Parsing of Lexicalised Linear Context-Free Rewriting Systems via Supertagging

Name : Alex Ivliev
Number : 4604723
Course : INF-PM-FPG
Supervisor : Prof. Dr.-Ing. habil. Heiko Vogler
Advisor : Dipl.-Inf. Richard Mörbitz

Date of submission : 6th May 2020

Aufgabenstellung

„Parsing of lexicalised linear context-free rewriting systems via supertagging“

Technische Universität Dresden
Fakultät Informatik

Student:	Alex Ivliev
Geburtsdatum:	3. Januar 1998
Matrikelnummer:	4604723
Studiengang:	Diplom Informatik
Immatrikulationsjahr:	2016
Modul:	INF-PM-FPG
Studienleistung:	360h (12LP)
Beginn am:	7. November 2019
Einzureichen am:	6. Mai 2020
Verantw. Hochschullehrer:	Prof. Dr.-Ing. habil. Heiko Vogler
Betreuer:	Dipl.-Inf. Richard Mörbitz

Parsing. Beim Parsing (Zerlegen) sollen mithilfe einer Grammatik syntaktische Analysen eines gegebenen Satzes berechnet werden. Eine solche syntaktische Analyse wird üblicherweise Parsebaum genannt. Für große Grammatiken ist diese Berechnung oft aufwendig und man versucht daher mit verschiedenen Methoden, den Suchraum zu beschneiden. Eine dieser Methoden ist Supertagging [JS94]. Dabei wird jedem Symbol des gegebenen Satzes zunächst ein Elementarbaum einer lexikalisierten Tree-Adjoining-Grammatik zugeordnet. Durch Kombination dieser Elementarbäume erhält man dann einen Parsebaum. Die Leistung von Supertagging-basierten Systemen lässt sich durch den Einsatz neuronaler Modelle steigern [Vas+16; Kas+17].

Linear context-free rewriting systems [VWJ86]. Natürliche Sprachen weisen Merkmale auf, die von kontextfreien Grammatiken nicht darstellbar sind, z.B. kann ein Teilsatz eine Lücke haben, in die vom Kontext abhängiger Inhalt eingefügt wird. Um die hohe Parsingkomplexität kontextsensitiver Grammatiken (PSPACE-complete) zu

vermeiden, untersucht man Formalismen, die solche Lücken zwar darstellen können, aber dennoch polynomiell parsebar sind. Man fasst solche Formalismen unter dem Begriff mildly context-sensitive formalisms zusammen. Dazu gehören z.B. head grammars, tree adjoining grammars, combinatory categorial grammars, linear indexed grammars, multiple context-free grammars, und minimalist grammars. Linear context-free rewriting systems (kurz: LCFRS) wurden eingeführt, um mildly context-sensitive formalisms einheitlich darzustellen. Alle oben genannten (und noch einige weitere) Formalismen erzeugen eine kleinere oder die gleiche Klasse von Stringsprachen wie LCFRS [Sek+91; VWJ86; WJ88; Vij88; Mic01b; Mic01a]. Das Parsing von LCFRS ist daher von besonderer Bedeutung für die Verarbeitung natürlicher Sprachen [Eva11]. Eine LCFRS wird lexikalisiert genannt, wenn jede ihrer Regeln genau ein Terminalsymbol, den sogenannten Anker, enthält.

Aufgabe. Der Student soll ein Verfahren zur Induktion von lexikalisierten LCFRS aus einem Dependency-Korpus [KS09] implementieren. Ein geeigneter Korpus wird zur Verfügung gestellt.

Der Student soll einen Supertagging-basierten Parser entwickeln, der sich in zwei Phasen unterteilt. In der ersten Phase (*Supertagger*) sollen, für eine gegebene LCFRS und einen gegebenen Satz, jeder Satzposition die n besten Grammatikregeln (*Supertags*) zugeordnet werden. Die Gewichtung der Regeln erfolgt durch ein neuronales Modell. Zu diesem Zweck soll ein geeignetes vortrainiertes neuronales Netz (z.B. BERT [Dev+18]) verwendet und für die Aufgabe angepasst werden. In der zweiten Phase (*Parser*) soll für die reduzierte LCFRS aus Phase 1 und den gegebenen Satz der wahrscheinlichste Dependency-Baum berechnet werden. Dazu soll ein geeigneter LCFRS-Parser verwendet oder ein eigener implementiert werden.

Der Student soll experimentell gute Metaparameter seines Systems (Anzahl der Supertags, Vereinfachungen der Grammatik) bestimmen. Abschließend soll der Supertagging-basierte Parser auf dem gegebenen Korpus hinsichtlich Genauigkeit und Laufzeit evaluiert werden. Die dabei erzielten Ergebnisse sollen mit denen eines LCFRS-Parsers, der kein Supertagging verwendet, verglichen werden.

Ende Januar 2020 soll der Student seinen aktuellen Stand im Rahmen des Freitagsseminars vorstellen. Weiterhin soll er eine Projektarbeit (schriftliche Ausarbeitung) anfertigen. Das Modul wird mit einem 45-minütigen benoteten Kolloquium (gemäß Modulbeschreibung) abgeschlossen. Die Gesamtnote des Moduls entspricht dem gewichteten Durchschnitt der Note für die Projektarbeit und der Note für das Kolloquium im Verhältnis 3 zu 1.

Form. Die Arbeit muss den üblichen Standards wie folgt genügen. Die Arbeit muss in sich abgeschlossen sein und alle nötigen Definitionen und Referenzen enthalten. Die Urheberschaft von Inhalten – auch die eigene – muss klar erkennbar sein. Fremde Inhalte, z.B. Algorithmen, Konstruktionen, Definitionen, Ideen, etc., müssen durch genaue Verweise auf die entsprechende Literatur kenntlich gemacht werden. Lange wörtliche Zitate sollen vermieden werden. Gegebenenfalls muss erläutert werden, inwieweit und zu welchem Zweck fremde Inhalte modifiziert wurden. Die Struktur der Arbeit muss klar

erkenntlich sein, und der Leser soll gut durch die Arbeit geführt werden. Die Darstellung aller Begriffe und Verfahren soll mathematisch formal fundiert sein. Für jeden wichtigen Begriff sollen Erläuterungen und Beispiele angegeben werden, ebenso für die Abläufe der beschriebenen Verfahren. Wo es angemessen ist, sollen Illustrationen die Darstellung vervollständigen. Bei Diagrammen, die Phänomene von Experimenten beschreiben, muss deutlich erläutert werden, welche Werte auf den einzelnen Achsen aufgetragen sind, und beschrieben werden, welche Abhängigkeit unter den Werten der verschiedenen Achsen dargestellt ist.

Für die Implementierung soll eine ausführliche Dokumentation erfolgen, die sich angemessen auf den Quelltext und die schriftliche Ausarbeitung verteilt. Dabei muss die Funktionsfähigkeit des Programms glaubhaft gemacht und durch geeignete Beispielläufe dokumentiert werden.

Der Student verpflichtet sich, ihm im Rahmen dieser Arbeit zugänglich gemachte Daten und Software (einschließlich Quellcode) lediglich zur Erledigung der Aufgaben zu verwenden und ansonsten vertraulich zu behandeln.

Dresden, 5. November 2019

Unterschrift von Heiko Vogler

Unterschrift von Alex Ivliev

Literatur

[Dev+18] Jacob Devlin, Ming-Wei Chang, Kenton Lee und Kristina Toutanova. „BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding". In: (2018). arXiv: 1810.04805 [cs.CL].

[Eva11] Kilian Evang. „Parsing discontinuous constituents in English". Masterarbeit. Universität Tübingen, 5. Jan. 2011. URL: http://kilian.evang.name/publications/mathesis.pdf.

[JS94] Aravind K. Joshi und B. Srinivas. „Disambiguation of Super Parts of Speech (or Supertags): Almost Parsing". In: *Proceedings of the 15th Conference on Computational Linguistics - Volume 1*. COLING '94. Kyoto, Japan: Association for Computational Linguistics, 1994, S. 154–160. DOI: 10.3115/991886.991912.

[Kas+17] Jungo Kasai, Robert Frank, R. Thomas Mccoy, Owen Rambow und Alexis Nasr. „TAG Parsing with Neural Networks and Vector Representations of Supertags". In: *Conference on Empirical Methods in Natural Language Processing*. Copenhague, Denmark, 2017, S. 1712–1722. URL: https://hal.archives-ouvertes.fr/hal-01771494.

[KS09] Marco Kuhlmann und Giorgio Satta. „Treebank Grammar Techniques for Non-projective Dependency Parsing". In: *Proceedings of the 12th Conference of the European Chapter of the Association for Computational Linguistics*. EACL '09. Athens, Greece: Association for Computational Linguistics, 2009, S. 478–486. URL: http://dl.acm.org/citation.cfm?id=1609067.1609120.

[Mic01a] Jens Michaelis. „Derivational Minimalism Is Mildly Context-Sensitive". English. In: *Logical Aspects of Computational Linguistics*. Hrsg. von Michael Moortgat. Bd. 2014. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2001, S. 179–198. ISBN: 978-3-540-42251-8. DOI: 10.1007/3-540-45738-0_11.

[Mic01b] Jens Michaelis. „Transforming Linear Context-Free Rewriting Systems into Minimalist Grammars". English. In: *Logical Aspects of Computational Linguistics*. Hrsg. von Philippe Groote, Glyn Morrill und Christian Retoré. Bd. 2099. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2001, S. 228–244. ISBN: 978-3-540-42273-0. DOI: 10.1007/3-540-48199-0_14.

[Sek+91] Hiroyuki Seki, Takashi Matsumura, Mamoru Fujii und Tadao Kasami. „On multiple context-free grammars". In: *Theoretical Computer Science* 88.2 (1991), S. 191–229. ISSN: 0304-3975. DOI: 10.1016/0304-3975(91)90374-B.

[Vas+16] Ashish Vaswani, Yonatan Bisk, Kenji Sagae und Ryan Musa. „Supertagging With LSTMs". In: *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. San Diego, California: Association for Computational Linguistics, 2016, S. 232–237. DOI: 10.18653/v1/N16-1027.

[Vij88] Krishnamurti Vijay-Shanker. „A study of tree adjoining grammars“. Diss. University of Pennsylvania, 1988. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.401.1695&rep=rep1&type=pdf.

[VWJ86] Krishnamurti Vijay-Shanker, David Jeremy Weir und Aravind K. Joshi. „Tree adjoining and head wrapping“. In: *Proceedings of the 11th coference on Computational linguistics*. Association for Computational Linguistics. 1986, S. 202–207. DOI: 10.3115/991365.991425.

[WJ88] David Jeremy Weir und Aravind K. Joshi. „Combinatory categorial grammars: Generative power and relationship to linear context-free rewriting systems“. In: *Proceedings of the 26th annual meeting on Association for Computational Linguistics*. Association for Computational Linguistics. 1988, S. 278–285. DOI: 10.3115/982023.982057.

The software to be submitted is uploaded on `https://github.com/aannleax/Supertagging`. All intermediate and end results are saved on ficus at `/home/s0803460/results`.

Contents

1 Introduction **7**

2 Preliminaries **8**
2.1 Graphs and trees 8
2.2 Dependency trees 8
2.3 LCFRS grammars 8

3 Parsing pipeline **10**
3.1 Grammar extraction 10
3.2 Parsing 12

4 Supertagging **13**

5 Using BERT for Supertagging **14**
5.1 Pre-training 15
5.2 Finetuning 15

6 Embedding of LCFRS rules **15**
6.1 Similarity measures 16
6.1.1 N-gram 16
6.1.2 Damerau-Levenshtein distance 16
6.2 Embedding algorithm 17
6.3 Decoding of vectors into LCFRS rules 18

7 Program **19**
7.1 Installation and setup 20
7.2 Training LCFRS rule embeddings 20
7.3 BERT 21
7.4 Parsing 21

8 Experiments **21**
8.1 Choosing hyperparameters for embedding 21
8.2 Training the embeddings 22
8.3 Training BERT 22
8.4 Parsing 24

9 Conclusion and future work **24**

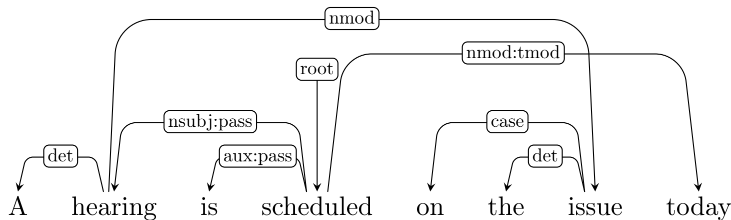

Figure 1: Example of a non-projective dependency tree

1 Introduction

Many natural language processing tasks require a syntactic representation of the input sentence. One way this can be done is with dependency trees. Dependency trees describe grammatical relations between words in a sentence as a directed edge between them. Figure 1 shows a dependency tree for the sentence "A hearing is scheduled on the issue today". Notice how the edge between "hearing" and "issue" crosses the edge between "scheduled" and "today". Dependency trees with crossing edges are called non-projective. Non-projective structures cannot be described by context free grammars. Parsing of context sensitive grammars, however, is PSPACE-complete and therefore not computationally feasible. Hence a number of grammar formalisms were proposed which can represent non-projective structures occuring in natural languages while still allowing a sentence to be parsed in polynomial time. These grammar formalisms are summarized under the term mildly context sensitive grammars. This category includes for example head grammars [10], tree adjoining grammars [5], combinatory categorial grammars [12], linear indexed grammars [4], multiple context-free grammars [10] and minimalist grammars [8]. Linear context-free rewriting systems (LCFRS) are a uniform way to represent these different grammar formalisms [14]. Therefore, parsing LCFRS grammars plays an important role for parsing natural languages.

Parsing of LCFRS with large grammars is still computationally expensive. A method called supertagging can be used to restrict the grammar in order to increase the speed of parsing and ideally without losing much accuracy. This approach was originally developed for lexicalised tree adjoining grammars [6]. It assigns a small set of so called elementary trees for each word in the input sentence. These trees are combined to generate a parse tree of the whole sentence. It is an open question whether, analogously, lexicalised LCFRS grammars, whose rules contain exactly one terminal on the left hand side, can also be used for supertagging. Instead of elementary trees, a small set of candidate rules would be selected for each input word. The selection can for example be done with neural networks. In this paper, supertagging is done by finetuning BERT which is a pretrained neural network developed by Google AI [3]. For that, a vector representation of LCFRS rules was created.

The goal of this paper is to present a method for how BERT can be used for supertagging and evaluate the impact on performance with regard to speed and precision of parsing. Sections 2-6 provide necessary background and algorithms used to implement the approach. Section 7 then descirbes the actual implemention while section 8 contains the description and discussion of the experiments.

2 Preliminaries

This section formally introduces the concepts needed for the rest of the paper. For a natural number $n \in \mathbb{N}$ we define the set $[n] = \{1, \dots, n\}$. We call a finite set $X \subseteq \mathbb{N}$ *contiguous* if $x = \max X$ or $x + 1 \in X$ for all $x \in X$. Furthermore $\binom{X}{n} = \{Y \subseteq X \mid |Y| = n\}$ denotes the set of all subsets of X with n elements. In addition we introduce the *span* of a finite set $J \subseteq \mathbb{N}$ as a tuple $\operatorname{span}(J) = (J_1, \dots J_k)$ such that $J = \bigcup_{i \in [k]} J_i$ where each set J_i is contiguous and for each $i, j \in \mathbb{N}$ it holds that $i < j$ implies $\min J_i < \min J_j$. So for example $\operatorname{span}(\{1, 2, 4\}) = (\{1, 2\}, \{4\})$. For $a, b \in \mathbb{R}$ we define $[a, b] = \{x \in \mathbb{R} \mid a \leq x \leq b\}$.

2.1 Graphs and trees

A directed graph $G = (V, E)$ consists of a set V of *nodes* and the set $E \subseteq V \times V$ of *edges*. Let $G = (V, E)$ and $G' = (V', E')$ be graphs. The function $\varphi \colon V \to V'$ is an *isomorphism* between G and G' if it is a bijection such that $(v, w) \in E \iff (\varphi(v), \varphi(w)) \in E'$. A sequence of nodes $v_1, \dots, v_n$ is a *path* in G if $(v_i, v_{i+1}) \in E$ or $(v_{i+1}, v_i) \in E$ for all $i \in [n-1]$. A node $w \in V$ *can be reached* from another node $v \in V$ if there exists a path from v to w. Furthermore a *cycle* is defined as a path $v_1, \dots, v_n$ with $v_1 = v_n$. A graph which contains no cycles is called *acyclic*. An acyclic graph is a *tree* if it contains exactly one node, called the root and denoted $\operatorname{root}(G)$, from which each other node can be reached. Let $v \in V$ be a node. Then $\operatorname{succ}(v) = \{v' \in V \mid (v, v') \in E\}$ is the set of all *successors* to v. Also $\operatorname{leaves}(v) = \{v \in V \mid \forall v' \in V \colon (v, v') \notin E\}$ is the set of *leaf nodes* in G. For a tree $G = (V, E)$, we define the *subtree* $G_{|v}$ to be the tree obtained from G by removing all nodes $v' \in V$ (and corresponding edges) which are not reachable from v.

2.2 Dependency trees

We represent dependency trees as trees where each node is a pair containing a position in the sentence and its incoming dependency. Let T be a set of *tokens*. Tokens are the smallest units of a sentence and can be words but also punctuation marks like for a example a period. Formally, a *sentence* over a token set T is considered to be a sequence $s = t_1, \dots, t_n$ where $t_i \in T$ for every $i \in [n]$. Now, let $s = t_1, \dots, t_n$ be sentence and N be a set. A *dependency tree* is a pair $D = (G, s)$ where $G = (V, E)$ with $V \subseteq N \times [n]$ is a tree such that the function $\operatorname{ord} \colon V \to [n], (d, i) \mapsto i$ is a bijection. Let $v \in V$ be a node. Then the *cover* of the node, denoted $\operatorname{cover}(v)$, is defined as the set $\{j \in \mathbb{N} \mid (v', j) \in V'\}$ where V' is the set of nodes in $G_{|v}$. A finite set C of dependency trees is called a *corpus*. For evaluation we will have to compare two dependency trees. For that, we define $\operatorname{edges}(D) = \{(i, j) \in \mathbb{N}^2 \mid ((A, i), (B, j)) \in E)\}$ and $\operatorname{edges}_{\text{labeled}}(D) = \{(i, j, B) \in \mathbb{N}^2 \times N \mid ((A, i), (B, j)) \in E\}$. Then the precision measure between two dependency trees D_{gold} and D_{given} is defined as

$$\operatorname{precision}_{\text{unlabeled}}(D_{\text{gold}}, D_{\text{given}}) = \frac{|\operatorname{edges}(D_{\text{gold}}) \cap \operatorname{edges}(D_{\text{given}})|}{|\operatorname{edges}(D_{\text{given}})|}$$

Lebeled precision is defined analogously.

2.3 LCFRS grammars

For simplicity, the syntax of range concatenation grammars are used for defining LCFRS rules [2].

Let N, T and V be pairwise disjoint sets. We call elements of N *non-terminals*, elements of V *variables* and T is a set of tokens. An *LCFRS grammar* is defined as the tuple $\mathbb{G} = (N, T, V, P, S)$ where $S \in N$ is the start non-terminal and P a set of LCFRS rules over N, T, V. For convenience

$x_{j,k}$ is written as a shorthand for $x_j, \ldots, x_k$. An LCFRS rule over N, T, V is an expression of the form

$$r = A(y_1, \ldots, y_n) \to A_1(x_1^{(1)}, \ldots, x_{m_1}^{(1)}) \ldots A_k(x_1^{(k)}, \ldots, x_{m_k}^{(k)})$$

where

- $n \in \mathbb{N}$ is the *fanout* of the rule and $k \in \mathbb{N}$ its *rank* denoted $\text{fanout}(r)$ and $\text{rank}(r)$ respectively,
- $A, A_1, \cdots A_k \in N$ where A is the *dependency* of the rule denoted $\text{dependency}(r)$,
- Each $x_j^{(i)}, x_{j'}^{(i')} \in V$ with $i, i' \in [k]$, $j \in [m_i]$ and $j' \in [m_{i'}]$ is pairwise distinct
- $y_i \in (V \cup T)^*$ with $i \in [n]$,
- $A(y_1, \ldots, y_n)$ is the left hand side and $A_1(x_{1,m_1}^{(1)}) \ldots A_1(x_{k,m_k}^{(k)})$ the right hand side of the rule and
- each variable on the right hand side appears exactly once in $y_1 \circ \cdots \circ y_n$.

Additionally, an LCFRS rule is *monotone* if

- for each pair of variables $x_j^{(i)}, x_{j'}^{(i)} \in V$ it holds that $j < j'$ implies that $x_j^{(i)}$ appears before $x_{j'}^{(i)}$ in $y_1 \circ \cdots \circ y_n$.

And finally, r is *lexicalised* if

- $y_1 \circ \cdots \circ y_n$ contains exactly one element from T called the *token* of the rule and denoted by $\text{token}(r)$.

Rules of rank 0 are called ε-rules. An LCFRS grammar is called monotone or lexicalised if each rule of the grammar is monotone or lexicalised, respectively. *In the following we assume that every LCFRS grammar is monotone and lexicalised.*

A probabilistic LCFRS grammar is a tuple $\mathbb{P} = (\mathbb{G}, p)$ consisting of an LCFRS grammar $\mathbb{G}$ and a function $p\colon P \to [0, 1]$ which assigns a probability to each rule such that for each $A \in N$

$$\sum_{r \in \{r' \in P \mid \text{dependency}(r') = A\}} p(r) = 1.$$

We now formally introduce the concept of parsing LCFRS grammars. Let $\mathbb{G} = (N, T, V, P, S)$ be an LCFRS grammar and $s = t_1, \ldots, t_n$ a sentence over T. Then a dependency tree $D = (G, s)$ with $G = (V, E)$ and $V \subseteq P \times [n]$ is called an *abstract syntax tree* for $\mathbb{G}$ and s if

- $\text{dependency}(r) = S$ and $\text{fanout}(r) = 1$ for $(r, i) = \text{root}(G)$
- for all $v = (r, i) \in V$ it holds that $\text{token}(r) = t_i$
- for all $v = (r, i) \in V$ where

  $$r = A(y_1, \ldots, y_n) \to A_1(x_1^{(1)}, \ldots, x_{m_1}^{(1)}) \ldots A_k(x_1^{(k)}, \ldots, x_{m_k}^{(k)})$$

  there exists a bijection $b\colon [k] \to \text{succ}(v)$ such that

  - $\text{dependency}(b(j)) = A_j$ and
  - $\text{fanout}(b(j)) = m_j$ for every $j \in [k]$.

A dependency tree $D = (G, s)$ with $G = (V, E)$ and $V \subseteq N \times [n]$ is a *parse tree* for the sentence $s = t_1, \ldots, t_n$ and a grammar $\mathbb{G}$ if N is the set of non-terminals of $\mathbb{G}$ and if there exists an abstract syntax tree $D' = (G', s)$ with $G' = (V', E')$ for $\mathbb{G}$ and s such that the bijection $\pi\colon V \to V'\colon (r, i) \mapsto (\text{dependency}(r), i)$ is a graph isomorphism between G and G'.

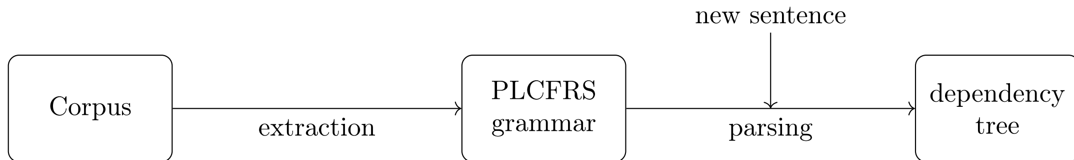

Figure 2: Pipeline for parsing without supertagging

3 Parsing pipeline

This section describes the process of parsing languages without the use of supertagging. Figure 2 shows the general pipeline. We start with a corpus containing dependency trees which were compiled by experts and are assumed to be correct. From this corpus a probabilistic LCFRS grammar is extracted. The goal is to extract a grammar which can describe sentences which did not appear in the original corpus. The following subsections contain the algorithms for grammar extraction and parsing.

3.1 Grammar extraction

The algorithm for extraction (1) on page 11 is taken from [7]. It produces one rule for each node of the dependency tree. The dependency and the token of the rule are taken from the node itself. If the current node is a leaf node then the produced rule is simply an ε-rule. Otherwise we introduce an expression $A_j(x^{(j)}_{1,m_j})$ on the right hand side for each successor of the current node. Each variable represents a contiguous section of the input sentence which is covered by the subtree of the successor node. The token of the rule covers the position of the token. The variables and the token are then arranged on the left hand side in order. Gaps are seperated by a comma. For example let $(A, 3)$ be a node and $(B, 2)$ and $(C, 6)$ be its successors which cover the positions $\{1, 2, 4\}$ and $\{6, 7\}$ respectively. Then the following rule is extracted:

$$A(x^{(1)}_1 t_3 x^{(1)}_2, x^{(2)}_1) \to B(x^{(1)}_1, x^{(1)}_2)C(x^{(2)}_1).$$

$x^{(1)}_2$ and $x^{(2)}_1$ are seperated because there is a gap between the $\{4\}$ and $\{6, 7\}$. An LCFRS grammar can be extracted from a corpus of dependency trees by applying algorithm 1 for every tree in the corpus. The set of rules of this grammar is the union of the individual rule sets extracted from each tree.

Algorithm 1 does not produce a probabilistic LCFRS grammar. Probabilities can be assigned to rules by, for example, considering the frequency the rule has been extracted with. Details of that will not be discussed here since the pipeline will be adjusted in section 4 so that the probability assignment will happen in a later stage.

Algorithm 1: Grammar extraction from a corpus of dependency trees

Input : Corpus C of dependency trees
Output: Set of LCFRS rules P

1 $P = \emptyset$
2 **for** $(G, s) \in C$ where $G = (V, E)$ and $s = t_1, \ldots, t_n$ **do**
3   **for** $v = (A, i) \in V$ **do**
4   **if** $v \in \text{leaves}(G)$ **then**
5             $P \leftarrow P \cup \{A(t_i) \rightarrow \varepsilon\}$
6   **end**
7   **else**
8             $J^{(0)} \leftarrow \text{spans}(\text{cover}(v))$
9             $k \leftarrow 0$
10   **for** $v' = (A', i') \in \text{succ}(v)$ **do**
11                 $k \leftarrow k + 1$
12                 $J^{(k)} \leftarrow \text{spans}(\text{cover}(v'))$
13                 $A_k \leftarrow A'$
14   **end**
15   **for** $j \leftarrow 1$ **to** $|J^{(0)}|$ **do**
16                 $y_j \leftarrow \varepsilon, U = J^{(0)}_j$
17   **while** $U \neq \emptyset$ **do**
18   **if** $\min U = i$ **then**
19                         $y_j \leftarrow y_j \circ t_i$
20                         $U \leftarrow U \setminus \{i\}$
21   **end**
22   **else**
23   Let $i' \in [k]$ and $j' \in [|J^{(i')}|]$ such that $\min U = \min J^{i'}_{j'}$
24                         $y_j \rightarrow y_j \circ x^{(i')}_{j'}$
25                         $U \leftarrow U \setminus J^{(i')}_{j'}$
26   **end**
27   **end**
28   **end**
29             $P \leftarrow P \cup \{A(y_1, \ldots, y_{|J^{(0)}|}) \rightarrow A_1(x^{(1)}_1, \ldots, x^{(1)}_{|J^{(1)}|}) \ldots A_k(x^{(k)}_1, \ldots, x^{(k)}_{|J^{(k)}|})\}$
30   **end**
31   **end**
32 **end**
33 **return** P

$_0$ A $_1$ hearing $_2$ is $_3$ scheduled $_4$ on $_5$ the $_6$ issue $_7$ today $_8$. $_9$

Figure 3: Numbering of the gaps in a sentence

3.2 Parsing

In the following we assume that the "gaps" of the tokens in a sentence are numbered like shown in figure 3. This allows us to refer to certain parts of a sentence. For example, the part "on the issue" is located between the positions 4 and 7.

The algorithm for parsing (2) which is presented on page 13 is based on [9]. It views parsing as a deductive process starting with axioms and using rules of inference based on the grammar to introduce new facts. The facts are called parse items and are of the form $u\colon [A, \kappa]$ where A is a dependency, $u \in [0,1]$ a probability and $\kappa = \kappa^{(1)}, \ldots, \kappa^{(n)}$ with $n \in \mathbb{N}$ and

$$\kappa^{(i)} \in \{(j,k) \in \mathbb{N} \times \mathbb{N} \mid j < k\} \text{ for every } i \in [n]$$

is called a *range vector*. Each parse item covers certain parts of the input sentences with a dependency. For example the parse item $0.5\colon [\text{nsub}, ((0,3),(6,7))]$ means that the sentence positions between 0 and 3 and between 6 and 7 are covered by the dependency nsub. We define the partial operation $\circ\colon \mathbb{N}^2 \to \mathbb{N}^2$ as

$$(i, j_1) \circ (j_2, k) = \begin{cases} (i,k) & \text{if } j_1 = j_2 \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Furthermore, let $r = A(y_1, \ldots, y_n) \to A_1(x^{(1)}_{1,m_1}) \ldots A_1(x^{(k)}_{k,m_k})$ be an LCFRS rule and $(\kappa^{(i)})_{i \in [k]}$ with $\kappa^{(i)} = \kappa^{(i)}_1, \ldots, \kappa^{(i)}_{m_i}$ a family of range vectors. Then $r(\kappa^{(1)}, \ldots, \kappa^{(k)})$ is defined to be the range vector obtained from $(y_1, \ldots, y_n)$ by replacing $x^{(i)}_j$ with $\kappa^{(i)}_j$ and concatenating, if possible, all the neighboring pairs. Let $s = t_1, \ldots, t_n$ be a sentence. If $\text{token}(r) = t_l$ is the token of the rule then the token is treated as the pair $(l-1, l)$. If concatenation is not possible, then the operation is undefined. The deduction system has three rules

$$\text{SCAN:} \quad \frac{}{p(r)\colon [A, ((i-1), i)]} \quad \text{if } r = A(t_i) \to \varepsilon$$

$$\text{RULE:} \quad \frac{u_1\colon [A_1, \kappa_1], \ldots, u_k\colon [A_k, \kappa_k]}{p(r) \cdot \prod_{i \in [k]} u_i\colon [A, r(\kappa_1, \ldots, \kappa_k)]} \quad \text{if } \text{rank}(r) = k \text{ and } r(\kappa_1, \ldots, \kappa_k) \text{ is defined}$$

$$\text{GOAL:} \quad u\colon [S, (0,n)]$$

Algorithm 2 manages two sets $\mathcal{A}$ and $\mathcal{C}$ where $\mathcal{A}$ is called the agenda and $\mathcal{C}$ is the chart. The agenda is initialized with all the parse items that can be obtained from applying the SCAN rule. Then we enter a while loop which removes the parse item with the highest probability from the agenda and adds it to the chart. If the selected item is a GOAL item, the algorithm stops. If it is not a goal item, new parse items are deduced using RULE which are then added to the agenda.

If $n \in \mathbb{N}$ is the length of the input sentence, $k \in \mathbb{N}$ the maximal rank of the LCFRS rules in the grammar and $f \in \mathbb{N}$ the maximal fanout, then the algorithm has a time complexity of $\mathcal{O}(n^{(k+1)\cdot f})$ [11].

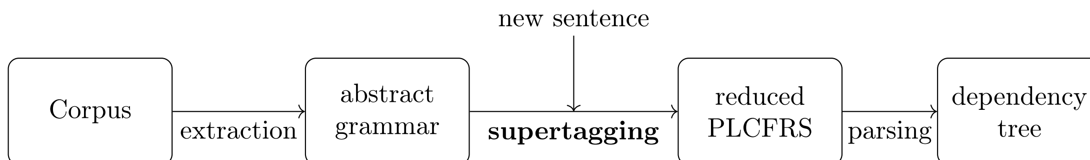

Figure 4: Pipeline for parsing with supertagging

Algorithm 2: Parsing of a sentence via lexicalised PLCFRS

Input : Sentence $s = t_1, \ldots, t_n$, PLCFRS $\mathbb{P} = (\mathbb{G}, p)$
Output: Dependency tree D

```
1  A ← ∅
2  C ← ∅
3  Add each parse item genereted by SCAN to the set A
4  while A ≠ ∅ do
5      (u: [A, κ]) ← argmax_{(u': [A', κ']) ∈ A} u'
6      A ← A \ {(u: [A, κ])}
7      C ← C ∪ {(u: [A, κ])}
8      if (u: [A, κ]) ist GOAL-Item then
9          Reconstruct the dependency tree and return it
10     end
11     else
12         foreach (v: [B, η]) deduced by RULE from (u: [A, κ]) and other items in C do
13             if there is no z where (z: [B, η]) ∈ A ∪ C then
14                 A ← A ∪ {(v: [B, η])}
15             end
16             else
17                 if (z: [B, η]) ∈ A for some z then
18                     A ← (A \ {(z: [B, η])}) ∪ {max{v, z}: [B, η]}
19                 end
20             end
21         end
22     end
23 end
24 return empty tree
```

4 Supertagging

While the algorithm presented in section 3.2 has a polynomial runtime with respect to the length of the sentence for a fixed grammar, parsing is still expensive for the large grammars generated in the extraction step. To speed up parsing we try to reduce the number of production rules with which the sentence is parsed, but still obtain an accurate parse tree. One such method of rule reduction is called supertagging and was originally developed for lexicalised tree adjoining grammars [6]. Each token in the input sentence is assigned a small number of so called elementary trees. These are combined into a parse tree for the whole sentence. This reduces the number of combinations of rules the parser has to consider significantly. Hence this step is also referred to as almost-parsing. Supertags are selected using discriminatory classifiers. In particular, neural

case(A) $\rightarrow \varepsilon$ | nsubj($x_1$hearing) $\rightarrow$ det(x_1) | aux(was) $\rightarrow \varepsilon$ | case(x_1, scheduledx_2) $\rightarrow$ cc(x_1)aux(x_2)

det(A) $\rightarrow \varepsilon$ | advmod(hearing) $\rightarrow \varepsilon$ | cc(was) $\rightarrow \varepsilon$ | root($x_1x_2$scheduled) $\rightarrow$ nsubj(x_1)aux(x_2)

A | hearing | was | scheduled

Figure 5: Example of supertagging with lexicalised LCFRS

networks have proven effective in this task [13]. The neural network used here will be introduced in the the next section.

Lexicalised LCFRS grammars allow for a similar method. But instead of elementary trees, each token is assigned a small number of lexicalised LCFRS rules. The selected rules form a smaller grammar with which the sentence can then be parsed. Figure 5 shows an example of supertagging for the sentence "A hearing was scheduled". Here, two rules per token are selected. All the selected rules form the grammar with which this sentence would be parsed. Notice how there is no point in assigning the rule nsubj($x_1$issue) $\rightarrow$ det(x_1) to the token "hearing" since "issue" does not even appear in the sentence. In fact, we can restrict rules which can be assigned to a token to those who contain the token. This allows the supertagger to only deal with abstract rules which contain a placeholder token instead of a real one. Formally, we define an *abstract* LCFRS grammar $\mathbb{G} = (N, T, V, P, S)$ as an LCFRS grammar whose token set $T = \{\langle\text{Token}\rangle\}$ only contains a placeholder element. For parsing, the placeholder is replaced with the actual token.

Figure 4 shows the adjusted pipeline for parsing with supertagging. First we extract an abstact LCFRS grammar from the corpus. This is done by applying a slightly altered version of algorithm 1 which replaces the actual token with the placeholder. The supertagger selects a small number of abstract rules for each token in the input sentence, fixes the token and assigns them probabilities. The resulting grammar is used to parse the sentence with algorithm 2.

5 Using BERT for Supertagging

BERT stands for Bidirectional Encoder Representations from Transformers and was published by the Google AI team in 2018 [3]. It is a pre-trained neural network based NLP model which uses the architecture of the so called transformer. We skip the details of how transformers work and consider BERT to be a blackbox which transforms one vector of real numbers into another. More formally, we represent BERT as a function $\text{bert}\colon \mathbb{R}^m \rightarrow \mathbb{R}^h$ where $m \in \mathbb{N}$ is the input dimension and $h \in \mathbb{N}$ is the output dimension. In order to input a sentence into a neural network, the sequence of tokens first has to be tranformed into a sequence of vectors. We call a function $e\colon X \rightarrow \mathbb{R}^n$ which maps an arbitrary set X into set of n-dimensional vectors an n-dimensional *embedding*. BERT uses the WordPiece embeddings [15]. This is a subtoken embedding meaning tokens can be split into multiple subtokens. The token "playing" for example might be split up into the subtokens "play" and "##ing". Since subtokens can be combined freely it reduces the chance of encountering an out of vocabulary token. So each input token is first split into its subtokens each of which are assigned an m-dimensional vectors. These are added with embeddings for positions and sections and are used as input for BERT. The neural network is applied to each of the input vectors in parallel. This produces one output vector of dimension h for each of the input tokens.

The actual training can be divided into two stages:

1. pre-training and
2. finetuning.

5.1 Pre-training

The pre-training step involves training the neural network on the following language tasks

- Masked language model and
- Next sentence prediction.

The Masked language model randomly replaces a small amount of the input tokens with a blank token. The network is then tasked with guessing what the original token was. For Next sentence prediction BERT is given a pair of sentences. The task is to determine whether or not the second one is a logical continuation of the first one.

In this step the neural network is supposed to obtain a general understanding of language. During finetuning the network can then be adapted to a specific NLP task.[1]

5.2 Finetuning

In the finetuning step BERT is initialized with the parameters learned from the pre-training. A simple fully connected neural network $\text{finetune}\colon \mathbb{R}^h \to \mathbb{R}^n$ with $(h+1) \cdot n$ parameters is applied to the output of BERT to learn the specific NLP task which in out case would be supertagging. So overall we get the network $\text{bert}_{\text{finetuned}} = \text{finetune} \circ \text{bert}$

Since the final output of the finetuned bert network is a vector for each of the input tokens, we have to translate them into a set of candidate rules. There are multiple conceiveable ways to achieve this. The first option would be to output a vector which has an entry for each rule in the trained grammar. For supertagging, the rules with the highest values in the corresponding vector positions would be picked. The problem with this is that it results in a huge number of parameters which would have to be learned during the finetuning step. The pretrained network used here has an output dimension of $h = 768$. The abstract grammar extracted from the training corpus consists of over 40.000 rules. This would equal over 30 million parameters. BERT itself was trained on 110 million parameters. The second option would be to modify the first approach by combining several rules into rule classes in order to reduce the number of parameters. The third option, which is the option explorered here, is to use an small dimensional embedding for the abstact rules. The vector which is outputted by BERT is compared to the embedding vectors of the rules extracted from the training corpus. From them the ones with the smallest distances are picked.

6 Embedding of LCFRS rules

To reduce the amount of parameters that have to be learned during finetuning the rules of the extracted grammar are embedded into a vector space. BERT assigns an output vector for each of the input tokens. The rules embedded into the vectors closest to the outputted vector are used as supertags. It is therefore beneficial to embed each rule in such a way that the

[1]Here we use the pre-trained model from `https://deepset.ai/german-bert`. It has roughly 110 million parameters and was trained on latest german Wikipedia dump, the OpenLegalData dump and news articles.

rules corresponding to the nearby embeddings are in some sense similar. So if BERT were to accidentally pick a neighboring vector the rules of resulting grammar would still be related to the actual rules. This is achieved by defining a similarity measure on the production rules.

6.1 Similarity measures

A function $\mu\colon P \times P \to [0,1]$ is called a *similarity measure* on the LCFRS grammar $\mathbb{G} = (N, T, V, P, S)$. There are several possible ways of defining similarity between rules.

6.1.1 N-gram

This similarity measure treats LCFRS rules as strings and defines similarity through the amount of n-grams shared between two rules viewed as strings. An n-gram is a sequence of symbols of size n which appears in a word. Formally, let Σ be an alphabet and $w \in \Sigma^*$. We define a function $\mathrm{ngram}_n(w)\colon \Sigma^n \to \mathbb{N}$ such that $(\mathrm{ngram}_n(w))(a_1, \ldots, a_n)$ is the number of times the n-gram $a_1, \ldots, a_n$ appears in w. To apply this to LCFRS-rules we define the string $\mathrm{str}(r) \in (N \cup \{,\})^*$ of an LCFRS rule

$$r = A(y_1, \ldots, y_n) \to A_1(x_1^{(1)}, \ldots, x_{m_1}^{(1)}) \ldots A_k(x_1^{(k)}, \ldots, x_{m_k}^{(k)})$$

as the sequence obtained from $y_0, \ldots, y_n$ by replacing each $x_j^{(i)}$ with A_i for $i \in [k]$ and $j \in [m_i]$. The rule $r = A(x_1x_3, tx_2) \to B(x_1, x_2)C(x_3)$ would be assigned the string $\mathrm{str}(r) = \mathrm{BC{,}tC}$. It contains the bigrams “BC”, “C,”, “,t” and “tC”. The n-gram distance of two LCFRS-rules r_1 and r_2 is then calculated as follows

$$d^{(n)}_{\mathrm{ngram}}(r_1, r_2) = 2 \cdot \delta(r_1, r_2) + \sum_{v \in (N \cup \{,\})^*, |v| = n} |(\mathrm{ngram}_n(\mathrm{str}(r_1))(v_{1,n}) - \mathrm{ngram}_n(\mathrm{str}(r_2))(v_{1,n})|$$

where

$$\delta(r_1, r_2) = \begin{cases} 0 & \text{if } \mathrm{dependency}(r_1) = \mathrm{dependency}(r_2) \\ 1 & \text{otherwise.} \end{cases}$$

To obtain a similarity measure, the n-gram distance is linearilly mapped into the range $[0,1]$. Hence

$$\mu^{(n)}_{ngram}(r_1, r_2) = 1 - \frac{d^{(n)}_{\mathrm{ngram}}(r_1, r_2) - m}{M - m}$$

where $M = \max d^{(n)}_{\mathrm{ngram}}(P \times P)$, $m = \min d^{(n)}_{\mathrm{ngram}}(P \times P \setminus \Delta_P)$ and $\Delta_P = \{(p, q) \in P \times P \mid p \neq q\}$.

6.1.2 Damerau-Levenshtein distance

The Damerau-Levenshtein distance is analogous to the n-gram distance in the sense that it also considers LCFRS rules as strings. It represents the amount of edit operations (insertion, deletion, substitution or transposition of nearby characters) to transform one string into another. Given two words $a = a_1, \ldots, a_m$ and $b = b_1, \ldots, b_m$ we define

$$d_{a,b}(i,j) = \min \begin{cases} 0 & \text{if } i = j = 0 \\ d_{a,b}(i-1, j) + 2 & \text{if } i > 0 \\ d_{a,b}(i, j-1) + 2 & \text{if } j > 0 \\ d_{a,b}(i-1, j-1) + 2_{(a_i \neq b_j)} & \text{if } i, j > 0 \\ d_{a,b}(i-2, j-2) + 0.5 & \text{if } i, j > 1, a_i = b_{j-1} \text{ and } a_{i-1} = b_j \end{cases}$$

where $2_{a_i \neq b_j}$ equals if 0 if $a_i = b_j$ and 2 otherwise. We have chosen 2 as the punishment for insertions, replacements and deletions and only 0.5 for transpositions. The Levenshtein distance of two LCFRS rules r_1 and r_2 is calculated as follows

$$d_{\text{leven}}(r_1, r_2) = \delta(r_2, r_2) + d_{\text{str}(r_1),\text{str}(r_2)}(|\text{str}(r_1)|, |\text{str}(r_2)|).$$

This distance is again mapped into the range $[0, 1]$ with

$$\mu_{leven}(r_1, r_2) = 1 - \frac{d_{\text{leven}}(r_1, r_2) - m}{M - m}.$$

where $M = \max d_{\text{leven}}(P \times P)$ and $m = \min d_{\text{leven}}(P \times P \setminus \Delta_P)$.

6.2 Embedding algorithm

A gradient descent algorithm is used to find a n dimensional embedding $e\colon P \to \mathbb{R}^n$ of the rules P from the abstract grammar $\mathbb{G} = (N, T, V, P, S)$ obtained during grammar extraction of the training corpus. The loss function is defined as follows

$$\mathcal{L}(e) = \sum_{\{r_1, r_2\} \in \binom{P}{2}} (1 - \mu(r_1, r_2) - ||e(r_1) - e(r_2)||_2^2)^2$$

where $||x||_2 = \sqrt{(} \sum_{i=1}^{n} (x_i)^2)$ for $x \in \mathbb{R}^n$ is the L2-norm of a vector. The algorithm requires the gradient of the inner term of the sum w.r.t. $e(r_1)$ denoted $\text{grad}(r_1, r_2) \in \mathbb{R}^n$ which is given by

$$\text{grad}(r_1, r_2)_i = -4 \cdot (1 - \mu(r_1, r_2) - ||e(r_1) - e(r_2)||_2^2) \cdot (e(r_1)_i - e(r_2)_i)$$

for every $i \in [n]$. Differentiating the inner term w.r.t. $e(r_2)$ simply yields $-\text{grad}(r_1, r_2)$.

Algorithm 3 on page 18 describes the training of the rule embedding. It is done over multiple iterations which are called epochs. The number of epochs is given by the hyperparameter $E \in \mathbb{N}$. In each iteration, the set of rules is partitioned into subsets of size $b \in \mathbb{N}$ called batches. For each rule r_1 in a batch a set of size $s \in \mathbb{N}$ of rules is randomly chosen from the whole set P. This is done to reduce the time of training. For each sampled rule r_2 the gradient $\text{grad}(r_1, r_2)$ is calculated. After this is completed for each pair of rules in a batch the embedding is updated according to AdaDelta [16]. This is an adaptive learning rate method which determines the magnitude of the update to the embedding (the direction of the updates is given by the gradient) based on past gradients and updates. AdaDelta introduces the hyperparameters δ and ε. The former, which is called decay, determines how quickly the the magnitude of the updates decreases. The latter affects the initial size of the updates.

Algorithm 3: Training embedding

```
   Input  : Similarity function μ: P × P → [0,1], hyperparameters n, s, b, E, δ, ε
   Output: Embedding e: P → ℝ^n
 1 Initialize e(r) ∈ [−1,1]^n randomly for each r ∈ P
 2 Initialize g(r) and u(r) with zeros for each r ∈ P
 3 for Epoch ← 1 to E do
 4 |   B ← |P| ÷ b
 5 |   Create random bijection σ: [|P|] → P
 6 |   FirstRule = 0
 7 |   for Batch ← 1 to B do
 8 |   |   Initialize d(r) with zeros for each r ∈ P
 9 |   |   for j ← 1 to b do
10 |   |   |   FirstRule ← FirstRule + 1
11 |   |   |   r_1 ← σ(FirstRule)
12 |   |   |   Create a random set S ⊆ P with s elements s.t. r_1 ∉ S
13 |   |   |   for r_2 ∈ S do
14 |   |   |   |   d(r_1)_i ← d(r_1)_i + grad(r_1, r_2)_i for all i ∈ [n]
15 |   |   |   |   d(r_2)_i ← d(r_1)_i − grad(r_1, r_2)_i for all i ∈ [n]
16 |   |   |   end
17 |   |   end
18 |   |   AdaDelta(e, g, u, d, δ, ε)
19 |   end
20 end
```

Algorithm 4 shows the updating procedure in detail.

Algorithm 4: Update embedding according to AdaDelta

```
   Input  : Embedding e, past gradients g, past updates u, current gradients d, dimension
            n, decay δ, ε
   Output: Update to e
 1 Function RMS(x, ε):
 2 |   return √(x + ε)
 3
 4 for r ∈ P do
 5 |   for i ∈ [n] do
 6 |   |   g(r)_i ← δ · g(r)_i + (1 − δ) · (d(r)_i)^2
 7 |   |   x ← −1 · RMS(u(r)_i, ε) / RMS(g(r)_i, ε) · d(r)_i
 8 |   |   u(r)_i ← δ · u(r)_i + (1 − δ) · x^2
 9 |   end
10 end
```

6.3 Decoding of vectors into LCFRS rules

We have established a method of encoding LCFRS rules of some grammar into embeddings. BERT returns a vector for each of input tokens. The following algorithm constructs a probabilistic LCFRS grammar from that output. It works by finding the $n \in \mathbb{N}$ closest rule vectors the outputted vectors. The probabilities are assigned with the softmax function. The parameter

$\beta \in \mathbb{R}$ is used to determine how much the resulting distribution is skewed towards the minimum value. A higher value for β means that the distribution will be more concentrated around the minimum value.

Algorithm 5: LCFRS rules from vectors

Input : Sentence $s = (t_1, \ldots, t_k)$, vectors $(o_1, \ldots, o_k)$, grammar $\mathbb{G} = (N, T, V, P, S)$, embedding $e\colon P \to \mathbb{R}^n$, $n \in \mathbb{N}$, $\beta \in \mathbb{R}$

Output: PLCFRS $\mathbb{P} = (\mathbb{G}', p)$ with $\mathbb{G}' = (N, T, V, P', S)$

1 $P' \leftarrow \emptyset$
2 **for** $i \in [k]$ **do**
3     $\text{Temp} \leftarrow P$
4   **for** $j \leftarrow 1$ **to** n **do**
5         $r \leftarrow \text{argmin}_{r' \in \text{Temp}} ||o_j - e(r')||_2$
6         $r \leftarrow \text{replace_token}(r, t_j)$
7         $\text{dist}(r) = ||o_j - e(r)||_2$
8         $\text{Temp} \leftarrow \text{Temp} \setminus \{r\}$
9         $P' \leftarrow P' \cup \{r\}$
10   **end**
11 **end**
12 $D \leftarrow \{\text{dependency}(r) \in N \mid r \in P'\}$
13 **for** $d \in D$ **do**
14     $M_d = \max\{\text{dist}(r) \mid r \in P', \text{dependency}(r) = d\}$
15   **for** $r \in \{r' \in P' \mid \text{dependency}(r') = d\}$ **do**
16         $p(r) = \frac{\exp(\beta \cdot (M_d - \text{dist(r)}))}{\sum\limits_{r' \in P', \text{dependency}(r') = d} \exp(\beta \cdot (M_d - \text{dist}(r')))}$
17   **end**
18 **end**
19 **return** $\mathbb{P}$

7 Program

As part of the task a software was developed. It includes an implemention of the grammar extraction algorithm and parsing algorithm from section 3 and also the gradient descent algorithm for training the LCFRS rule embeddings from section 6. Code for reading `.conllu` files is found in `Corpus.cpp` while the extraction of the grammar from the corpus takes place in `Grammar.cpp`. `SimilarityFunctions.cpp` implements the similarity function used in the embedding algorithm. The algorithm for the hyperparameter search is found in `HyperParameters.cpp` while the actual training is done in `Training.cpp`. The parsing algorithm is implemented in `Parse.cpp`.

As an implemention for BERT, the python library simpletransformers[2] was chosen. This library contains transformer models for many different NLP tasks such as Token Classification. Token Classification seeks to categorise tokens of the input sentence into predefined categories. It is therefore similar to our task of assigning each token a vector representing an LCFRS rule. Hence the model was adjusted to work with rule embeddings instead of token categories. In addition, python scripts have been written to train the model and test its results which interface with the software.

[2] `https://github.com/ThilinaRajapakse/simpletransformers`

7.1 Installation and setup

To install the simpletransformers library follow the instructions from the setup section of their readme. To make it work for LCFRS embeddings, replace the file `simpletransformers/ner/ner_model.py` with `Python/ner_model.py`. To compile the C++ program run `make`. The only dependency needed is OpenMP. The compiled executable will be placed in `Build/supertag`.

Running the executable for the first time starts a setup process. The program will ask you to input the paths to the training, test and development corpus. The last one is used for determining hyperparameters for the LCFRS rule embedding. If one corpus consists of multiple files you can enter multiple paths seperated by a new line. Entering an empty line confirms the input. From now on, everytime you need to input a path to a corpus you can use the abbreviation "train", "test" or "dev" instead. Also it asks you to enter an output path. All outputted files will be saved relative to this path. Running the program without parameters returns a list of all subprograms while running it with the subprogram as the only parameter shows you how to use it.

The setup process creates a file `settings.json` where all the information is stored. It also contains the hyperparameters for the hyperparameter search. They are initialized to the combinations tried during the experiments.

7.2 Training LCFRS rule embeddings

Before training BERT, an embedding for the LCFRS rules extracted from the training corpus must be obtained. This process is divided into three parts

1. computing a similarity matrix,
2. finding hyperparameters and
3. training the embedding.

The first step computes a matrix representing the chosen similarity measure. This is done in order to save time during training. It is created by calling the matrix subprogram like so

```
$ supertag matrix output_path train function_name cutoff
```

Function name is the name of the similarity function and can either be 2gram, 3gram or leven. The optional parameter cutoff clamps the similarity distance to the cutoff value.

The next step is to find the best hyperparameters to use for the training. You can skip this step and simply select the hyperparameters used during the experiments. In order to do that use

```
$ supertag hyper default out_path function_name
```

The output file contains the hyperparameters. If you want to perform the search yourself use the command

```
$ supertag hyper search out_path function_name dev devmatrix epochs
```

For that you will have to create a similarity matrix for the dev corpus by replacing train with dev in the first command. This command iterates over all combinations of hyperparameters set in the settings file. The output file will contain information of how well the embedding is representing the similarity measure for each combination. An finally, to run the training use

```
$ supertag embedding out_path matrix_path hyperparameters_file 1
```

The last parameter determines after how many epochs the current loss is displayed. Set it to 0 to turn this off.

7.3 BERT

This section describes how to train BERT and use it to tag the test sentences. For that we first have to create a file containing all the training sentences together with the correct LCFRS rules.

```
$ supertag tagcorpus out_path train
```

Also we need to create a file containing all of the test sentences.

```
$ supertag savecorpus out_path test
```

Now, to train BERT simply run

```
$ python3 train.py train_sentence_path embedding_path
```

This will train the BERT model and save it in the folder `outputs`. To change the output folder you can provide an additional argument. To run the trained model on the test sentences use

```
$ python3 test.py out_path test_sentence_path embedding_path
```

If you changed the output path during training you have to provide the new folder as an additional argument. The file outputted by this command needs to be entered to the parse subprogram as the `bert_result` argument.

7.4 Parsing

For parsing you have to input the path to the embedding of the grammar and the file outputted by `test.py`. The parameter `N` determines the number of supertags per token while `beta` sets the value for β in algorithm 5.

```
$ supertag parse output_path train embedding_path bert_result N beta
```

8 Experiments

For the experiments, the UD German-HDT[3] treebank was used which includes a training, test and development corpus [1]. For training, only part A of the training corpus was used. The 1115th test sentence has been removed because the BERT implementation used was unable to tokenize it correctly for unknown reasons. To do this automatically, run

```
$ supertag remove output_path test 1115
```

Remember to set the path in the `settings.py` to the new file.

The experiments were done with the bigram, trigram and Levenshtein similarity measure for which the cutoffs 24, 22 and 19 were chosen, respectively. This corresponds to roughly 0.01% of all rule pairs which receive the minimum similarity for the respective similarity measure.

8.1 Choosing hyperparameters for embedding

Before training the embedding, suitable hyperparameters must be chosen. For the decay δ the values 0.9, 0.95 and 0.99, for ε the values 10^{-4}, 10^{-5} and 10^{-6} and for the sample size the values 10 and 100 were considered. The dimension has been set to 512 and the batch size to 128. Each combination has been trained on the development corpus for 50 epochs and evaluated using the Top10 metric. The Top10 metric averages the similarity distance between each rule and the 10 closest rules to it (in the embedding). Table 1 shows the hyperparameters chosen for each similarity measure.

[3] `https://github.com/UniversalDependencies/UD_German-HDT`

Parameter	Bigram	Trigram	Levenshtein
δ	0.95	0.9	0.9
ε	10^{-6}	10^{-6}	10^{-6}
s	100	100	100
E	100	30	35

Table 1: Hyperparameters chosen for each similarity measure

	Bigram	Trigram	Levenshtein
Top10	3.69	5.52	3.13
Percentile	0.05%	1.45%	5.53%

Table 2: Evaluation of the trained embedding

8.2 Training the embeddings

Using the hyperparameters from table 1, an embedding for each similarity measure was trained. In order to evaluate how well the embedding represents its similarity measure, Top10 values were computed for each embedding which are shown in table 2. The precentile row shows the percentage of rule pairs with a smaller similarity distance than the top10 value. So for example, 0.05% of all rule pairs in the training set have a smaller bigram distance than 3.69. This can be used to compare the embeddings against each other. We can observe that the bigram embedding performs significantly better than both the embedding for trigram and Levenshtein.

8.3 Training BERT

Before evaluating the parse trees we can evaluate the quality of the rules BERT picked for the test sentences by comparing them to the correct rules using the respective similarity measure. We distinguish between correct rules that appeared during training and the ones who did not. In table 3, the first row shows the average placement of the correct rule in the list obtained by sorting the distances of the vector outputted by BERT to the embedding vectors of the grammar extracted from the training corpus. For reference, the grammar has 40.569 rules in total. The other rows show how similar the rule corresponding to the vector closest to the outputted vector are to the correct rule according to the respective similarity distance.

All measures perform significantly worse if the correct rule was not extracted during training. This happens 2.08% of the time. Bigram and trigram measures behave quite similarly. Levenshtein performs worst and even manages to guess worse than random on rules only appearing in the test sentences.

	Bigram	Trigram	Levenshtein
Average Placement	1454	1415	8034
Similarity training Top1	2.98 0.01%	2.27 0.03%	3.52 6.68%
Similarity test Top1	8.38 6.31%	7.07 8.14%	8.95 74.52%

Table 3: Quality of rules outputted by BERT

	Bigram		Trigram		Levenshtein	
	$\beta = 1$	$\beta = 5$	$\beta = 1$	$\beta = 5$	$\beta = 1$	$\beta = 5$
$N = 25$						
Failure [%]	66.61	66.61	95.76	95.76	99.57	99.31
Timeout [%]	0.00	0.00	0.00	0.00	0.00	0.26
Time (Average) [ms]	44	42	30	27	6	165
Time (Median) [ms]	19	19	17	15	1	1
Precision (labeled) [%]	66.21	76.14	42.22	44.46	90.45	90.79
Precision (unlabeled) [%]	77.83	82.00	62.14	64.32	96.65	96.65
$N = 50$						
Failure [%]	13.15	13.15	10.11	10.09	98.84	90.00
Timeout [%]	0.80	0.98	0.12	0.12	0.00	8.84
Time (Average) [ms]	1781	1965	958	922	15	5401
Time (Median) [ms]	185	189	191	191	5	3
Precision (labeled) [%]	43.39	60.74	5.62	8.82	89.91	90.42
Precision (unlabeled) [%]	54.79	67.54	19.23	21.51	96.58	96.64
$N = 75$						
Failure [%]	1.37	1.33	0.76	0.75	97.86	71.72
Timeout [%]	7.89	8.6	0.94	1.52	0.01	26.16
Time (Average) [ms]	8407	8468	2849	3344	29	15691
Time (Median) [ms]	627	422	436	540	14	4
Precision (labeled) [%]	37.46	60.44	4.48	10.77	89.46	90.10
Precision (unlabeled) [%]	49.37	67.95	19.76	23.59	96.98	97.00
$N = 100$						
Failure [%]	0.08	0.07	0.42	0.38	96.94	52.56
Timeout [%]	20.23	19.53	1.09	3.69	0.07	44.53
Time (Average) [ms]	17679	16725	3400	4923	77	27170
Time (Median) [ms]	2743	1815	633	705	18	14
Precision (labeled) [%]	31.19	61.98	3.80	12.49	88.73	89.91
Precision (unlabeled) [%]	42.85	69.76	17.72	25.15	96.22	96.35

Table 4: Evaluation of the parse trees

8.4 Parsing

Table 4 compiles the results of parsing the sentences from the test corpus using 25, 50, 75 and 100 supertags per token and a value for β of 1 and 5. Parsing was canceled if it took longer than 60 seconds. The first row of each section shows how often the parser failed to parse the sentence with the given grammar without reaching the timeout. The next row shows how often the timout has been reached. Rows 3 and 4 of each section contain the average and medium run time respectively. The last two rows show the precision of the parse tree as defined in section 2.2.

The similarity measure Levenshtein which performed worst in all previous measures also proves to be unsuitable for parsing as it fails to parse over 96% percent of the sentences with each parameter. Bigram and trigram, although performing similarly in previous measures, differ in the quality of the parse trees and in runtime. While trigram is able to output a result faster, bigram is way more accurate.

A higher β value of 5 improves the precision in every experiment. This is because a higher value of β means that lower probabilities are assigned to rules whose vetors are farther away from the vector outputted by BERT (which are presumably worse).

9 Conclusion and future work

This work examined whether supertagging could be used for lexicalised LCFRS grammars. The pre-trained neural network BERT was finetuned to select candidate rules. For that, a representation as vectors of real numbers for LCFRS rules has been trained using different similarity measures. Out of the three tested similarity measures bigram performed the best having a precision value of 67.95% (unlabeled) and 60.44% (labeled) while being able to successfully parse 91.07% of the sentences ($N = 75$).

The work can be extended in the future by trying different similarity measures to train the embedding. One could also consider to develope a general vector representation for arbitrary LCFRS rules. This would remove one limition of the current approach which is only able to supertag tokens with rules that already appeared during training. Also adjustments to algorithm 5 could be explored. Right now the probability assignment only considers the distance of the embedded rules to outputted vector but not how likely the rule should be by itself. This could further improve the precision. Lastly, as mentioned in section 5.2, an approach where rules are categorized into rule classes could be examined.

References

[1] Borges Völker, E. ; Wendt, M. ; Hennig, F. ; Köhn, A. : HDT-UD: A very large Universal Dependencies Treebank for German. In: *Proceedings of the Third Workshop on Universal Dependencies (UDW, SyntaxFest 2019).* Association for Computational Linguistics, 46–57

[2] Boullier, P. : Range Concatenation Grammars. In: Carroll, J. (Hrsg.): *Proceedings of the Sixth International Workshop on Parsing Technology (IWPT 2000)*

[3] Devlin, J. ; Chang, M.-W. ; Lee, K. ; Toutanova, K. : BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In: *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational*

Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers). Association for Computational Linguistics, 4171–4186

[4] Gazdar, G. : Applicability of Indexed Grammars to Natural Languages. In: *Natural Language Parsing and Linguistic Theories*. Springer Netherlands, 1988, S. 69–94

[5] Joshi, A. K. ; Levy, L. S. ; Takahashi, M. : Tree adjunct grammars. In: *Journal of Computer and System Sciences* 10 (1975), Febr., Nr. 1, S. 136–163. http://dx.doi.org/10.1016/s0022-0000(75)80019-5. – DOI 10.1016/s0022–0000(75)80019–5

[6] Joshi, A. K. ; Srinivas, B. : Disambiguation of Super Parts of Speech (or Supertags): Almost Parsing. In: *COLING 1994 Volume 1: The 15th International Conference on Computational Linguistics*

[7] Kuhlmann, M. ; Satta, G. : Treebank Grammar Techniques for Non-Projective Dependency Parsing. In: *Proceedings of the 12th Conference of the European Chapter of the ACL (EACL 2009)*. Association for Computational Linguistics, 478–486

[8] Lecomte, A. ; Retoré, C. : Extending Lambek grammars. In: *Proceedings of the 39th Annual Meeting on Association for Computational Linguistics - ACL 01*, Association for Computational Linguistics, 2001

[9] Nederhof, M. : Weighted Deductive Parsing and Knuth's Algorithm. In: *Computational Linguistics* 29 (2003), Nr. 1, 135–143. http://dx.doi.org/10.1162/089120103321337467. – DOI 10.1162/089120103321337467

[10] Pollard, C. J.: Generalized phrase structure grammars, head grammars, and natural language, 1984

[11] Seki, H. ; Matsumura, T. ; Fujii, M. ; Kasami, T. : On multiple context-free grammars. 88 (1991), Nr. 2, S. 191–229. http://dx.doi.org/10.1016/0304-3975(91)90374-B. – DOI 10.1016/0304–3975(91)90374–B. – ISSN 0304–3975

[12] Steele, S. : Mark Steedman, Surface structure and interpretation (Linguistic Inquiry Monographs 30). Cambridge, MA: MIT Press, 1996. Pp. xiv+126. In: *Journal of Linguistics* 34 (1998), Sept., Nr. 2, S. 489–549

[13] Vaswani, A. ; Bisk, Y. ; Sagae, K. ; Musa, R. : Supertagging With LSTMs. In: *Proceedings of the 2016 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, 232–237

[14] Weir, D. J. ; Joshi, A. K.: Characterizing mildly context-sensitive grammar formalisms, 1988

[15] Wu, Y. ; Schuster, M. ; Chen, Z. ; Le, Q. V. ; Norouzi, M. ; Macherey, W. ; Krikun, M. ; Cao, Y. ; Gao, Q. ; Macherey, K. ; Klingner, J. ; Shah, A. ; Johnson, M. ; Liu, X. ; Kaiser, L. ; Gouws, S. ; Kato, Y. ; Kudo, T. ; Kazawa, H. ; Stevens, K. ; Kurian, G. ; Patil, N. ; Wang, W. ; Young, C. ; Smith, J. ; Riesa, J. ; Rudnick, A. ; Vinyals, O. ; Corrado, G. S. ; Hughes, M. ; Dean, J. : Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. In: *ArXiv* abs/1609.08144 (2016)

[16] Zeiler, M. D.: ADADELTA: An Adaptive Learning Rate Method. In: *CoRR* abs/1212.5701 (2012). http://arxiv.org/abs/1212.5701