TRABALLO FIN DE GRAO
GRAO EN ENXEÑARÍA INFORMÁTICA
MENCIÓN EN COMPUTACIÓN

A Linearization Framework for Dependency and Constituent Trees

Estudante: Diego Roca Rodríguez

Dirección: David Vilares Calvo

Carlos Gómez Rodríguez

A Coruña, September 2022.

Acknowledgements

First, I have to thank my research supervisors, David and Carlos. Without the help they provided this work would have been impossible. I would also like to thank the LyS research group for providing the tools needed for developing this project and all the teachers that through the whole degree got me interested in this line of research. Finally, I wanted to thank my family for their support during the long months that this work required and my friends for always being there. Thank you all.

Abstract

Parsing is a core natural language processing problem in which, given an input raw sentence, a model automatically produces a structured output that represents its syntactic structure. The most common formalisms in this field are constituent and dependency parsing. Although both formalisms show differences, they also share limitations, in particular the limited speed of the models to obtain the desired representation, and the lack of a common representation that allows any end-to-end neural system to obtain those models. Transforming both parsing tasks into a sequence labeling task solves both of these problems. Several tree linearizations have been proposed in the last few years, however there is no common suite that facilitates their use under an integrated framework. In this work, we will develop such a system. On the one hand, the system will be able to: (i) encode syntactic trees according to the desired syntactic formalism and linearization function, and (ii) decode linearized trees into their original representation. On the other hand, (iii) we will also train several neural sequence labeling systems to perform parsing from those labels, and we will compare the results

El análisis sintáctico es una tarea central dentro del procesado del lenguaje natural, en el que dada una oración se produce una salida que representa su estructura sintáctica. Los formalismos más populares son el de constituyentes y el de dependencias. Aunque son fundamentalmente diferentes, tienen ciertas limitaciones en común, como puede ser la lentitud de los modelos empleados para su predicción o la falta de una representación común que permita predecirlos con sistemas neuronales de uso general. Transformar ambos formalismos a una tarea de etiquetado de secuencias permite resolver ambos problemas. Durante los últimos años se han propuesto diferentes maneras de linearizar árboles sintácticos, pero todavía se carecía de un software unificado que permitiese obtener representaciones para ambos formalismos sobre un mismo sistema. En este trabajo se desarrollará dicho sistema. Por un lado, éste permitirá: (i) linearizar árboles sintácticos en el formalismo y función de linearización deseadas y (ii) decodificar árboles linearizados de vuelta a su formato original. Por otro lado, también se entrenarán varios modelos de etiquetado de secuencias, y se compararán los resultados obtenidos.

Keywords:

Natural Language Processing

Tree Linearization

Sequence Labeling

- Constituent Parsing
- Dependency Parsing
- Multi Task Learning

Palabras chave:

- Procesamiento Lenguaje Natural
- Linearizacion de arboles
- Etiquetado de secuencias
- Analisis Sintactico de Constituyentes
- Analisis Sintactico de Dependencias
- Aprendizaje Multitarea

Contents

1 Introduction **1**

1.1 Natural Language Processing . . . 3

1.2 Motivation and Challenges . . . 3

1.3 Structure . . . 5

2 Preliminaries **7**

2.1 Constituent Grammars . . . 7

2.1.1 Statistical Models for Constituent Parsing . . . 8

2.1.2 Transition-based Systems for Constituent Parsing . . . 9

2.2 Dependency Grammars . . . 9

2.2.1 Transition-based Systems for Dependency Parsing . . . 11

2.2.2 Graph-based methods for Dependency Parsing . . . 11

3 Project Development **12**

3.1 Resources . . . 12

3.1.1 Human Resources . . . 12

3.1.2 Tools . . . 12

3.1.3 Data Sets . . . 13

3.2 Evaluation Methods . . . 14

3.3 Project Planning and Methodology . . . 15

3.3.1 Tasks Definition . . . 15

4 Sequence Labeling **18**

4.1 NLP tasks traditionally cast as sequence labeling . . . 18

4.2 Architectures for sequence labeling systems . . . 19

4.2.1 Probabilistic models . . . 19

4.2.2 Artificial Neural Network Models . . . 21

4.2.3 Transformer-based Sequence Labeling Systems . . . 23

4.3 Architecture of a Sequence Labeling System . . . 23
4.4 Adding Multitask Learning to Sequence Labeling Systems . . . 25

5 Constituent Parsing as a Sequence Labeling task 27
5.1 Constituent Parsing as Sequence Labeling . . . 27
5.2 Encoding the Constituent Tree . . . 28
5.3 Encodings . . . 32
5.3.1 Naive Absolute Encoding . . . 32
5.3.2 Naive Relative Encoding . . . 32
5.3.3 Dynamic Encoding . . . 33
5.4 Decoding the Labels . . . 34
5.5 Ensuring Correctness . . . 35
5.6 Chapter conclusion . . . 37

6 Dependency Parsing as a Sequence Labeling Task 38
6.1 Dependency Parsing as Sequence Labeling . . . 38
6.1.1 Encoding the Dependency Tree . . . 39
6.1.2 Decoding the Labels . . . 39
6.2 Encodings . . . 40
6.2.1 Naive Absolute Encoding . . . 41
6.2.2 Naive Relative Encoding . . . 42
6.2.3 PoS Based Encoding . . . 44
6.2.4 Bracketing Based Encoding . . . 45
6.3 Ensuring Correctness . . . 52
6.4 Chapter Conclusion . . . 52

7 Software Details 54
7.1 Models . . . 54
7.1.1 Output Labels . . . 56
7.2 Software System . . . 56
7.2.1 Software Architecture . . . 56
7.2.2 User interaction and usage . . . 58
7.3 Training of Sequence Labeling Systems . . . 61

8 Experimental Results 63
8.1 Experiments Setup . . . 63
8.1.1 Datasets . . . 63
8.1.2 Evaluation Methods . . . 64
8.2 Experimental Results . . . 64

8.2.1 Constituent parsing experiments 65
8.2.2 Dependency parsing experiments 67

9 Conclusion and Future Work **70**
9.1 Conclusion 70
9.2 Future work 71

Bibliography **72**

List of Figures

1.1 Constituent tree (a) and Dependency tree (b) for a given sentence 2

2.1 Example of a constituent parse tree for the sentence "Agent Cooper loves black coffee." represented in graphical format (a) and in bracketed format (b). 8

2.2 Example a Dependency Tree for the sentence "Agent Cooper loves black coffee." in graphical format (a) and in CoNLL-U format (b). 10

2.3 Projective dependency tree (a) and non-projective dependency tree (b). Dependency tree (b) edges are separated in two planes indicated by red and blue colored edges. 11

3.1 Gantt Diagram of the project. 17

4.1 Diagram of a LSTM cell where (a) is the Forget Gate, (b) is the Input Gate, (c) is the Output Gate and (d) is the Cell State inspired by `https://colah.github.io/posts/2015-08-Understanding-LSTMs/`. 22

4.2 Sample architecture of a sequence labeling system. In the token representation layer we have pre-trained word embeddings (red), character-level word embeddings (blue) and feature embeddings (green). This is forwarded to the encoding layer built on 2 stacked bidirectional long short term memories (yellow and pink) and its output is sent to a softmax layer (green) to get a predicted label. 24

4.3 Architecture of a Multi Task Learning System with Hard Parameter Sharing . 26

5.1 Step by step linearization of the first 3 nodes from a constituent tree. The red arrow represents the path to leaves of w_i and the blue arrow represents the path to leaves of w_{i+1}. They are compared in order to obtain the values of nc_i and lc_i 31

5.2 Constituent tree with intermediate unary branches and leaf unary branches (a) and its collapsed form (b). 32
5.3 Example of a constituent tree (a) and its label representation in naive absolute encoding (b). In the column l_i, field nc_i indicates the number of common ancestors encoded in naive absolute encoding, field lc_i indicates the lowest common ancestor and field uc_i is the leaf unary chain. 33
5.4 Example of a constituent tree (a) and its label representation in naive relative encoding (b). In the column lc_i, field nc_i indicates the number of common ancestors encoded in naive relative encoding, field lc_i indicates the lowest common ancestor and field uc_i is the leaf unary chain. 33
5.5 Example of a conflicting tags error while decoding a constituent tree. The conflicting nodes are NP (blue) and VP (red). This is solved by setting both separated by a '|' separation character. 35
5.6 Example of a unexpected length problem while decoding a constituent tree. The ill-predicted label is of word 'not' having the field n_c a 3 instead of a 2, causing it to leave a '-' node in the tree. 37

6.1 Encoding of a given dependency tree with the x_i's encoded according to the naive absolute encoding. 41
6.2 Encoding of a given dependency tree with the x_i's encoded according to the naive relative encoding. 42
6.3 Encoding of a given dependency tree with the x_i's encoded according to the part-of-speech based encoding. 44
6.4 Encoding of a given dependency tree with the x_i's encoded according to bracketing-based encodings. 46
6.5 47
6.6 Example of a Right Dependency (blue) and a Left Dependency (red). 47
6.7 2-planar tree encoded with bracket based encoding (a) and resulting tree from decoding those labels (b). 50
6.8 Planar separation for bracketing encoding using different strategies. 51
6.9 Post-process of a dependency tree decoded from a bad prediction 53

7.1 Diagram explaining the process of obtaining the parse tree for a given sentence. 55
7.2 Sample of an output file representing a linearized dependency tree from the sentence "I am the one who knocks" with bracket encoding. The columns represent: (i) word, (ii) part-of-speech feature, (iii) person, (iv) number and (v) label. 57

7.3 SPMRL French bracketed constituent tree with features (indicated in red color) between ## characters and separated by "|" character (a) and sample of a CoNLL-U file from English$_{EWT}$ with additional features in 'feats' field (b) 57

7.4 Class diagram for the constituent and dependency linearization system. In the diagram the different layers and sub-modules that were taken into consideration when developing the system are remarked. 58

7.5 Architecture of the NCRF++ sequence labeling toolkit extracted from `https://github.com/jiesutd/NCRFpp` 62

List of Tables

2.1 Example of the substitution principle used in constituent grammars. 7

5.1 F-Score from predicted labels for the test set of the Penn Treebank, in relation with the tree depth measured for the absolute scale encoding and the relative scale encoding 34

5.2 Label sparsity (number of distinct labels obtained) for each encoding for the English Penn Treebank of constituent Trees. 34

6.1 Label sparsity for the naive absolute encoding and the naive relative encoding for the sum of all the trees in the universal dependencies English$_{EWT}$ treebank. 43

8.1 F-Score (higher is better) for constituent parsing on the test sets of the English$_{PTB}$, Basque$_{SPMRL}$, French$_{SPMRL}$, German$_{SPMRL}$, Hebrew$_{SPMRL}$, Hungarian$_{SPMRL}$, Korean$_{SPMRL}$, Polish$_{SPMRL}$, Swedish$_{SPMRL}$ treebanks. 65

8.2 Speed measured in sentences per second (higher is better) using the NCRF++ tool for the test sets of the English$_{PTB}$, Basque$_{SPMRL}$, French$_{SPMRL}$, German$_{SPMRL}$, Hebrew$_{SPMRL}$, Hungarian$_{SPMRL}$, Korean$_{SPMRL}$, Polish$_{SPMRL}$, Swedish$_{SPMRL}$ treebanks., using a i5-1155G7 CPU. 66

8.3 Comparison of the metrics for constituent parsing for the test set from the Penn Treebank. The speeds are reported by authors running on their own hardware[1]. 66

8.4 F-Score (higher is better) for the dependency parsing encodings on the test sets of the UD treebanks used in this work. 68

8.5 Speed measured in sentences per second (higher is better) for the UD test sets, using a i5-1155G7 CPU. 69

Chapter 1

Introduction

Natural Language Processing is one of the main fields in artificial intelligence (AI), concerned with the capability of a machine to process human languages. A core part of NLP is parsing, concerned with obtaining a syntactic representation of a given a raw sentence. These syntactic structures usually take the form of a tree, and they are generated from the application of rules from a given grammatical theory. Two of the most common formalisms used to obtain these trees are constituent and dependency grammars, each one allowing us to obtain a different type of representation.

On the one hand, constituent parsing aims to obtain the syntactic structure of a sentence as a constituent tree [2] where the sentence is formed of smaller parts that fulfill different syntactic roles (e.g., noun or prepositional phrase). This idea is based on the concept of *constituency relationships* between words, a derivative from the subject-predicative division from Greek and Latin and further refined in context free grammars. On the other hand, the aim of dependency parsing is to generate a parse tree as a set of binary directed relations between words called dependencies that describe the syntactic roles (e.g., direct object, subject, adverbial modifier) that participate in the sentence. Dependency parsing is based on the idea that words in a sentence are dependent between them. Figure 1.1 shows an example of constituent tree (a) and dependency tree (b) for the same sentence.

These differences make them useful for different sets of tasks: constituent trees are commonly used in tasks like *grammar checking* [3], where the constituent tree is used for detecting ill-formed sentences; *question answering* [4], using the constituent tree for extracting syntactic information about the sentence; or *named entity recognition* [5], by taking advantage of the sentence decomposition provided by constituent parse trees. Dependency trees have also been used in several tasks such as *machine translation* [6, 7], where dependency trees are used to improve the translation results, *sentiment analysis* [8], or *information extraction* [9], by using the sentence's dependency structure to increase the accuracy of the extraction systems.

The tasks of constituent and dependency parsing have been around for decades, with tech-

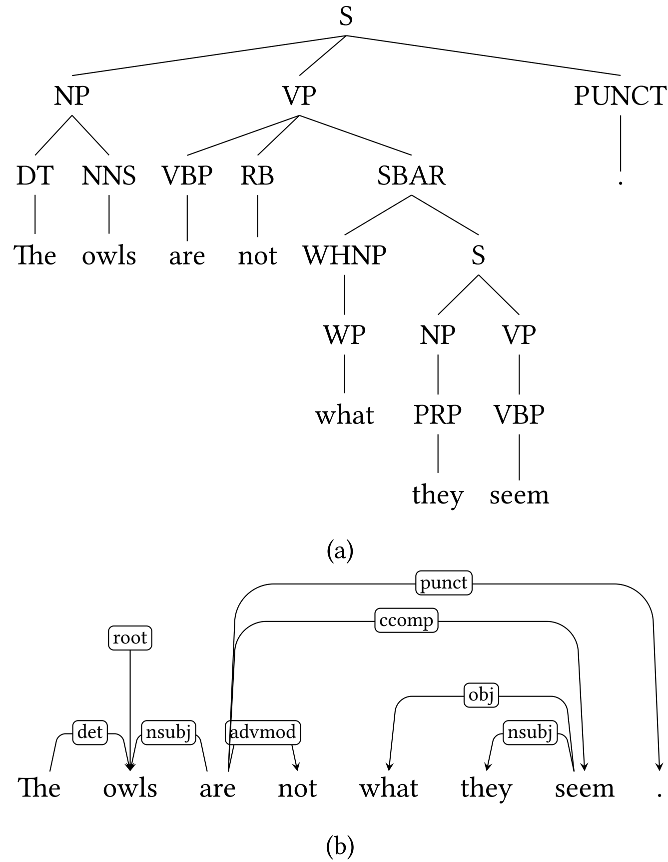


Figure 1.1: Constituent tree (a) and Dependency tree (b) for a given sentence

niques ranging from rule-based and transition-based statistical approaches up to the revolution that came from artificial neural networks. This change of paradigm allowed the parsing tasks to increase accuracy without relying so much on human-crafted features. This transition however did not solve other main problems, such as the speed of the models used to obtain the desired representation. Also, most of the neural approaches to parsing tasks have the disadvantage that they do not work with out-of-the-box with black-box neural systems and require ad-hoc parsing algorithms. This work deals with one of the most recent approaches to neural constituent and dependency parsing, that is, casting them as a sequence labeling task. This recast has already been proposed in previous work [1, 10] , but it lacked a unified system including all the available encodings and being developed with scalability and maintainability in mind, apart from working out of the box with most sequence labeling systems.

1.1 Natural Language Processing

The study of natural language processing (NLP) began in the 1940s after World War II, when people became aware of the importance of tasks such as language translation and hoped to create a machine that allowed them to automate it. But it was not until the late 50's, with researchers like Noam Chomsky [11], that grammars began to be studied with the creation of advanced language models that attempted to allow machines to understand human language. This line of research continued during the late 20th Century developing more complex tools and statistical methods of handling human language. In the most recent years the focus of the natural language processing researchers has been set on neural and machine learning systems, allowing for the creation of the first neural language models [12].

Among the different parts [13] of NLP tasks, the one that this work relates to is syntactic analysis. Obtaining the structure of a sentence based only on the tokens that compose it is a task that relates directly to the way that we, humans, shape our thought and understand language as a whole. This task can also act as a support task to other tasks by providing a structure that can be used to further refine their results. The main challenges that natural language processing tries to overcome are related to understanding the most ambiguous aspects of human language or phrase ambiguities. Generating the syntactic structure of a sentence can help to eventually overcome them.

1.2 Motivation and Challenges

The motivation from this work stems from the need for a reliable and fast way to predict syntactic structures of input sentences, both for constituent and dependency formalisms. Developing a fast and accurate way to obtain such structure can provide advantages in tasks like named entity recognition, with the structure from constituent trees providing a good basis to find entities in a sentence, or sentiment analysis, by the usage of the dependency tree for shortening the distance between opinion words or propagating sentiment information along the syntactic tree.

Traditional approaches for parsing can provide good accuracy or speed, but usually not both. That limitation was overcome by the approach of casting the task into a sequence labeling task, but the developed systems lacked scalability or extendability, making it hard to add new pieces to them or to change the way they work. The creation of a framework that integrates the different existing linearization techniques for parsing, and evaluating its utility to train neural sequence labeling systems in a black-box fashion, are the two main components that motivate and justify this work.

The tasks that the development of this system requires can be grouped as follows:

1. **Dependency tree encoding and decoding**: Given a dependency tree in its standard format, encode it into a unique sequence of labels that are representative of the tree and that can be decoded back into the original representation.

2. **Constituent tree encoding and decoding**: Given a constituent tree in its standard bracketing format, encode it into a unique sequence of labels that are representative of the syntactic structure of the sentence and that can be decoded back into the original tree.

And the main challenges faced in the development of the aforementioned tasks are the following:

1. **Scalability**: Due to the fact that this line of work is a very recent one it is probable that new ways to encode the trees are found as years come by, so the implementation of the encoding and decoding mechanisms must be as scalable as possible. That would be one of the characteristics of the developed system. Also, the possibility of encoding parse trees from new grammatical theories is always there, so the system must be open to refinements without changing the core.

2. **Loose dependency from any concrete sequence labeling system**: Since sequence labeling is still an evolving research topic and new systems often appear that outperform the existing ones, the linearization framework must not be attached to any sequence labeling system, but instead it must generate the data in a way that allows it to be used by, potentially, any generic system.

3. **Usability**: The system must present an unified and usable command line interface that allows for fast encoding and decoding of the selected files. That interface layer must also not be coupled to the business logic layer in case it needs to be attached to other types of user interface.

Verifying the system efficacy and efficiency

Apart from developing the tree linearization system, the other goal of this project is to train sequence labeling models that provide baselines that prove the linearization approach efficiency. To accomplish it, different architectures of sequence labeling models will be trained and tested on a diverse set of languages. This not only would prove the system efficiency, but will also make it more attractive, as it can be used for a wider audience.

This work will take advantage of some of the most commonly used sequence taggers. The performed experiments will use techniques like (i) the inclusion of different word features into training to help the tagger to improve its predictions, (ii) the inclusion of Multi-language

BERT [14] language model that will help increase the metrics thanks to the embedded data they provide, and (iii) the usage of multi-task learning to share the training process of multiple tasks with a single model.

1.3 Structure

This bachelor's thesis can be divided into four sections. The first part is a brief presentation to the preliminary knowledge required to understand the technical aspects introduced in the following chapters. The second part gives a rundown of how the project was developed and the tools that we used. The third part focuses on the linearization algorithms for constituent and dependency parsing. The last part shows the training and testing phases of the parsers using sequence taggers in a black box fashion. Specifically, the chapters are broken as follows:

- PART 1:
  - **Chapter 2 - Preliminaries**: Introduces the theoretical concepts for both constituent and dependency formalisms from a language theory viewpoint.
- PART 2:
  - **Chapter 3 - Project Development**: Breakdown of the development of this project, with focus on the resources employed, the methodology and the task division.
- PART 3:
  - **Chapter 4 - Sequence Labeling**: This chapter explains what sequence labeling is and how it has evolved during the years until the present time. In order to explain modern sequence labeling systems we will include explanations of modern neural network architectures like long short term memories and bidirectional neural networks. Also, in this chapter the concept of multi-task learning will be explained more in detail.
  - **Chapter 5 - Constituent Parsing**: Explanation of the algorithms implemented to linearize and de-linearize constituent trees (5.3.1, 5.3.2, and 5.3.3); and the heuristics implemented to ensure correctness from predicted outputs (5.5).
  - **Chapter 6 - Dependency Parsing**: Explanation of the algorithms implemented for linearize and de-linearize dependency trees (6.2.1, 6.2.2, 6.2.3, 6.2.4, and 26); and the heuristics implemented to ensure correctness from predicted outputs (6.3).
- PART 4:

- **Chapter 7 - Dependency and Constituent Tree Linearization System**: This chapter takes a look into how the previous algorithms were implemented into code, with the system architecture and usage. Also, it shows how the sequence labeling systems were trained.
- **Chapter 8 - Results and Performance**: In this chapter we show how the different algorithms perform for the designated treebanks and for the two sequence labeling systems (NCRFpp [15] and MACHAMP [16]).
- **Chapter 9 - Conclusions and Future Developments**: Final part of the bachelor thesis. In this chapter the contents of the whole work are put into contrast with lessons learned from developing it, and possible future developments are commented as well.

Chapter 2
Preliminaries

In order to understand natural language processing, familiarity is required with aspects from different disciplines, ranging from the fields of linguistics to mathematics applied to computational theory. This chapter will discuss the necessary concepts required for a solid understanding of the following chapters. Specifically, it will cover (i) the aspects needed to understand parsing tasks, (ii) what are parse trees and (iii) previous developments on systems to obtain them.

2.1 Constituent Grammars

Constituent grammars [17] are a set of grammatical theories based on the concept of *constituency relations*, defining a sentence as a group of constituents in a hierarchical structure. This concept is founded on the idea that a sentence is built of smaller structural pieces that have their own intrinsic meaning and give meaning to the sentence as a whole. This relates to the idea that groups of words can behave a single units or constituents, what is commonly referred as the substitution principle (see table 2.1).

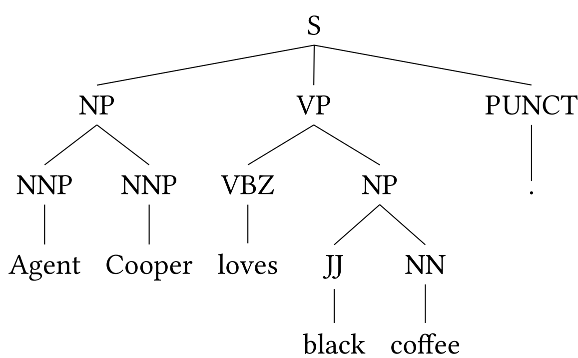
Sentence		Substituted Sentence
The FBI agent solved the mystery.	$\xrightarrow{\text{Nominal Phrase}}$	He solved the mystery.
He solved the mystery.	$\xrightarrow{\text{Nominal Phrase}}$	He solved it.

Table 2.1: Example of the substitution principle used in constituent grammars.

Constituent grammars are modeled after context free grammars (CFG), that follow a similar theoretical principle. Generally, CFGs consist of a set of production rules, that express the ways in what the symbols of the language can be grouped and ordered, and a lexicon, the set of words and symbols that make up the language. For example, a noun phrase (NP) could be

composed either by a proper noun or a determiner followed by any nominal, and a nominal could be made up or one or more nouns.

The symbols of CFGs are divided into terminal symbols (when they correspond to tokens in the lexicon of the language) and non-terminal symbols (when they express abstraction over them). All CFGs also require to have a starting non-terminal from which we can, by using the grammar rules, reach any terminal belonging to the lexicon. The representation of the sequence of rule expansions that we use to go from the root Non-Terminal to the Terminals is called parse tree. In order to represent constituent parse trees in files we employ the bracketing (or parenthesized) [18] tree format. This format represents the span of a given level of a tree surrounding all its children nodes with a pair of parenthesis (see figure 2.1).



(a)

(S (NP (NNP Agent) (NNP Cooper)) (VP (VBZ loves) (NP (JJ black) (NN coffee))) (PUNCT .))

(b)

Figure 2.1: Example of a constituent parse tree for the sentence "Agent Cooper loves black coffee." represented in graphical format (a) and in bracketed format (b).

As we have seen in chapter 1, the computation of the syntactic structure of a sentence is a core aspect in NLP, so it is very important to design tools that allow us to efficiently parse this kind of trees.

2.1.1 Statistical Models for Constituent Parsing

Statistical model based systems were one of the first approaches to obtain constituent parse trees. Even though these systems offered good accuracy in their predictions, they had a big downside in terms of speed. Generally speaking, the statistical models work by deciding among all the possible constituent trees the most accurate one, that is, for each possible tree T for the sentence S they compute the conditional probability of $P(T/S)$. To build the possible trees these systems used rule-based approaches, dynamic programming approaches or *beam-search* algorithms. These statistical models were refined as time passed, adding increased

heuristics to get the probability of an element being a pre-terminal (i.e., a PoS tag), a terminal or a non terminal [19] or by refining the grammar via split-merge techniques [20].

2.1.2 Transition-based Systems for Constituent Parsing

Later works tried to obtain faster models by taking inspiration from transition-based parsing techniques used in dependency parsing (as we will see in section 2.2). This change of paradigm allowed to do constituent parsing in linear times [21]. The transformations between grammars relied on transforming the constituent tree into a binary tree via the introduction of intermediate nodes.

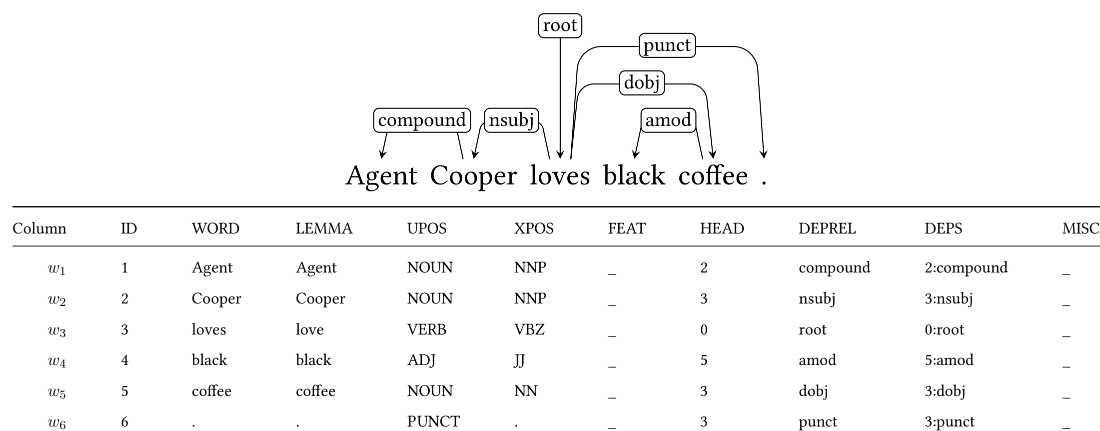
In this kind of algorithms the binary constituent tree is parsed via a *bottom-up shift-reduce parsing* trying to generate the tree from the leaves back to the root by using a stack to store symbols and a set of transitions to manipulate said stack and build the output tree.. The workflow of these systems consisted in the application of rules to obtain the start symbol of the grammar from the leaves. Shift-Reduce algorithms were developed and improved in the recent years by using different classifiers (support vector machines, decision trees or memory learning) [22]; by developing systems that can avoid the step of transforming the constituent tree into a binary tree and still use shift-reduce algorithms [23] or by the inclusion of neural structures in the statistical computation like recurrent neural networks [24].

2.2 Dependency Grammars

Dependency grammars are a group of grammatical theories based on the idea of *dependency relations*. This idea can be traced back to first grammatical theories, but modern work dates back to research from the last century [25] refining the mathematical aspects of this grammar. This kind of grammars emphasize word semantics, assuming that the whole sentence is a derivation from the dependency relations between them. This is motivated by the concept of grammatical function, i.e., that a word depends on another if it complements or modifies the meaning or structure of the latter. The dependency structure of a sentence in a defined dependency grammar can be represented as an edge-labeled tree $T = (V, E)$ where V will correspond to the indexes of the words of the sentence and E will correspond to the set of edges that capture the dependent relationships between elements in V shaped as a 3-tuple (h, t, d), where h indicates the index of the head of the relation, d indicates the index of the dependant of the relation and t indicates the dependency type among them. The task of building this tree from the sentence $W = [w_1, ..., w_n]$ is what we call dependency parsing.

The standard format for representing dependency trees in files is the X Conference on Computational Natural Language Learning (CoNLL-X) [26] format. For this work we will

employ the revised version CoNLL-U[1] employed by the Universal Dependencies treebanks. This format represents the dependency trees as a set of lines of fields containing the annotations of each word/token from the input sentence. The nomenclature employed in these fields follows the Universal Dependencies nomenclature[2] standards. In the figure 2.2 we can see the dependency parse tree for a given sentence and its CoNLL-U representation.



Column	ID	WORD	LEMMA	UPOS	XPOS	FEAT	HEAD	DEPREL	DEPS	MISC
w_1	1	Agent	Agent	NOUN	NNP	_	2	compound	2:compound	_
w_2	2	Cooper	Cooper	NOUN	NNP	_	3	nsubj	3:nsubj	_
w_3	3	loves	love	VERB	VBZ	_	0	root	0:root	_
w_4	4	black	black	ADJ	JJ	_	5	amod	5:amod	_
w_5	5	coffee	coffee	NOUN	NN	_	3	dobj	3:dobj	_
w_6	6	.	.	PUNCT	.	_	3	punct	3:punct	_

Figure 2.2: Example a Dependency Tree for the sentence "Agent Cooper loves black coffee." in graphical format (a) and in CoNLL-U format (b).

Projectivity An important notion we can derive from the order of words in the input and the dependency relations among them is *projectivity*. An arc from the head to a dependent is said to be *projective* if there is a path from the head to every word that lies between the head and the dependent in the sentence. A dependency tree is said to be projective if all the arcs in it are projective, that is, we can draw all its arcs without any crossings. As non-projective trees often appear in practice, the capability of deal with them is a desirable property for dependency parsers.

A related property is k-planarity. Formally, we say that a dependency graph $G = (V, E)$ is k-**planar** for each $k \geq 1$ if we can make partitions of E into $E1, ..., E_k$ planes such that the edges in each partition don't cross, that is, they are projective. If $k = 1$, the dependency tree is projective, otherwise, it is not. Even though theoretically we could have any number of planes, in practice most of the trees are solved by a 2-planar system. An example from a 1-planar dependency tree and a 2-planar one is shown in figure 2.3.

[1] https://universaldependencies.org/format.html
[2] https://universaldependencies.org/u/pos/index.html

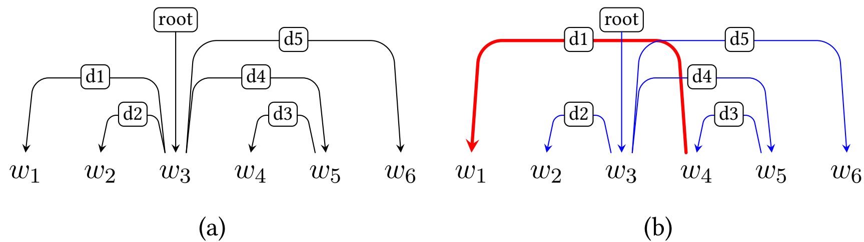


Figure 2.3: Projective dependency tree (a) and non-projective dependency tree (b). Dependency tree (b) edges are separated in two planes indicated by red and blue colored edges.

2.2.1 Transition-based Systems for Dependency Parsing

One of the main paradigms traditionally employed for dependency parsing systems are the transition-based parsers. This kind of systems are built using a stack to navigate through the sentence, a buffer, that contains the input tokens, and an oracle, that predicts the action that the parser will take at each step. Transition-based systems have been applied to dependency parsing since long time ago [27]. The main advantage of these systems is their speed (they can run in $\mathcal{O}(n)$), making them very useful for tasks where the speed factor is a critical one. Through the years, they also have been adapted in order to improve performance. Such improvements have come from exploiting language-dependent hand-crafted features [28] (e.g. morphological information) and for transition-based systems that included neural networks [29, 30]. However, a recurrent problem of this type of parsers is that they are better at local (shorter) dependencies than at longer ones. This problem arises in part due to the greedy nature of this kind of systems, that may lead to false early predictions, contributing to error propagation, and messing up the final tree.

2.2.2 Graph-based methods for Dependency Parsing

The other main paradigm employed for dependency parsing are graph-based methods [31]. These systems take the whole dependency tree into consideration in order to make the head assignment decisions. This allows graph-based systems to out-perform transition-based systems for sentences where the head and the dependant are far away, but under-performs in sentences where they are close. This kind of parsers work by searching through the entire space of possible trees for a given sentence for a tree that maximizes some arbitrary score. The critical task of graph-based algorithms is defining that scoring function, which will define what (sub)trees are considered for computation. Traditionally, the scoring was done by a *feature-based algorithm*. The problem of this kind of approach is that it needed too many hand-crafted features that made the training process excessively long. This was overcome by the usage of neural networks [32]. In spite of this change, the theoretical limitations of graph-based parsers remain, e.g. their computational complexity is $\mathcal{O}(n^2)$.

Chapter 3

Project Development

This chapter will discuss the planning details related to the development of the constituent and dependencies linearization. We will talk about the methodology used and how the planning of time and resources was done. Finally, we will do a short rundown of the tools and resources used during the development and evaluation stages.

3.1 Resources

The resources employed in this project can be differentiated between the human resources that participate on it, the tools employed during the development and the data employed for training the models and verifying that the system reaches the desired levels of performance.

3.1.1 Human Resources

The people involved in this project will be split into *supervisors* and *developers*, where the assigned project directors will act as supervisors and I will myself be the developer. The supervisors were involved during the whole project development, participating with weekly control sessions where they validated the correct development of the system and ensured that time constraints were respected. They also were available any time on demand for problem solving or to provide resources.

3.1.2 Tools

1. **High level programming language**: Python 3.8. Python is a high-level language commonly used in this kind of projects due to the high amount of libraries available related with the fields of natural language processing and machine learning.

2. **Libraries**: The project required of libraries that could read input/output data from the sets of data employed and also required libraries that coud represent them in an efficient

way. In order to read dependency trees the library CONLLu was employed. For reading constituent trees, the libraries NLTK and Stanza were considered, but due to Stanza being faster in preliminary tests, it was the chosen one.

3. **Support tools**:
   - Code Development: Visual Studio Code. Chosen due to the great amount of extensions related with Python development available.
   - Version Control: git, and specifically, Git Hub desktop application.
   - Package Management: Conda. Chosen because it allows to create virtual environments and allowed for a good compatibility with the CUDA libraries needed for the sequence labeling tools used in this project.

4. **Sequence labeling systems**: To demonstrate the system's viability, the frameworks used to train the parsing models were: (i) NCRF++, a state-of-the-art neural sequence labeling tool, and (ii) MACHAMP, a multi-purpose neural system based on multi-task learning that can be used with large language models embeddings in an straightforward way.

5. **Evaluation tools**: The testing scripts used for this project are EVAL-B with COLLINS parameters and EVAL-SPMRL, for constituent parsing, and Conll-Eval for dependencies. These tools are the most used when evaluating other constituent and dependency parsing systems, therefore using them makes our system more easy to compare with other works.

3.1.3 Data Sets

These are the treebanks that we will use to train, evaluate, and test the models. The data banks are the following:

1. Penn Treebank [18]: Corpus from the Wall Street Journal annotated and represented as bracketed constituent trees. Licensed under PTB.

2. SPMRL [33]: Collection of multilingual annotated constituent treebanks. The license was provided by the project coordinators from the SPMRL shared task to the LyS research group. This collection included the following languages: Basque, French, German, Hebrew, Hungarian, Korean, Swedish and Polish.

3. Universal Dependencies Treebanks [34]: Collection of dependency treebanks for different languages, in CoNLL-U format. Most of treebanks are freely available and have

been adopted as the standard collection for the evaluation of dependency parsing models. The languages selected for this work are the ones that are also included in SPMRL, for comparison purposes.

In order to train the sequence labeling systems that are going to be used we follow the standard split in supervised automatic learning to divide our data into three groups:

1. Training set: Used to train the neural system and adjust its weights.
2. Development set: Used by the neural system to verify the quality of the predictions during training.
3. Test set: Used afterwards to test and evaluate the system on a set of samples that emulate a real environment.

3.2 Evaluation Methods

The evaluation metrics employed to prove that the developed system reaches the desired levels of performance are different depending on the formalism being evaluated. Such metrics are the following:

1. **Dependency trees**: The most employed metric for dependency tree evaluation is the attachment score. This metric computes the number of well assigned dependency edges. We can distinguish two variations of attachment score:
   (a) **Unlabelled Attachment Score** (UAS): Percentage of dependency edges where the head is well assigned.
   (b) **Labelled Attachment Score** (LAS): Percentage of dependency edges where the head *and* the relationship type are well assigned.
2. **Constituent trees**: For constituent parsing evaluation we will employ the *F-Score* value. This metric is computed as the harmonic mean of the precision and the recall of the decoded constituent trees, where precision is the number of correctly predicted constituent relative to the total number of predicted constituent and recall is the number of correctly predicted constituent relative to the number of *gold* constituent. We consider that a constituent in the predicted tree is correct if it matches the constituent in the same position of the gold tree. F-Score is computed as follows:

$$F - Score = \frac{2 \times P \times R}{P + R} \quad (3.1)$$

3.3 Project Planning and Methodology

The development methodology employed for this project, due to its nature, was based on an *incremental model*. This development methodology provided the advantage of an easy way to debug and test each component of the system and share on-going results with the interested parties. The core of the incremental methodology is the division of the project into a set of clearly defined iterations or cycles, with each one having their own analysis, design, implementation and testing phase.

3.3.1 Tasks Definition

The first block of the project consists in a single start-up task where the working environment is created and the remote access tools are set up. This task also includes time for the familiarization with the tools and libraries that will be used in the project.

1. **Start up** - Set the working environment and become familiar with the state-of-the-art developments in the constituent parsing and dependency parsing areas.

The next block of the project consists in developing the tree linearization system. These tasks are developed in a fully incremental fashion. The number of iterations and what they accomplish are closely related to the encodings discussed in sections 5.3 and 6.2. These tasks are:

2. **Constituent trees - naive absolute encoding**: Develop the encoding and decoding functions for transforming constituent trees into labels using the naive absolute encoding as discussed in section 5.3.1. Create all the classes for representing the constituent trees and input/output subsystems. Create the heuristic functions to ensure tree correctness.

3. **Constituent trees - naive relative encoding**: Develop the encoding and decoding functions for the naive relative encodings, as discussed in section 5.3.2.

4. **Constituent trees - dynamic encoding**: Develop the encoding and decoding functions for the dynamic encoding, as discussed in section 5.3.3.

5. **Dependency trees - naive absolute encoding**. Develop the encoding and decoding functions for dependency trees following the absolute naive encoding, as discussed in section 6.2.1. Create all the classes for representing dependency trees and input/output subsystems. Create the heuristic functions to ensure tree correctness.

6. **Dependency trees - naive relative encoding**: Develop the encoding and decoding functions for naive relative encodings, as discussed in section 6.2.2.

7. **Dependency trees - part-of-speech based encoding**: Develop the encoding and decoding functions for part-of-speech based encodings, as discussed in section 6.2.3.

8. **Dependency trees - bracketing encoding**: Develop the encoding and decoding functions for the bracketing-based encodings, as discussed in section 6.2.4. Develop the planar-separation algorithms, as discussed in section 26.

Once the linearization system is developed the next block of the work consist in training standalone sequence labeling systems, to provide metrics that demonstrate the system's accuracy and efficiency. To do so, for each sequence labeling system we considered the following steps:

- For each encoding, we linearize the selected constituent and dependency treebanks. In total, 9 languages were trained for 3 different constituent encodings and 9 dependency encodings. The treebanks and languages employed are discussed in-depth in section 3.1.3.
- Training the selected sequence labeling systems for each one of the encodings and for each treebank.
- Decoding the predicted test files from the corresponding treebank.
- Applying the evaluation tools to obtain accuracy metrics from that prediction.

The tasks added to the project to deal with the sequence labeling systems training are the following ones:

9. **NCRF++ Training** - Train the NCRF++ sequence labeling tool, and evaluate results.

10. **Machamp BERT Embeddings Training** - Train the MACHAMP tool with Multi-Lingual Bert [14] language model embeddings added to the training; and evaluate results.

11. **Machamp BERT-MTL Training** - Train the MACHAMP tool with Multi-Lingual Bert [14] language model embeddings added to the training, as well as in multi-task learning setup, using features extracted from the treebanks; and evaluate results.

The final block of the project consists in only one task related to writing the bachelor's thesis.

12. **Results and Evaluation** - Final task of the project. Evaluate the whole development and write the thesis.

The time span to complete all the defined tasks of this project ranged from February to September. That makes 7 months to complete the project. The first 4 months were dedicated to the development of the linearization system, and the last 3 months were employed to train the sequence labeling tools, and to write the bachelor thesis. To do that management and keep track of deviations, a Gantt diagram was used. This diagram is shown in figure 3.1.

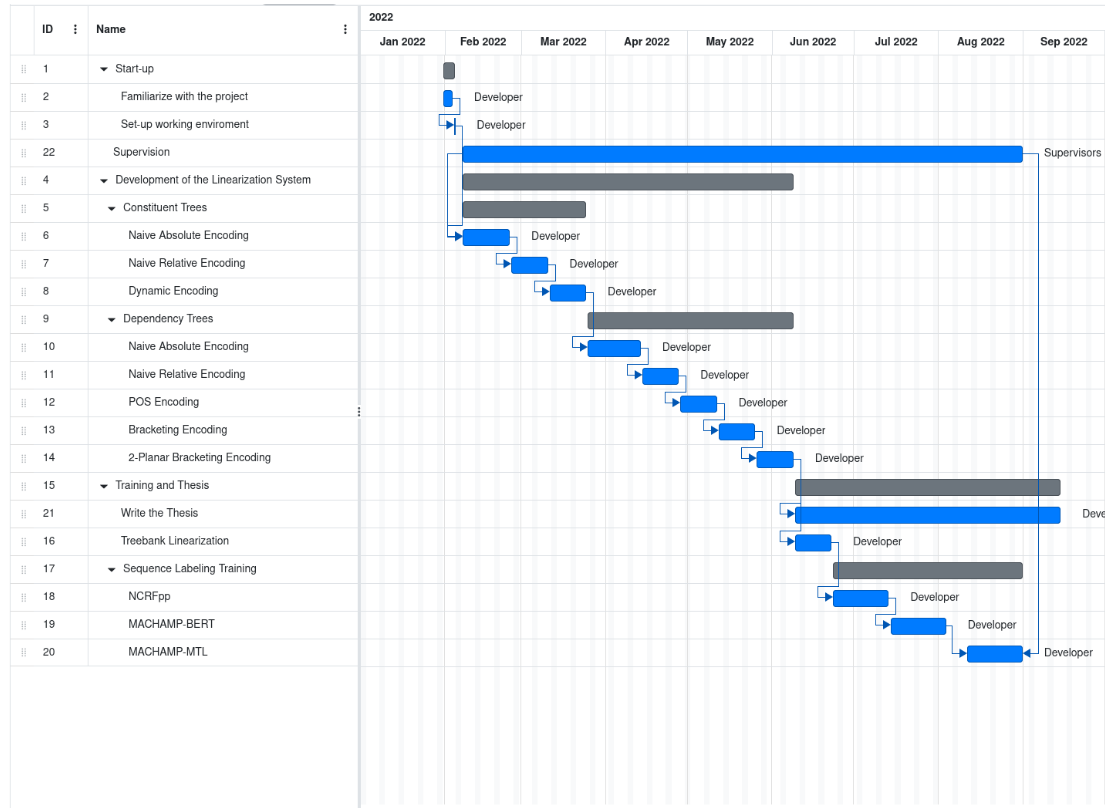


Figure 3.1: Gantt Diagram of the project.

Chapter 4

Sequence Labeling

Sequence Labeling is a structured prediction task whose goal is to map an input sequence of tokens $W = [w_1, ..., w_n]$, that in NLP usually correspond to the words of a sentence, to a unique sequence of output labels $L = [l_1, ..., l_n]$, where for each input token there is one and only one assigned label. In natural language, the context of a sentence is very important to understand its meaning, as the intent of a word w_i can change depending on its left (or right) context. Good sequence labeling systems should therefore be able to model context. Also, sequence labeling is an interesting and popular paradigm because it is easy to understand and implement, while usually offering a good speed-accuracy trade-off. However, not all NLP tasks can be cast under this paradigm, and sometimes casting some tasks in this way is not trivial, as it is the case of parsing that we will discuss in the upcoming chapters.

In this chapter, we will explore how sequence labeling relates to NLP, and how sequence labeling models have evolved through the years until the neural architectures that we will be using in this work.

4.1 NLP tasks traditionally cast as sequence labeling

We briefly introduce here some core NLP tasks that have been traditionally cast as sequence labeling:

1. Shallow Parsing [35]: This task consists in extracting the different meaningful parts of a sentence in the form of chunks by indicating the words that mark the start of the chunk and the words that mark the end. The beginning and end of chunks are usually marked by IOB-based (Inside-Outside-Beginning) labels (e.g. in But_1 it_2 could_3 be_4 much_5 worse_6, the label for could_3 is B-VP, indicating that it begins a verbal phrase and be_4 label is I-VP because it is inside the verbal phrase). Because those labels are dependent of the context, these tasks can take advantage of casting them as a sequence labeling problem.

2. Part-of-speech tagging (PoS tagging) [36]: This task consists in assigning to an input sentence a sequence of labels that correspond to the grammatical properties of each word (noun, pronoun, verb...). Sequence labeling provides an advantage for this task due to be able to predict the tag for a word w_i based on the rest of the sentence. This can provide an advantage when predicting the part-of-speech tags for polysemous words i.e. in the sentence "The kid went out to run" the word "run" acts like a *verb* but in the sentence "It was a great run", the word "run" acts like a *noun.* POS tagging tasks usually act as a pre-processing step for more complex natural language processing problems, including syntactic parsing.

3. Named entity recognition (NER) [37]: This is a task where for a given input sentence, the model must find and classify named entity chunks (person, organization, location, etc) inside a text and assign the corresponding label to them. The performance of this task greatly benefits from context (i.e. in the sentence "Paris$_1$ Hilton$_2$ visited$_3$ the$_4$ Paris$_5$ museum$_5$." Paris$_1$ is a person and Paris$_5$ is a location) and in consequence greatly benefits from casting it as a sequence labeling task. The output from this task can be useful for problems where we want to identify proper nouns in a sentence or can act as a pre-processing step for more advanced tasks (i.e. *text mining*).

4.2 Architectures for sequence labeling systems

The architectures of sequence labeling models have evolved in the last years from statistical models to transformer-based models. The limitations that forced this evolution of the task were usually related to aspects such as limited performance, the training process requiring many hand-crafted features, and the impossibility to model long contexts.

4.2.1 Probabilistic models

A traditional and popular approach to build sequence labeling models are machine learning statistical models, specifically hidden Markov models and conditional random fields. The computation of probabilities for these models rely on hand-crafted features obtained from rule-based models, where the rules are designed based on language characteristics [38], or feature-based models [39], where the rules are designed based on vectors which encode word-level or sentence-level features. These first approaches to sequence labeling were mathematical and statistical models founded on the notion of conditional probabilities (see definition 4.2.1).

Definition 4.2.1. Conditional probability is defined as the likelihood of an event A occurring based on the occurrence of a previous event B. This is written as $P(A/B)$ and it reads as "The

probability of A happening taken that B has happened".

Hidden Markov Models

Hidden Markov models (HMMs) are mathematical models developed at the start of the twentieth century based on the concept of *Markov chains*. These models compute the probabilities of sequences of random variable states, where each one can take on values from some predefined set. The main assumption of Markov chains are that in order to predict a state s_i the only state that matters is the state s_{i-1}, not caring about states s_{i-2}, s_{i-3}, ... s_{i-n}. For any given Markov chain, the only data taken into account for the computation of probabilities is observable data, for example, words in the input sentence. Hidden Markov models are a more advanced model of the Markov chains where the computation of probabilities can take into account non-observable data. This makes it possible to take into account "hidden" events into the computation, refining the probabilistic computation even more. In the field of Natural Language Processing, these "hidden" fields are usually the word features (for example, the number of a noun or the person of a verb) or word tags (for example, part-of-speech tags).

The main drawbacks of HMMs are (i) the need for hard hand-crafted input data and (ii) the limitations that the model topology cause to how the predictions are computed:

1. The Transition Matrix $A = [a_{11}, a_{12}, ..., a_{1n}, ..., a_{mn}]$ is a static structure, this means, the transition from one token to another is independent of their position in the input sequence.

2. The transitions in a HMM only take into account the previous element, meaning that any token in the input sequence at a distance $d > 1$ will not be taken into account for the probability computation.

Conditional Random Fields

Conditional random fields models seek to avoid these limitations by changing the topology from a chain, where the arrows affect directly the element that comes after, to a non directed graph, where the set of observed states $S = [s_1, ...s_n]$ and the set of hidden observations $O = [o_1, o_2, ..., o_t]$ can be linked in any possible way. Another big change from hidden Markov models, that makes conditional random fields models more powerful tools, is the change from being a *Generative Model* (i.e. model that attempts to guess how a prediction is generated) into a *Discriminative Model* (i.e. model that searches for the correct prediction in a search space), meaning that we can compute P(X|Y) directly without the application of the Bayes rule.

4.2.2 Artificial Neural Network Models

A challenge to overcome in sequence labeling statistical models is the dependency on hand-crafted features to better model context in sentences. This kind of models also need a huge lexicon during training to obtain good accuracy metrics. For these reasons recent sequence labeling tools have changed the core of the system to neural architectures. This kind of systems do not rely on hand-crafted features, and can better model long contexts, making them a very powerful tool.

Multi-layer Perceptron models

The first artificial neural network approaches to sequence labeling tasks came with the usage of a multi-layer perceptron [40] model. Perceptrons allow to predict outputs based on an input using weights computed through several hidden layers. These models were developed in the middle of the 20th century and are still the basis for many more advanced neural network systems. This kind of models are formed by three components:

1. Input layer: Receives the input signal to be processed.
2. Output Layer: Performs the prediction task.
3. Hidden Layer(s): Intermediate layers in charge of learning the latent representations from the input.

The weights of the network connections between these layers are initialized randomly and are updated during training via error backpropagation [41], an algorithm that allows a network with any number of hidden layers to adjust the weights based on a loss defined in the output layer. More specifically, multi-layer perceptrons treat sequence labeling tasks as a set of independent predictions, where for each input word w_i in a sequence $W = [w_1, ..., w_n]$, we predict an output label l_i based on the window of surrounding n tokens (this can be left, but also right context if wished). The main limitation of this kind of models comes from the fact that the context that the predictions use is of a limited size n: modeling long context becomes sparse, contributing to inaccurate predictions.

Recurrent Neural Network models

Recurrent neural networks (RNNs) [42] are a type of neural networks that differentiate themselves from traditional neural networks by adjusting the hidden layers recursively, allowing the system to extract large context from sequential data. RNNs use a recurrent architecture where for processing a given input w_i, the hidden state h_i will be computed using the previous hidden state h_{i-1}. This approach to recursion allows the system to overcome one of

the main shortcomings of traditional neural networks and be able to 'remember' the previous context for a given token in the sequence.

Even if RNNs provide a very elegant way of dealing with sequential data, their architecture makes them only effective in practice between data points that are spatially close in the sequence. This makes recurrent neural networks only perform well with shorter sequences of input data and not being able to keep a good representation of context for large ones. This problem is caused by the way in which this kind of architectures implement backpropagation, resulting in that any error caused by an input element decreases its effect (or vanishes) over the system as time passes. In order to avoid it, most of sequence labeling systems implement a more advanced version of recurrent neural networks called long short-term memories (LSTMs) [43]. LSTMs handle the problem of vanishing weights by the inclusion of a more advanced gate-based architecture to the network. This architecture is based on the usage of a cell state (indicated in (d) in figure 4.1) that contains the flow of information in the memory and modifies the next hidden state of the network. This cell state can be modified by the usage of different gates that allow information to be forgotten, added or updated.

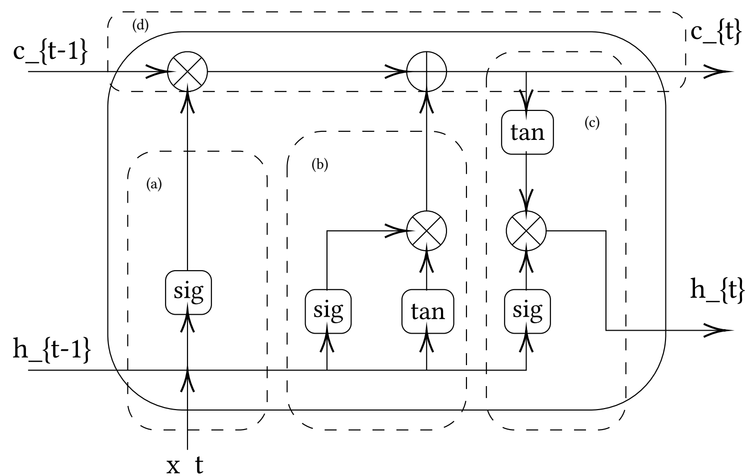


Figure 4.1: Diagram of a LSTM cell where (a) is the Forget Gate, (b) is the Input Gate, (c) is the Output Gate and (d) is the Cell State inspired by https://colah.github.io/posts/2015-08-Understanding-LSTMs/.

An easy improvement that can help better model context in RNNs is to include right context from the sentence. This can be done by running two RNNs in opposite directions. In this way, *Bidirectional Recurrent Neural Networks*, can capture information both from left-to-right and right-to-left. In this case, the output is a concatenation of the outputs from both RNNs in different directions. We show the formal definition for the particular case of bidirectional LSTMs (BiLSTMs) in definition 4.2.2.

Definition 4.2.2. Let $LSTM(w)$ be an abstraction of a Long Short-Term Memory RNN and

let $w = [w_1, ..., w_{|w|}]$ be the sequence of tokens that the system has as input, then, the output for each w_i in as $o_i = BiLSTM(w, i) = LSTM^l(w_{[1:i]}) \circ LSTM^r(w_{[|w|:i]})$. This reasoning can also be applied to stacked bidirectional recurrent neural networks, where the output o_i of the nth BiLSTM layer is fed as input to the $(n+1)^{th}$ layer.

4.2.3 Transformer-based Sequence Labeling Systems

Transformer architectures [44] have become a standard architecture for many natural language processing tasks, while being the basis of most well-known language models like *BERT* [14] or *GPT* [45]. Transformers introduce the concept of *self-attention mechanism*, that allows a token from the input sentence to attend to any other token without regarding or penalizing the distance between them. Also, this allows for parallelization at the token level, instead of at the layer level (such as for RNNs). The inner workings of this are based on a multi-layer encoder and a multi-layer decoder modules. The encoder will transform each item in the input sequence and codify its information into a vector (called *context*) that will be passed onto the decoder. The decoder will translate these vectors into the output sequence. Both encoder and decoder are usually built using RNNs.

4.3 Architecture of a Sequence Labeling System

In this section we will introduce a generic architecture for a sequence labeling system based on some of the aforementioned neural models. Specifically, we will discuss a BiLSTM based sequence labeling model with a softmax output layer. A sample diagram of this architecture can be seen in figure 4.2.

Token representation layer

This module will deal with the transformation of the sequence of raw input data into a sequence of vectors representing each word. These representations are built using *embeddings*, a low-dimensional vector representation of features of words from a sentence. This embeddings can come from pre-trained systems and their inclusion during training of other models can provide increments in the accuracy of its predictions. Some examples of these representations are (i) pre-trained word embeddings that extract the information at the word level (i.e. *word2vec* [46]; GloVe [47]), (ii) character-level word embeddings, vector representations computed using neural models (i.e. LSTMs[48], GRUs[49] or CNNs[50]) using the characters of the input word, and (iii) feature embeddings (e.g. POS tags, NER tags...) that are initialized randomly and are updated during training by the system itself. One of the main drawbacks of word embeddings, both pre-trained and at character-level, is that they are static representations of the words, meaning, they don't take into account the context where the word is.

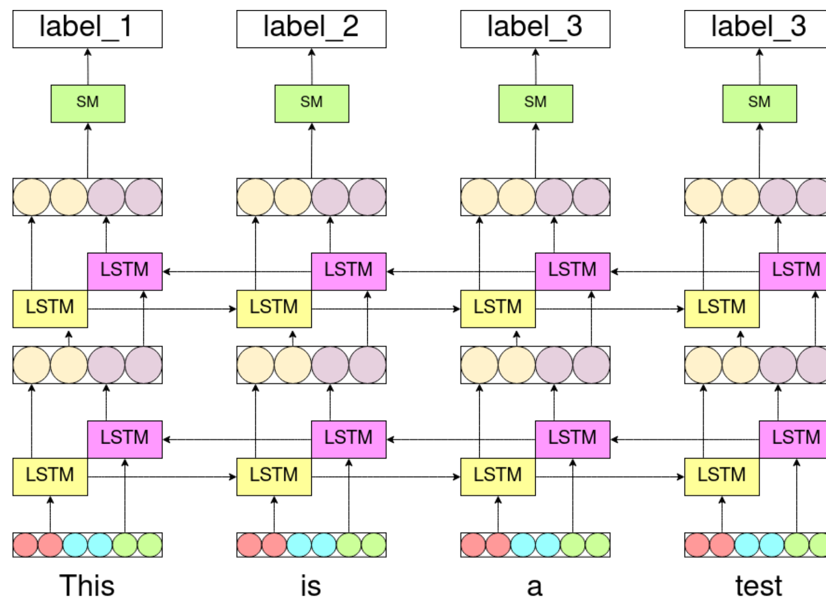


Figure 4.2: Sample architecture of a sequence labeling system. In the token representation layer we have pre-trained word embeddings (red), character-level word embeddings (blue) and feature embeddings (green). This is forwarded to the encoding layer built on 2 stacked bidirectional long short term memories (yellow and pink) and its output is sent to a softmax layer (green) to get a predicted label.

To overcome this, research switched to contextualised word-embeddings or, language modeling embeddings. Language models have the advantage over traditional word embeddings because they are trained using more advanced architectures (e.g. Transformers) and thus allowing them to better model context. Some well known examples are ELMo [51] or BERT [14].

Encoder

The second module of this type of sequence labeling systems is the encoder. The inputs to this module are the word representations computed in the token representation layer, which may include the character-level word embeddings, pre-trained word embeddings or feature specific embeddings. This model can contain multiple layers, that can be stacked to build a even deeper feature extractor. The most common neural models used as encoders are stacked CNNs or LSTMs.

Decoder

The last component of a typical neural sequence labeling model is the inference or decoder module. This module takes the contextualized vectors corresponding to each word from the

sentence and obtained from the encoder and assigns one and only one label. Essentially, the module is responsible from transforming the hidden representation into the sequence of output labels. This can be done in different ways, e.g.,: (i) with a multi layer perceptron that uses a *softmax* function to output a probability distribution over the output vocabulary space and select the most likely label, or (ii) using conditional random fields. CRFs were already explained in section 4.2.1, and the softmax function is a mathematical function that converts a given hidden representation (h_i) into a vector of probabilities $p = [p_1, p_2, ..., p_m]$, where each probability p_i indicates the probability of each possible output label from a set $L_{|m|}$ to be the correct output for a given h_i. The decision of what type of decoder to use depends on different factors. While CRFs [52, 53] are better to model interdependence across labels, they become slow when the output label space is large. On the contrary, straightforward decoders that simply rely on a feed-forward network with softmaxes are very fast [54], but might obtain worse performance.

4.4 Adding Multitask Learning to Sequence Labeling Systems

A way to improve generalization for any automatic learning system, and in this situation for sequence labeling systems, comes with the addition of multi-task learning [55]. The main premise is that solving many tasks together provides a better understanding that leads to models with better generalization abilities. This potential improvement comes from the idea that; (i) learning different talks might help to extract potential hidden patterns that would remain unexploited if learned tasks are trained separately, and (ii) possible biases that would appear in a single learning system and that would lead the model to over-fit do not occur thanks to information obtained from other tasks. To implement this idea, multitask learning (MTL) systems can take different approaches depending on how the parameters are shared across tasks:

1. **Hard parameter sharing**: The hidden layers are shared between all tasks, that is, different tasks share the same representation and parameters. The output layer is a task-specific layer that is updated differently for each task. Figure 4.3 shows the architecture of a hard-parameter sharing multi task learning system. This is the approach employed for the multi-task learning in this work.

2. **Soft Parameter Sharing**: Each task has its own hidden layer and output layer, but the parameters among those layers are regularized among them by penalizing the distance between different the different models parameters.

As we can see, MTL fits naturally in situations where we need to obtain predictions for multiple tasks with a single model. In relation to the field of natural language processing the

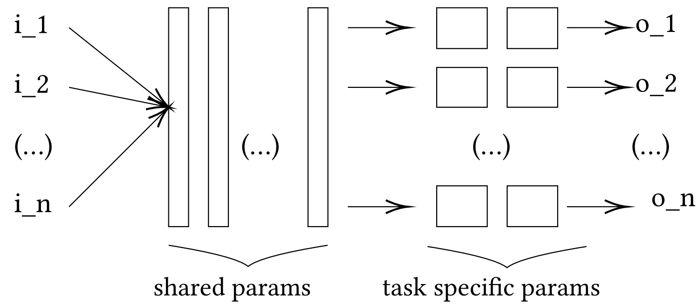


Figure 4.3: Architecture of a Multi Task Learning System with Hard Parameter Sharing

potential of this kind of systems is that as several tasks are trained in parallel, several language features (i.e. POS tags, lemmas, NER...) can also be learned. An additional advantage that multi-task learning provides to sequence labeling systems, and that we will exploit in this work, is to reduce the label space into different sub-spaces when possible, i.e. when a label can be decomposed. For example, for a label with 3 atomic components, $l_i = A + B + C$, we can decompose the task of learning the label into 3 different tasks where the system has to learn each field A, B and C separately. The output label space from performing this system as a single-task learning problem potentially is $|A|$x$|B|$x$|C|$, while treating it as a multi-task learning problem we can get a output label space of only $|A|$+$|B|$+$|C|$.

Chapter 5

Constituent Parsing as a Sequence Labeling task

Constituent grammars are a set of grammatical theories based on the idea that a sentence is made of several smaller parts organized in a hierarchical structure called *constituents*. Obtaining such tree is a task called *constituent parsing*. In this chapter we will discuss how to cast the problem of constituent parsing into a sequence labeling task by showing how a constituent tree can be encoded into a sequence of labels. We will also show how those labels can be decoded back into a constituent tree and all the heuristics used to ensure the tree correctness.

5.1 Constituent Parsing as Sequence Labeling

Transforming a constituent tree into a sequence of one label per word allows us to use all advantages of sequence labeling systems [1] such as the speed or the black-box capabilities. This process however faces the challenge to encode a tree structure into a sequential list structure. The goal of the developed system is to implement different ways to encode a constituent tree as a sequence of labels in an unified system.

In order to explain the linearization process that transforms the constituent tree Tc corresponding to a sentence $W = [w_1, w_2, ..., w_{|W|}]$, we first must define a set of labels L and a function $Fc_{|W|} : V_{|W|} \rightarrow T_{|W|}$ that allows us to encode Tc as an unique sequence of labels in $L^{|W|}$. Once the function is defined we need to develop its inverse function $Fc^{-1}_{|w|}$ that allows us to restore the predicted labels back into the constituent tree shape. The different implementations for that function will be explored in sections 5.3.1, 5.3.2 and 5.3.3. Once the encoding and decoding functions are implemented, the generated labels can be fed to a black-box sequence labeling system that will learn to predict a function $\Gamma_{|w|,\theta}$ that generates the sequence of labels $L^{|w|}$ that represent an encoded constituent tree for a given sentence w. This process is formally defined in definition 5.1.1.

Definition 5.1.1. Given the following elements:

1. V: A given lexicon of tokens.
2. $W = [w_1, w_2, ..., w_{|W|}$: An input sequence of words represented as such that every $w_i \in V$
3. $Tc_{|W|}$: The set of constituent trees with $|W|$ leaf nodes.
4. L: The set of possible labels that allows us to encode each tree in $Tc_{|W|}$ as an unique sequence of labels $L^{|W|}$.
5. $L^{|W|}$: The set of sequence of labels that allows to encode any tree $T_{|w|}$.
6. $Fc_{|w|} : T_{|w|} \to L^{|w|}$: The encoding function that allows us to map the constituent trees with their label sequence representation.

Therefore, the core of constituent parsing as sequence labeling task is predicting the function that maps every input token from a sentence into a label, defined as:

$$\Gamma_{|w|,\theta} : V_{|w|} \to L^{|w|}$$

where θ is the set of parameters to be learned during training.

Note that the encoding function $Fc_{|w|}$ should guarantee completeness and injectivity, problems that will be discussed in section 5.5. Another challenge that we must take into account for the correct constituent linearization is the encoding of *unary chains*, that will be discussed in section 14.

5.2 Encoding the Constituent Tree

For all the different encodings implemented, the labels will have a similar pattern: for every word w_i located at position i in the sentence, we will assign a 2-tuple label $l_i = (nc_i, lc_i)$ where:

1. nc_i is the encoded *number of common* ancestors between the words w_i and w_{i+1}. The value of nc_i will be dependent of the encoding algorithm used.
2. lc_i is the tag [1] of the *lowest common ancestor* node between w_i and w_{i+1}.

[1] We will use 'tag' to refer to the label of the nodes to avoid confusion with the labels that represent the linearized tree.

In order to encode constituent trees the first step that our system needs to perform is to compute the path from the root to the leaves. Comparing the path for words w_i and w_{i+1} will allow us to get the encoded number of common ancestors, nc_i, and also the lowest common ancestor lc_i. This approach was inspired by other [56] algorithms used to encode constituent trees as labels, but those algorithms were focused on binary trees while this one can encode any constituent tree. Also, it appends to the node tags their index in the path in order to be able to differentiate them during the following steps of the encoding process as shown in algorithm 1.

Algorithm 1: Given a constituent tree Tc, it returns a list p of paths from the root to the leaves

Input: Tc: The constituent tree to get the path to leaves
Output: P: Set of paths from the root to the leaves in tree Tc

```
1  Function PathR(node, path_current, list_paths, idx):
2      if node is leaf then
3          path_current ∪ [node.tag];
4          list_paths ∪ path_current
5      else
6          path_temp ← path_current;
7          path_temp ∪ [node.tag + idx];
8          foreach child in node.children do
9              PathR(child, path_temp, list_paths, idx);
10             idx ← (idx + 1);
11     return list_paths;
12 return PathR(Tc, [], [], 0);
```

Once we have the array of paths computed, we can start the comparison between them to craft the labels. To accomplish this, an algorithm will compare two consecutive paths element by element until a different one is found. When a different element between the paths is found, we create the label and add it to the labels list. One of the problems to overcome when implementing this algorithm was to deal with the last word of the sentence, w_n, because there is no w_{n+1} and to overcome it we added an empty node (ε) to the end of the constituent tree. The critical path of the encoding process is the way in which the number of commons field (nc_i) is represented in the label, and that will be explained further below for each one of the different encodings. In algorithm 2 we show the common part of the encoding function and in figure 5.1 we show a visual example of how the values of nc_i and lc_i are computed.

Algorithm 2: Encoding function for a constituent tree Tc

Input: Tc: The constituent tree to encode.
Output: $L^{|w|}$: Sequence of labels that represent the tree Tc.

```
1  Tc.children ← Tc.children ∪[ε];
2  paths ← PathToLeaves(Tc);
3  L^|w| ← [];
4  for i ← 1; i < |paths| − 1; i ← i + 1 do
5      path_a ← paths[i]; path_b ← paths[i + 1];
6      n_i ← 0;
7      c_i ← "";
8      forall a, b ∈ path_a, path_b do
9          if a ≠ b then
10             n_i ← ComputeN_i;
11             L^|w| ← L^|w| ∪ [Label(n_i, c_i)];
12         n_i ← n_i + 1;
13         c_i ← a;
14 return L^|w|;
```

Handling Unary Branches

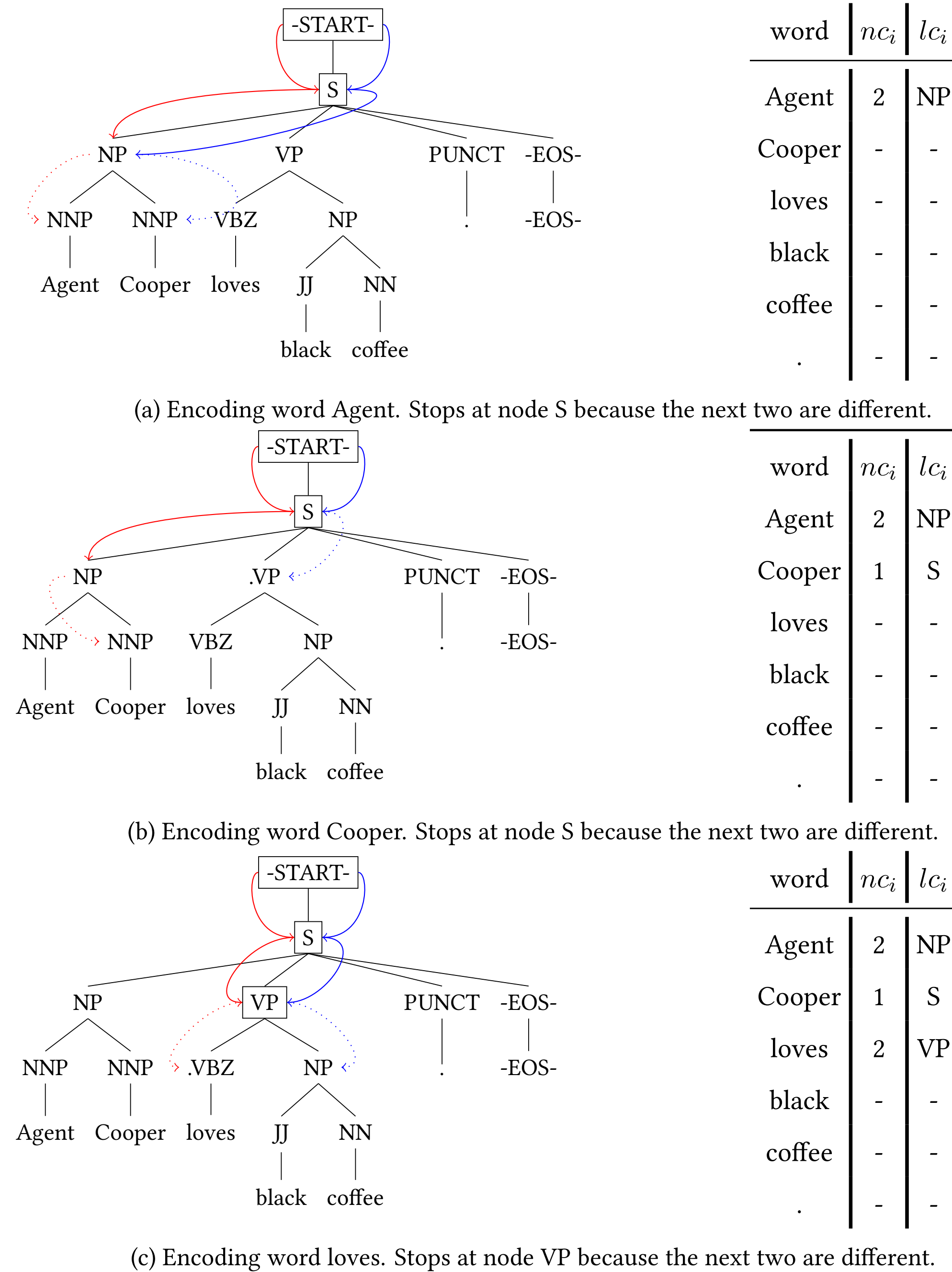
A problem we had to deal with when implementing the encoding algorithm is the loss of information when the trees present *unary branches*, as we cannot encode them correctly. We call unary branches to those nodes of a tree that have only one child. We can differentiate two kinds of unary branches:

1. **Intermediate unary branches**: Branches where the *parent* and the *child* are both non terminal nodes, i.e., the unary branch occurs in the middle of the tree (see figure 5.2 (a), A $\rightarrow$ B).

2. **Leaf unary branches**: Branches where the *child* is a pre-terminal node. In the constituent trees the leaf unary branches will end in the part-of-speech tags from the sentence (see figure 5.2 (b), C $\rightarrow$ p5).

To deal with the *intermediate* unary branches the system will follow a collapsing approach, similar to other tree linearization systems. This will convert all middle nodes joined by an unary branch into a single node with a tag composed of the joined nodes with a token separator sep among them (see figure 5.2 (b) for a collapsed tree). The *leaf* unary branches are a little bit more complex to deal with. The following two approaches to deal with them have been considered in the past [1]:

1. Use an extra function to enrich the part-of-speech tags with the collapsed unary branches needed for the decoding. This approach requires of the creation of another function



word	nc_i	lc_i
Agent	2	NP
Cooper	-	-
loves	-	-
black	-	-
coffee	-	-
.	-	-

(a) Encoding word Agent. Stops at node S because the next two are different.

word	nc_i	lc_i
Agent	2	NP
Cooper	1	S
loves	-	-
black	-	-
coffee	-	-
.	-	-

(b) Encoding word Cooper. Stops at node S because the next two are different.

word	nc_i	lc_i
Agent	2	NP
Cooper	1	S
loves	2	VP
black	-	-
coffee	-	-
.	-	-

(c) Encoding word loves. Stops at node VP because the next two are different.

Figure 5.1: Step by step linearization of the first 3 nodes from a constituent tree. The red arrow represents the path to leaves of w_i and the blue arrow represents the path to leaves of w_{i+1}. They are compared in order to obtain the values of nc_i and lc_i

$\Phi_{|w|} : V^{|w|} \rightarrow C^{|w|}$ that maps every word w_i to a collapsed unary label $uc_i \in C^{|w|}$ if it exists or $\varnothing$ otherwise. The addition of that function has the disadvantage that will make the system require two steps of sequence labeling.

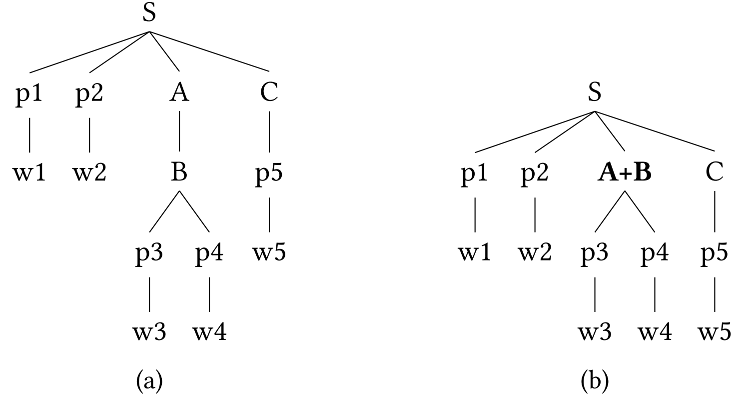


Figure 5.2: Constituent tree with intermediate unary branches and leaf unary branches (a) and its collapsed form (b).

2. Use an extended label encoding function $Fc'_{|w|} : T_{|w|} \rightarrow L^{|w|}$ where the label $l_i \in L$ will be now represented as a tuple of three elements, i.e., $l_i = (nc_i, lc_i, uc_i)$, where uc_i is the collapsed leaf unary chain. This is the approach that we follow in this work.

5.3 Encodings

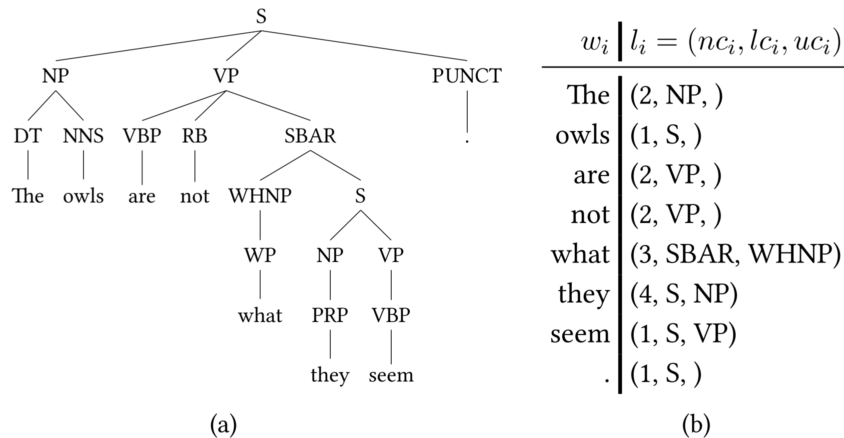
The most critical step in the process of linearization of constituent trees is the way in which we encode the value of nc_i. Specifically, the way in which this field is encoded greatly affects the sparsity of the labels obtained, which could eventually harm the performance that we obtain when training models to predict sequences of labels.

5.3.1 Naive Absolute Encoding

The first and most simple encoding function is $Fc^{abs}_{|W|} : Tc_{|w|} \rightarrow L^{|W|}$, a function that will encode nc_i directly as the number of common ancestors between the word w_i and w_{i+1}. An example for the encoding of a sample tree with this encoding is shown in figure 5.3.

5.3.2 Naive Relative Encoding

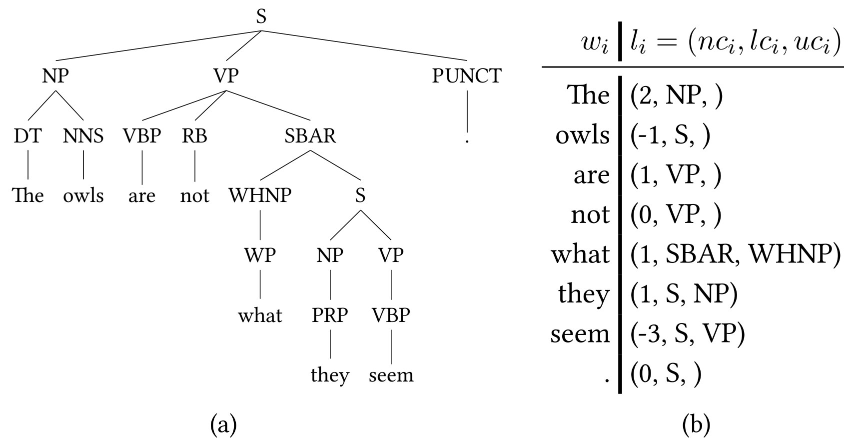
One of the main problems of the naive absolute encoding is that as the trees grow in depth so does the number of different possible labels, which complicates training an effective sequence labeling system. A proposed solution [1] to this problem is to create a different encoding function $Fc^{rel}_{|w|} : T_{|w|} \rightarrow L^{|w|}$ that encodes nc_i as the difference with respect to the number of common ancestors encoded in nc_{i-1}. This allows the set of labels to become much smaller (see Table 5.2). In order to implement the decoding function for this encoding we will apply a preprocessing step to the labels that will turn the field nc_i into absolute scale, so we can



w_i	$l_i = (nc_i, lc_i, uc_i)$
The	(2, NP,)
owls	(1, S,)
are	(2, VP,)
not	(2, VP,)
what	(3, SBAR, WHNP)
they	(4, S, NP)
seem	(1, S, VP)
.	(1, S,)

Figure 5.3: Example of a constituent tree (a) and its label representation in naive absolute encoding (b). In the column l_i, field nc_i indicates the number of common ancestors encoded in naive absolute encoding, field lc_i indicates the lowest common ancestor and field uc_i is the leaf unary chain.

apply the same decoding function that we use for the previous encoding. For a sample tree, the resulting labels in a relative scale is shown in 5.4.



w_i	$l_i = (nc_i, lc_i, uc_i)$
The	(2, NP,)
owls	(-1, S,)
are	(1, VP,)
not	(0, VP,)
what	(1, SBAR, WHNP)
they	(1, S, NP)
seem	(-3, S, VP)
.	(0, S,)

Figure 5.4: Example of a constituent tree (a) and its label representation in naive relative encoding (b). In the column lc_i, field nc_i indicates the number of common ancestors encoded in naive relative encoding, field lc_i indicates the lowest common ancestor and field uc_i is the leaf unary chain.

5.3.3 Dynamic Encoding

While the absolute scale encoding outperforms (see Table 5.1) the relative scale encoding for shallow trees (max depth smaller than 10 levels), the accuracy of the labels predicted decreases for deeper trees, where the relative scale outperforms it. The last encoding for nc_i

implemented aims to solve this problem by mixing the relative and absolute encodings [57] and is based on using a different encoding function scale depending on the number of levels that it descends or ascends into the tree. The hybrid encoding is better for encoding large constituent trees, showing the lowest label sparsity of the three encodings as shown in table 5.2.

Depth	2-4	5-7	8-10	11-13	14-16	17-19	20-22	23-25	26-27
Absolute encoding	95.84	94.32	93.25	93.04	92.06	91.99	86.55	85.51	82.11
Relative encoding	94.90	94.18	93.39	93.22	92.96	92.23	91.76	93.31	88.46

Table 5.1: F-Score from predicted labels for the test set of the Penn Treebank, in relation with the tree depth measured for the absolute scale encoding and the relative scale encoding

Encoding	Test Set	Dev Set	Train Set
Naive absolute encoding	854	690	2244
Naive relative encoding	665	548	1761
Dynamic encoding	552	473	1506
Total tree count	2416	1700	39832

Table 5.2: Label sparsity (number of distinct labels obtained) for each encoding for the English Penn Treebank of constituent Trees.

Formally, the encoding function $F^{dyn}_{|w|} : Tc_{|w|} \to L^{|w|}$ of this algorithm is defined as a conditional function using both $F^{rel}_{|w|}$ and $F^{abs}_{|w|}$. The decision of what encoding function to use depends on the value of the number of common ancestors nc_i of the node that we are encoding being inside or outside of a range defined by some threshold values. The threshold values were computed empirically for this work and are $nc_i = 3$ for the upper threshold th_{up} and $nc_i = -2$ for the lower threshold th_{lw}. The encoding function is defined as follows:

$$F^{dyn}_{|w|} \begin{cases} F^{abs}_{|w|} & \text{if } th_{lw} < nc_i < th_{up} \\ F^{rel}_{|w|} & \text{otherwise} \end{cases}$$

5.4 Decoding the Labels

To explain how to decode a sequence of labels, we assume that a well-formed output is given, and describe the measures to fix corrupted outputs (e.g., like those that a sequence labeller could produce at inference time) in section 5.5. The general decoding process consists in parsing the sequence of labels $l_i = (nc_i, lc_i, uc_i)$ from $L^{|W|}$ one at a time while generating

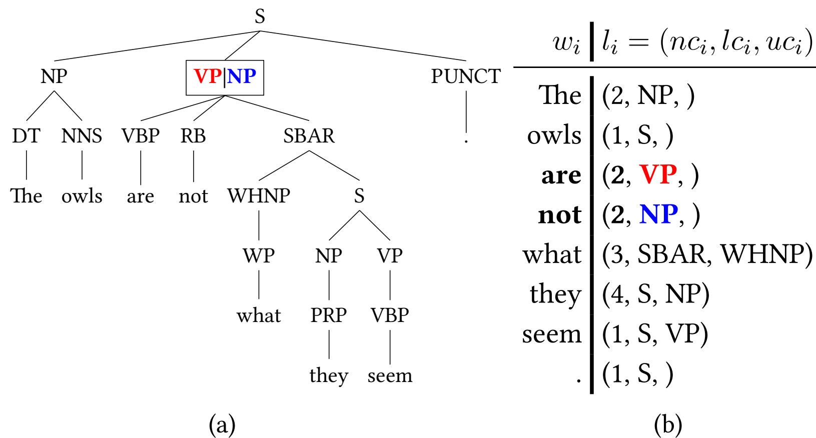
the tree from root to leaves. The number of levels to descend in each step is determined by the field nc_i of the label, and, once reached that level, we set the node tag to lc_i. After that, to obtain the tree T_c, we have to append: (i) the leaf unary chain uc_i, (ii) the part-of-speech tag p_i, and (iii) the word w_i. The critical part of the decoding process comes with how the nc_i is represented, as it is different depending of the encoding employed. A pseudocode from this process is shown in algorithm 3.

5.5 Ensuring Correctness

Until now we considered a perfect sequence labeling system that allowed us to decode grammatically correct constituent trees. However, in practical scenarios, where a model will be trained to predict linearized outputs, this will be not the case. In this context, the main errors that can happen during the decoding step can be classified into the following groups:

1. **Conflicting tag predictions**: Two labels from the decoded sequence propose different tags for the same non-terminal node in the constituent tree (this can happen when a non-terminal node has more than two children, see figure 5.5).

2. **Unexpected length prediction**: A label from the decoded sequence puts a non-terminal node in a deeper level than it was expected, potentially resulting in empty non-terminal nodes during the decoding phase (see figure 5.6).



w_i	$l_i = (nc_i, lc_i, uc_i)$
The	(2, NP,)
owls	(1, S,)
are	(2, **VP**,)
not	(2, **NP**,)
what	(3, SBAR, WHNP)
they	(4, S, NP)
seem	(1, S, VP)
.	(1, S,)

(b)

Figure 5.5: Example of a conflicting tags error while decoding a constituent tree. The conflicting nodes are NP (blue) and VP (red). This is solved by setting both separated by a '|' separation character.

In order to solve both of these problems we need to (i) modify the encoding function to deal with the conflicting tags problem and (ii) implement a post-processing function to clean

Algorithm 3: Decoding function for a given sequence of labels L .

Input: $L^{|W|}$: The labels to decode.
Input: $W^{|W|}$: The set of words from the sentence that generated the labels $L^{|W|}$.
Input: $P^{|W|}$: The set of part-of-speech tags of sentence $W^{|W|}$.
Output: T_c: The decoded constituent tree of sentence $W^{|W|}$.

1. $T_c \leftarrow \varepsilon$;
2. $nc_{old} \leftarrow 0$;
3. $lc_{old} \leftarrow$ "";
4. $lvl_{old} \leftarrow$ None;
5. **foreach** $l_i, w_i, p_i \in L^{|W|}, W^{|W|}, P^{|W|}$ **do**
6. $lvl \leftarrow T_c$;
7. $nc_i \leftarrow DecodeNc(l_i)$ `/* Get the` nc_i `field in absolute scale */`
8. ;
9. $lc_i \leftarrow DecodeLc(l_i)$ `/* Get the` lc_i `field as a list of nodes to deal with collapsed intermediate unary chains */`
10. ;
11. **for** $i \leftarrow 0; i < nc; i \leftarrow i+1$ **do**
12. **if** *(GetNumberChildren(lvl) == 0)* ***or*** *(i ≥ nc_{old})* **then**
13. lvl.children = lvl.children + ε;
14. lvl = $GetRightmostChild(lvl)$;
15. `/* Set the last common tag and insert the corresponding unary chain if needed */`
16. **foreach** $lc_i \in l_i.lc$ **do**
17. **if** $lvl.tag == \varepsilon.tag$ **then**
18. lvl.tag $\leftarrow$lc$_i$;
19. $lvl \leftarrow GetRightmostChild(lvl)$
20. `/* Add the pos tag and word tree in the corresponding level */`
21. **if** $l_i.nc \geq nc_{old}$ **then**
22. $lvl_{append} \leftarrow lvl$
23. **else**
24. $lvl_{append} \leftarrow lvl_{old}$
25. $t_p \leftarrow \varepsilon$;
26. t_p.tag $= p_i$; t_p.children $= [w_i]$;
27. **foreach** $n \in uc_i$ **do**
28. $t_{temp} \leftarrow \varepsilon$;
29. t_{temp}.tag $\leftarrow n$; t_{temp}.children $\leftarrow$t$_p$;
30. $t_p \leftarrow t_n$;
31. lvl_{append}.children $\leftarrow$lvl$_{append}$.children $+ t_p$;
32. **return** T_c;

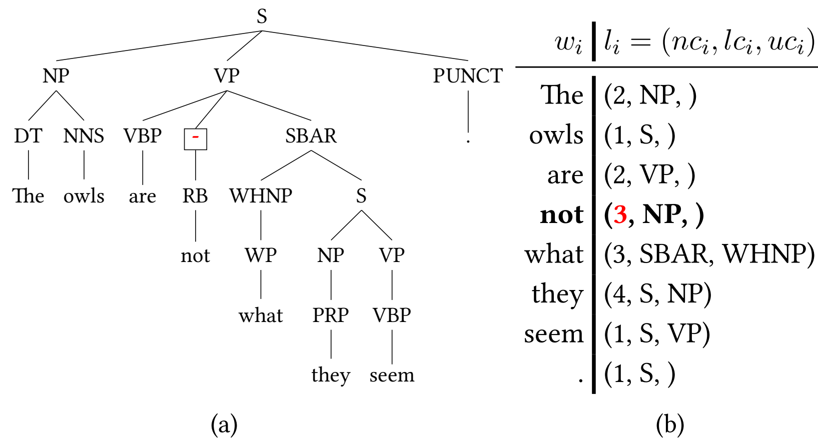


Figure 5.6: Example of a unexpected length problem while decoding a constituent tree. The ill-predicted label is of word 'not' having the field n_c a 3 instead of a 2, causing it to leave a '-' node in the tree.

the tree of the conflicts and the *null* nodes. The modification of the encoding function consists in changing the step of replacing the tag from the node with the lc_i field of the label and add all the predictions with a separating character between them ; the post-processing function will consist in a depth-first traversal of the tree that (i) replaces all conflicting node tags according to a strategy (e.g., taking the first predicted tag, taking the last one, or a voting approach that takes the most repeated one) and (ii) if it finds an empty non-terminal node, appends all its children to the parent of that node and removes it.

5.6 Chapter conclusion

In this chapter we discussed how to perform the linearization of constituent trees in order to perform constituent parsing as a sequence labeling task, and its advantages against previous ways of dealing with this problem. Three encoding techniques were presented, with the last one being a combination of the first two. Finally, we also showed how to decode a sequence of labels back to a constituent tree, and correct corrupted outputs if required (for instance, because such sequence of labels have been predicted by a trained model).

Chapter 6

Dependency Parsing as a Sequence Labeling Task

Dependency grammars are a set of grammatical theories based on the idea that sentences are formed by a sequence of tokens related by syntactic dependency relations. The task of obtaining such relations in the shape of a tree is called dependency parsing. Traditional methods of performing this task relied on: (i) transition-based systems where the task of parsing is done in linear time, but they require ad-hoc parsing algorithms and auxiliary structures, and (ii) on graph-based systems that explore the search space of all possible dependency trees and use statistical methods to discern the correct one, causing the problem to only be solvable in at best O(n^2). This chapter will discuss an alternative to those methods by casting the dependency parsing task into a sequence labeling one. To accomplish so, we will show how to encode the dependency trees into a set of labels by means of several encodings. This chapter will also explore how to decode those labels back into the corresponding dependency tree and all heuristics needed to ensure correctness.

6.1 Dependency Parsing as Sequence Labeling

In order to explain the linearization process that transforms the constituent tree $Td_{|W|}$ corresponding to a sentence $W = [w_1, w_2, ..., w_n]$, we first must define a set of labels Ld and a function $Fd_{|W|} : Td_{|W|} \rightarrow Ld_{|W|}$ that allows us to encode $Td_{|W|}$ as a unique sequence of labels $Ld^{|W|}$ in Ld. Once the function is defined we need to develop its inverse function $Fd^{-1}_{|W|}$ that allows us to restore the predicted labels back into the dependency tree shape. The different implementations for that function will be explored in sections 6.2.1, 6.2.2, 6.2.3 and 6.2.4. Once the encoding and decoding functions are implemented, the generated labels can be fed to a black-box sequence labeling system that will learn to predict a function $\Phi_{|W|,\theta}$ that generates the sequence of labels $L^{|W|}$ that represent an encoded constituent tree for a given

sentence W. This process is formally defined in definition 6.1.1.

Definition 6.1.1. Given the following elements:

1. V: A given lexicon of tokens.
2. $W = [w_1, w_2, ..., w_n]$: An input sequence of words where every $w_i \in V$.
3. $Td_{|W|}$: The set of possible dependency trees with $|W|$ nodes.
4. Ld: The set of labels that allows us to encode each tree in $Td_{|W|}$ as a unique sequence of labels in $Ld^{|W|}$.
5. $Fd_{|W|} : Td_{|W|} \rightarrow Ld^{|W|}$: The encoding function that allows us to translate the dependency trees into a unique sequence of labels.

Therefore, the core of the dependency parsing as sequence labeling task is predicting the function that maps every input token from a sentence into a label, defined as:

$$\Phi_{|W|,\theta} : V_{|W|} \rightarrow Ld^{|W|}$$

where θ is the set of parameters that the sequence labeling tool has to learn during training.

6.1.1 Encoding the Dependency Tree

For every word w_i located at the position i in the input sentence W, we will assign a 2-tuple label $ld_i = (x_i, t_i)$ where each one of the fields represents the following:

1. t_i: Representation of the type of relation between the head and the dependent for word w_i, as defined in the context of the dependency relation $r_i = (h_i, d_i, t_i)$ where h_i represents the head of the relation, d_i represents the dependant of the relation and t represents the syntactic relation (e.g. subject, or direct object) that exists among them.
2. x_i: Encoding-specific value that defines how the head-dependant relations will be encoded for each word.

A overview of the encoding process for a given dependency tree T_d can be seen in algorithm 4, where we leave the x_i field assignation to be performed by the different encodings implemented.

6.1.2 Decoding the Labels

The general decoding process that we use to rebuild a dependency tree back from the sequence of output labels can be seen in algorithm 5. As the head position was encoded in the x_i field of the label and it was an encoding-specific task, this algorithms leaves the implementation of the function $\texttt{DecodeH}_i$ to each of the specific encodings.

Algorithm 4: Encode a Dependency Tree $Td_{|W|}$ into a Sequence of Labels $Ld^{|W|}$

```
   Input: Dependency Tree Td_|W|
   Output: Set of encoded labels Ld^|W|
1  R ← Td_|W|.edges;
2  Ld^|W| ←[];
3  foreach r_i ∈ R do
4      h_i, d_i, t_i ← r_i;
5      if d_i! = 0 then
6          /* When creating the labels, the field x_i is left to be
              filled later                                        */
7          Ld^|W| ← Ld^|W| ∪ [Label(−, t_i)];
8  /* Encoding-specific segment                                  */
9  Ld^|W| ← EncodeX_i(Ld^|W|, R);
10 return Ld^|W|;
```

Algorithm 5: Algorithm to decode Dependency Trees

```
   Input: W: Sequence of words forming a sentence.
   Input: P: Sequence of part-of-speech tags corresponding to sentence W.
   Input: Ld^|W|: Sequence of labels ld_i = (x_i, t_i)
   Output: T_d|W|: Decoded dependency tree for sentence W.
1  R ←[ ]; V ←[ ];
2  i ← 0;
3  foreach ld_i ∈ Ld^|W| do
4      x_i, t_i ← ld_i;
5      R ← R ∪ (0, i, t_i);
6      V ← V ∪ i
7  /* Encoding-specific segment                                  */
8  R ← DecodeH_i(R) ;
9  Td_|W|.edges ←R;Td_|W|.nodes ←V
10 return Td_|W|
```

6.2 Encodings

The following sections will study different approaches to encode the x_i field of the label. The decision of what encoding to use will have an impact on the performance and accuracy of the final system, as it will affect the sparsity and learnability of the generated labels.

6.2.1 Naive Absolute Encoding

The first of the proposed encodings consists in setting directly the h_i component of the relationship on the x_i field of the label. The resulting labels from this encoding for a sample tree is shown in figure 6.1. The main disadvantages of this encoding are (i) the high label sparsity that it produces when encoding the dependency trees of long sentences and (ii) the relation between the fields x_i and t_i of the label $ld_i = (x_i, t_i)$ is nearly non-existent.

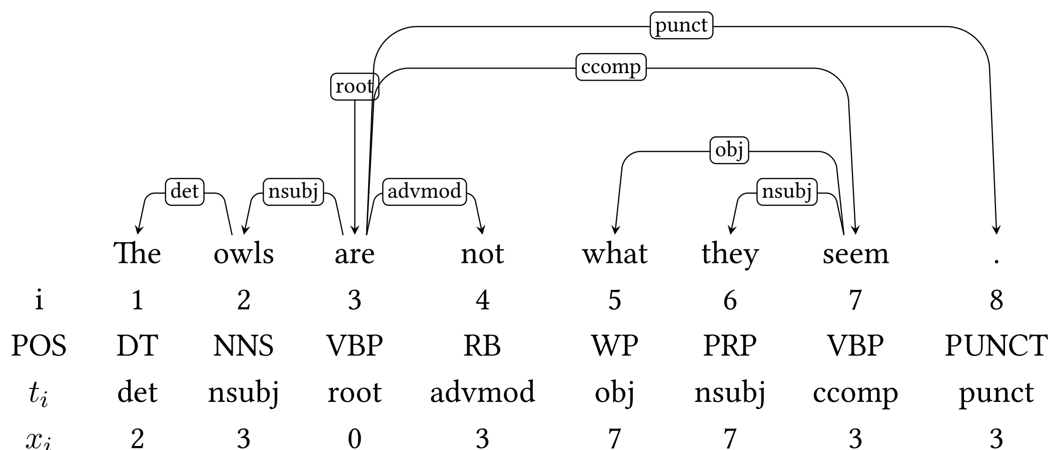


Figure 6.1: Encoding of a given dependency tree with the x_i's encoded according to the naive absolute encoding.

Encoding This encoding can be derived trivially from the CoNLL-U format to represent a dependency tree. This encoding is expressed as the function $Fd^{ABS}_{|W|}$ and simply encodes the head position h_i of the edge $r_i = (h_i, d_i, t_i)$ directly into the field x_i of the label, setting $x_i = 0$ for the root of the tree (see algorithm 6).

Algorithm 6: Encoding function for the x_i's components with the naive absolute encoding

Input: R: Sequence of dependency relations $r_i = (h_i, d_i, t_i)$ representing a dependency tree
Input: $Ld^{|W|}$: Sequence of labels $ld_i = (x_i, t_i)$ without the x_i field
Output: Ld: Sequence of labels $ld_i = (x_i, t_i)$ with the x_i field

1 **foreach** $r_i, ld_i \in R, Ld$ **do**
2 $\quad x_i, t_i \leftarrow l_i$;
3 $\quad h_i, d_i, t_i \leftarrow r_i$;
4 $\quad x_i \leftarrow h_i$;
5 **return** Ld

Decoding The h_i decoding process is very straightforward, due to the labels x_i field encoding directly the head of the dependency relationship. This process is explained in algorithm 7.

Algorithm 7: Decoding function for the h_i's components in naive absolute encoding

---

Input: $Ld^{|W|}$: Sequence of labels $ld_i = (x_i, t_i)$
Input: P: Sequence of part-of-speech tags
Input: R: Sequence of dependency relations $r_i = (h_i, d_i, t_i)$ without the h_i field
Output: R: Sequence of dependency relations $r_i = (h_i, d_i, t_i)$ with the h_i field

```
1 foreach ld_i, r_i ∈ Ld^|W|, R do
2     x_i, t_i ← ld_i;
3     h_i, d_i, t_i ← r_i;
4     h_i ← x_i;
5 return R
```

6.2.2 Naive Relative Encoding

We now propose a more advanced way to encode a dependency [58] tree into labels with lower label sparsity than the *naive absolute encoding*. The resulting labels from this encoding for a sample tree is shown in figure 6.2. The main advantage from this encoding comes from the decreased label sparsity with respect to the absolute encoding. This is illustrated in table 6.1, where we compare the different labels obtained for both encodings for the universal dependencies English$_{EWT}$ treebank.

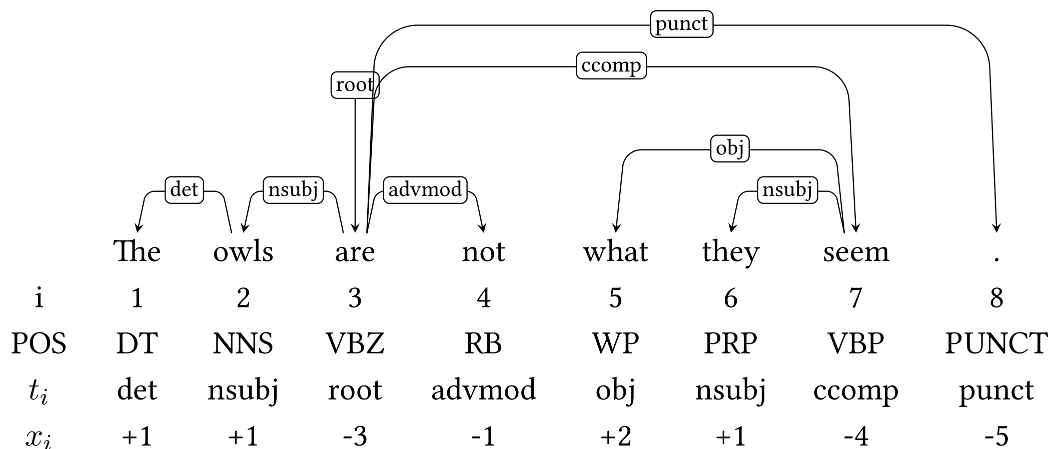


Figure 6.2: Encoding of a given dependency tree with the x_i's encoded according to the naive relative encoding.

Encoding	Test Set	Dev Set	Train Set
Total Trees	2077	2001	12543
$Fd^{ABS}_{\|W\|}$ Unique Labels	1521	1447	2701
$Fd^{REL}_{\|W\|}$ Unique Labels	743	745	1386

Table 6.1: Label sparsity for the naive absolute encoding and the naive relative encoding for the sum of all the trees in the universal dependencies English$_{EWT}$ treebank.

Encoding This approach encodes the distance between the dependent and the head. It is represented as the function $Fd^{REL}_{|w|}$, and encodes the head of any dependency edge $r_i = (h_i, d_i, t_i)$ in the x_i field of the label as $h_i - d_i$ (see algorithm 6).

Algorithm 8: Encoding function for the x_i's components in naive relative encoding

Input: R: Set of dependency relations $r_i = (h_i, d_i, t_i)$ representing a Dependency Tree
Input: $Ld^{|W|}$: Set of labels $ld_i = (x_i, t_i)$ without the x_i field
Output: $Ld^{|W|}$: Set of labels $ld_i = (x_i, t_i)$ with the x_i field

```
1 foreach r_i, ld_i ∈ R, Ld^|W| do
2     x_i, t_i ← ld_i;
3     h_i, d_i, t_i ← r_i;
4     x_i ← h_i − d_i;
5 return Ld^|W|
```

Decoding The decoding process for this encoding is shown in algorithm 9, where the value of h_i is decoded as an addition between the label field x_i and the i position.

Algorithm 9: Decoding function for the h_i's fields in naive relative encoding

Input: $Ld^{|W|}$: Sequence of labels $ld_i = (x_i, t_i)$
Input: P: Sequence of part-of-speech tags
Input: R: Sequence of dependency relations $r_i = (h_i, d_i, t_i)$ without the h_i field
Output: R: Sequence of dependency relations $r_i = (h_i, d_i, t_i)$ with the h_i field

```
1 foreach ld_i, r_i ∈ Ld^|W|, R do
2     x_i, t_i ← ld_i;
3     h_i, d_i, t_i ← r_i;
4     h_i ← x_i + d_i;
5 return R
```

6.2.3 PoS Based Encoding

This encoding attempts to solve the lack of correlation between the fields x_i and t_i of the label by encoding x_i using information extracted from the part-of-speech tags of the sentence. This algorithm, even if accurate and fast, presents the drawback of needing labeled input. This supposes a problem for low-resource languages [59] or languages where the part-of-speech tagger is not accurate. An example of this encoding for a given sentence is shown in the figure 6.3, whose details will be explained below.

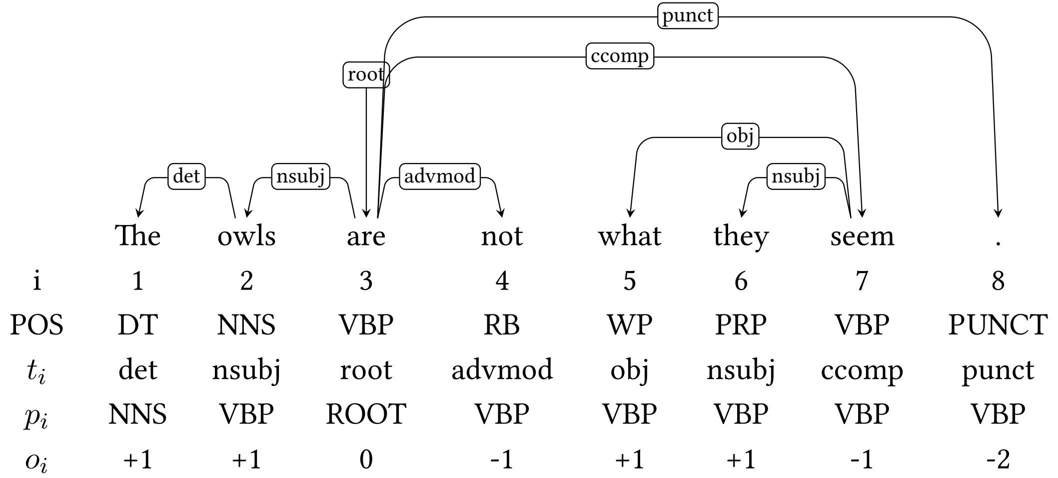


Figure 6.3: Encoding of a given dependency tree with the x_i's encoded according to the part-of-speech based encoding.

Encoding For this encoding, we define a function $Fd^{POS}_{|W|}$ where for each node from the input tree the algorithm generates a label $l = (x_i, t_i)$ where x_i is a tuple (p_i, o_i) such that:

- p_i encodes the part-of-speech tag for the head of the word w_i or "ROOT" if w_i is the root of the sentence.
- o_i encodes two things: (i) the direction to the head, meaning that is to the left if $o_i < 0$, or to the right if $o_i > 0$, and (ii) the number of POS tags with the value p_i that occur in the sentence in that direction until the desired one.

The pseudocode for this encoding is shown in algorithm 10.

Decoding The decoding process will consist in traversing the sequence of part-of-speech tags and the sequence of labels according to the x_i field of the labels. A pseudocode of the implementation of this idea is shown in Algorithm 11.

Algorithm 10: Encoding function for the x_i's components in part-of-speech encoding

Input: R: Sequence of dependency relations $r_i = (h_i, d_i, t_i)$ representing a dependency tree
Input: P: Sequence of part-of-speech tags from the input sentence
Input: $Ld^{|W|}$: Sequence of labels $l_i = (x_i, t_i)$ without the x_i field
Output: $Ld^{|W|}$: Sequence of labels $l_i = (x_i, t_i)$ with the x_i field

1 **foreach** $r_i, ld_i \in R, Ld^{|W|}$ **do**
2     $x_i, t_i \leftarrow ld_i$;
3     $h_i, d_i, t_i \leftarrow r_i$;
4     $o_i \leftarrow P[h_i]$; $p_i \leftarrow 0$;
5   **if** $o_i == ROOT$ *or* $p_i == 0$ **then**
6         $x_i \leftarrow (p_i, o_i)$; continue;
7   **if** $d_i < h_i$ **then**
8         $s \leftarrow 1$;
9   **else**
10         $s \leftarrow -1$;
11   **for** $i \leftarrow d_i; i < (h_i + s); i \leftarrow i + s$ **do**
12   **if** $o_i == P[i]$ **then**
13             $p_i \leftarrow p_i + s$;
14     $x_i \leftarrow (p_i, o_i)$;
15 **return** $Ld^{|W|}$

6.2.4 Bracketing Based Encoding

The last implemented encoding is based on the works of [10, 60, 61], where the task of parsing dependency trees is approached from the viewpoint of dependency bracketing (see figure 6.4). This allows us to encode the dependency edges as a set of brackets that indicate outgoing or incoming arcs from neighbor tokens. As opposed to the other encodings the bracket based encoding is unable to encode non-projective trees as we will show later.

Encoding For this encoding we will define a function $Fd^{BRK}_{|W|}$ that for each node of the input tree will generate labels shaped as $l_i = (x_i, t_i)$ where the value of x_i is encoded as a string formed from the set of characters $B = <, \backslash, /, >$ and defined by the regular expression (<)?((\)*|(/)*)(>)? where the presence of each character means:

1. < : w_{i-1} has an incoming arrow from the right.
2. \: w_i has an outgoing arrow towards the left. This character can appear as many times as outgoing arrows w_i has.

Algorithm 11: Decoding function for the h_i's field in part-of-speech encoding

Input: $Ld^{|W|}$: Sequence of labels $ld_i = (x_i, t_i$
Input: P: Sequence of part-of-speech tags
Input: R: Sequence of dependency relations $r_i = (h_i, d_i, t_i)$ without the h_i field
Output: R: Sequence of dependency relations $r_i = (h_i, d_i, t_i)$ with the h_i

1 **foreach** $ld_i, r_i \in Ld^{|W|}, R$ **do**
2 $\quad x_i, t_i \leftarrow Ld$;
3 $\quad p_i, o_i \leftarrow x_i$;
4 $\quad h_i, d_i, t_i \leftarrow r_i$;
5 $\quad$ **if** $o_i == 0$ *or* $p_i == ROOT$ **then**
6 $\quad\quad h_i = 0$;
7 $\quad\quad$ continue;
8 $\quad$ **if** $oi > 0$ **then**
9 $\quad\quad s \leftarrow 1; f \leftarrow |P|$;
10 $\quad$ **else**
11 $\quad\quad s = -1; f = 0$;
12 $\quad c \leftarrow o_i$;
13 $\quad$ **for** $i \leftarrow d_i; i \leftarrow f; i \leftarrow i + s$ **do**
14 $\quad\quad$ **if** $p_i == P[i]$ **then**
15 $\quad\quad\quad c \leftarrow c - s$;
16 $\quad\quad$ **if** $c == 0$ **then**
17 $\quad\quad\quad$ break;
18 $\quad h_i \leftarrow i$;
19 **return** R

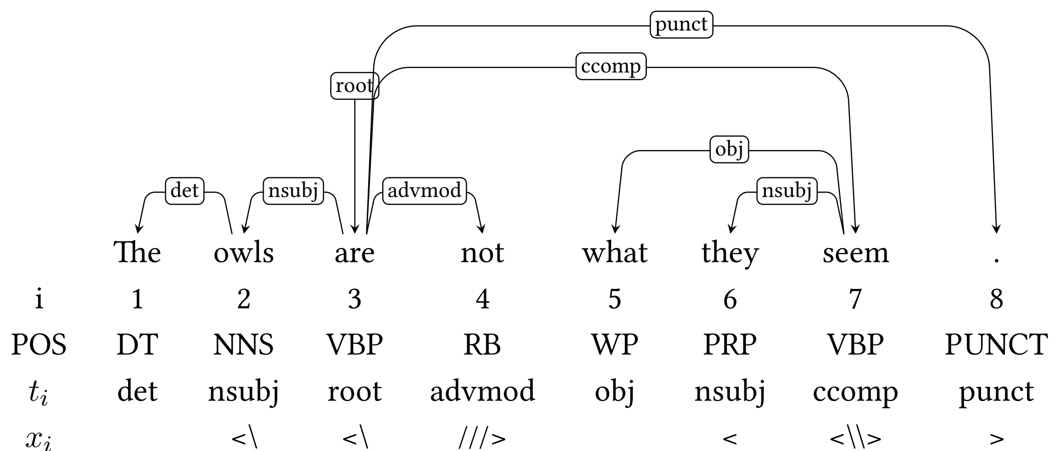


Figure 6.4: Encoding of a given dependency tree with the x_i's encoded according to bracketing-based encodings.

3. / : w_{i-1} has an outgoing arrow towards the right. This character can appear as many times as outgoing arrows w_{i-1} has.

4. > : w_i has an incoming arrow from the left.

This process can be understood as encoding the dependency arcs according to their direction as follows:

1. **Left Dependency**: for a given dependency relationship $r_{right} = (w_i, w_j, t_i)$ such that $w_i < w_j$, $Fd^{BRK}_{|W|}$ will encode in label l_{i+1} a '>' character and will encode in label l_j a '/' character (see figure 6.6, blue).

2. **Right Dependency**: for a given dependency relationship $r_{left} = (w_j, w_i, t_i)$ such that $w_i < w_j$, $Fd^{BRK}_{|W|}$ will encode in the label l_{i+1} a '<' character and will encode in the label l_j a '\' character (see figure 6.6, red).

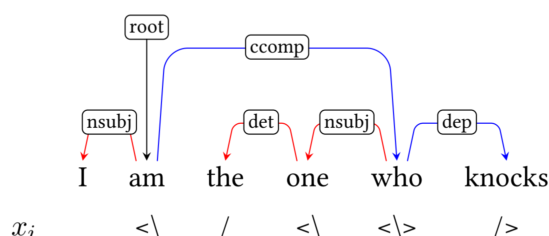


Figure 6.5

Figure 6.6: Example of a Right Dependency (blue) and a Left Dependency (red).

Note that in order to reduce the label sparsity in projective trees we encode the characters '<' and '/' in w_{i+1} instead of w_i. This allows to reduce the number of labels, because we avoid having the characters '/' and '\' in the same x_i field. The general encoding process is shown in algorithm 12.

Decoding For the decoding algorithm, we will also need create beforehand an empty tree that we will fill using the labels and two stacks as auxiliary structures. Each stack will store the dependencies in one direction, meaning we will have a stack that will deal with the dependencies from *left to right* and another one will store the dependencies from *right to left*. This stacks will fill the fields of the empty dependency tree.

Bracketing Based Encoding for 2-Planar Trees

This encoding as shown in figure 6.7 has the disadvantage of being unable to encode non-projective dependency trees. If we recall from chapter 2, we say that a non-projective tree is

Algorithm 12: Encoding function for the x_i's components in bracket based encoding

```
   Input: R: Sequence of dependency relations r_i = (h_i, d_i, t_i) representing a
          dependency tree
   Input: P: Sequence of part-of-speech tags from the input sentence
   Input: Ld^|W|: sequence of labels ld_i = (x_i, t_i) without the x_i field
   Output: Ld^|W|: sequence of labels l_i = (x_i, t_i) with the x_i field
 1 b =[ ];
 2 foreach r_i ∈R do
 3     h_i, d_i, t_i ← r_i;
 4     if h_i == 0 or d_i == 0 then
 5         continue;
 6     if d_i < h_i then
 7         b[d_i + 1]← b[d_i + 1] + "<";
 8         b[h_i]← b[h_i] + "\";
 9     else
10         b[d_i]← b[d_i] + ">";
11         b[h_i + 1]← b[h_i + 1] + "/";
12 foreach l_i, r_i ← L, R do
13     x_i, t_i ← l_i;
14     h_i, d_i, t_i ← r_i;
15     x_i ← b[d_i]
16 return Ld^|W|
```

k-planar if its dependency edges can be separated in k projective planes (that is, sets of arcs where there are no crossing edges.) It has been proven by previous research [26, 62, 63] that most non-projective dependency trees can be separated in only 2 planes.

Encoding To adapt the bracketing algorithm to encode 2-planar trees we will (i) separate the input sentence into two planes, (ii) encode the first plane using the bracketing encoding and the default set of characters $B = <, \backslash, /, >$ and (iii) encode the second plane with also the bracketing encoding but with a different set of characters $B* = < *, \backslash *, /*, > *$.

Decoding The decoding process will have to change by using 4 stacks instead of 2. Each pair of stacks will store the dependencies for each plane, and, in that stack pair, each stack will store the dependencies in each direction. Therefore for the decoding process we need:

1. $Stack^r_{p1}$: Stores the dependencies from left to right for dependencies associated with Plane 1, that is, having the characters from B.

Algorithm 13: Decoding function for the h_i's field in brackets based encoding

```
   Input: Ld^|W|: Sequence of labels ld_i = (x_i, t_i
   Input: P: Sequence of part-of-speech tags
   Input: R: Sequence of dependency relations r_i = (h_i, d_i, t_i) without the h_i field
   Output: R: Sequence of dependency relations r_i = (h_i, d_i, t_i) with the h_i
 1 Stack_l =[ ];
 2 Stack_r =[ ];
 3 foreach ld_i, r_i ∈ Ld^|W|, R do
 4     x_i, t_i ← ld_i;
 5     h_i, d_i, t_i ← r_i;
 6     foreach c ∈ x_i do
 7         switch c do
 8             case "<" do
 9                 Stack_l.push(d_i − 1);
10             case "\" do
11                 if Stack_l.pop() != None then
12                     j ← Stack_l.pop();
13                 else
14                     j ← 0
15                 h_j, d_j, t_j ← R[j];
16                 h_j ← d_i;
17             case "/" do
18                 Stack_r.push(d_i − 1);
19             case ">" do
20                 if Stack_r.pop() != None then
21                     j ← Stack_r.pop();
22                 else
23                     j ← 0
24                 j ← Stack_r.pop();
25                 h_i ← j;
26 return R
```

2. $Stack^{l}_{p1}$: Stores the dependencies from right to left for dependencies associated with Plane 1, that is, having the characters from B.

3. $Stack^{r}_{p2}$: Stores the dependencies from left to right for dependencies associated with Plane 2, that is, having the characters from $B*$.

4. $Stack^{l}_{p2}$: Stores the dependencies from right to left for dependencies associated with Plane 2, that is, having the characters from $B*$.

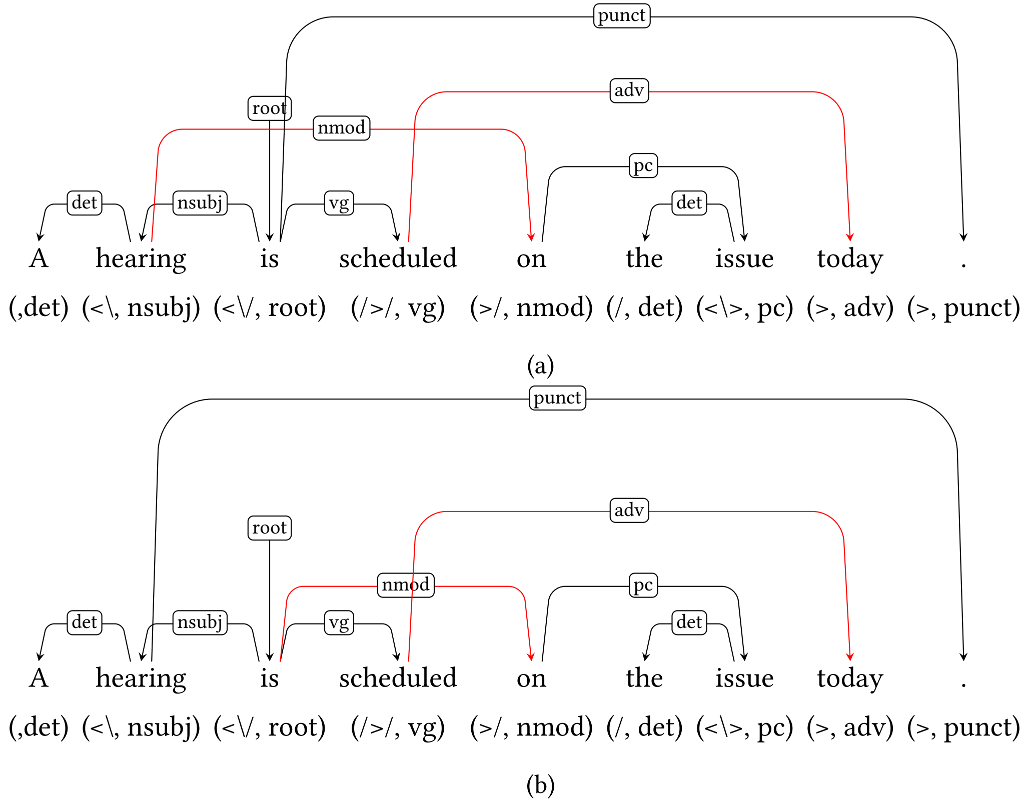
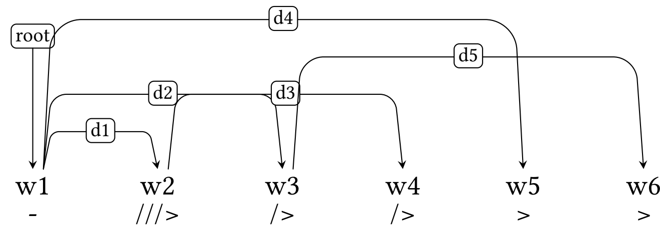


Figure 6.7: 2-planar tree encoded with bracket based encoding (a) and resulting tree from decoding those labels (b).

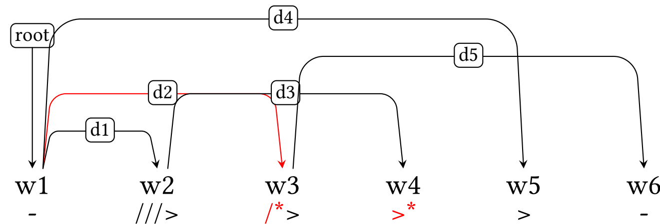
Planar Separation In order to implement this encoding we will need to first separate the dependency tree into the two non-crossing planes. For this task, we will consider two different planar-separation algorithms [64]. Due to the scarcity of crossing dependencies in the treebanks [65], both approaches deal with the planar separation by parsing the dependency tree and only setting the dependency relations to a new plane when needed, therefore reducing the number of unique output labels that the encoding produces.

Greedy planar separation The first planar separation strategy implemented is the *greedy planar separation*. For a given tree $Td_{|W|} = (N, R)$ in order to split the relationship set R into two partitions without crossing dependencies R_1 and R_2, we will traverse the set of nodes N from left to right and we will check the relationships that each node participates in. Those relationships will be assigned to R_1 by default, to R_2 if the relationship crosses any relationship already in R_1 or will be removed if it crosses relationships in both R_1 and R_2 and therefore not being able to encode all 2-planar trees. An example of greedy algorithm doing a bad assignment is shown in figure 6.8 (b), where the tree is divisible in two planes but the greedy nature of the algorithm cannot separate it.

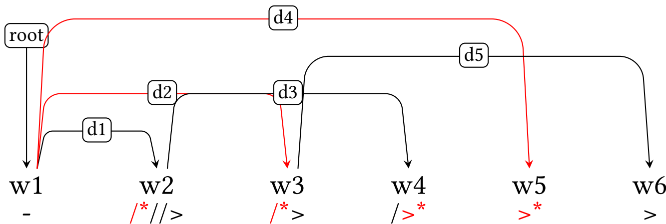
Propagation planar separation The *propagation* planar algorithm aims to solve the problem with the inner workings of the greedy algorithm by propagating the plane assignment along the tree. To do that propagation of changes we use the crossings graph, which is a special graph $G_c = (V, E)$ where the edges E indicate the set of relationships in the dependency tree T_d that cross among them, that is, that cannot be assigned to the same plane. The goal of this algorithm is to do a left to right pass through the dependencies in T_d and every time an edge is assigned to each plane, all its neighbours in the crossing graph mark that plane as forbidden, meaning they can't be assigned to it. An example of a planar separation performed by this algorithm is shown in figure 6.8 (c).



(a) Dependency tree with bracketing based encoding. There 2 are crossing edges and therefore the decoded tree will go wrong.



(b) Dependency tree with 2-planar bracketing based encoding using greedy planar separation. There is 1 crossing edge and therefore the decoded tree be wrong.



(c) Dependency tree with 2-planar bracketing based encoding using propagation-based planar separation. There are no crossing edges and therefore the decoded tree be correct.

Figure 6.8: Planar separation for bracketing encoding using different strategies.

6.3 Ensuring Correctness

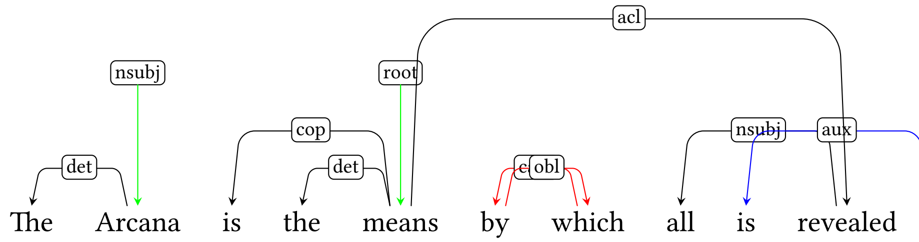
Until now, we considered that the labels used in the decoding process are predicted with a 100 percent accuracy, something that is rarely the case. In real life, the normal thing is that some of the labels are wrongly predicted. The problems that these wrongly predicted labels can cause for a decoded dependency tree can be divided in three groups:

1. **Node has the head outside the tree bounds**: One or more nodes from the decoded tree have the head outside the bounds of the sentence, that is, the relationship $r_i = (h_i, d_i, t_i)$ associated with the word w_i has value $h_i > |W|$ or $h_i < 1$.
2. **Loops inside the tree**: One or more nodes in the decoded tree have circular dependencies. This collides frontally with one of the restrictions of the tree structure, which is that they must present no cycles.
3. **One element of the sentence must be the root**: The decoded tree has no root element, meaning, no $r_i = (h_i, d_i, t_i) \in R$ has $h_i = 0$.
4. **Only one element of the sentence must be the root**: Some data sets enforce uniqueness of root in the sentences, therefore the decoded set of relations R must have only one relation r_i such that $h_i = 0$.

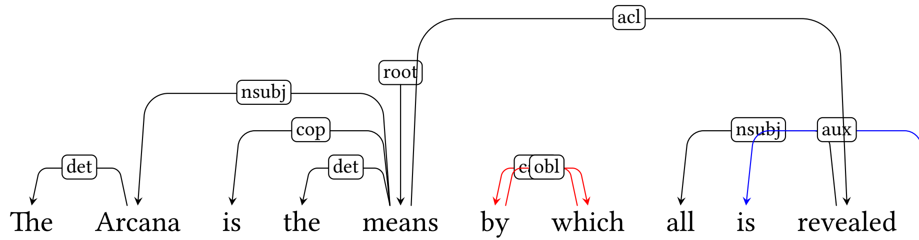
To solve this problem, the proposed solution is the usage of a post-processing function that solves all the issues. That function will deal with the problem by (i) ensuring the uniqueness of the root by selecting as true root the first token that has a relation with $h_i = 0$ and $t_i ='ROOT'$, or the first one that has $h_i = 0$ or the first token in the sentence (figure 6.9, image (a)) and (ii) will break any loop in the resulting tree by setting the h_i field in the relationship edge that causes the loop to the index of the current root of the sentence (figure 6.9, image (b)), and (iii) fix any relationship where the field h_i is out of the bounds of the sentence by setting its value to the index of the root of the sentence (figure 6.9, image (c)).

6.4 Chapter Conclusion

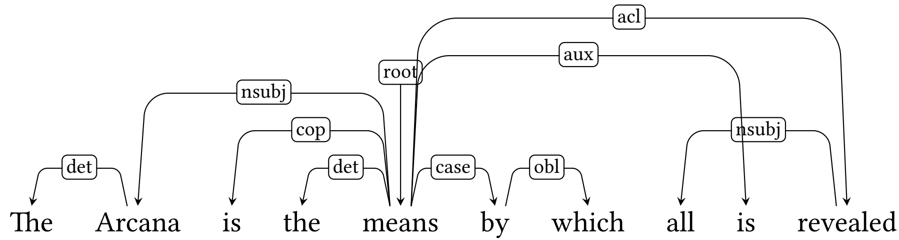
In this chapter we have seen the different strategies to cast the dependency parsing problem into a sequence labeling problem. A total of four different ways to encode a dependency tree into an unique sequence of labels were explained, from encodings based on word positions to more abstract bracketing-based encodings. All the implemented algorithms can encode a CoNLL-U treebank into labels and decode it back, resulting in a very powerful tool for training sequence labeling systems for parsing. Finally, we explained the heuristics used to



(a) Decoded dependency tree with loops (red), out of bounds heads (blue) and multiple root nodes (green)



(b) Decoded dependency tree with loops (red) and out of bounds heads (blue)



(c) Valid dependency tree. Root is unique and has no loops or out of bound heads.

Figure 6.9: Post-process of a dependency tree decoded from a bad prediction

ensure that the resulting dependency tree is correct, under the assumption that a trained parser could produce corrupted outputs.

Chapter 7

Software Details

Once the constituent and dependency linearization system has been developed and the sequence labeling systems models have been trained, we can now obtain the parse trees for given sentences. Generally, this process is illustrated in figure 7.1 and consists in (i) inputting the sentence to a sequence labeling system with a trained model that can predict our system labels, (ii) sending the labels obtained to our system in decoding mode and (iii) obtaining the decoded parse tree. This chapter will deal with the software details of the developed system and how the final product is used. We will take a look at the final system structure and the input and output formats available. Finally, we will show some execution examples from command line that can be executable.

7.1 Models

This section will focus on the different inner representations of the labels inside the system and how they are translated into plain text files. We will also explain how the different word features employed during the training of the sequence labeling system are extracted from the treebanks and how are included in the output files.

Constituent Tree Labels

Each of the constituent tree labels is a 3-tuple $l_i = (lc_i, nc_i, uc_i)$ that encodes the lowest common ancestor, the number of common ancestors and the unary chain needed to decode as shown in section 5.1. The main challenges found during the translation of the information of this tuples to a string are the following ones:

1. **Storing how the field nc_i was encoded**: When encoding a tree using the *dynamic encoding* we encode the value of the number of common ancestors both using relative scale and absolute scale. When labels are classes this can be stored as a field, but when

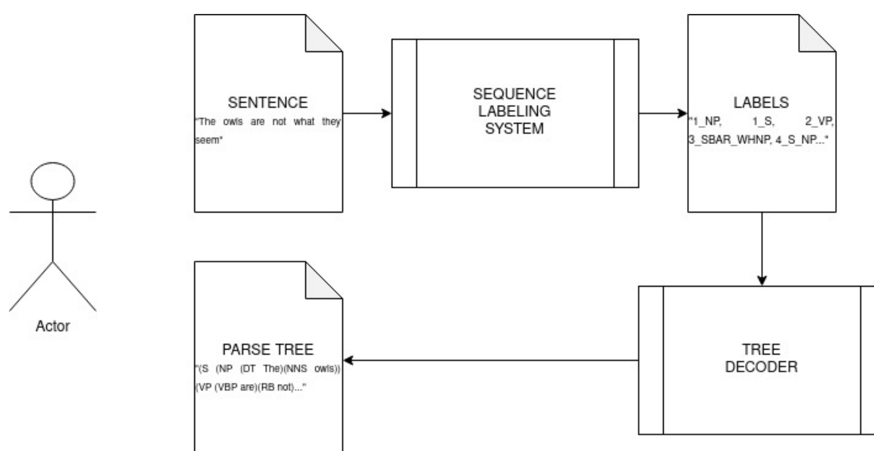


Figure 7.1: Diagram explaining the process of obtaining the parse tree for a given sentence.

writing it to a file this information has to be stored somewhere. In order to store this information in the files in this work we opted for including a **"*"** character into the nc field whenever the number was obtained employing a relative scale (see 6* in label: "6*_NP+QP_S+VP").

2. **Storing Unary Chains**: The unary chains from a tree are the structures that are formed when a node has only one child. These structures can be of any length, so the fields need to be stored separated in the labels too. The intermediate unary chains in labels can be found in the lc field from collapsing the nodes before the encoding of the tree (see NP+QP in label "6*_NP+QP_S+VP") or in the uc field where we store the leaf unary chains (see S+VP in label "6*_NP+QP_S+VP"). Collapsed nodes from unary chains combine the labels of the nodes that forms him with a separator between the labels. This separator at first was a "+" character, but some languages employ the "+" character in their part-of-speech tags, so this field was made customizable.

Dependency Tree Labels

Each of the dependency tree labels is a tuple $l_i = (t_i, x_i)$ that stores the *deprel* field from the CONLL file into the t_i field of the label and encodes the position on the head in the x_i field. The encoding of the head varies a lot depending on the encoding, with only a few of them resulting in problems:

1. Part-of-speech encoded labels: When encoding part-of-speech labels, the field x_i is separated into two sub-fields $x_i = (o_i, p_i)$ that represent the part-of-speech tag of the head and the number of ocurrences of that tag that appear in the sentence between the word and the head. To write the two fields to the output labels we employed a separator

string (e.g. ”–”, see ”-1–NP in label -1–VERB_nsubj). When reading the strings the part-of-speech encoding algorithm deals with the separation of those two fields.

2. 2-planar bracketing based encoding: When encoding non projective dependency trees using any of the 2-planar bracketing algorithms we employ a second set of bracket characters to encode the arcs from the second plane and avoid crossings. To differentiate these characters from the normal ones, we append the ”*” char to the bracket.

7.1.1 Output Labels

The output file of the system will be the files employed to train the sequence labeling system. These files will represent a sentence as a sequence of lines of columns representing: (i) the word in the sentence (column 1 in figure 7.2), (ii) the extracted features that will be employed during the sequence labeling system training (columns 2, 3 and 4 in figure 7.2), (iii) the encoded constituent or dependency label (column 5 in figure 7.2). The features are extracted from the gold tree during the encoding step and the process is different depending on the formalism:

1. **Constituent tree features**: The part-of-speech tags feature is extracted from the pre-terminal nodes and it is available both in the SPMRL and the Penn Treebank. Word specific features (such as *number*, *gender*, *verb tense*...) are only available in the SPMRL treebanks encoded in the leaf nodes (the words) surrounded by ”##” characters and separated by ”|” character (see 7.3 (a)). After their extraction, these features are removed from the ”word” column when writing the word into the output file.

2. **Dependency tree features**: The features from the dependency trees are extracted directly from the columns of the CoNLL-U files. The main feature extracted from this are the part-of-speech tags and the lemma features. In the *feats* column treebanks can include language-specific features that are also taken into consideration (see 7.3 (b))

7.2 Software System

This section will deal with the system architecture and the details involved in its development. We will show a class diagram of the system and provide an user manual with some command line examples.

7.2.1 Software Architecture

The architecture of this system can be divided into three clearly defined layers:

-BOS-	-BOS-	-BOS-	-BOS-	-BOS-
I	PRP	first	sing	_nsubj
am	VBP	first	sing	<_root
the	DT	_	_	/_det
one	CD	_	_	<_nsubj
who	WP	_	_	<\>_ccomp
knocks	VBZ	third	sing	/>_dep
-EOS-	-EOS-	-EOS-	-EOS-	-EOS-

Figure 7.2: Sample of an output file representing a linearized dependency tree from the sentence ”I am the one who knocks” with bracket encoding. The columns represent: (i) word, (ii) part-of-speech feature, (iii) person, (iv) number and (v) label.

```
(S
    (NP
        (NNP ##lem=PRAGUE|cpos=N|n=s|s=p##PRAGUE)
        (NC##lem=correspondance|cpos=N|g=f|n=s|s=##correspondance)
    )
)
```

(a)

w_i	lemma	UPOS	CPOS	feats	head	deprel	dep	misc
two	two	NUM	CD	NumType=Card	7	nummod	7:nummod	_

(b)

Figure 7.3: SPMRL French bracketed constituent tree with features (indicated in red color) between ## characters and separated by ”|” character (a) and sample of a CoNLL-U file from English$_{EWT}$ with additional features in 'feats' field (b)

1. **User-Interaction Layer**: Deals with the interaction with the user. Currently the interaction is done through command line.

2. **Input-Output Layer**: Deals with the input and the output of trees and labels. Translates the different input files into the system objects and deals with writing the label objects into files.

3. **Encoding Layer**: Deals with the linearization and de-linearization of trees and decoding of labels for both formalisms. This layer also includes the post-processing module for ensuring the correctness of the decoded trees.

This code structure decision comes after the need to keep the system as loosely coupled and extensible as possible to ease future developments (i.e. including new encodings or adding new post-processing algorithms). Also, keeping the user interaction layer separated from the encoding layer is useful if the project decides to implement any new interface (i.e. a web interface or interaction with other programs). The layered architecture greatly fits this system, thanks to how the requests flow across each layer. These layers are indicated in the class diagram of figure 7.4.

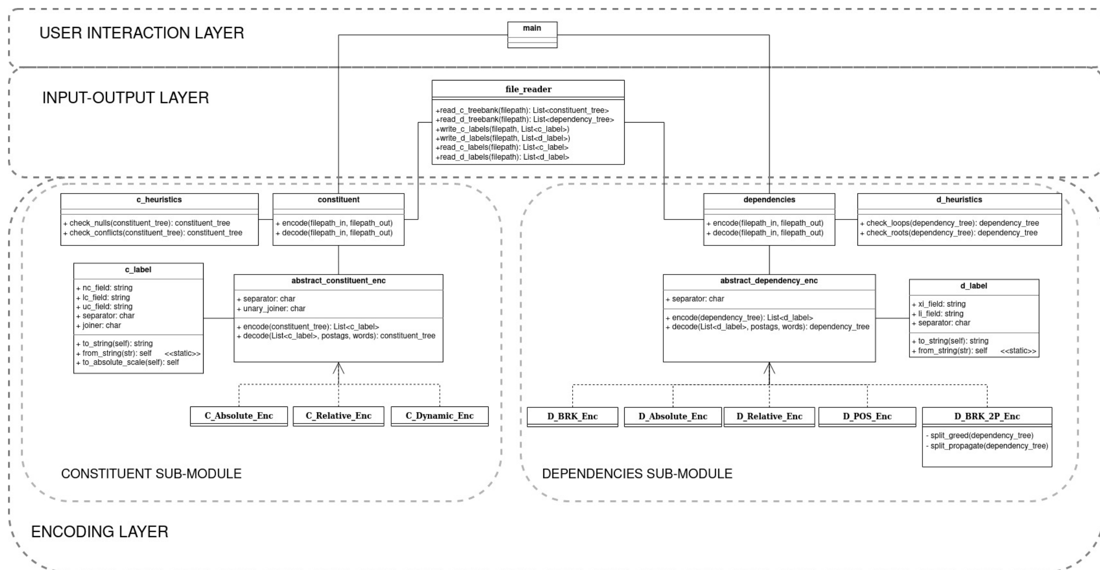


Figure 7.4: Class diagram for the constituent and dependency linearization system. In the diagram the different layers and sub-modules that were taken into consideration when developing the system are remarked.

7.2.2 User interaction and usage

The user interaction with this system will be through command line interface, where the parameters taken as input are the following:

1. **Formalism**: Positional argument that indicates the formalism that we want to use; currently the system supports dependency and constituent formalisms. The allowed values for this field are CONST for constituent trees and DEPS for dependency trees.

2. **Operation**: Positional argument that indicates the operation that the system must do; can be ENC for encoding or DEC for decoding.

3. **Encoding**: Positional argument that indicates the type of encoding used to linearize or de-linearize the given tree; the encodings are the ones mentioned in sections 5.1

(ABS for naive absolute, REL for naive relative and DYN for dynamic) and 6.1 (ABS for naive absolute, REL for naive relative, POS for part-of-speech encoding, BRK for bracket encoding and BRK_2P for bracket encoding with 2-planar separation).

4. **Input file**: Positional argument that indicates the path of the bracketed tree file or CoNLL-U file to encode or the labels file to decode.

5. **Output file**: Positional argument that indicates the path of the decoded or encoded file resulting from the operation indicated.

6. **Separator**: Character or string used to separate the different fields of a label. Indicated by the `--sep` flag. The default separator character is _.

7. **Joiner**: Character or string used to join the different labels that form a unary chain in the constituent formalism. Indicated by the `--joiner` flag. The default joiner character is +.

8. **Displacement**: Flag that if present will use a character displacement for the dependency bracket based encodings. The default option is to use displacement. The flag is `--disp`.

9. **Planar algorithm**: Argument indicating the type of planar separation algorithm employed for separating the nodes in a 2-planar bracket based encoding. The algorithms are the ones defined at 26. Indicated by the flag `--planar`. The allowed values for this parameter will be GREED or PROPAGATE and the default option is GREED.

10. **Single root**: Flag that if present will post-process a decoded dependency tree in order to ensure uniqueness of the root. This was added due to non Universal Dependencies treebanks allowing multiple roots. The flag is `--sroot`.

11. **Root Search**: Field that indicates the method of root selection used by the post-processing function in dependency trees (if the root has to be unique) whenever more than one root is found. The root selection can be done by searching for nodes where the head is zero or nodes where the dependency relation is 'root'. Indicated by the `--rsearch` flag. The different options for this field are `strat_gethead` (root will be the first node found having a dependency relation with head set as 0) or `strat_getrel` (the root will be the first node found having a dependency relation with the relation type set as 'root'). The default option is `strat_gethead`.

12. **Conflict**: Field that indicates the conflict resolution strategy to apply during the decoding of constituent trees when two or more different tags are predicted for a given node of the tree. Indicated by the `--conflict` flag. The different values that this

parameter can take are `strat_first` (take the first predicted tag), `strat_max` (take the most repeated tag and, if tied, take the first one) or `strat_last` (take the last predicted tag). The default option is `strat_max`.

13. **Allow nulls**: Flag such that, if present, the system will not apply the post-process step when decoding constituent trees to remove null nodes. The flag is `--nulls`.

14. **Part-of-speech tags**: Flag such that, if present, the system will predict the part-of-speech tags for the sentence using the Stanza tagger. The flag is `--postags`.

15. **Language**: Field that indicates the language (as indicated from the stanza available models language code) that we want to use for the part-of-speech tags prediction. If the part-of-speech tags flag is missing, this field will be ignored. If this parameter is not present and the part-of-speech tags is, the default language will be English. The flag for this field is `--lang`.

16. **Time**: Flag such that, if present, will the system will output the encoding/decoding time, the number of trees processed per second and the number of labels processed per second. The flag is `--time`.

17. **Features**: Field that indicates the features to extract during the encoding process of the constituent or dependency tree. The features will be outputted to the labels file as a column with the desired value if the feature exists or a '_' if it does not. The flag is `--feats`.

Example of execution

For the constituent formalism command line example of usage we will (i) encode (and display the run time for) the constituent trees file `test_c.trees` with dynamic encoding using `[_]` as the separator character, `[+]` as the unary chain joiner and with 'pos tags' feature extracted into `test_c.labels`; (ii) decode (and display the run time for) the `test_c.labels` file back into a constituent trees file `test_c_decode.trees` using the 'take first predicted tag' as conflict resolution strategy, allowing null nodes in the decoded tree and predicting the part-of-speech tags for English language.

1. `$python main.py CONST ENC DYN test_c.trees test_c.labels --sep [_] --joiner [+] --feats pos_tags --time`

2. `$python main.py CONST DEC DYN test_c.labels test_c_decode.trees --sep [_] --joiner [+] --conflict strat_first --nulls --postags --language en --time`

For the dependency formalism command line example we will (i) encode (and display the run time for) the dependency CoNLL-U file `test_d.conllu` with 2-planar bracketing encoding with displacement using propagation algorithm and with 'pos tags' feature extracted into `test_d.labels` using `[_]` as a separator character; (ii) decode (and display the run time for) the `test_d.labels` file back into the dependency tree file `test_d_dec.conllu` using the 'take head' strategy for discerning the correct root, forcing the decoded trees to have a single root and predicting the part-of-speech tags for French language.

1. `$python main.py DEPS ENC BRK_2P test_d.conllu test_d.labels --planar PROPAGATE --disp --sep [_] --feats pos_tags --time`

2. `$python main.py DEPS DEC BRK_2P test_d.labels test_d_dec.conllu --disp --rsearch strat_gethead --rsingle --postags --language fr --time`

7.3 Training of Sequence Labeling Systems

In order to train a black-box sequence labeling system for predicting the labels of the linearized trees, we used the *train*, *dev* and *test* splits [66] already given in the treebanks that we use in this work. More particularly, to train the sequence labeling models we rely on two freely available software systems that are widely used by the community: NCRF++ and MACHAMP, which we proceed to describe below.

NCRF++

NCRF++ [54] is a toolkit for neural sequence labeling built over PyTorch [1] using recurrent networks. This tool allows for customizing the architecture of the sequence labeling system, and, for this work, we follow this particular configuration (see also chapter 4 for a review of further details about how this type of architecture works):

1. **Token Input Layer** (Char Seq Layer): Uses a character-level BiLSTM with randomly initialized word embeddings, part-of-speech embeddings and feature-specific embeddings.

2. **Encoding Layer** (Word Seq Layer): Built using 2 stacked BiLTSTMs.

3. **Decoding Layer** (Inference Layer): This layer uses a *softmax* function to determine the output.

[1] https://pytorch.org/

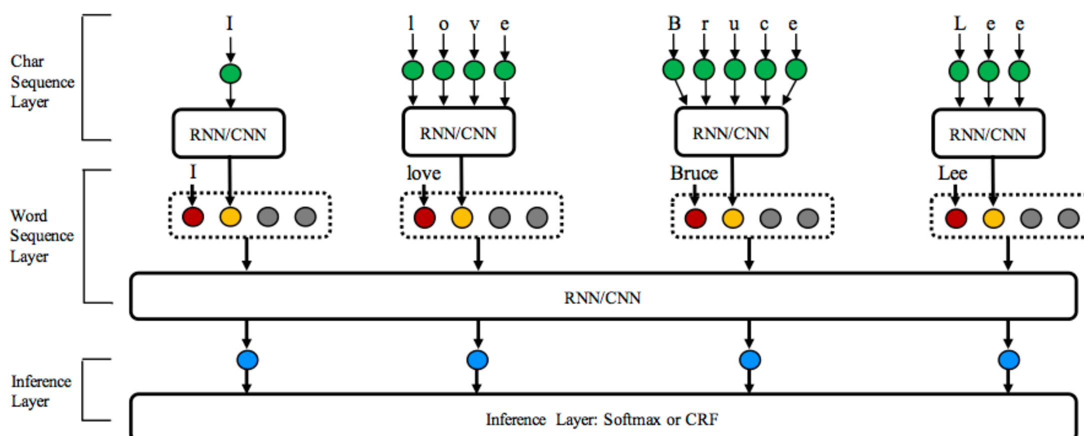


Figure 7.5: Architecture of the NCRF++ sequence labeling toolkit extracted from https://github.com/jiesutd/NCRFpp

For the specific hyper-parameters for this network, we rely on existing configuration used on previous similar works [67].

MACHAMP

MACHAMP [16] is a multi-purpose toolkit for natural language processing tasks, that focuses on providing a black box framework for multi-task learning, based on AllenNLP [68]. This toolkit offers the possibility to train simultaneously different types of tasks, ranging from text classification to text generation, and includes sequence labeling too. More in detail, we here consider MACHAMP for two reasons. First, we will exploit the multi-task learning capabilities to decompose the output label space of the syntactic tasks that we introduced in previous chapters. Secondly, we will exploit the capabilities of adding deep-contextualized word embeddings using language models, such as BERT [14]. All tasks evaluated (whether we run single task or multi task experiments) here are sequence labeling tasks, and as in NCRFpp, we simply use a softmax layer to map contextualized vectors to output labels. The hyper-parameters for this experiment are the ones already given in MACHAMP, and obtained by fine tuning the GLUE [69] data sets and the English$_{EWT}$ [70].

Chapter 8

Experimental Results

In this chapter we will discuss how the experiments to test the developed system were performed. We will see what datasets were chosen and why, how the training of the different sequence labeling systems was performed and the different tools and metrics employed in order to evaluate the results. Finally, we will show the baseline resulting from those experiments and draw conclusions from them.

8.1 Experiments Setup

In this section we will explain the datasets employed in the experiments and the testing tools to score the predicted outputs.

8.1.1 Datasets

The experiments for constituent treebanks are done on the Penn Treebank for the English language, and the SPMRL treebanks, which contains treebanks for French, Basque, German, Hebrew, Hungarian, Korean, Polish and Swedish languages. For each of these treebanks we will extract all the part-of-speech tags and in the case of SPMRL we will also extract all available word features and use them in the training in order to get the best results as possible. As some treebanks have 25+ different language features they will not be listed here, but some of them are *gender*, *lemma*, *number*, *person*, *tense*, *degree*, or *mood*.

For the dependency formalism, the treebanks come from the Universal Dependency collection, and for a homogeneous comparison, we run our experiments on the same set of languages. For some of these languages, UD has more than one available treebank. When that is the case, we choose the biggest ones. The selected treebanks are English$_{EWT}$, French$_{GSD}$, German$_{GSD}$, Hebrew$_{IAHLTWIKI}$, Hungarian$_{SZEGED}$, Korean$_{GSD}$, Polish$_{PDB}$ and Swedish$_{LINES}$. The information extracted from these treebanks that will be used during training of the sequence labeling systems will be the universal part-of-speech tags, the language-specific part-

of-speech tags and the different universal features [1] available in the treebanks. Same as with the constituent treebanks, for some languages there are more than 25 features in the treebank, so they wont be listed here.

8.1.2 Evaluation Methods

In order to compute the system's speed the metric employed will be the number of processed sentences per time unit. This metric is represented as *s/t* and will be computed as the number of parse trees processed each second. To evaluate how well the predicted models perform we will employ the F-Score metric for the constituent parsing module and LAS-F1 for the dependency parsing module.

Dependency Evaluation To evaluate the performance of the system with dependency trees we will employ the universal dependencies CoNLL-U evaluation tool. That tool will take the gold dependency trees file and the predicted dependency trees file and will compare them. The tool outputs both the labeled attachment score (LAS) and the unlabeled attachment score (UAS). As the LAS score provides more accurate information about the system performance, when discussing the results we will employ that metric.

Constituent Evaluation When evaluating the constituent trees we will employ the EVAL-B tool. Given a gold constituent tree file and a predicted constituent tree file, this tool computes the precision, recall and f-score. In order to compute those scores the evaluation tool will first transform the gold trees into standard gold trees [71], to do so it will apply a pre-processing step where (i) unknown words and additional labels will be removed, (ii) constituent trees that ended up empty from the previous step will be removed, and (iii) a list of constituent equivalences will be applied (e.g. VB = VERB). In order to define the parameters for the pre-processing step we will employ the specific parameters file for each treebank, this being *COLLINS.prm* for the Penn treebank and *SPMRL.prm* for the SPMRL treebanks.

8.2 Experimental Results

In this section we will discuss the baselines obtained for both constituent and dependency parsing formalisms. We will provide results proving that the developed systems provide good enough predictions comparing the different encodings implemented. In order to obtain those metrics the decoded files will be the test files available in the selected datasets. We will use gold word features and part-of-speech tags for the NCRF++ experiments. For MACHAMP experiments we will use (i) multilingual BERT embeddings and (ii) features as learned by the

[1] https://universaldependencies.org/u/feat/index.html

multi task learning capabilities that the tool offers. We will also provide metrics showing the speed of the sequence labeling system predictions. We will only compute speeds for the NCRF++ tool, because the MACHAMP experiments have to deal with the bottle-neck caused by the usage of language model embeddings.

8.2.1 Constituent parsing experiments

Table 8.1 shows the baselines of this work for the SPMRL and PTB datasets on constituent parsing. The metrics shown on that table are the results of computing the F-Score between the decoded trees of the .test treebank of the different languages and the gold .test file. We performed experiments using labels generated using *naive absolute*, *naive relative* and *dynamic* encoding represented with $Fc^{ABS}_{|W|}$, $Fc^{REL}_{|W|}$ and $Fc^{DYN}_{|W|}$ respectively. The heuristics employed when decoding predicted constituent trees are *Null Removal* and *Most voted conflict resolution.* In the table we show the comparison of the results of those encodings for each one of the tested sequence labeling systems. In order to provide some baselines for the decoding speed of this system, table 8.2 shows the number of decoded sentences per second with the NCRF++ tool.

Tool	Encoding	EN	EU	FR	DE	HE	HU	KO	PL	SV
NCRF++	$Fc^{ABS}_{\|w\|}$	80.21	77.27	**88.91**	88.21	87.05	92.41	87.18	93.28	71.82
	$Fc^{REL}_{\|w\|}$	87.21	82.39	90.00	88.47	88.10	91.42	86.30	93.49	73.56
	$Fc^{DYN}_{\|w\|}$	89.91	**84.85**	90.67	89.69	**90.49**	91.98	87.24	94.69	77.53
Machamp^{BERT}	$Fc^{ABS}_{\|w\|}$	92.11	78.16	90.80	89.61	87.92	**93.57**	**88.14**	95.05	76.81
	$Fc^{REL}_{\|w\|}$	92.23	80.38	90.04	88.86	88.65	93.23	87.21	94.79	79.27
	$Fc^{DYN}_{\|w\|}$	**93.35**	80.93	90.38	**89.99**	88.92	93.47	87.60	**95.55**	**80.95**
Machamp$^{BERT}_{MTL}$	$Fc^{ABS}_{\|w\|}$	92.87	75.23	89.00	87.12	87.36	91.61	85.68	93.40	74.45
	$Fc^{REL}_{\|w\|}$	92.51	78.25	87.10	88.45	88.39	90.63	85.03	92.86	75.95
	$Fc^{DYN}_{\|w\|}$	93.09	79.15	87.36	88.66	88.63	91.29	85.58	94.47	79.49

Table 8.1: F-Score (higher is better) for constituent parsing on the test sets of the English$_{PTB}$, Basque$_{SPMRL}$, French$_{SPMRL}$, German$_{SPMRL}$, Hebrew$_{SPMRL}$, Hungarian$_{SPMRL}$, Korean$_{SPMRL}$, Polish$_{SPMRL}$, Swedish$_{SPMRL}$ treebanks.

Encoding	EN	EU	FR	DE	HE	HU	KO	PL	SV
$Fc_{\|w\|}^{ABS}$	132.37	256.74	160.30	196.20	152.42	163.12	358.12	392.66	190.12
$Fc_{\|w\|}^{REL}$	**195.89**	**285.39**	**170.06**	**203.45**	**164.22**	**220.23**	**369.12**	**433.12**	**255.31**
$Fc_{\|w\|}^{DYN}$	189.76	245.53	158.12	182.44	142.13	152.14	319.81	380.31	182.27

Table 8.2: Speed measured in sentences per second (higher is better) using the NCRF++ tool for the test sets of the English$_{PTB}$, Basque$_{SPMRL}$, French$_{SPMRL}$, German$_{SPMRL}$, Hebrew$_{SPMRL}$, Hungarian$_{SPMRL}$, Korean$_{SPMRL}$, Polish$_{SPMRL}$, Swedish$_{SPMRL}$ treebanks., using a i5-1155G7 CPU.

Penn Treebank Comparison

In addition to showing our baselines for the multiple languages tested, in table 8.3 we will compare the baseline obtained from this work to previous parsers. We show the speed comparison on the PTB, where the bracketing F-score also shows that sequence labeling parsers are competitive with respect to other parsers.

Model	Sents (CPU)	F-Score
Collins 1999 [72]	3.5	88.2
Sagae and Lavie 2006 [73]	2.2	87.9
Petrov and Klein 2007 [74]	6.2	90.1
Zhu et al 2013 [75]	101	89.9
Vinyals et al 2015 [76]	120	88.3
Gomez and Vilares 2018 [1]	126	90.7
Our baseline: $Fc_{\|w\|}^{ABS}$	132	80.21
Our Baseline: $Fc_{\|w\|}^{REL}$	195	87.21
Our Baseline: $Fc_{\|w\|}^{DYN}$	189	89.91

Table 8.3: Comparison of the metrics for constituent parsing for the test set from the Penn Treebank. The speeds are reported by authors running on their own hardware[1].

Results analysis

From the experimental results we can deduce that, for most languages, the best encoding for constituent parsing as sequence labeling is the dynamic encoding. That encoding ranks higher

in almost all tested treebanks, with the exception of BASQUE$_{SPMRL}$, HUNGARIAN$_{SPMRL}$ and KOREAN$_{SPMRL}$. For those languages the absolute encoding was the best-performing one, which makes us think that the difference in performance could be caused by the fixed values of thresholds used in the dynamic encoding. As different languages usually have different constituent tree topology (i.e. some languages can have deeper or shallower constituent trees on average), more experiments could be performed for those languages changing the threshold values and see if they perform better. We can also see that, as expected, the inclusion of Multilingual BERT embeddings provided a increase in the F-Score metrics of most decoded treebanks, with some of them such ENGLISH$_{PTB}$ gaining up to 4 points. Not so good were the experiments with Multilingual BERT and multi task learning capabilities, making most experiments rank lower than only with the BERT embeddings. Particularly, SPMRL treebanks presented losses even when compared with the NCRF++ experiments. This could be caused by the inclusion of all available features into the training of the system instead of taking a curated subset, but this would need more experiments to be proven true. In terms of the speed with which the NCRF++ tool predicted our labels, we can see that the faster encoding is the naive relative one, but we can't infer any reason for this being the case. However, we can make the remark that the differences between the speed metrics of the different tested treebanks are happening due to languages having longer or shorter sentences. This could be proven by performing the experiments in different hardware, but this falls outside the scope of this work.

8.2.2 Dependency parsing experiments

Table 8.4 shows the baselines of this work for the Universal Dependencies datasets in dependency parsing. The metrics shown on that table are the labeled attachment score computed between the decoded dependency trees from the test files and the gold test files. We performed the experiments using the labels generated with *naive absolute*, *naive relative*, *part-of-speech based* and *bracketing based* encodings, represented by $Fd^{ABS}_{|W|}$, $Fd^{REL}_{|W|}$, $Fd^{POS}_{|W|}$ and $Fd^{BRK}_{|W|}$ respectively. For the part-of-speech based encoding we also performed test using (i) the 2-planar encoding with the two different planar separation algorithms, represented by $Fd^{BRK-2PG}_{|W|}$ and $Fd^{BRK-2PP}_{|W|}$ for greed separation and propagation separation respectively and (ii) using the displacement modification for the three algorithm variations, represented by $Fd^{BRKd}_{|W|}$, $Fd^{BRKd-2PG}_{|W|}$ and $Fd^{BRKd-2PP}_{|W|}$. For these experiments, the heuristics employed when decoding the predicted labels back into dependency trees are *loop removal*, *hang from root* out of bounds dependencies and *take first node* as default root. In order to provide some baselines for the decoding speed of this system, table 8.5 shows the number of decoded sentences per second with the NCRF++ tool.

Tool	Encoding	EN	EU	FR	DE	HE	HU	KO	PL	SV
NCRFpp	$Fd_{\|W\|}^{ABS}$	79.09	71.99	74.94	79.88	76.98	40.76	71.69	82.25	70.27
	$Fd_{\|W\|}^{REL}$	84.27	77.31	80.72	80.40	82.40	69.02	76.92	87.47	79.80
	$Fd_{\|W\|}^{POS}$	80.13	73.04	78.23	76.24	80.23	62.21	72.23	85.12	76.17
	$Fd_{\|W\|}^{BRK}$	84.12	72.04	76.11	77.67	83.16	68.17	74.23	86.24	79.09
	$Fd_{\|W\|}^{BRK-2PG}$	83.11	72.31	74.11	77.86	83.19	67.22	74.11	86.41	79.01
	$Fd_{\|W\|}^{BRK-2PP}$	82.12	72.04	74.89	77.68	83.34	67.82	74.22	86.32	79.18
	$Fd_{\|W\|}^{BRKd}$	84.02	72.49	76.85	78.29	84.02	67.94	74.63	86.78	79.82
	$Fd_{\|W\|}^{BRKd-2PG}$	84.47	72.36	75.70	78.23	84.49	69.55	74.45	86.92	80.13
	$Fd_{\|W\|}^{BRKd-2PP}$	84.12	72.38	75.86	78.89	84.40	69.06	73.89	86.85	80.18
Machamp^{BERT}	$Fd_{\|W\|}^{ABS}$	83.53	87.82	84.53	95.21	66.70	35.82	69.99	97.14	66.24
	$Fd_{\|W\|}^{REL}$	85.92	88.90	85.92	**97.53**	82.68	62.23	72.95	**98.55**	78.26
	$Fd_{\|W\|}^{POS}$	86.42	87.56	83.42	89.96	83.51	**75.68**	51.33	92.11	78.69
	$Fd_{\|W\|}^{BRK}$	88.30	90.22	88.42	95.02	88.47	74.26	80.49	97.11	83.42
	$Fd_{\|W\|}^{BRK-2PG}$	88.18	90.74	88.30	95.12	88.13	74.23	80.38	97.13	83.54
	$Fd_{\|W\|}^{BRK-2PP}$	88.07	90.49	88.18	95.07	88.24	73.51	**80.87**	97.08	83.28
	$Fd_{\|W\|}^{BRKd}$	88.20	90.86	88.07	96.27	**89.17**	74.42	77.07	97.58	83.83
	$Fd_{\|W\|}^{BRKd-2PG}$	88.34	**91.07**	88.20	97.11	89.03	73.91	77.29	97.56	**83.88**
	$Fd_{\|W\|}^{BRKd-2PP}$	**88.34**	90.58	**88.34**	97.25	89.06	73.89	76.57	97.88	83.74
Machamp$_{MTL}^{BERT}$	$Fd_{\|W\|}^{ABS}$	78.93	81.82	73.93	93.63	61.93	20.74	58.83	95.98	51.37
	$Fd_{\|W\|}^{REL}$	82.92	86.23	82.14	93.48	77.20	51.45	71.52	96.12	71.61
	$Fd_{\|W\|}^{POS}$	85.73	86.93	81.92	88.16	74.12	71.82	51.12	92.30	77.71
	$Fd_{\|W\|}^{BRK}$	84.96	89.15	84.84	94.86	83.71	60.13	75.66	96.01	76.09
	$Fd_{\|W\|}^{BRK-2PG}$	85.16	89.01	85.05	94.77	83.93	68.88	75.70	95.82	76.62
	$Fd_{\|W\|}^{BRK-2PP}$	85.12	89.12	84.04	94.13	84.13	68.84	76.05	95.91	76.33
	$Fd_{\|W\|}^{BRKd}$	86.41	89.96	86.31	95.02	85.03	58.88	76.19	96.13	78.11
	$Fd_{\|W\|}^{BRKd-2PG}$	86.12	89.12	86.07	95.11	84.87	67.23	76.06	96.02	78.05
	$Fd_{\|W\|}^{BRKd-2PP}$	86.36	89.66	86.13	95.07	82.05	67.24	76.18	96.11	77.89

Table 8.4: F-Score (higher is better) for the dependency parsing encodings on the test sets of the UD treebanks used in this work.

ENC	ENG	FR	BQ	GER	HB	HG	KR	PL	SW
$\Gamma^{ABS}_{\|W\|}$	301.65	124.76	243.13	194.33	162.15	213.97	279.50	255.75	231.25
$\Gamma^{REL}_{\|W\|}$	316.96	168.75	254.37	202.49	174.42	222.63	384.55	266.62	283.03
$\Gamma^{POS}_{\|W\|}$	317.89	167.39	253.73	209.30	172.82	225.71	385.89	268.80	286.26
$\Gamma^{BRK}_{\|W\|}$	316.83	163.85	251.03	204.66	198.73	227.19	346.22	266.81	287.12
$\Gamma^{BRK-2PG}_{\|W\|}$	319.89	174.93	**258.91**	**215.84**	197.89	222.82	389.46	264.66	284.45
$\Gamma^{BRK-2PP}_{\|W\|}$	318.00	171.18	256.81	212.08	195.93	222.85	398.53	264.06	281.54
$\Gamma^{BRK-D}_{\|W\|}$	**324.44**	**175.22**	252.19	183.19	200.18	224.05	**401.73**	**271.44**	**287.57**
$\Gamma^{BRKd-2PG}_{\|W\|}$	317.98	147.54	244.27	182.45	**202.11**	**225.84**	397.20	263.93	287.13
$\Gamma^{BRKd-2PP}_{\|W\|}$	322.19	170.38	245.44	170.69	200.94	224.40	395.03	269.65	286.03

Table 8.5: Speed measured in sentences per second (higher is better) for the UD test sets, using a i5-1155G7 CPU.

Results analysis

From the results obtained we can infer that the bracketing-based encodings are the ones that perform better, ranking higher in almost all languages. This may be due to this encoding being independent of any language feature. For the bracketing encoding we can also see that the default algorithm without planar separation ($Fd^{BRK}_{|W|}$) usually performs betters than its two planar modification, this could be because the reduced label sparsity produced by not including the additional set of bracketing characters $B*$. The problem with this, is that the small increase in accuracy does not outweigh the innability of decoding non projective trees. Another important note that we can make on the results is the great increase in the LAS score that produces the inclusion of Multilingual BERT language embeddings, making some treebanks (e.g. Polish$_{PDB}$) reach close to perfect scores. Not so good were the experiments where the multitask learning capabilities were included, making most treebanks perform worse overall. As in the constituent case, this could be caused by performing the multitask training with all available features instead of curated ones, but more experiments that fall outside of the scope of this work would be needed to confirm that hypothesis. In terms of the speed with which the sequence labeling tools can predict our labels most encoding rank the same, with the differences between languages being based mostly on differences in sentence lengths. However, we can make the remark that overall the bracketing based encodings seem to be slightly faster than the other ones, but again, to prove this we would need to perform more experiments with different hardware.

Chapter 9

Conclusion and Future Work

In this final chapter of the thesis, we summarize the contributions of the work and its usefulness. We will also talk about future room for improvements, and new lines of research that could arise from this project.

9.1 Conclusion

Syntactic parsing is a core task in natural language processing that focuses on automatically inferring the syntactic structure of natural language sentences, which is useful for machines to understand and generate human languages. Constituent and dependency parsing are two of the most popular formalisms to represent syntax. Although their theoretical motivations are different, when it comes to design models that can automatically parse sentences, they share certain weaknesses such as the need for dedicated systems to parse each formalism or limited speed.

To solve these problems, researchers have worked on casting such tasks as a sequence labeling problem, i.e., each input token receives an output label, such that the whole sentence encodes a linearized syntactic tree that can be de-linearized back if wished. This is interesting because sequence labeling models are generic, fast, and are already used for a variety of natural language processing tasks. In the context of parsing, such strategy has been successful for both formalisms and multilingual setups, but to date there was no unified framework that could allow the community to work with parsing as sequence labeling using a single system.

Specifically, in this work we have implemented a system that includes 3 different linearizations for constituent parsing (a top-down approach, a relative-scale approach and a mixed one) and 4 different linearizations for dependency encodings (two simple encodings based on absolute and relative indexing, a part-of-speech-based offsetting approach, and a bracketing-based approach as well as its variants). Keeping in mind that new encodings or syntactic formalisms might be added in the future, we kept a scalable and extensible imple-

mentation.

To show empirical results and demonstrate that the approach, and more particularly the system, are practical, we trained different sequence labeling systems using modern neural models, and ran experiments on multiple languages. In this context, we also defined and implemented a set of heuristics to ensure correctness for both constituent and dependency trees generated by a trained model, that eventually could produce corrupted sequences of labels that cannot be naturally decoded into the desired syntactic tree.

9.2 Future work

The system implemented in this project supports dependency and (continuous) constituent formalisms, but thanks to its modularity and extendability, additional syntactic or semantic formalisms can be added in the future. Some encodings for some other formalisms have been left out of this work, e.g., discontinuous constituent grammars [77], although it is already known that they perform successfully when cast as sequence labeling too. Also, we plan to research new encodings for formalisms that remain unexplored, such as head-driven phrase structure grammars [78] and certain flavours of semantic dependency parsing [79], which are theoretically feasible, although their practical utility needs to be tested empirically.

Another line of research is to optimize linearizations that already exist. An example of this could be to transform the constituent encoding to a purely incremental algorithm. In its current version, given a word w_i we need to look to the next one w_{i+1} to encode the label for w_i. To train a model that learns the linearization function, the standard approach is to look at models that either use bidirectional LSTMs (that look to the left and right context, and therefore they are not incremental) or Transformers (which suffer a similar problem). We could create a purely incremental parser by forcing the model to look either just to the left of to the right context (e.g. using simply a strict right-to-left LSTM, or using a left-to-right LSTM and encode w_i with respect to w_{i-1} instead of w_{i+1}). We also would like to explore how to exploit linearized trees for downstream NLP tasks, such as named-entity recognition. Specifically, we could embed such labels so they can be concatenated with other input embeddings, such as pre-trained word embeddings, or PoS tag embeddings. Last but not least, thanks to the loosely coupled design of the system, new user interfaces can be implemented. For instance, it is common that software tools become more used thanks to a web interface that allows users to try and visualize how the tool works before downloading it. The current user interaction layer is intended to be used from command line, but it could be adapted easily to offer a web service that could answer requests from, for instance, a java-script interface.

Bibliography

[1] C. Gómez-Rodríguez and D. Vilares, "Constituent parsing as sequence labeling," in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. Brussels, Belgium: Association for Computational Linguistics, Oct.-Nov. 2018, pp. 1314–1324. [Online]. Available: https://aclanthology.org/D18-1162

[2] D. J. . J. H. Martin, *Speech and Language Processing, Chapter 12*. [Online]. Available: https://web.stanford.edu/~jurafsky/slp3/12.pdf

[3] G. Thurmair, "Parsing for grammar and style checking," in *COLING 1990 Volume 2: Papers presented to the 13th International Conference on Computational Linguistics*, 1990. [Online]. Available: https://aclanthology.org/C90-2063

[4] U. Hermjakob, "Parsing and question classification for question answering," 2001.

[5] S. Yang and K. Tu, "Bottom-up constituency parsing and nested named entity recognition with pointer networks," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Dublin, Ireland: Association for Computational Linguistics, May 2022, pp. 2403–2416. [Online]. Available: https://aclanthology.org/2022.acl-long.171

[6] E. Bugliarello and N. Okazaki, "Enhancing machine translation with dependency-aware self-attention," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Online: Association for Computational Linguistics, Jul. 2020, pp. 1618–1627. [Online]. Available: https://aclanthology.org/2020.acl-main.147

[7] K. Chen, R. Wang, M. Utiyama, L. Liu, A. Tamura, E. Sumita, and T. Zhao, "Neural machine translation with source dependency representation," in *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*. Copenhagen, Denmark: Association for Computational Linguistics, Sep. 2017, pp. 2846–2852. [Online]. Available: https://aclanthology.org/D17-1304

[8] K. Sun, R. Zhang, S. Mensah, Y. Mao, and X. Liu, "Aspect-level sentiment analysis via convolution over dependency tree," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. Hong Kong, China: Association for Computational Linguistics, Nov. 2019, pp. 5679–5688. [Online]. Available: https://aclanthology.org/D19-1569

[9] P. Gamallo, M. Garcia, and S. Fernández-Lanza, "Dependency-based open information extraction," in *Proceedings of the Joint Workshop on Unsupervised and Semi-Supervised Learning in NLP*. Avignon, France: Association for Computational Linguistics, Apr. 2012, pp. 10–18. [Online]. Available: https://aclanthology.org/W12-0702

[10] M. Strzyz, D. Vilares, and C. Gómez-Rodríguez, "Viable dependency parsing as sequence labeling," 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)*. Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 717–723. [Online]. Available: https://aclanthology.org/N19-1077

[11] E. Roberts, "Natural language processing - overview," 2004. [Online]. Available: https://cs.stanford.edu/people/eroberts/courses/soco/projects/2004-05/nlp/overview_history.html

[12] Y. Bengio, R. Ducharme, P. Vincent, and C. Jauvin, "A neural probabilistic language model," 2003. [Online]. Available: https://www.jmlr.org/papers/volume3/bengio03a/bengio03a.pdf

[13] D. Mehta, "Stages of natural langauge processing," 2020. [Online]. Available: https://byteiota.com/stages-of-nlp/

[14] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[15] J. Yang and Y. Zhang, "Ncrf++: An open-source neural sequence labeling toolkit," *arXiv preprint arXiv:1806.05626*, 2018.

[16] R. van der Goot, A. Üstün, A. Ramponi, I. Sharaf, and B. Plank, "Massive choice, ample tasks (machamp): A toolkit for multi-task learning in nlp," *arXiv preprint arXiv:2005.14672*, 2020.

[17] N. Chomsky, "Syntactic structures," in *Syntactic Structures*. De Gruyter Mouton, 2009.

[18] M. A. Marcinkiewicz, "Building a large annotated corpus of english: The penn treebank," *Using Large Corpora*, vol. 273, 1994.

[19] E. Charniak, "A maximum-entropy-inspired parser," in *1st Meeting of the North American Chapter of the Association for Computational Linguistics*, 2000.

[20] S. Petrov, L. Barrett, R. Thibaux, and D. Klein, "Learning accurate, compact, and interpretable tree annotation," in *Proceedings of the 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics*, 2006, pp. 433–440.

[21] K. Sagae and A. Lavie, "A classifier-based parser with linear run-time complexity," in *Proceedings of the Ninth International Workshop on Parsing Technology*, 2005, pp. 125–132.

[22] M. Wang, K. Sagae, and T. Mitamura, "A fast, accurate deterministic parser for chinese," in *Proceedings of the 21st International Conference on Computational Linguistics and 44th Annual Meeting of the Association for Computational Linguistics*, 2006, pp. 425–432.

[23] D. Fernández-González and C. Gómez-Rodríguez, "Faster shift-reduce constituent parsing with a non-binary, bottom-up strategy," *Artificial Intelligence*, vol. 275, pp. 559–574, 2019.

[24] J. R. Brennan, C. Dyer, A. Kuncoro, and J. T. Hale, "Localizing syntactic predictions using recurrent neural network grammars," *Neuropsychologia*, vol. 146, p. 107479, 2020.

[25] H. Gaifman, "Dependency systems and phrase-structure systems," *Information and control*, vol. 8, no. 3, pp. 304–337, 1965.

[26] S. Buchholz and E. Marsi, "Conll-x shared task on multilingual dependency parsing," in *Proceedings of the tenth conference on computational natural language learning (CoNLL-X)*, 2006, pp. 149–164.

[27] J. Nivre, "An efficient algorithm for projective dependency parsing," in *Proceedings of the eighth international conference on parsing technologies*, 2003, pp. 149–160.

[28] M. Ballesteros and J. Nivre, "MaltOptimizer: A system for MaltParser optimization," in *Proceedings of the Eighth International Conference on Language Resources and Evaluation (LREC'12)*. Istanbul, Turkey: European Language Resources Association (ELRA), May 2012, pp. 2757–2763. [Online]. Available: http://www.lrec-conf.org/proceedings/lrec2012/pdf/715_Paper.pdf

[29] D. Chen and C. D. Manning, "A fast and accurate dependency parser using neural networks," in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 2014, pp. 740–750.

[30] M. Straka, J. Hajic, and J. Straková, "Udpipe: trainable pipeline for processing conll-u files performing tokenization, morphological analysis, pos tagging and parsing," in *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC'16)*, 2016, pp. 4290–4297.

[31] R. McDonald and F. Pereira, "Online learning of approximate dependency parsing algorithms," in *11th Conference of the European Chapter of the Association for Computational Linguistics*, 2006, pp. 81–88.

[32] T. Dozat and C. D. Manning, "Deep biaffine attention for neural dependency parsing," *arXiv preprint arXiv:1611.01734*, 2016.

[33] R. Tsarfaty, D. Seddah, Y. Goldberg, S. Kübler, Y. Versley, M. Candito, J. Foster, I. Rehbein, and L. Tounsi, "Statistical parsing of morphologically rich languages (spmrl) what, how and whither," in *Proceedings of the NAACL HLT 2010 First Workshop on Statistical Parsing of Morphologically-Rich Languages*, 2010, pp. 1–12.

[34] J. Nivre, M.-C. de Marneffe, F. Ginter, J. Hajič, C. D. Manning, S. Pyysalo, S. Schuster, F. Tyers, and D. Zeman, "Universal dependencies v2: An evergrowing multilingual treebank collection," *arXiv preprint arXiv:2004.10643*, 2020.

[35] F. Zhai, S. Potdar, B. Xiang, and B. Zhou, "Neural models for sequence chunking," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 31, no. 1, 2017.

[36] M. Yasunaga, J. Kasai, and D. Radev, "Robust multilingual part-of-speech tagging via adversarial training," *arXiv preprint arXiv:1711.04903*, 2017.

[37] J. Nothman, N. Ringland, W. Radford, T. Murphy, and J. R. Curran, "Learning multilingual named entity recognition from wikipedia," *Artificial Intelligence*, vol. 194, pp. 151–175, 2013.

[38] E. Brill, "A simple rule-based part of speech tagger," in *Proceedings of the Third Conference on Applied Natural Language Processing*, ser. ANLC '92. USA: Association for Computational Linguistics, 1992, p. 152–155. [Online]. Available: https://doi.org/10.3115/974499.974526

[39] T. Brants, "TnT – a statistical part-of-speech tagger," in *Sixth Applied Natural Language Processing Conference*. Seattle, Washington, USA: Association for Computational Linguistics, Apr. 2000, pp. 224–231. [Online]. Available: https://aclanthology.org/A00-1031

[40] F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain." *Psychological review*, vol. 65, no. 6, p. 386, 1958.

[41] A. Meyer-Baese and V. Schmid, "Chapter 7 - foundations of neural networks," in *Pattern Recognition and Signal Analysis in Medical Imaging (Second Edition)*, second edition ed., A. Meyer-Baese and V. Schmid, Eds. Oxford: Academic Press, 2014, pp. 197–243. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780124095458000078

[42] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," California Univ San Diego La Jolla Inst for Cognitive Science, Tech. Rep., 1985.

[43] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[44] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2017. [Online]. Available: https://arxiv.org/abs/1706.03762

[45] A. Radford, K. Narasimhan, T. Salimans, I. Sutskever *et al.*, "Improving language understanding by generative pre-training," 2018.

[46] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[47] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, 2014, pp. 1532–1543.

[48] G. Lample, M. Ballesteros, S. Subramanian, K. Kawakami, and C. Dyer, "Neural architectures for named entity recognition," 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, Jun. 2016, pp. 260–270. [Online]. Available: https://aclanthology.org/N16-1030

[49] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," 2014. [Online]. Available: https://arxiv.org/abs/1412.3555

[50] J. P. Chiu and E. Nichols, "Named entity recognition with bidirectional LSTM-CNNs," *Transactions of the Association for Computational Linguistics*, vol. 4, pp. 357–370, 2016. [Online]. Available: https://aclanthology.org/Q16-1026

[51] J. Sarzynska-Wawer, A. Wawer, A. Pawlak, J. Szymanowska, I. Stefaniak, M. Jarkiewicz, and L. Okruszek, "Detecting formal thought disorder by deep contextualized word representations," *Psychiatry Research*, vol. 304, p. 114135, 2021.

[52] X. Ma and E. Hovy, "End-to-end sequence labeling via bi-directional lstm-cnns-crf," *arXiv preprint arXiv:1603.01354*, 2016.

[53] Y. Shao, C. Hardmeier, J. Tiedemann, and J. Nivre, "Character-based joint segmentation and pos tagging for chinese using bidirectional rnn-crf," *arXiv preprint arXiv:1704.01314*, 2017.

[54] J. Yang, S. Liang, and Y. Zhang, "Design challenges and misconceptions in neural sequence labeling," *arXiv preprint arXiv:1806.04470*, 2018.

[55] R. Caruana, "Multitask learning," *Machine learning*, vol. 28, no. 1, pp. 41–75, 1997.

[56] Y. Shen, Z. Lin, A. P. Jacob, A. Sordoni, A. Courville, and Y. Bengio, "Straight to the tree: Constituency parsing with neural syntactic distance," *arXiv preprint arXiv:1806.04168*, 2018.

[57] D. Vilares, M. Abdou, and A. Søgaard, "Better, faster, stronger sequence tagging constituent parsers," *arXiv preprint arXiv:1902.10985*, 2019.

[58] Z. Li, J. Cai, S. He, and H. Zhao, "Seq2seq dependency parsing," in *Proceedings of the 27th International Conference on Computational Linguistics.* Santa Fe, New Mexico, USA: Association for Computational Linguistics, Aug. 2018, pp. 3203–3214. [Online]. Available: https://aclanthology.org/C18-1271

[59] A. Muñoz-Ortiz, M. Strzyz, and D. Vilares, "Not all linearizations are equally data-hungry in sequence labeling parsing," *arXiv preprint arXiv:2108.07556*, 2021.

[60] A. Yli-Jyrä, "On dependency analysis via contractions and weighted fsts," in *Shall We Play the Festschrift Game?* Springer, 2012, pp. 133–158.

[61] A. Yli-Jyrä and C. Gómez-Rodríguez, "Generic axiomatization of families of noncrossing graphs in dependency parsing," *arXiv preprint arXiv:1706.03357*, 2017.

[62] C. Gómez-Rodríguez and J. Nivre, "A transition-based parser for 2-planar dependency structures," in *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, 2010, pp. 1492–1501.

[63] ——, "Divisible transition systems and multiplanar dependency parsing," *Computational Linguistics*, vol. 39, no. 4, pp. 799–845, 2013.

[64] M. Strzyz, D. Vilares, and C. Gómez-Rodríguez, "Bracketing encodings for 2-planar dependency parsing," *arXiv preprint arXiv:2011.00596*, 2020.

[65] R. F. i Cancho, C. Gómez-Rodríguez, and J. Esteban, "Are crossing dependencies really scarce?" *Physica A: Statistical Mechanics and its Applications*, vol. 493, pp. 311–329, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0378437117310580

[66] A. Gholamy, V. Kreinovich, and O. Kosheleva, "Why 70/30 or 80/20 relation between training and testing sets: a pedagogical explanation," 2018.

[67] M. Strzyz, D. Vilares, and C. Gómez-Rodríguez, "Viable dependency parsing as sequence labeling," *arXiv preprint arXiv:1902.10505*, 2019.

[68] M. Gardner, J. Grus, M. Neumann, O. Tafjord, P. Dasigi, N. F. Liu, M. Peters, M. Schmitz, and L. Zettlemoyer, "AllenNLP: A deep semantic natural language processing platform," in *Proceedings of Workshop for NLP Open Source Software (NLP-OSS)*. Melbourne, Australia: Association for Computational Linguistics, Jul. 2018, pp. 1–6. [Online]. Available: https://aclanthology.org/W18-2501

[69] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman, "Glue: A multi-task benchmark and analysis platform for natural language understanding," *arXiv preprint arXiv:1804.07461*, 2018.

[70] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. Bowman, "GLUE: A multi-task benchmark and analysis platform for natural language understanding," in *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*. Brussels, Belgium: Association for Computational Linguistics, Nov. 2018, pp. 353–355. [Online]. Available: https://aclanthology.org/W18-5446

[71] T. Kikas and M. Treumuth, "Automatic parser evaluation," *Retrieved December*, vol. 16, p. 2014, 2007.

[72] M. Collins, "Head-driven statistical models for natural language parsing," *Computational linguistics*, vol. 29, no. 4, pp. 589–637, 2003.

[73] K. Sagae and A. Lavie, "Parser combination by reparsing," in *Proceedings of the Human Language Technology Conference of the NAACL, Companion Volume: Short Papers*. New York City, USA: Association for Computational Linguistics, Jun. 2006, pp. 129–132. [Online]. Available: https://aclanthology.org/N06-2033

[74] S. Petrov and D. Klein, “Improved inference for unlexicalized parsing,” in *Human Language Technologies 2007: The Conference of the North American Chapter of the Association for Computational Linguistics; Proceedings of the Main Conference*. Rochester, New York: Association for Computational Linguistics, Apr. 2007, pp. 404–411. [Online]. Available: https://aclanthology.org/N07-1051

[75] M. Zhu, Y. Zhang, W. Chen, M. Zhang, and J. Zhu, “Fast and accurate shift-reduce constituent parsing,” in *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2013, pp. 434–443.

[76] O. Vinyals, L. u. Kaiser, T. Koo, S. Petrov, I. Sutskever, and G. Hinton, “Grammar as a foreign language,” in *Advances in Neural Information Processing Systems*, C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, Eds., vol. 28. Curran Associates, Inc., 2015. [Online]. Available: https://proceedings.neurips.cc/paper/2015/file/277281aada22045c03945dcb2ca6f2ec-Paper.pdf

[77] D. Vilares and C. Gómez-Rodríguez, “Discontinuous constituent parsing as sequence labeling,” *arXiv preprint arXiv:2010.00633*, 2020.

[78] J. Zhou and H. Zhao, “Head-driven phrase structure grammar parsing on penn treebank,” *arXiv preprint arXiv:1907.02684*, 2019.

[79] S. Oepen, M. Kuhlmann, Y. Miyao, D. Zeman, S. Cinková, D. Flickinger, J. Hajic, and Z. Uresova, “Semeval 2015 task 18: Broad-coverage semantic dependency parsing,” in *Proceedings of the 9th International Workshop on Semantic Evaluation (SemEval 2015)*, 2015, pp. 915–926.