive by far the most weight, followed—with ample margin—by columns 0 and 1, corresponding to features s_1, s_2. Under our interpretation that more weight means more importance, we can hypothesize that these features are responsible for a large part of the performance of the parser. We will test this hypothesis, and discuss the findings, below.

5.2.2 One-hot

Figure 13 shows the heatmap of $\sum_{i=1}^{200} |W_i|$ for a MLP trained on the one-hot embeddings. Noticeable is again the concentration of weight in columns 18 and 19, although this concentration is now no longer in the form of a solid bar, but rather fractured and located at a number of distinct rows.[11] The reason for this is that every time a one-hot vector e_j, for $1 \geq j \leq n$, is fed into the neural network, it selects, for each internal node only one of the 50 weights that transform it: the weight node corresponding to the one non-zero entry j of e_j. Hence only this weight is activated upon that input. And since some POS tags are much more frequent than others, some of the 50 weights corresponding to an internal node will receive much more weight during training than other weights. To illustrate this with our plot: we assigned the one-hot vector e_{44} to the NN POS tag, and columns 18 and 19 have much weight in row 44. On the other hand, we assigned e_{33} to UH[12], and this has almost no weight in columns 18 and 19.

5.3 Importance of features

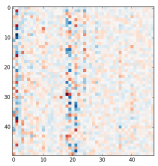
The trained weights appear to hold much interpretative value. In this section, we attempt to evaluate whether we were right to hypothesize that features with higher associated weights contribute more to the parsing performance.

[11] Note that the the first columns (0-17) of this plot still show solid bars. This is because here the word-embeddings are not one-hot—for these the GloVe embeddings are still used.

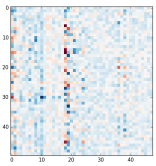
[12] UH denotes an interjection.

$$\begin{bmatrix} w_1 \\ \vdots \\ w_{900} \\ w_{901} \\ \vdots \\ w_{1800} \\ w_{1801} \\ \vdots \\ w_{2400} \end{bmatrix} \xrightarrow{reshape} \underbrace{\begin{bmatrix} w_1 & w_{51} & \dots & w_{851} & w_{901} & \dots & w_{1801} & w_{1851} & \dots & w_{2351} \\ w_2 & w_{52} & \dots & w_{852} & w_{902} & \dots & w_{1802} & w_{1852} & \dots & w_{2352} \\ \vdots & \vdots & & \vdots & \vdots & & \vdots & \vdots & & \vdots \\ \vdots & \vdots & & \vdots & \vdots & & \vdots & \vdots & & \vdots \\ w_{50} & w_{100} & \dots & w_{900} & w_{950} & \dots & w_{1850} & w_{1900} & \dots & w_{2400} \end{bmatrix}}_{48}$$

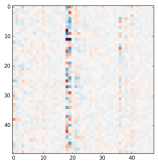
Figure 10: Reshaping the vector w_i to the matrix W_i.



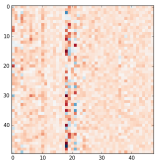
(a) Node 5



(b) Node 6



(c) Node 14



(d) Node 26

Figure 11: Heatmaps of reshaped weight vectors W_i for a selection of nodes. Blue indicates negative value, red indicates positive value. Lighter shades are closer to zero.

5.3.1 Feature experiment

The following experiment is carried out. First, the model is trained with the features that are expected to be the least informative: these are the features corresponding to the lightest regions of Figure 12. This means that we remove $all\ features$ out of 48 $except$ for the following 20:

- s_i for $i \in \{1, 2, 3\}$
- b_i for $i \in \{1, 2, 3\}$
- $s_i.t$ for $i \in \{1, 2, 3\}$
- $b_i.t$ for $i \in \{1, 2, 3\}$
- $lc_1(s_i).l$ for $i \in \{1, 2\}$
- $lc_2(s_i).l$ for $i \in \{1, 2\}$
- $rc_1(s_i).l$ for $i \in \{1, 2\}$

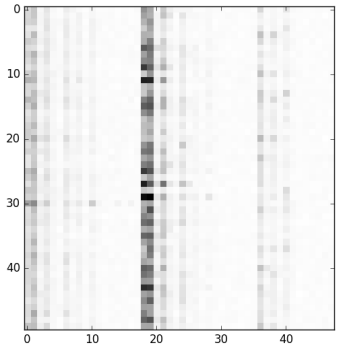


Figure 12: Heatmap of $\sum_{i=1}^{200} |W_i|$ for the model trained on the pre-trained embeddings. Note: we write $|A| = (|a_{ij}|)$ for $A \in \mathbb{R}^{m \times n}$.

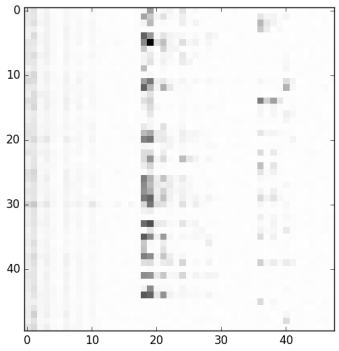


Figure 13: Heatmap of $\sum_{i=1}^{200} |W_i|$ for the model trained on the one-hot vectors.

- $rc_2(s_i).l$ for $i \in \{1, 2\}$

Besides this, the model is trained exactly as before.

The same procedure is carried out with the features that were identified as being the most informative. These are the features corresponding to

the darkest regions of Figure 12. Now we remove only the following 4 features:

- s_i for $i \in \{1, 2\}$
- $s_i.t$ for $i \in \{1, 2\}$

Again, the model is trained as before.

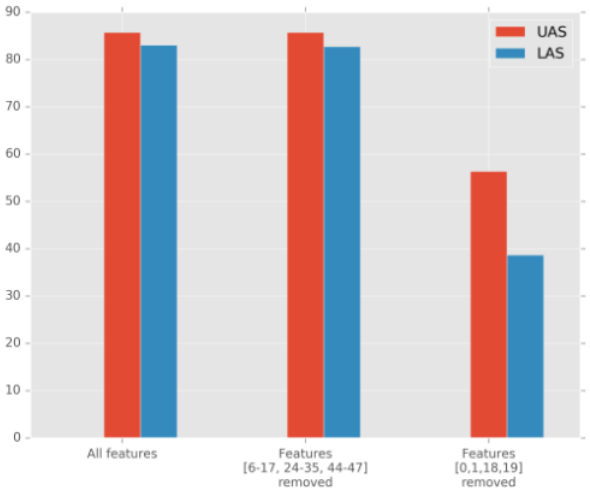


Figure 14: Results of parsers with indicated features removed.

The performance of the trained models on the test set is shown in Figure 14, where they are compared to the performance upon training with all features.

The effects of removing more than half of the low-weight features is almost negligible: the performance of the parser drop by only less than 0.5%. Removing the four features $s_1, s_2, s_1.t, s_2.t$, however, completely cripples the parser: the UAS drops to 56%, and the LAS drops even further, under 40%.

5.3.2 Interpreting results of experiment

It is very clear from these results that the four features corresponding to the first two words on the stack and their POS tags are pivotal for the MLP to predict correct transitions. Intuitively, it is clear that the decision to perform a left- or right-reduction for two words depends crucially on what these words are, and what POS tag they have.

From the results in Figure 14 we can see that, nevertheless, predicting a correct dependency arc is still possible without these four features: the UAS remains above 50%. The biggest effect can be seen in the LAS: without the POS tags or word information of the two words that are assigned a dependency relation, predicting the correct arc label becomes almost impossible. The MLP now has almost no information to go by.

From the results of these experiments we can conclude that our hypothesis is correct: the more weight a feature receives in our trained network, the more important that feature is for the classification task. Likewise, features that receive almost no weight have hardly any importance.

In addition, we can conclude that studying the weight vectors in the way shown in Figure 12 is very useful. The particular representation shown in Figure 12 is an innovation over (Chen and Manning, 2014), who only considered plots of individual nodes (three to be exact). As we can see when contrasting Figures 11(a)-(d) with Figure 12, this method is very limited. The plots of individual weight matrices are highly variable, and only show a glimpse of the underlying pattern. The plot of the absolute sum of the weight matrices on the other hand is much more representative and quite beautifully shows the underlying regularity.

6 Discussion

This paper meant to study the dependency parser described by (Chen and Manning, 2014), and to identify the features that play the most important role in parsing a sentence. Following the same approach, we have reached a degree of accuracy that is some 6 % lower than the one obtained in the paper, but the results are in a sufficient order of magnitude to justify some concluding remarks about the method.

In the preceding section, we analysed the weights of the MLP that are obtained after training. The higher weights in regions associated with particular features fueled the hypothesis that these features may be most crucial in making correct parsing decisions. This hypothesis turned out to be correct after we removed the features concerned and tested the resulting network. No significant change in performance was observed after removing 20 of the least influential features, whereas accuracy scores dropped dramatically after removing the 4 most heavily weighted ones.

This result is positive in the sense that we appear to have identified the most important linguistic features for a transition-based parser. It is negative in the sense that it shows how many of the included features of (Chen and Manning, 2014) are virtually redundant. The rich feature set they

define mostly consists of superfluous information, requiring more computational effort while hardly improving performance.

Initially, we planned to study how parsing performance could be improved by extending the set of features. This plan was aborted after obtaining the described result, as features to be added in the style of (Chen and Manning, 2014) could only encode information of words further down the parse tree, whose influence would be even less significant.

This is not to say that there are no features of a different kind than the ones implemented in (Chen and Manning, 2014) that may improve performance of the parser. Departing from their approach, it would be interesting to study such features in future work. Rather than only looking at words, their POS tags and arc labels, we could consider even richer sets of features, as is done in e.g. (Zhang and Nivre, 2011). We could consider the distance between a pair of head and dependent, following (Goldberg and Elhadad, 2010). Also, we could include the number of dependents attached to a head word. This is a variable known as 'valency', and it is used in (Zhang and Clark, 2008). It would be interesting to study the effects of adding such alternative features to our parser, or to invent more options.

A number of possible reasons may explain why our parser did not achieve the same accuracy scores as the one in (Chen and Manning, 2014). (a) we used a different activation function, namely tanh instead of the cubic function. (b) we used a differnt optimization method: SGD instead of AdaGrad. (c) (Chen and Manning, 2014) may have trained their POS tag and arc label embeddings in a more informed way than we did. (d) The pretrained word vectorizations we adopted did not include all words of the corpus. It is likely that (Chen and Manning, 2014) accommodated such cases more intelligently than we did.

Finally, looking beyond (Chen and Manning, 2014), we can suggest many future directions. For instance, we could consider the evaluation scripts in more detail to see what kinds of errors the parser makes most often, and base modifications on this information. Also, we would like to train on corpora in different languages, such as Czech and Arabic, to see how performance decreases and consider how this could be fixed. We could consider a different arc system, such as arc-hybrid or arc-eager. Finally, it would be interesting to experiment with the network topology, and consider alternatives such as a recurrent neural network for classification.

7 Team Responsibility

- **Daan** Coded depparser together with Mathijs (50/50), did much of the training, predicting, and evaluating of models; experimented with different embeddings (also many random embeddings not used in the end); experimented with removing features. Made presentation together with Mathijs, using basic layout prepared by Selina. Made neural network in tikz (very proud). Wrote sections 4.4-4.5 and all of section 5 of the paper. Made all the graphs, barplots and heatmaps of the Results section.

- **Mathijs** Coded depparser together with Daan (50/50). Did initial testing, training and predicting, pre- and post-processing data, set-up of basic architecture, extension to arc labels. Made presentation together with Daan, using basic layout prepared by Selina. Wrote Abstract, Relevant Literature (Section 3), Sections 4.1-4.3.1, Discussion; prepared some figures.

- **Selina** Worked on word2vec_basic implementation and tuning, a more advanced (scrapped) word2vec, structuring the paper and presentation, repairing layout. Worked on Introduction, Problem and word embeddings visualizations in the paper.

- **May** Trained embeddings and t-SNEs on word2vec basic implementation, carried out experiments with different parameters. Trained embeddings of words, POS tags and arc labels jointly (not implemented). Wide reading of relevant literature. Wrote Section 4.3.2 on the theory and process of training the embeddings, Wrote original long version of Relevant Literature (considered too detailed and technical and summarised in Section 3).

References

Bernd Bohnet. 2010. Very high accuracy and fast dependency parsing is not a contradiction. In *Proceedings of the 23rd international conference on computational linguistics*, pages 89–97. Association for Computational Linguistics.

Danqi Chen and Christopher D Manning. 2014. A fast and accurate dependency parser using neural networks. In *EMNLP*, pages 740–750.

James Cross and Liang Huang. 2016. Incremental parsing with minimal features using bi-directional lstm. *arXiv preprint arXiv:1606.06406*.

Jason M Eisner. 1996. Three new probabilistic models for dependency parsing: An exploration. In *Proceedings of the 16th conference on Computational linguistics-Volume 1*, pages 340–345. Association for Computational Linguistics.

Yoav Goldberg and Michael Elhadad. 2010. An efficient algorithm for easy-first non-directional dependency parsing. In *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 742–750. Association for Computational Linguistics.

Yoav Goldberg and Joakim Nivre. 2012. A dynamic oracle for arc-eager dependency parsing. In *COLING*, pages 959–976.

Keith Hall. 2007. K-best spanning tree parsing. In *Annual Meeting-Association for Computational Linguistics*, volume 45, page 392.

Liang Huang, Wenbin Jiang, and Qun Liu. 2009. Bilingually-constrained (monolingual) shift-reduce parsing. In *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing: Volume 3-Volume 3*, pages 1222–1231. Association for Computational Linguistics.

Diederik P. Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980.

Eliyahu Kiperwasser and Yoav Goldberg. 2016. Simple and accurate dependency parsing using bidirectional LSTM feature representations. *TACL*, 4:313–327.

Lingpeng Kong and Noah A Smith. 2014. An empirical comparison of parsing methods for stanford dependencies. *arXiv preprint arXiv:1404.4314*.

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

Taku Kudo and Yuji Matsumoto. 2002. Japanese dependency analysis using cascaded chunking. In *proceedings of the 6th conference on Natural language learning-Volume 20*, pages 1–7. Association for Computational Linguistics.

Marco Kuhlmann, Carlos Gómez-Rodríguez, and Giorgio Satta. 2011. Dynamic programming algorithms for transition-based dependency parsers. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies-Volume 1*, pages 673–682. Association for Computational Linguistics.

Miikkulainen Risto Mayberry, Marshall R. 1999. Sardsrn: a neural network shift-reduce parser.

Tomas Mikolov. 2013. word2vec: Tool for computing continuous distributed representations of words.

Jens Nilsson, Sebastian Riedel, and Deniz Yuret. 2007. The conll 2007 shared task on dependency parsing. In *Proceedings of the CoNLL shared task session of EMNLP-CoNLL*, pages 915–932. sn.

Joakim Nivre and Ryan T McDonald. 2008. Integrating graph-based and transition-based dependency parsers. In *ACL*, pages 950–958.

Joakim Nivre. 2003. An efficient algorithm for projective dependency parsing. In *Proceedings of the 8th International Workshop on Parsing Technologies (IWPT*. Citeseer.

Joakim Nivre. 2004. Incrementality in deterministic dependency parsing. In *Proceedings of the Workshop on Incremental Parsing: Bringing Engineering and Cognition Together*, pages 50–57. Association for Computational Linguistics.

Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12(Oct):2825–2830.

Jeffrey Pennington, Richard Socher, and Christopher D. Manning. 2014. Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543.

Pontus Stenetorp. 2013. Transition-based dependency parsing using recursive neural networks. In *NIPS Workshop on Deep Learning*.

Yue Zhang and Stephen Clark. 2008. A tale of two parsers: investigating and combining graph-based and transition-based dependency parsing using beam-search. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, pages 562–571. Association for Computational Linguistics.

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: short papers-Volume 2*, pages 188–193. Association for Computational Linguistics.