MASARYK
UNIVERSITY

FACULTY OF INFORMATICS

RapCor Support

Master's Thesis

MICHAL HALA

Brno, Spring 2022

MASARYK UNIVERSITY

FACULTY OF INFORMATICS

RapCor Support

Master's Thesis

MICHAL HALA

Advisor: doc. Mgr. Pavel Rychlý, Ph.D

Department of Machine Learning and Data Processing

Brno, Spring 2022


FACULTAS ARTIS INFORMATICÆ
UNIVERSITAS MASARYKIANA

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Michal Hala

Advisor: doc. Mgr. Pavel Rychlý, Ph.D

Acknowledgements

I would like to thank my supervisors doc. Mgr. Pavel Rychlý, Ph.D and doc. PhDr. Alena Polická, Ph.D. for guidance and help they provided whenever it was needed.

Abstract

RapCor is a specialized corpus for the French language comprising entirely of rap songs. Over the past several years, many tools were created for management and annotation of the songs.

This thesis has set four main goals: first, to examine available part of speech taggers (TreeTagger, FreeLing, UDPipe2) and choose one for the purposes of RapCor; second, to make the work of annotators easier by implementing a new vertical format, which will shorten the time needed for the annotation; third, to combine the new tagger and new vertical into a new pipeline; fourth, to improve overall quality of life of the RapCor annotators by updating existing tools or creating new ones to solve the matters of the day.

All four goals set have been successfully achieved. UDPipe2 was chosen as the new part of speech tagger and a model was trained using the UD French GSD corpus. New vertical format was created, including the new dictionaries. Finally, both have been combined into the new French pipeline.

Keywords

NLP, corpora, morphological analysis, part of speech tagger, RapCor, FreeLing, UDPipe

Contents

List of Tables

List of Figures

Introduction

RapCor is a corpus of francophone rap songs, which is being created by the staff and students of the Faculty of Arts, Masaryk University. The project aims to collect, catalogue and analyse rap songs of mainly French and Quebecois artists. Most of the manual work spent on annotating the corpus is done by the students, providing them with experience of linguistic work, while being significantly more intriguing then annotating heaps of newspaper texts or Wikipedia articles.

This work has one clear goal, to ease the work of the annotators, and improve the overall quality of their work by updating the tools used in the process.

The first goal is to choose a new part of speech tagger to be used in the RapCor pipeline. TreeTagger, which is being used, is an obsolescent software, and there is a plenty of modern open source taggers waiting to be used. In this work, FreeLing and UDPipe have been examined, and a successor to the TreeTagger has been chosen. Choosing the new tagger will provide more accurate pre-annotation for the users by including a more descriptive morphological tagset.

The second goal is to ease the work of the annotators by implementing a new vertical format used in the corpus. In the previous version, the annotators were burdened by a dozen of tasks associated with creating the vertical. The new version has most of the content pre-filled, and user must only check the veracity of the data.

The third goal is to put the tagger and new vertical together, creating a new French pipeline for generating vertical text, while keeping most of the existing functional programs similar in use to their predecessors.

The fourth goal is to improve the overall quality of life of the RapCor annotators by updating any existing software, or creating new one to solve the matters of the day.

This thesis was written as a project of the NLP Centre of Faculty of Informatics, Masaryk University, and the text of the work is published under the Creative Commons Zero (CC0) license.

1 Natural Language Processing

Natural Language Processing (NLP) is a field of study utilizing computer science to process the natural languages (such as English, French, or Czech), in order to create software able to help humans (or other software for that matter) with all sorts of linguistic tasks, such as spell checking, translation, information retrieval, question answering, automatic correction, and many others.[1] [2]

Natural languages are alien to the computers, because as opposed to formal languages, which computers understand, are full of exceptions, hard to describe rules, and ambiguities in basically all layers of language. Attempts to translate natural languages to formal languages, for example using propositional, first order, or even intentional logic, have never been truly successful. And so the processing of the language by analysing each layer of language in detail is the way to make computers work with human languages.

Analysis of the natural language can be typically split into several phases, each corresponding to a layer of the language. Each layer is processed differently because of the nature of the data, and wanted results, utilizing a knowledge from a wide array of scientific fields.

This process is often split into five stages. First is the phonetic analysis[3], which studies the sounds of a language, and the two most prominent tasks are speech recognition and speech generation. Then follows the morphological analysis[4], which studies individual words, how they are created, from which parts they constitute, how they flex, and how they can be categorized. On that follows the syntactical analysis[5] which tries to capture the relationship between words in a sentence. The goal of such analysis is the ability to tell whether a sentence is correct with respect to a language's grammar. A typical output of semantic analysis is a syntactic tree depicting the dependencies in a sentence. Then semantic analysis[6] can be performed, with attempt to disambiguate the meaning of a sentence. And finally the pragmatic analysis[7] tries to find the meaning of an utterance in a wider context.

This work mostly concerns itself with the morphological analysis of the natural languages. The unit of the morphological analysis is the *morpheme*, defined as the smallest unit of a language able to carry any

meaning. Morphemes are grouped together creating words. The most typical tasks are morphological disambiguation, and lemmatization.

The goal of morphological disambiguation, also called part of speech tagging, is the process of assigning a part of speech tag to the given word, typically in the context of its sentence. The part of speech tag can comprise simply of the morphological category (such as noun, verb, adjective), or from the more descriptive features (such as number, person, tense, gender) which can be used to provide further information about flexible word forms.

Lemmatization is the process of finding the base form of a given word. What the definition of base form, or lemma, is can differ from language to language, but it typically depends on the features of the particular part of speech. For example, the lemma of a noun is commonly a singular number (dogs $\rightarrow$ dog) or nominative case if the language has cases; the lemma of a verb is typically an infinitive (bought $\rightarrow$ buy); and so on.

Both morphological disambiguation and lemmatization vary wildly based on the language in question. For less inflectional languages, such as English, the lemmatization can be immensely easier than for highly inflectional languages like Czech. The part of speech tagging is instead quite a similar task, at least for the Indo-European languages. Assigning the features also depends heavily on a language, because of the different feature sets.

2 Part of Speech Tagging

Part of speech (POS) tagging is the process of specifying a part of speech, such as noun, verb, adjective, etc., of words in a text, which is an important part of many NLP pipelines. POS taggers typically operate on the sentence level of the document, and often utilize parsers and tokenizers, which can provide further insight into the grammatical and sometimes even semantic composition of the sentence.[8]

POS tagger assigns a POS tag (typically an abbreviation) chosen from a *tagset*, which is specified by the language of the document. For example, in English we distinct the following parts of speech: noun, verb, adjective, adverb, pronoun, preposition, conjunction, interjection, and article; which we can describe using the following tagset: NOUN, VERB, ADJ, ADV, PRON, PREP, CONJ, INTJ, DET, sometimes including tags for numbers (NUM), punctuation (PUNCT), and others.

POS taggers discussed in this work also assign each word a *lemma*, or a base form of the word, for verbs typically infinitive, for nouns singular (nominative if the language has noun cases), etc. In English the lemmatization is a rather straightforward process, for French, however, is the process a bit more complex, since its parts of speech are more inflectional.

In table 2.1 is a typical output of a POS tagger on English sentence *"Dogs ran around in the rain."*.

Table 2.1: Tagged English sentence *Dogs ran around in the rain.*

Word	Lemma	Tag
Dogs	dog	NOUN
ran	run	VERB
around	around	ADV
in	in	PREP
the	the	DET
rain	rain	NOUN
.	.	PUNCT

2.1 Challenges of POS tagging

Among the typical problems encountered, when tagging a text, the following stand out:

- **Unknown words** which are not included in the POS tagger's dictionary can cause confusion. They are typically either named entities (such as people, organizations, or locations), which should be detected by the named entity recognition (NER) module; or phrases uttered in different language than expected (in RapCor is common to notice English or Arabic phrases in the middle of French text).

- **Polysemy** is an omnipresent problem in NLP which does not evade POS tagging as well. Polysemy can affect POS tagging in the preprocessing of the input sentence, for example during the syntactic analysis, and also during assignment of the lemma. If no context is provided, the tagger usually chooses the more frequent meaning of the word.

- **Human error** can affect tagging in many ways. Humans who manually annotate the text sometimes make mistakes (often by misspelling the tags) or don't follow the guidelines for ambiguous situations.

2.2 Types of POS taggers

We can categorize POS taggers from several points of view, the two main ones are (a) how the tagger determines which tag to use, (b) how is it trained (if it is trained), and (c) on how long word phrases it operates.

- **Rule based taggers** have a fixed ruleset, often hand made by humans, they use to assign tags to words. They typically have a lexicon which is used to determine a set of tags for each word, and then use the rules to pick out the best one. They are quite inflexible and perform poorly on unknown words.

- **Stochastic taggers** use statistics or machine learning methods, such as neural networks, in order to find the most probable tags using their internal probabilistic model obtained by learning from the training data.

- **Hybrid taggers** utilize both rules and stochastic models retaining the strengths of both and minimizing their weaknesses for the cost of being more difficult to put together.

Supervised taggers are trained using data, in which each word contains the exact correct tag, or at least a set of probabilities for each plausible tag. **Unsupervised taggers** don't have the luxury of having labeled data so they often rely on clustering methods. Completely unsupervised methods are much less common. The more common unsupervised approach is to use a supervisedly trained model and further improve it using the unlabeled data.

Taggers also differ based on word phrases, or **n-grams**, they use, unigrams are the easiest to use, but bigrams or trigrams carry more information about the context.

2.3 Training

As explained above, tagging is the process of morphological disambiguation, the mapping of a tag V_t from tagset $C_1, C_2, ..., C_3$ to each token w in the input text. In order to do that, statistical models, trained using both supervised and unsupervised approaches, are used.

2.3.1 Rule-Based Approach

Simple training process could be described as follows: take the annotated data, remove the tags from them and let the tagger-in-training assign tags to them. Then compare these interim tagged data with the true annotations, and inspect whether the tagger guessed any tagging rules consistent with reality. If yes, add the best of them to the ruleset. Repeat the process as long as any new rules of quality are being discovered.[9]

This raises the question of the initial interim annotation. It is possible to use either an arbitrary pretrained already working model,

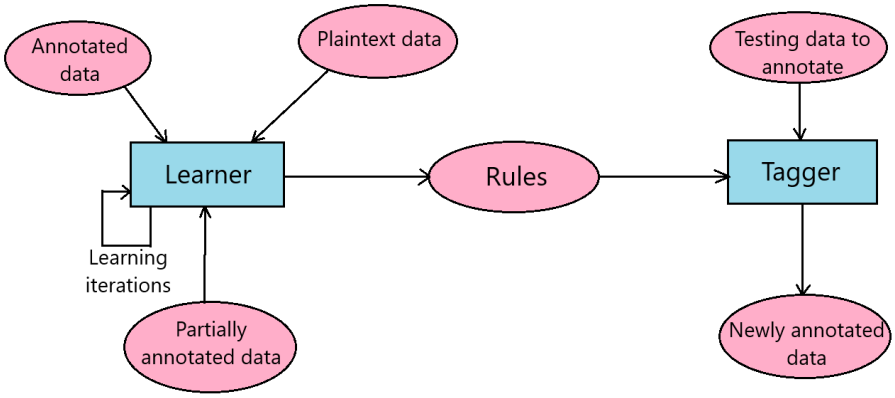

Figure 2.1: Brill Tagger learning schema

simple statistics such as assigning the most frequent tag relative to the token, or even use a trained neural network.

When choosing the new rules during the training process, the following criterion can be used: compute how adding each new rule (ceteris paribus) affects the model's error rate (or accuracy). Then choose a rule which minimizes the error rate (or maximizes accuracy). Repeat the process as long as the improvement to the model is good enough. Picture 2.1 depicts the Brill tagger learning schema described here.

2.3.2 Neural Approach

The current state-of-the-art approach of training models for part of speech tagging is utilizing deep neural networks. Modern implementations of such models often use large language models or recurrent networks for computing contextualized word embeddings of the input text, and then using a simple feed forward neural network of few layers to classify the tokens into part of speech categories.[10][11][12]

3 Corpora

Corpus (plur. corpora) is a collection of text documents nowadays mostly in a digital form. Most of the currently widely used corpora are a organized collection of documents sharing a common topic or form. Corpora can be in plaintext form, such as Project Gutenberg[1] which contains a plethora of books of the canonical western literature, a collection of texts from varying sources or topics, such as Oxford Text Archive[2], or a database of vertical text made specifically for training of models or statistical analysis.

The documents in the corpus show in a very varied forms. They can contain metadata about the author, source of the text, data of its creation; and may consist of unstructured plaintext, or be split into chapters, paragraphs, sentences, headlines, or even verses. Some corpora, typically in vertical form, contain morphological data about the words, most commonly part of speech tags and lemmas.

Corpora also very much vary in terms of size. The most commonly used corpora usually contain from hundreds of thousands of words to several million. However, some corpora obtained from web crawling[13], contain hundreds of millions or even small billions of words. A subtype of corpora especially useful for machine translation are parallel corpora, which contain aligned sentences uttered in multiple languages.[14] Corpora are often gathered from newspaper, Wikipedia articles, blog posts, or official government documents, but there also exist smaller yet no less useful corpora collected from spoken language, which can differ significantly from the written texts.

Corpora are a immensely helpful tool for modern NLP, because they can be used to train models used for the vast majority of the NLP tasks. Aside from that, the linguists find a use in corpora as well, because during the creation process or after statistical analysis of the corpus can be performed, an insight is given into the examined language and its evolution.

1. https://www.gutenberg.org/
2. https://ota.bodleian.ox.ac.uk/repository/xmlui/

3.1 Universal Dependencies (UD)

UD[3] is a project which aims to create a framework for morphological and syntactical annotation of natural languages. The project now supports more than 100 languages with over 200 treebanks (corpora).[15]

UD aims to create a corpora annotation standard useful across as many natural languages as possible, both in the area of morphological analysis with part of speech tags and advanced morphological features, and in syntactical analysis with the dependency tree system of heads and predicates describing the types of dependencies.

UD prefers to view the word as the smallest unit of a text, seeing it as a most general approach across the human languages, as opposed to morpheme based approaches (or even phrasal approaches).

3.1.1 UD Tagset

UD utilizes its own set of part of speech tags split into 16 categories depictedin talbe 3.1.

Table 3.1: UD tagset

Open Class		Closed Class and Others	
Tag	Description	Tag	Description
ADJ	Adjective	ADP	Adposition
ADV	Adverb	AUX	Auxiliary
INTJ	Interjection	CCONJ	Coordinating Conjunction
NOUN	Noun	DET	Determiner
PROPN	Proper Noun	NUM	Numeral
VERB	Verb	PART	Participle
		PRON	Pronoun
		SCONJ	Subordinating Conjunction
		PUNCT	Punctuation Symbol
		SYM	Symbol
		X	Other

3. https://universaldependencies.org/

3.1.2 UD French GSD Corpus

GSD is a French treebank which has been a part of UD since its v1.0 release.[4] The corpus consists of 400,399 words in 16,341 sentences split into a train (14,449 sentences), test (416 sentences), and dev (1,476 sentences) files. It consists mostly of blog, news, reviews, and wiki texts.

GSD utilizes the UD tagset along with the morphological features depicted in table 3.2.

Table 3.2: GSD feature set

Nominal features	
Gender	Fem, Masc
Number	Plur, Sing
Definite	Def, Ind
Verbal features	
Mood	Cnd, Imp, Ind, Sub
Tense	Fut, Imp, Past, Pres
Pronouns, Determiners, Quantifiers	
PronType	Art, Dem, Exc, Ind, Int, Neg, Prs, Rel
NumType	Ord
Reflex	Yes
Person	1, 2, 3
Number[psor]	Plur, Sing
Other Features	
Foreign	Yes
Person[psor]	Yes
Typo	Yes
Degree and Polarity	
Polarity	Neg

4. https://universaldependencies.org/treebanks/fr_gsd/index.html

4 RapCor

RapCor[1] is a corpus of francophone rap song texts created at the Faculty of Arts Masaryk University[2] under the leadership of doc. PhDr. Alena Polická, Ph.D.[16]

The corpus is comprised entirely of spoken French texts, and aims to progress the social-lexical research. Texts in the corpus contain plenty of substandard vocabulary and phrases, and allows for the deeper study of the evolution of the francophone rap culture, its word formation, various neologisms, and the modern poetry.

The collection of data for the corpus is done in large part by the students of the Faculty of Arts, who decide to enroll in the RapCor courses. The secondary aim of the project is to provide the students with the experience of applied linguistics work, which would be more interesting and engaging for the average student, than for example processing the newspaper texts, or European Union Parliament proceedings transcriptions.

4.1 RapCor Size

RapCor is a sizeable corpus with nearly 1300 fully finished songs. An exploratory analysis of lengths of 1411 texts of songs is depicted in table 4.1 and figure 4.1.

Table 4.1: RapCor exploratory analysis

Number of songs	1411
Number of words	668706
Average length of song	473.9
Shortest song	32
Longest song	2318

1. https://is.muni.cz/do/phil/Pracoviste/URJL/rapcor/index.html
2. https://www.phil.muni.cz/en

Figure 4.1: RapCor song length frequency analysis

4.2 RapCor Pipeline

Most of the work on RapCor could be divided into three distinct actions:

- **Organization**, which is done via Google Spreadsheets tables. There are several rather large spreadsheets, which hold information about songs, albums, interprets, dictionaries, and more.
- **Acquisition of albums and texts** necessary for the linguistic work itself. Currently, gathering the texts from the albums covers is the preferred way, but in case of albums not being available, or songs simply not being released on the albums, texts are usually gathered from online sources.
- **Work with the texts**, and processing them into annotated formats stored in the information system.

4.2.1 Spreadsheets

Most of the project organizational data is stored in Google Spreadsheets. This section will attempt to describe the spreadsheets currently used. For the future, a transition towards a more sophisticated relation database model would be preferred.

RAPCOR is a spreadsheet comprised of three main lists:

- **Library**, which contains information about songs. In the table are stored mainly: ID of the song, interpret, featuring interprets,

name of the song, year of release, album, country of origin, and others. From the Library spreadsheet is generated the file **Rapcor_data.js**, which is used by the song search engine on the project website.

- **Used albums** with data about albums, interprets, year of release, IDs associated with the album, work progress, and others.
- **Metadata** holding the known information about the interprets.

RapCor - albums is a spreadsheet dedicated to the albums. Most important is the list *ALBUMS-LISTE vérifiés et à vérifier*, which holds most of the information about the albums, such as interpret, title, year of release, whether there are texts on a booklet, number of processed songs, cover picture, ID of the album, and among others, whether the albums is already in possession of the project.

From the ALBUMS-LISTE vérifiés et à vérifier spreadsheet is generated the file **Rapcor_albums.js**, which is used by the album search engine on the project website. This is also the source for generating the **Wanted Albums** website, which lists all the albums which have not yet been acquired for the purposes of the project.

Dictionnaire RapCor VALIDÉ is a many-listed spreadsheet which serves as a dictionary for the substandard words appearing in the rap songs. It is divided into lists based on the part of speech, and also contains various types of phrases. Entries in the dictionary contain the part of speech tag, lemma, additional linguistic tags (marking for example foreign terms), sense, and pronunciation in the IPA[3] (in French known as API) alphabet.

4.2.2 Processing the texts

The arguably slowest part of the whole RapCor project, is processing and annotating the texts. One student can typically finish around 5 songs during one semester. This section will describe everything a student needs to do in order to finish one annotation of a song.

3. https://en.wikipedia.org/wiki/International_Phonetic_Alphabet

1. Student scans the texts from the album booklets into high resolution images. Over 700 albums had already been scanned.
2. Student cuts and vertically pastes the text using a image manipulation program, typically GIMP[4]. Some booklets are rather straightforward to cut, some have the texts set into complex shapes or wrapped around pictures, making them a bit more difficult to process.
3. Student uses an automation optical text recognition software (OCR) to create a .pdf with the song lyrics.
4. If the booklet is not available, student searches for the text online. For that purpose is used an online tool which searches for song texts on several websites. Usually, the most viable source is the webpage Genius.com[5].
5. Student uses MS Word to create **P (pochet)**, the text from booklet, and **S (sung)**, text transcribed from listening to the song, versions of the song. Metadata are added into the files. These include the song name, interpret, separation the song into verses and refrains, and which singer raps each section of the song. Student then marks the differences between these two texts by coloring the text. coloring is depicted in table 4.2.
6. Student creates a plaintext version of the song from the S version (P version if S is unavailable), strips it of any metadata, and saves it in a .txt format.
7. Student copies the content of the plaintext version of the song, opens up the website with the *fr_pipe* interface, pastes the text into a field, and clicks a button to generate the vertical data in .ods format.
8. Student opens up the automatically annotated song in .ods format and performs the following actions:
   (a) Check the veracity of the part of speech tag and the lemma.
   (b) Fill column F with structural metadata (intro, refrain, first verse, etc.).

4. https://www.gimp.org/
5. https://genius.com/

(c) Fill column H with interpret metadata (who sings this particular line).

(d) If the token was marked in step 4, add what kind of edit has been performed into column I (correction, addition, etc.).

(e) If column I was filled, write down the difference between P and S tokens into column J.

(f) If the token is a phrase, mark it into the column K.

(g) Student resolves any marked ambiguities in tags and lemmas.

9. Upload the P version (.docx), S version (.docx), plaintext version (.txt), annotated vertical (.ods) into the university information system.

Table 4.2: MS Word text coloring table

Type	P Style	S Style	Description
Correction	Gray	Saffron	Grammatical or typographical error in booklet.
Intention	Lime	Green	Error found in booklet transcription.
Change	Pink	Red	Booklet inconsistent with what is sung.
Addition	Underline	Blue	Added passage to P version.
Omission	Brown	Underline	Text in P version is not sung.
Position	Purple	Purple	Text in P version is misspositioned.
Pronunciation	Cyan	Cyan	Nonstandard pronunciation.
Incomprehension	-	Pink	Song is incomprehensible.

4.3 RapCor Code System

In RapCor, each album and song have their own identifying unique code. For albums the code format *XXX_N*, where *X* stands for a capital letter, and *N* for a number, albums might look like *ACB_6, HOR_1*, etc. The song format is similiar: *XXXNN*, for example *AEK01, VKB07*, etc.

Codes are assigned manually to each newly acquired album or processed song. The capital letter prefix typically represents the interpret (Akhenaton = *AKH*, Manau = *MNU*), the numbers are assigned in order of the songs/albums being processed.

The codes are being organized in tables *RAPCOR* and *RapCor - albums* and their management grows ever more cumbersome with the expanding number of songs and albums. For the purposes of easier creation of new codes, a script was made to create a spreadsheet with all possible code combinations and their usage.

4.4 RapCor Data Formats

As briefly described in the RapCor Pipeline section, song texts are being kept in several different formats. There are the album scans in the image formats, and the OCR texts in .pdf. The text formats are threefold and all of them are using the UTF-8 encoding:

- P (pochet) and S (song) versions of the songs are in MS Word .docx format.
- Plaintext song made from the S version is in .txt format.
- Vertical text pre-annotated by TreeTagger and then manually annotated by the students is in Open Office .ods spreadsheet format.

4.4.1 Old Vertical Format

The .ods vertical is a spreadsheet of 11 columns, which hold the data depicted in table 4.3.

If tagger didn't recognize a token, it would assign tag *NOUN* and lemma <*unknown*>. Then the token would be searched for in the

Table 4.3: Old vertical text contents

Column	Content	Annotation
A	Token	Automatic
B	Tag	Automatic
C	Lemma	Automatic
D	Advanced tags	Automatic
E	Sense	Automatic
F	Song structure	Manual
G	Pronunciation	Skipped
H	Current singer	Manual
I	P and S version difference	Manual
J	P version of token (if different)	Manual
K	Phrase tokenization	Manual

dictionary, and if found, tag, lemma, advanced tags, and sense would be filled.

The vertical would also contain coloring which marked certain phenomena for the annotators to correct, the coloring is described in table 4.4.

Table 4.4: Old vertical text coloring

Color	Phenomenon
Yellow	For <*unknown*> tokens.
Red	For ambiguous lemmas.
Cyan	For tokens tagged from dictionary.

4.4.2 New Vertical Format

As a part of new RapCor pipeline, a new vertical format has been implemented, with the goal of making the work of the annotators a bit easier. As the new format was chosen the MS Excel **.xlsx** format, as there have been occasional problems with opening the .ods files on Windows computers, and .xslx doesn't seem to exhibit this behaviour. The structure of the vertical is the very same as before, but now all columns aside from J (the P version of token) are automatically filled and just need to be checked by the annotator.

A	B	C	D	E	F	G	H	I
ça	PRON:yD	ça	FAM SYN	cela, ceci	C2	[sa]	Orelsan	correction
tombe	VERB:nSp3vFmItR	tombe			C2		Orelsan	
bien	ADV	bien			C2		Orelsan	
,	PUNCT	,			C2		Orelsan	addition
j'	PRON:nSp1yP	j'	ELIS		C2		Orelsan	
ai	AUX:nSp1vFmItR	ai			C2		Orelsan	
pris	VERB:gMvPtP	pris			C2		Orelsan	
le	DET:gMnSyAkD	le			C2		Orelsan	
soixante	NUM:gMnS	soixante			C2		Orelsan	
.	PUNCT	.			C2		Orelsan	addition
Les	DET:nPyAkD	Les			C2		Orelsan	
jaloux	NOUN:gMnP	jaloux			C2		Orelsan	
croient	VERB:nPp3vFmItR	croient			C2		Orelsan	
que	SCONJ	que			C2		Orelsan	
j'	PRON:nSp1yP	j'	ELIS		C2		Orelsan	
suis	AUX:nSp1vFmItR	suis			C2		Orelsan	
blindé	VERB:gMnSvPtP	blindé	FAM	ivre ; plein, complet	C2		Orelsan	
,	PUNCT	,			C2		Orelsan	addition
c'	PRON:gMnSp3yD	c'			C2		Orelsan	
est	AUX:nSp3vFmItR	est			C2		Orelsan	
pas	ADV:eN	pas			C2		Orelsan	
l'	DET:nSyAkD	l'	ELIS		C2		Orelsan	
avis	NOUN:gMnS	avis			C2		Orelsan	
de	ADP	de			C2		Orelsan	
ma	DET:gFnSoSq1yP	ma			C2		Orelsan	
banque	NOUN:gFnS	banque			C2		Orelsan	
.	PUNCT	.			C2		Orelsan	addition

Figure 4.2: Example of the new vertical

Also, new coloring format has been introduced. The new *xlsxwriter*[6] python library allows for more advanced formatting, so multiple visual hints have been added to the vertical. The text formatting from the S version is transferred into the vertical, so if a token is colored in a S version (.docx), the A column of the token will be colored the same. Also the coloring of the cells has been expanded, as described in table 4.5. An example of the new vertical is shown in figure 4.2.

Table 4.5: New vertical text coloring

Color	Phenomenon
Yellow	For <*unknown*> tokens.
Green	For foreign phrases.
Light red	For ambiguous lemmas.
Cyan	For tokens tagged from the old dictionary.
Pink	For tokens tagged from the verified new dictionary.
Purple	For tokens tagged from the unverified new dictionary.

6. https://xlsxwriter.readthedocs.io/

5 Comparison of POS Taggers

RapCor has used TreeTagger for several years as the main part of speech tagger. However, with the advent of neural taggers, the new competitors claim to perform better than the older system using hidden Markov models. New systems are also designed to use more descriptive tagsets, which might provide the annotator with additional information useful for their work. This chapter will examine three part of speech taggers: the currently used TreeTagger, FreeLing, and UDPipe.

5.1 TreeTagger

TreeTagger was created by Helmut Schmid as an alternative approach to POS tagging which was at the time mostly done using hidden Markov models on trigrams, which had problems accurately assigning tags to the infrequent trigrams.[17]

Schmid improved the HMM's sparse data problem by training a decision tree paying attention to the context of the tokens, which helps estimate the transition of probabilities of the model. The decision tree is recursively constructed using the ID3 algorithm[18] which in each step (each newly created node) searches for a way to divide the trigram training set into two as distinct as possible sets, with regard to the tag probability distribution of the third token in a trigram by considering either both, one, or neither of its preceeding tokens. When the algorithm finds a good enough way to split the set (meaning the splitting rule yield enough information gain), the algorithm is yet again recursively called on the original set.

The decision tree obtained from the algorithm is then pruned by merging leaves if their weighted information gain is below a set threshold.

And finally in order to obtain the optimal tag sequence for an input sentence, the TreeTagger uses an implementation of the Viterbi algorithm[19] with HMM. TreeTagger utilizes a set of lexicons, specifically a full-form and suffix lexicons, and a default entry setting for unrecognized tokens. Each word in a lexicon has a corresponding probability vector of all possible tags. TreeTagger first searches in the full-form

lexicon, if it fails, it turns the token to lowercase and tries again, if it is unsuccessful still, it searches through the suffix lexicon, and if the search fails it returns the default entry.

5.1.1 Example

Table 5.1 shows an example of the fr_pipe output for a French sentence *Le vie est un vrai film d'action.*

Table 5.1: TreeTagger tagged sentence *Le vie est un vrai film d'action.*

Token	Tag	Lemma
<s>		
La	DET :ART	le
vie	NOM	vie
est	VER :pres	être
un	DET :ART	un
vrai	ADJ	vrai
film	NOM	film
d'	PRP	de
<g/>		
action	NOM	action
<g/>		
.	SENT	.
</s>		

Fr_pipe adds tags <*s*>, </*s*> to mark beggining/ending of an sentence, and <*g*/> glues to mark where the tokenizer has split the text in a place where previously hasn't been a white space. These tags, however, are eliminated in the postprocessing and are not present in the final .ods file.

5.1.2 fr_pipe

Fr_pipe (abbreviated from French pipeline) is an in-house developed pipeline used for text processing for the purposes of RapCor. The tools used in the pipeline were created at the FI MUNI in cooperation with Lexical Computing. The pipeline itself consists of the following programs:

- **Uninorm**, a tool for normalization of non-alphanumeric symbols, such as quotation marks, apostrophes, hyphens, and others. Functionality of Uninorm tool is mostly handled by the Unitok tool.
- **Unitok**[20], a tool for tokenization of a text. Nowadays performs the normalization of symbols as well.
- **TreeTagger** trained on French data.
- **Postprocess** for creating the tagged output in a form of an .ods file. Postprocess also attempts to tag tokens marked by the TreeTagger as <*unknown*>, by searching in a lexicon.

5.1.3 Assessment

TreeTagger has served successfully for several year for the purposes of RapCor. However, the tagger's tagset is limited, and more descriptive morphological annotation is desired. The model's age also begins to show and more current, more accurate models have been examined.

5.2 FreeLing

FreeLing is an open source library developed by TALP research center[1] providing a wide array of tools for NLP. FreeLing was constructed with the idea of extensibility and customizability in mind, and the current 4.2 version supports 15 languages by default. This work aims at using the FreeLing tools for morphological analysis on French, thus utilizing the tokenizer, splitter, morphological analyser, NER analyser, tagger, parser, dependence parser, and coreference solver.[21]

FreeLing was constructed as a two layer library, the first layer providing the linguistic functionality being run as a server, and the second an API on the client side.

5.2.1 Inner Structure

From the internal point of view, FreeLing utilizes two main data types, the linguistic and processing objects.

Linguistic objects are the results of the analysis itself. They contain the following objects: *Analysis*, containing POS tag, lemma, probabilites, and a sense list. *Word*, being a word form with an assigned list of possible analyses. *Sentence*, a list of word objects, which can contain parse, or dependency trees. *Paragraph*, a list of sentences. *Document*, a list of paragraphs.

Processing objects contain the functionality for the analysis. The early analysis tools are the *language identifier*, *tokenizer* (parses the document into a sequence of words), and *splitter* (parses the document into a sequence of sentences).

After the initial analysis, the objects inheriting the processor functionality come into action.

- *Morfo* annotates words in a sentence, is able to recognise for example dates, numbers, proper nouns, or multiword locutions.
- *Tagger* (for disambiguation of POS tags for each word in a sentence) implemented either by HMM[22], or relaxation labeling[23], which is using the hybrid approach.

1. https://www.talp.upc.edu/

- *NE Classifier* used for named entity recognition based on CoNLL-2002 system[24].
- *Sense annotator* adds synset information to the selected analysis of each word in a given sentence.
- *Word sense disambiguator* for the given sentence ranks all possible word sense for each word and its analyses in the provided context using the pagerank system[25].
- *Chunk parser* for enriching of sentences with a parse tree, reimplementing the system Atserias and Rodriguez[26].
- *Dependency parser* for enriching of sentences with a dependency tree. The algorithm is based on a paper by Atserias et al.[27]
- *Co-reference solver* which searches for co-references in a given document, which is based on a system descrbied by Soon et al.[28]

5.2.2 Example

Table 5.2 shows an example of the FreeLing output for a French sentence *Le vie est un vrai film d'action.*

Table 5.2: FreeLing tagged sentence *Le vie est un vrai film d'action.*

Token	Lemma	Tag	Prob
La	le	DA0FS0	0.955121
vie	vie	NCFS000	1
est	être	VMIP3S0	0.752751
un	un	DI0MS0	0.956036
vrai	vrai	AQ0MS00	0.823065
film	film	NCMS000	1
d	d	AQ0MS0	0.238508
’	’	Frc	1
action	action	NCFS000	1
.	.	Fp	1

FreeLing in its default version uses EAGLES tagset[2], and shows the probability of the tags.

5.2.3 Assessment

The FreeLing was tested by annotating large quantities of texts from the corpus. By comparing the result with the human annotated data, FreeLing results were at best underwhelming.

FreeLing's tokenizer and sentence splitter were often at odds with varying punctuation symbols and usually weren't interpreting the triple dot, or triple exclamation mark symbols as the end of the sentence.

FreeLing was also unable to work with the French apostrophe symbols, kept splitting them away from pronouns and determiners (which were subsequently misinterpreted as nouns or verbs), and interpreting them as quoting punctuation.

Problems with named entity recognition were also noticed. FreeLing has shown overall troubles with NER, and if the proper noun was recognized, the lemma was for some reason lowercased.

As a plus side, FreeLing was the only examined tagger which was able to recognize multiword proper nouns and dates.

The most thwarting downside were the excessive amounts of time needed to load the application. FreeLing is supposed to be running in the server regime in the background and serving the requests, and if it is run this way, the serving times are admissible. However, it is preferred to not have the server running at all times, and launch the application just for the tagging of one song when needed. The loading times of the software, spanning several **minutes**, were simply too long to even consider using it in the offline mode.

The combination of poor performance compared to the adversaries, slow loading time, and the fact that the competitor UDPipe is being developed at a faculty in contact with the RapCor supervisors, led to the decision to not use FreeLing for any further testing or even training of the RapCor's own model.

2. https://freeling-user-manual.readthedocs.io/en/latest/tagsets/tagset-fr/

5.3 UDPipe

UDPipe is an open source trainable pipeline for morphological and syntactical analysis developed at ÚFAL[3]. First version of the software **UDPipe 1** was written mainly in C++, and comprised of several small and fast models. Performance of models was, however, lower compared to the contemporary neural models, thus the creators opted to create the **UDPipe 2**, the current version, which is available as a prototype written in Python. The size of the models was increased, and artificial neural models were used to improve performance by 40-50 %.[29]

UDPipe 2 was tested on *CoNLL 2018 UD Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies* and performed best of all models. The pipeline itself allows for tokenizing, splitting multi-word tokens, part of speech tagging, lemmatization, and dependency parsing. Tokenizer and multi-word token splitter was reused from UDPipe 1.

5.3.1 Model

Tokenizing and multi-word token splitting is done by first embedding the input characters, and batching them by 50 (batches of size 200 can be used during the training to improve performance). These batches are then fed into a bidirectional GRU[30] which classifies each character into one of three categories:

- **Sentence break**, a token which ends a sentence.
- **Token break**, a multi-word token to be split.
- Or the token isn't a break of any kind.

Word embeddings are used in three ways:

- **Pretrained word embeddings** are constant during the training, and have been computed from large plaintext data. These can be either *CoNLL 2017 UD Shared Task* or Wikipedia *fastText* embeddings.

3. https://ufal.mff.cuni.cz/

- **Trained word embeddings** which are initialized randomly at the beginning, are trained for each individual word.
- **Character level word embeddings** trained using a bidirectional GRU[31].

POS tagging first processes the word embedding through a multi-layer bidirectional LSTM [11] to compute the contextual embeddings. Then a dictionary of part of speech tags is constructed from the training data. A softmax classifier, enhanced by a tanh dense layer, is used to determine the final tag. UDPipe assigns the following types of part of speech tags:

- **UPOS**, the universal part of speech tags.[4]
- **XPOS**, the language specific part of speech tags.
- **UFeats**, the universal morphological features.[5]

Lemmatization process is done by classifying input word into lemma generation rules. Generation rules are created from pairs of word forms and lemmas. The algorithm first find the longest common substring of the word form and the lemma, if no such substring is found, the lemma is used instead. If the common substring is found, computes the shortest edit script to turn the word form prefix into the lemma prefix, and analogously the word form suffix into the lemma suffix. The script allows the following operations: deletion of a character, insertion of a character, and optionally copying of a character. And because the aforementioned process is wholly case insensitive, finally the algorithm determines the correct casing of the lemma.

Dependency parser is a reimplementation of a biaffine attention parser[32], which uses the contextualized embeddings and an attention mechanism enhancing the classifier. As the parser is used alongside the tagger, they can either share only the embeddings, or both embeddings and their contexts.

4. https://universaldependencies.org/u/pos/
5. https://universaldependencies.org/u/feat/index.html

5.3.2 CoNLL-U format

UDPipe2 is set to use the CoNLL-U format[6] of vertical text which holds additional data about each token. If any CoNLL-U attribute couldn't be filled by the tagger or parser, an underscore sign _ is filled instead. Tabke 5.3 describes the CoNLL-U format.

Table 5.3: CoNLL-U legend

Column	Description
ID	Index of each word.
FORM	The token.
LEMMA	The lemma.
UPOS	Part of speech tag from UPOS tagset.
XPOS	Part of speech tag, language specific.
FEATS	List of morphological features.
HEAD	Head of the current word.
DEPREL	Universal dependency relation.
DEPS	Dependency graph data.
MISC	Holds any other information.

For the purposes of this work, the UD French GSD corpus was used, which lacks the XPOS and DEPS attributes entirely. Thus, in all the following examples and results, these attributes are always blank.

5.3.3 Example

An example of the UDPipe output for a French sentence *Le vie est un vrai film d'action.* Since the CoNLL-U format is more descriptive than the TreeTagger or FreeLing formats, the example will be split into three tables. See tables 5.4, 5.5, and 5.6.

6. https://universaldependencies.org/format.html

Table 5.4: UDPipe2 example, part 1

ID	FORM	LEMMA	UPOS	XPOS
1	La	le	DET	_
2	vie	vie	NOUN	_
3	est	être	AUX	_
4	un	un	DET	_
5	vrai	vrai	ADJ	_
6	film	film	NOUN	_
7	d'	d'	ADP	_
8	action	action	NOUN	_
9	.	.	PUNCT	_

Note that XPOS property of each token is blank. Universal Dependencies do not support any French specific tagset. Thus the XPOS column is always empty.

Table 5.5: UDPipe2 example, part 2

FORM	FEATS
La	Definite=Def | Gender=Fem | Number=Sing | PronType=Art
vie	Gender=Fem | Number=Sing
est	Mood=Ind | Number=Sing | Person=3 | Tense=Pres | VerbForm=Fin
un	Definite=Ind | Gender=Masc | Number=Sing | PronType=Art
vrai	Gender=Masc | Number=Sing
film	Gender=Masc | Number=Sing
d'	_
action	Gender=Fem | Number=Sing
.	_

FEATS contains valuable morphological data about the tokens. The set of additional categories is described by the Universal Dependencies standard[7].

7. https ://universaldependencies.org/fr/feat/index.html

Table 5.6: UDPipe2 example, part 3

ID	FORM	HEAD	DEPREL	DEPS	MISC
1	La	2	det	_	_
2	vie	6	nsubj	_	_
3	est	6	cop	_	_
4	un	6	det	_	_
5	vrai	6	amod	_	_
6	film	0	root	_	_
7	d’	8	case	_	SpaceAfter=No
8	action	6	nmod	_	SpaceAfter=No
9	.	6	punct	_	SpacesAfter=/r/n/r/n

UDPipe by default also runs a tree parser on the sentence and attempts to create a dependency tree. Each token has its unique ID, which also hints the original token order in the sentence. HEAD defines the ID of a token on which the current token is dependent on, and DEPREL defines the type of dependency, as depicted in figure 5.1.

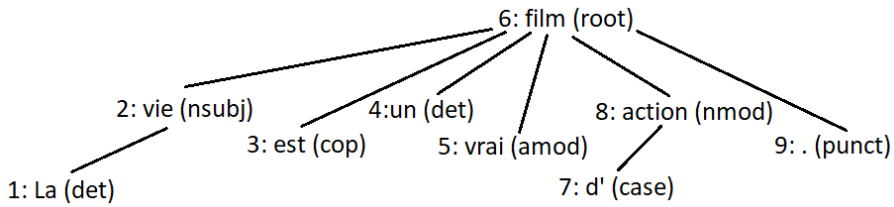

Figure 5.1: Dependency tree of sentence *Le vie est un vrai film d’action.* as reconstructed from table 5.6

However, since the results of syntactical analysis are often disputable, and the overall usefulness for the purposes of RapCor is quite low, a decision was made to omit the syntactical data from the final vertical data.

5.3.4 Assessment

UDPipe was from the beginning the expected successor to the TreeTagger. At first, the UDPipe1 full official release was tested, which lacked accuracy and performed rather poorly.

Then, the UDPipe2 was tested using the online API to tag the data with much more success. The 2.6 version available as LINDAT REST API Service[8] performed very well, but still made a considerable number of errors mainly with contracted verb forms, the adposition *du* almost always wrongly separated into two word forms *de les*, and frequent erroneus handling of possesive pronouns. Using the 2.6 online version was always meant as a temporary solution until the RapCor's own version of the software would be running on NLP Lab's machines.

Further inquiry into UDPipe2 uncovered the python source code, from which the new version could be trained on updated 2.9 version of UD corpora (mainly GSD), which is free of many errors mentioned about the 2.6 version. However, the UDPipe2 github[9] is completely void of any documentation about the use of the software, running scripts, and mainly training the models and running the inference on input data. Also, the trained models used for the 2.6 version have not been publicly released and are not included in the github. The tokenizer model needed for preparing the input data is also not present.

Nevertheless, UDPipe2 still seems like the most viable alternative to the TreeTagger and will be used in the future.

8. https://lindat.mff.cuni.cz/services/udpipe/
9. https://github.com/ufal/udpipe/tree/udpipe-2

5.4 Comparison

Two methods were used to compare the taggers with each other. Firstly a test batch of plaintext was created from 100 RapCor songs. Then each tagger was used on these data. Outputs of the taggers were then processed by a script which created vertical comparison tables in HTML with coloring for easier comparison. The comparison utilized four vertical files, output of TreeTagger (fr_pipe), FreeLing, UDPipe2, and human annotated data.

First each tagged vertical needed to be processed to the common format so that comparison could be performed in a reasonable way. First problem was assuring the same parsing of the text to sentences, because some taggers don't interpret the newline (or multiple newline) symbol as the end of a sentence, and if a song was formatted without the punctuation symbols, the tagger would interpret the songs as a one long sentence. These songs were ultimately taken out because the punctuation couldn't be reliably put in the song text split into verses.

When aligning the vertical by sentences was accomplished, aligning by words needed to be performed. Different taggers tokenize differently, and they mostly disagree when it comes to the contracted word forms and named entities. Synchronization thus needed to be computed. See figure 5.2 for an example.

fr_pipe			FreeLing			UDPipe			Human		
Token	Lemma	Tag	Token	Lemma	Tag	Token	Lemma	Tag	Token	Lemma	Tag
Bienvenue	bienvenir	VERB	**Bienvenue**	bienvenue	NOUN	**Bienvenue**	bienvenue	NOUN	**Bienvenue**	bienvenir	VERB
dans	dans	ADP	**dans**	dans	ADP	**dans**	dans	ADP	**dans**	dans	ADP
le	le	DET	**le**	le	DET	**le**	le	DET	**le**	le	DET
zoo	zoo	NOUN	**zoo**	zoo	NOUN	**zoo**	zoo	NOUN	**zoo**	zoo	NOUN
			de	de	ADP						
des	du	ADP	**les**	le	DET	**des**	de le	ADP	**des**	du	ADP
hommes	homme	NOUN	**hommes**	homme	NOUN	**hommes**	homme	NOUN	**hommes**	homme	NOUN
Paname		NOUN	**Paname_City**	paname_city	NOUN	**Paname**	Paname	PRO	**Paname**		NOUN
City		NOUN				**City**	City	PRO	**City**		NOUN

Figure 5.2: Comparison of the taggers on sentence *Bienvenue dans le zoo des hommes Paname City.* showcasing the synchronization of the vertical text

The comparison included for each tagger a token, tag, and a lemma. For further inspection were added also FEATS from UDPipe. Each entry's tag and lemma was compared to the human annotated data

and if they were in agreement, they would be colored in green, and in red if they wouldn't. For easier reading, the punctuation symbols were taken out of the comparison. See figure 5.3 for an example.

fr_pipe			FreeLing			UDPipe			Human		
Token	Lemma	Tag	Token	Lemma	Tag	Token	Lemma	Tag	Token	Lemma	Tag
Qu'	que	CONJ	**Qu**	qu	NOUN	**Qu'**	qu'	CONJ	**Qu'**	que	CONJ
on	on	PRON	**on**	on	PRON	**on**	on	PRON	**on**	on	PRON
a	avoir	VERB	**a**	avoir	VERB	**a**	avoir	VERB	**a**	avoir	VERB
			de	de	ADP						
du	du	ADP	**le**	le	DET	**du**	de le	ADP	**du**	du	ADP
temps	temps	NOUN	**temps**	temps	NOUN	**temps**	temps	NOUN	**temps**	temps	NOUN
à	à	ADP	**à**	à	ADP	**à**	à	ADP	**à**	à	ADP
perdre	perdre	VERB	**perdre**	perdre	VERB	**perdre**	perdre	VERB	**perdre**	perdre	VERB
et	et	CONJ	**et**	et	CONJ	**et**	et	CONJ	**et**	et	CONJ
merde	merde	NOUN	**merde**	merde	NOUN	**merde**	merde	VERB	**merde**	merde	NOUN

Figure 5.3: Comparison of the taggers on sentence *Qu' on a du temps á predre et merde.*

Statistics showing the token, lemma, and tag agreement with the human annotated data were computed, but the statistics were mostly skewed by the frequent desynchronization of the sentences. TreeTagger (fr_pipe) was usually around 100 % in agreement with the human annotated data, UDPipe was second on average around 95 % token agreement, 90 % tag agreement and 80 % lemma agreement, FreeLing was always the third with around 80 % token agreement, 75 % tag agreement, and 70 % lemma agreement. However, these numbers carry little value since different tokenization causes some discord, and assignment of part of speech tags is often unclear, and even human annotators have trouble reaching over 95 % agreement in some cases. As shown by a screenshot in figure 5.4.

Tagger	Token accord (%)	Lemma accord (%)	Tag accord (%)
fr_pipe	100.	100.	100.
FreeLing	80.2	71.8	76.0
UDPipe	94.2	80.7	88.0

Token count: 192

Figure 5.4: Statistics of comparison for the song AAA01

Eventually, five recently created song texts were tagged and ran through the comparison script. The comparison tables were then sent to the RapCor linguists to examine them closely. This directly led to abandonment of FreeLing as a viable TreeTagger successor, and choosing the UDPipe2, whose 2.6 version's mistakes (and their potential remedy) were described. For more details about the FreeLing's and UDPipe2.6's mistakes explore the Assessment chapters 5.2.3 and 5.3.4.

6 New Developments

During the past year, a lot of work has been made to improve RapCor. This chapter documents all of the new or updated tools which have been created to make the life of annotators easier.

6.1 New Tagset

Since the decision to use UDPipe2 was made, the question arose of which tagset to use. On one hand, UD tags could be easily translated into already used TreeTagger tags. However, one of the main motivations of choosing a new tagger was having a more descriptive tagset. New tagset is demonstrated in table 6.1.

Table 6.1: New tags with their features

Tag	Feats
ADJ	Degree, Number, Gender, Poss
ADP	-
ADV	Degree, Number, Gender, Polarity, PronType
AUX	Number, Gender, Tense, Person, VerbForm, Aspect, Voice, Mood
CCONJ	-
DET	Number[psor], Gender, Person, Person[psor], Definite, Poss, PronType
INTJ	-
NOUN	Number, Gender, Case, Animacy
NUM	Number, Gender, PronType, NumType
PRON	Number, Gender, Person, Reflex, Poss, PronType
PROPN	Number, Gender
PUNCT	-
SCONJ	-
SYM	-
VERB	Number, Gender, Tense, Person, VerbForm, Aspect, Voice, Mood
X	Foreign

Consequently, the decision was made to create a new tagset, utilizing the UD UPOS and FEATS tags. The goal was to produce a tagset which would be descriptive, yet could fit into just one spreadsheet cell and wouldn't be overly verbose. The solution was inspired by the Brno tagset for Czech, which uses an attribute system for the tag and the features. New features are described in table 6.2.

Table 6.2: New tagset features

Feature		Values
Polarity	e	Neg → N
Gender	g	Fem → F, Masc → M
Animacy	i	Anim → A, Inan → I, Hum → H, Nhum → N
Number	n	Plur → P, Sing → S
Number[psor]	o	Plur → P, Sing → S
Case	c	Nom → N, Acc → A, Dat → D
Person	p	1, 2, 3
Person[psor]	q	1, 2, 3
VerbForm	v	Fin → F, Inf → I, P → P
Aspect	a	Imp → I, Perf → P, Prosp → R, Prog → G, Hab → H, Iter → T
Mood	m	Cnd → C, Imp → I, Ind → D, Sub → S
Tense	t	Fut → F, Imp → I, Past → P, Pres → R
PronType	y	Art → A, Dem → D, Exc → E, Ind → I, Int → T, Neg → N, Prs → P, Rel → R
Poss	s	Yes → Y
NumType	x	Ord → O
Definite	k	Def → D, Ind → I
Reflex	r	Yes → Y
Foreign	f	Yes → Y

The Brno attribute tagset system works in a following way: the tag is a concatenation of pairs attribute-value, where attribute is described by a lowercase letter (e.g. k for part of speech, g for gender, t for tense, etc.), and a uppercase letter or a number for the value. For example a pair gM would be interpreted as gender=Masculine. The attributes in a tag follow a predefined order, which eases the reading. The first

attributes is always the part of speech, and if a value for an attribute is unknown, the attribute is completely omitted.

The new tagset would comprise from two parts: the **part of speech tag** equivalent to the UPOS tag (e.g. ADJ, VERB, AUX), and optional **feats** part, in the Brno tagset style, separated from the UPOS tag by a **:** sign. For example: *est*, a *VERB* with feats *Mood=Ind | Number=Sing | Person=3 | Tense=Pres | VerbForm=Fin* would translate into *VERB:nSp3vFtR*.

6.2 New fr_pipe

The tagging pipeline has been reworked as well, implementing the new, more informative, tagset, new dictionary, and improved vertical which makes the work for annotators a bit easier.

6.2.1 New Tagger

Since the UDPipe2 was selected as the new part of speech tagger, and it's online 2.6 version has shown some deficiencies, a new model based on UD French GSD v2.9 corpus was trained. The default training parameters were used, and the model went through 60 training epochs, reaching over 98 % accuracy on the development dataset, and over 97 % accuracy on the training dataset. Details are shown in table 6.3.

Table 6.3: Accuracy of the new model

Task	Dev data (Acc %)	Test data (Acc %)
UPOS	98.13	97.84
Lemma	98.74	98.55
UFeats	98.31	97.92

6.2.2 Tagging Pipeline

The pipeline had to be modified to accommodate the new tagger, since the UDPipe2 takes as input a tokenized vertical in the CoNLL-U format, and outputs it as well. The UNITOK tokenizer used previously with the TreeTagger was kept. A script for translating the traditional

.vert format into the .conllu, and vice versa .conllu into .vert, was written.

6.2.3 Post-processing

The post-processing used previously was kept as it was. Any further post-processing was added to the new vertical generating scripts, such as search for tokens in the new dictionary, coloring the cells, and automatic metadata filling.

6.3 New Vertical

As mentioned in section 4.4.2, the new vertical format offers an array of improvements over the old format. It mainly aims to achieve two goals: to make the annotation easier for the annotator, and making the annotation faster. Both of the goals also yield a beneficial byproduct of lowering the probability of the annotator making a mistake, since the number of opportunities to do so was significantly lowered.

6.3.1 New format

In the past, the Open Office .ods format was used mainly because the Open Office tools are widely available to anyone, and that the .ods can be used by most spreadsheet processors. However, the .ods was troubling at times, sometimes it couldn't be opened in MS Excel. Switching to .xlsx format seemed like a natural evolution, especially since every student and employee of the Masaryk University has an access to the license for the MS Excel, which so far didn't exhibit any erroneous behaviour, and thanks to the *xlsxwriter* library the python script can add more advanced formatting to the vertical.

6.3.2 Automatic Filling of Columns

Previously, only columns for token, tag, lemma, advanced tags (from dictionary), and sense (from dictionary) were automatically filled. The annotator still had to input all the other necessary columns manually, aside from checking the veracity of the machine input data. The user would annotate the data by opening the vertical, P and S versions of

the songs, and filling the required columns. The newly automated process saves the annotator the following work, which previously had to be done manually:

- User doesn't have to fill the column F with metadata about the song structure. This action could be done rather easily by copying the values, and saves at most a few minutes.
- User doesn't have to fill the column H with metadata about the current singer of the token. This is also a rather trivial task.
- User doesn't have to fill the column I with the information of the type of difference between the token in the P and S versions of the song. This is also accompanied by coloring the token in the same color as it was in the S version, making the orientation in the vertical easier. Student now only needs to fill the column J with the original version of the token in the P version. This automation doesn't necessarily save a lot of time, but makes the search for the phenomena much more convenient.
- User doesn't have to fill the column K with the phrasal tokenization, if the token is recognized as a phrase, which is now also done automatically. The amount of currently listed phrases in the dictionary is quite low (in the low hundreds), so the detection of phrases is still quite rare.

6.3.3 Summary

On account of saving the annotators' time, automated filling of columns will save the annotator roughly 10 to 60 minutes of their time based on the complexity of the vertical. Some songs need just a few minuscule touches, such as filling out the singer and song part, but some songs with heavy annotation, some extreme cases (typically songs with many foreign or slang phrases) contain tens of tokens needed to be annotated manually with filling out the additional columns. Also, with the addition of automated filling and the inclusion of new dictionaries, the risk of annotator making a mistake is significantly diminished.

6.4 Other

Apart from creating the new pipeline and vertical generation, some other work for RapCor has been done.

6.4.1 Search for Substandard Tags

Human error is a cause of a lot of noise in the corpus. In spite of both students' and teachers' diligence, some mistakes always slip through, especially since automated mistake detection in MS Office software hasn't been implemented yet. A simple script was written to scan through the entire corpus and look for any erroneous part of speech tags. Hundreds of mistakes in over 100 documents were found and manually corrected. Vast majority of them were typographic errors, where a student misspelled the tag or used uppercase/lowercase in a wrong way.

6.4.2 Tables of Codes

To help with keeping track of identification codes assigned for albums and songs, a script was written to create a spreadsheet of all possible codes with the additional data about which codes are already being used to identify either albums and songs, and to which interprets they relate.

6.4.3 Dictionary Update

RapCor used by now the *Verlan* dictionary. During the work on the new pipeline, new dictionary was implemented during generation of the vertical text along with scripts for automated updates of the dictionaries which are being continually worked on in the Google Spreadsheets.

6.4.4 Updating the Website

Some updates have been made also to the RapCor website. Mainly, the search engine for albums has been improved, along with the *findsong* web application students use to search for the song texts.

6.4.5 Evaluating the New Tagger

Since a new tagset was introduced with the new tagger, the idea of automated retagging of the RapCor verticals came up. This can of course be done manually, but such an endeavor would take exuberant amount of work. Thus, 46 songs from the corpus were selected and tagged using the UDPipe tagger. Results (see table 6.4.) were then compared with the annotated TreeTagger pretagged data, if there is high enough agreement of the taggers, so that the existing part of speech tags would be enhanced by the UDPipe provided features.

Table 6.4: Agreement of the taggers

Part of speech	86.34 %
Lemma	85.43 %

However, since the agreement of the part of speech tags wasn't high enough, detailed analysis of the agreement of the features wasn't performed. Table 6.5 shows the most common part of speech disagreements between the taggers at 24108 examined tokens.

Table 6.5: Ten most common tag disagreements

TreeTagger	UDPipe	Count
ADP	DET	222
PRON	SCONJ	115
NOUN	ADJ	113
ADV	PRON	103
VERB	NOUN	101
CCONJ	ADV	95
ADJ	VERB	90
CCONJ	ADP	85
ADJ	ADV	83
ADJ	NOUN	82

Note: from the comparison were excluded mistakes *VERB:AUX* and *CCONJ:SCONJ*, because TreeTagger doesn't differentiate between those categories. Foreign token were also excluded.

7 Conclusion

As a result of the work done for RapCor in the past year, plenty of new developments have been made. At the beginning the comparison of the part of speech taggers was done. FreeLing unfortunately didn't seem like a viable alternative to TreeTagger because of frequent mistakes and the need to run the software in the server mode. UDPipe was thus selected as the new part of speech tagger in spite of it's downsides.

Then new vertical format for RapCor was designed with the goal of making the work of annotators easier. Firstly, the .xlsx format was adopted instead of the .ods. Then, the vertical generating software was made, which pre-fills most of the data in the vertical. This eases the burden on the annotator's time, and decreases the probability of the human error. New dictionaries have been included as well, giving the annotator hints on the meaning of the substandard language.

Finally, the pipeline for generating tagged vertical was updated to use the new UDPipe2 tagger, which has been successfully trained on the UD French GSD Corpus.

Aside from the main part of the work, plenty of smaller updates were made on the corpus, along with many quality of life enhancements for the annotators. A search for substandard tags was made to find out mistakes in the corpus, tables of codes for songs and albums were created and posted online, new dictionaries were created to help the annotation, and the RapCor websites have been continually updated.

Looking towards the future, there is still a lot to improve on the RapCor pipeline. Most of the administrative data, including the dictionaries, is being managed using several Google Spreadsheets. Even though this approach is relatively intuitive and easy to use for the annotators, there is still much to be desired in terms of error detection, efficiency and most importantly - scalability. If RapCor continues to grow, a new, preferably database, solution is inevitable.

The new tagger itself isn't flawless and will make mistakes. Plenty of which are going to be mistakes of a kind previously not encountered with the TreeTagger. To remedy this, the annotators will have to vigilantly inspect the tagged data and report its mistakes. Another way to plausibly improve the tagger's performance would be retraining

the model on different data. The UD French GSD corpus could still be used, possibly joined with a spoken language corpus such as UD French Sequoia, or preferably, if a way is found, RapCor itself.

And finally the whole corpus should be converted into the new vertical form. If we suppose that each song is correctly annotated, translating the old TreeTagger tags into the new RapCor tags is in order. However, as shown in the previous chapter, this will probably not be a trivial task. Automatic pre-filling of the tags can be done, but they still must be manually checked to assure correctness. One solution might entail tagging of the texts by UDPipe and taking the UDPipe tags where they agree with TreeTagger at least on the part of speech level, the disagreements would either have to be manually settled by the annotator, or the original TreeTagger tag could be kept in a minimal form without the features.

Bibliography

1. CHOWDHARY, KR1442. Natural language processing. *Fundamentals of artificial intelligence*. 2020, pp. 603–649.
2. NADKARNI, Prakash M; OHNO-MACHADO, Lucila; CHAPMAN, Wendy W. Natural language processing: an introduction. *Journal of the American Medical Informatics Association*. 2011, vol. 18, no. 5, pp. 544–551.
3. HARRINGTON, Jonathan. *Phonetic analysis of speech corpora*. John Wiley & Sons, 2010.
4. ÁLVAREZ, Asunción; RITCHEY, Tom. Applications of general morphological analysis. *Acta Morphologica Generalis*. 2015, vol. 4, no. 1.
5. CHOMSKY, Noam. Systems of syntactic analysis. *The Journal of Symbolic Logic*. 1953, vol. 18, no. 3, pp. 242–256.
6. GODDARD, Cliff. *Semantic analysis: A practical introduction*. Oxford University Press, 2011.
7. KUMAR, Bhavesh; MARINGANTI, Hima Bindu; ASAWA, Krishna. Adaptive pragmatic analysis of natural language. In: *Proceedings of the First International Conference on Intelligent Interactive Technologies and Multimedia*. 2010, pp. 236–240.
8. KUMAWAT, Deepika; JAIN, Vinesh. POS tagging approaches: A comparison. *International Journal of Computer Applications*. 2015, vol. 118, no. 6.
9. ACEDAŃSKI, Szymon. A morphosyntactic Brill tagger for inflectional languages. In: *International Conference on Natural Language Processing*. 2010, pp. 3–14.
10. SCHMID, Helmut. Part-of-speech tagging with neural networks. *arXiv preprint cmp-lg/9410018*. 1994.
11. HOCHREITER, Sepp; SCHMIDHUBER, Jürgen. Long short-term memory. *Neural computation*. 1997, vol. 9, no. 8, pp. 1735–1780.

12. WANG, Peilu; QIAN, Yao; SOONG, Frank K; HE, Lei; ZHAO, Hai. Part-of-speech tagging with bidirectional long short-term memory recurrent neural network. *arXiv preprint arXiv:1510.06168*. 2015.

13. OLSTON, Christopher; NAJORK, Marc. *Web crawling*. Now Publishers Inc, 2010.

14. ZANETTIN, Federico. Parallel corpora in translation studies: Issues in corpus design and analysis. *Intercultural Faultlines*. 2000, pp. 105–118.

15. DE MARNEFFE, Marie-Catherine; MANNING, Christopher D; NIVRE, Joakim; ZEMAN, Daniel. Universal dependencies. *Computational linguistics*. 2021, vol. 47, no. 2, pp. 255–308.

16. PODHORNÁ-POLICKÁ, Alena. RapCor, Francophone Rap Songs Text Corpus. In: *RASLAN*. 2020, pp. 95–102.

17. SCHMID, Helmut. TreeTagger-a language independent part-of-speech tagger. *http://www. ims. uni-stuttgart. de/projekte/corplex/TreeTagger/*. 1994.

18. BHARDWAJ, Rupali; VATTA, Sonia. Implementation of ID3 algorithm. *International Journal of Advanced Research in Computer Science and Software Engineering*. 2013, vol. 3, no. 6.

19. FORNEY, G David. The viterbi algorithm. *Proceedings of the IEEE*. 1973, vol. 61, no. 3, pp. 268–278.

20. MICHELFEIT, Jan; POMIKÁLEK, Jan; SUCHOMEL, Vıt. Text Tokenisation Using unitok. In: HORÁK, Aleš; RYCHLÝ, Pavel (eds.). *RASLAN 2014*. Brno, Czech Republic: Tribun EU, 2014, pp. 71–75. ISBN 2336-4289.

21. PADRÓ, Lluıs; STANILOVSKY, Evgeny. Freeling 3.0: Towards wider multilinguality. In: *LREC2012*. 2012.

22. BRANTS, Thorsten. TnT-a statistical part-of-speech tagger. *arXiv preprint cs/0003055*. 2000.

23. PADRÓ, Lluıs. A hybrid environment for syntax-semantic tagging. *arXiv preprint cmp-lg/9802002*. 1998.

24. CARRERAS, Xavier; MARQUEZ, Lluis; PADRÓ, Lluıs. Named entity extraction using adaboost. In: *COLING-02: The 6th Conference on Natural Language Learning 2002 (CoNLL-2002)*. 2002.

25. AGIRRE, Eneko; SOROA, Aitor. Personalizing pagerank for word sense disambiguation. In: *Proceedings of the 12th Conference of the European Chapter of the ACL (EACL 2009)*. 2009, pp. 33–41.

26. ATSERIAS, Jordi; RODRIGUEZ, Horacio. Tacat: Tagged corpus analizer tool. *Technical report Isi-98-2-t*. 1998.

27. ATSERIAS BATALLA, Jordi; COMELLES PUJADAS, Elisabet; MAYOR MARTINEZ, Aingeru. TXALA un analizador libre de dependencias para el castellano. *Procesamiento del lenguaje natural, nº 35 (sept. 2005); pp. 455-456*. 2005.

28. SOON, Wee Meng; NG, Hwee Tou; LIM, Daniel Chung Yong. A machine learning approach to coreference resolution of noun phrases. *Computational linguistics*. 2001, vol. 27, no. 4, pp. 521–544.

29. STRAKA, Milan. UDPipe 2.0 Prototype at CoNLL 2018 UD Shared Task. In: *Proceedings of the CoNLL 2018 Shared Task: Multilingual Parsing from Raw Text to Universal Dependencies*. Brussels, Belgium: Association for Computational Linguistics, 2018, pp. 197–207. Available from DOI: 10.18653/v1/K18-2020.

30. CHO, Kyunghyun; VAN MERRIËNBOER, Bart; BAHDANAU, Dzmitry; BENGIO, Yoshua. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*. 2014.

31. LING, Wang; LUIS, Tiago; MARUJO, Luıs; ASTUDILLO, Ramón Fernandez; AMIR, Silvio; DYER, Chris; BLACK, Alan W; TRANCOSO, Isabel. Finding function in form: Compositional character models for open vocabulary word representation. *arXiv preprint arXiv:1508.02096*. 2015.

32. DOZAT, Timothy; QI, Peng; MANNING, Christopher D. Stanford's graph-based neural dependency parser at the conll 2017 shared task. In: *Proceedings of the CoNLL 2017 shared task: Multilingual parsing from raw text to universal dependencies*. 2017, pp. 20–30.

Index

A An appendix

In this section will be listed and described all scripts made in scope of this thesis. Source codes are freely available with the thesis under the GNU GPL license. However, python scripts released with the thesis are not accompanied by any RapCor data. The source codes could be split into two categories, the scripts used for tagging and creating new vertical, and one time use scripts which were quickly made to solve a quick problem. These scripts lack any documentation and were never meant to be used again.

A.1 create_tsv_dictionary.py

create_tsv_dictionary.py is a script used for processing the new dictionary, downloaded from Google Spreadsheets as .ods, into .tsv format. The original spreadsheet has several lists which are joined together into one large spreadsheet used during creation of the vertical.

```
$ python3 create_tsv_dictionary.py
   dictionary_valide.ods dictionary_valide.tsv
```

A.2 compile_xlsx.py and parse_html.py

New vertical is created by the following process: input in the form of .docx document is received from the .cgi script, and converted to .html using the *lowriter*; .html is then passed to the *parse_html.py* to create the text, contents, and styles .txt files. Text contains song text sent to the fr_pipe for tagging. Styles contains information about all coloring styles in the document. And contents holds information about which tokens have which styles, which part of song do they belong to, and who sings them. These files plus the tagged vertical are passed to the *compile_xlsx.py* which creates the final .xlsx vertical and the .cgi scripts sends it to the user.

```
$ python3 parse_html.py song.html

$ python3 compile_xlsx.py song.vert song.xlsx
   dictionary.tsv
```

A.3 fr_pipe_v2

fr_pipe_v2 is the modernized version of the pipeline. It substitutes the TreeTagger with the UDPipe, and two vertical conversion scripts. The tokenizer produces vertical text in TreeTagger format, but UDPipe needs input in the .conllu vertical, which is solved by *vert_to_conllu.py* script. UDPipe then outputs tagged .conllu vertical which is then converted back into TreeTagger vertical by *conllu_to_vert.py*, which also translated UD tags into new RapCor tags using the *tag_convertor.py* script.

```
$ ./fr_pipe_v2 < song.txt > song.vert
```

A.4 generate_codes_tsv.py

This script loads up *RAPCOR-Knihovna.tsv*, and *rapcor_albums.js* tables which are being automatically generated from Google Spreadsheets, and creates table of all codes in .tsv, and then uses it to write tables of codes in .html format, which are then put online.

```
$ python3 generate_codes_tsv.py
    RAPCOR-Knihovna.tsv rapcor_albums.js
    codes.tsv ./html/
```

A.5 Utils

Utils contain all sorts of python scripts which were intended for one time use and lack any documentation. These contain scripts for tagger comparison (compare), creating test batch for the tagger comparison (create test batch), comparing trained UDPipe2 with the manual annotated data (evaluation), creating .xslx spreadsheets of RapCor codes (get codes list), analysing tagsets of UD corpora (get tagset), and searching for erroneous tags in the corpora (search for substandard tags).

A.6 UDPipe2

The UDPipe2 was used both for training and tagging. The training was launched with the following parameters:

```
$ python3 udpipe2.py fr-gsd-model
   --train fr_gsd-ud-train.conllu
   --dev fr_gsd-ud-dev.conllu
   --test fr_gsd-ud-test.conllu
   --parse 0
```

However, since the udpipe2.py script doesn't take the input file from stdin and doesn't ouptut it to stdout, a variant called udpipe2_cmd.py was created just for that purpose, so it can be included in the pipeline. There it is launched with the following parameters:

```
$ python3 udpipe2_cmd.py fr-gsd-model --predict
   --parse 0 < input.conllu > output.vert
```