PyRATA, Python Rule-based feAture sTructure Analysis

Nicolas Hernandez and Amir Hazem
Laboratoire des Sciences du Numérique de Nantes (LS2N)
Université de Nantes, 44322 Nantes Cedex 3, France
Nicolas.Hernandez@univ-nantes.fr

Abstract

In this paper, we present a new Python 3 module named *PyRATA*, which stands for "*P*ython *R*ule-based fe*A*ture s*T*ructure *A*nalysis". The module is released under the Apache V2 license. It aims at supporting rules-based analysis on structured data. PyRATA offers a language expressiveness which covers the functionalities of all the concurrent modules and more. Designed to be intuitive, the pattern syntax and the engine API follow existing standard definitions; Respectively Perl regular expression syntax and Python re module API. Based on simple native Python data structures (i.e. sequence of feature sets), PyRATA is able to deal with various kinds of data (textual or not) at various levels, such as a list of words, a list of sentences, a list of posts in a forum thread, a list of events in a calendar... This specificity makes it free from any (linguistic) process.

Keywords: rules-based analysis, semantic annotations, regular expression, information extraction, Python 3

1. Introduction

Rules-based approaches must not be set in opposition to machine-learning-based (ML) approaches. The former provide the advantages of quickly generating some self-explanatory results with a few rules, even with little expert knowledge and no training data. But they have the pitfalls of becoming difficult to maintain with the number and the complexity of rules increasing. Cleverly trained, ML models are capable of generalizing i.e. they present the ability to perform well on new unseen data and offers so a larger recall performance. But their major drawbacks are to depend on large quantity of training data which often results from a costly manual annotation task. In addition, their decisions are harder to explain.

Rules are currently unpopular in Natural Language Processing (NLP) research community, but there are still good reasons to use them: 1) Since they do not require training data, they are often a good first cut to explore the data and define more precisely a problem; 2) In some languages, some problems can be done deterministically with rules pretty well; 3) To prototype a model by using rules and use it to generate training data for a ML system; 4) To define features (feature engineering) and let the ML component learn how to combine them; 5) To augment ML models with rules in pre- or post-processing stages to achieve an algorithm of 100 % performance (Manning, 2011). The rules are so used to tune input/output and handle some specific unwanted system behaviors.

To the best of our knowledge, the NLP community benefits from two software solutions[1] which allow to define patterns of annotations with some additional actions to perform on the matched annotations, namely *GATE JAPE*[2] (Cunningham et al., 1999) and *UIMA RUTA*[3] (Kluegl et al., 2016). Apart from learning the rule languages, the use of these options is not straightforward since both of them are part of a global text analysis framework, and consequently assume a basic understanding of the framework concepts as well as some technical skills (e.g. Eclipse workbench). Accidentally the programmer would have to develop in Java to integrate them in their own solution.

Python users do not benefit from the same advanced tools. At least they have some modules to formulate search patterns and extract resulting matches namely the *Python nltk chunk* module (Bird, 2006), the *clips pattern.search* (De Smedt and Daelemans, 2012) and the *spaCy* module.

In this paper, we present a new Python 3 module named *PyRATA*, which stands for "*P*ython *R*ules-based fe*A*ture s*T*ructure *A*nalysis". The module is released under the Apache V2 license. It aims at supporting rules-based analysis on structured data. PyRATA offers a language expressiveness which covers the functionalities of all the concurrent modules and more. Designed to be intuitive, the pattern syntax and the engine API follow existing standard definitions; Respectively Perl regular expression syntax and Python re module API. Using a simple native Python data structure (i.e. sequence of feature set) allows it to deal with various kinds of data (textual or not) at various levels, such as a list of words, a list of sentences, a list of posts of a forum thread, a list of events of a calendar... This specificity makes it free from any (linguistic) process.

Section 2. describes the PyRATA pattern language, the module API and its implementation. And Section 3. discusses PyRATA with respect to the other various Python alternatives.

2. PyRATA module

Regular expressions (RE) are traditionally known to define patterns of possible sequences of characters, which are in turn used by search algorithms on strings for actually finding matching sequences. In Natural Language Processing (NLP), RE are the essence of the rules-based approach for text analysis.

[1] We did not consider here environments such as Nooj (Silberztein, 2005) http://www.nooj4nlp.net or Unitex (Paumier et al., 2009) http://unitexgramlab.org which are deeply rooted in linguistic analysis.

[2] https://gate.ac.uk/sale/tao/splitch8.html, Java 8, GNU

[3] https://uima.apache.org/ruta.html, Java 8, Apache v2

```
1 >>> import nltk
  >>> data = 'Chuck Norris is cooler than Dolph Lundgren.'
3 >>> pyrata_data = [{'raw':word, 'pos':pos, 'lem':nltk.WordNetLemmatizer().lemmatize(word.
    lower())} for (word, pos) in nltk.pos_tag(nltk.word_tokenize(data))]
```

Figure 1: Generating a PyRATA data structure from a String (here used to store a simple sentence) using Python and the nltk module.

```
1 >>> pyrata_data
  [{'raw': 'Chuck', 'pos': 'NNP', 'lem': 'chuck'},
3  {'raw': 'Norris', 'pos': 'NNP', 'lem': 'norris'},
   {'raw': 'is', 'pos': 'VBZ', 'lem': 'is'},
5  {'raw': 'cooler', 'pos': 'JJR', 'lem': 'cooler'},
   {'raw': 'than', 'pos': 'IN', 'lem': 'than'},
7  {'raw': 'Dolph', 'pos': 'NNP', 'lem': 'dolph'},
   {'raw': 'Lundgren', 'pos': 'NNP', 'lem': 'lundgren'},
9  {'raw': '.', 'pos': '.', 'lem': '.'}]
```

Figure 2: Example of a PyRATA Data Structure resulting from the process described in Figure 1.

2.1. The data structure

A string can be seen as somehow a list of character tokens. But a character string is a poor data structure while in many cases the matter of interest are linguistic phenomena at various text levels (e.g. words, sentences, ...).

The main innovative feature of PyRATA is may be to deal with *lists* of *associative arrays* (i.e. `list` of `dicts` in the Python jargon). The dict data structure is a set of values indexed by unique keys (i.e. a name-value feature set).

Figure 2 illustrates the data structure[4] the PyRATA engine takes in input. Line 3 Figure 1 shows how to generate it in one simple instruction line in Python; Here thanks to the nltk module. The process takes the `String` sentence in line 2, then performs a tokenization, pos tagging and lemmatization, and stores the result in a `list` where each value is a `dict` which represents the various forms of a word of the given sentence. The names, `raw`, `pos`, and `lem`, are freely chosen to mean the feature keys.

2.2. Pattern syntax

The pattern syntax used by PyRATA is a subset of the PERL regular expression syntax. The subset is also common to the POSIX extended syntax. This subset suffices to describe all regular languages. In some aspects, the syntax looks like the language used to define queries in Corpus Query Processor (CQP) of the IMS Open Corpus Workbench (CWB) (Christ, 1994)[5].

A *pattern* can be seen as a sequence of elements separated by whitespace characters. In its minimal form, an *element* specifies a *constraint* on the value of a feature that a data token should satisfy. For a given feature name, you can specify its required exact value (e.g. `raw="Chuck"`), a regular expression definition of its value (e.g. `pos~"JJ.?"`), a list of possible values –a lexicon– (e.g. `lemma@"POSITIVE"`) or if the value correspond to a IOB[6] tag and should match a sequence consequently (e.g. `chunk-"NP"`).

More complex elements are quantified elements, element classes, groups of elements, alternatives or combinations of these various types. A *quantified element* allows to specify *optional* elements (`?`), elements which should occur *at least one* (+), or *zero or more* (⋆). An *element class* aims at specifying a logical combination of minimal constraints on the data token features. The combination is delimited by squared brackets (`[]`) and the constraints are combined with usual logical operators namely *parenthesis* (`()`) and logical connectors such as *and* (&), *or* (|) and *not* (!). A *group of elements* is delimited by parenthesis (`()`) and can be used to refer to and process subsequently subparts of a match. An *alternative* defines expected sets of data token candidates at a specific point of the data stream. The *wildcard element* can be set by a dot character.

Line 1 and 2 of Figure 3 show some concurrent patterns to define noun phrases chunks. They illustrate the use of equal and regular expression constraints as well as element classes and quantifiers. After importing the PyRATA module (line 3), line 4 shows the search instruction of the latter pattern over the previously defined data structure. Figure 4 shows the result as a list of `Match` objects, each one made of the matched sequence and its offsets.

2.3. Finding and editing operations

The API is developed to be familiar for whom who develops with the *python re module*[7] API. The module defines several common functions such as `search`, `findall`, or `finditer`. The functions are also available for compiled regular expressions. They generally take at least two arguments including the pattern to recognize and the data to explore (e.g. `re.search(pattern, data)` or `compiledPattern.search(data)`). Depending

[4] In Python, squared brackets delimit list while curly brackets mark dicts. The feature of a dict are separated by a comma and the feature name and its value are separated by a colon.

[5] `cwb.sourceforge.net`

[6] I for Inside, B for Begin and O for Other

[7] `https://docs.python.org/3/library/re.html`, Python 3, PSF

```
>>> pyrata_np_pattern = 'pos="DT"? [pos="JJ"|pos="NN"]* [pos="NN"|pos="NNS"|pos="NNP"]+'
>>> pyrata_np_pattern = 'pos="DT"? pos~"JJ|NN"* pos~"NN.?"+'
>>> import pyrata.re as pyrata_re
>>> found_noun_phrases = pyrata_re.finditer(pyrata_np_pattern , pyrata_data)
```

Figure 3: Recognizing simple Noun Phrases patterns with PyRATA (taking the PyRATA data structure of Figure 1 as input).

```
>>> found_noun_phrases
<pyrata.re MatchesList object; matcheslist="[
  <pyrata.re Match object; groups=[[[{'raw': 'Chuck', 'pos': 'NNP', 'lem': 'chuck'}, {'raw
    ': 'Norris', 'pos': 'NNP', 'lem': 'norris'}]], 0, 2]]>,
  <pyrata.re Match object; groups=[[[{'raw': 'Dolph', 'pos': 'NNP', 'lem': 'dolph'}, {'raw
    ': 'Lundgren', 'pos': 'NNP', 'lem': 'lundgren'}]], 5, 7]]>
]">
```

Figure 4: Display of a PyRATA `MatchesList` object resulting from the `finditer` operation in Figure 3.

on the selected function, the result can be a featured object which stands for the first match of a given pattern, all the non-overlapping matches of a pattern, or an iterator yielding match objects over all non-overlapping matches of a pattern. Figure 3 and 4 illustrate the `finditer` method returning an iterator yielding match objects.

In addition to the exploration methods, the module offers methods to edit the data structure either by substitution (`sub`), update (`update`) or extension (`extend`) of the data feature structures. These methods are very common in NLP annotation tasks.

Figure 2 shows that the nltk lemmatizer did not assign the correct lemma to the verb *to be*. Figure 5 illustrates how to easily fix this hassle thanks to the `update` method. Figure 6 illustrates the annotation method `extend` to add new features to token in order to create IOB tags corresponding to a noun phrase (NP) chunk. It uses the NP pattern defined in Figure 3. Since the resulting data structure is also a PyRATA data structure, such methods can be used to simulate transducers and rewriting rules.

Line 1 Figure 7 defines two lexicons while Line 2 both illustrates the combination in one pattern of the lexicons and IOB-chunk constraints, as well as the use of a sequence group. PyRATA assigns a group id to the chunks. Group 0 corresponds to the whole match.

More examples are available in the user documentation[8] and the demo directory of PyRATA.

2.4. Implementation

Since version `v0.4`, PyRATA has been based on a simple implementation[9] of Thompson's algorithm which aims at converting Regular Expressions (RE) to Non-deterministic Finite Automata (NFA) and simulate them in a multiple-state approach which offers a linear time efficiency in $\mathcal{O}(n)$ (Thompson, 1968).

The following description is based on the explanations of Russ Cox[10]. Every regular expression has an equivalent NFA. The NFA is built up from partial NFAs for each subexpression, with a different construction for each operator. The partial NFAs have no matching states: instead they have one or more dangling arrows, pointing to nothing. The construction process will finish by connecting these arrows to a matching state.

Running an NFA using some data as input requires tracking the current states that the NFA is in, and the next set of states that the NFA will be in, after processing the current token. The pattern is matched if the NFA reaches a final state.

For a regular expression of length m run on text of length n, the Thompson NFA requires $\mathcal{O}(mn)$ time. Since we only need to scan the pattern string once in order to build the corresponding NFA, the time efficiency for this step is linear $\mathcal{O}(n)$.

3. Comparison to alternatives

This section deals with the differences between the various Python alternatives to perform regular expressions over annotations.

Table 1 summarizes the main functionalities of the alternatives Python modules. Data structure dependency is set to dependent if the module works on an internal data structure. Concerning *clips.pattern* there was no successful straightforward `2to3` conversion.

The Python *nltk chunker*[11] module (Bird, 2006) offers to define chunk and chink grammars on word POS tags consisting of rules (expressed in regular-expression style) that indicate how sentences should be chunked (e.g. `"NP: {<DT>?<JJ>*<NN>}"`). By respecting the (Python) data structure, the engine can be applied on vari-

[8] `https://github.com/nicolashernandez/PyRATA/blob/master/docs/user-guide.rst`

[9] Gui Guan, "A Beautiful Linear Time Python Regex Matcher via NFA", August 19, 2014, `https://www.guiguan.net/a-beautiful-linear-time-python-regex-matcher-via-nfa/`, free to use and modify in any way.

[10] Russ Cox, "Regular Expression Matching Can Be Simple And Fast (but is slow in Java, Perl, PHP, Python, Ruby, ...)', January, 2007, `https://swtch.com/~rsc/regexp/regexp1.html`, Last consulted on September 30th, 2017

[11] `http://www.nltk.org/book/ch07.html` and `http://www.nltk.org/_modules/nltk/chunk/regexp.html`, Python 3, Apache v2

```
>>> updated_pyrata_data = pyrata_re.update ('[lem="is" | lem="are" | lem="wa"]', {'lem':'
    be'}, pyrata_data)
[{'raw': 'Chuck', 'pos': 'NNP', 'lem': 'chuck'},
 {'raw': 'Norris', 'pos': 'NNP', 'lem': 'norris'},
 {'raw': 'is', 'pos': 'VBZ', 'lem': 'be'},
 {'raw': 'cooler', 'pos': 'JJR', 'lem': 'cooler'},
 {'raw': 'than', 'pos': 'IN', 'lem': 'than'},
 {'raw': 'Dolph', 'pos': 'NNP', 'lem': 'dolph'},
 {'raw': 'Lundgren', 'pos': 'NNP', 'lem': 'lundgren'}]
```

Figure 5: Updating (edit operation) a PyRATA data structure by modifying a feature value. The PyRATA data structure of Figure 1 is taken as input.

```
>>> extended_pyrata_data = pyrata_re.extend (pyrata_np_pattern, {'chunk':'NP'},
    updated_pyrata_data, iob=True)
[{'raw': 'Chuck', 'chunk': 'B-NP', 'pos': 'NNP', 'lem': 'chuck'},
 {'raw': 'Norris', 'chunk': 'I-NP', 'pos': 'NNP', 'lem': 'norris'},
 {'raw': 'is', 'pos': 'VBZ', 'lem': 'be'},
 {'raw': 'cooler', 'pos': 'JJR', 'lem': 'cooler'},
 {'raw': 'than', 'pos': 'IN', 'lem': 'than'},
 {'raw': 'Dolph', 'chunk': 'B-NP', 'pos': 'NNP', 'lem': 'dolph'},
 {'raw': 'Lundgren', 'chunk': 'I-NP', 'pos': 'NNP', 'lem': 'lundgren'}]
```

Figure 6: Extending (edit operation) a PyRATA data structure by adding a new feature. The PyRATA data structure of Figure 1 is taken as input.

```
>>> lexicons = {'POSITIVE':['cooler', 'stronger'], 'NEGATIVE':['poor', 'worst']}
>>> pyrata_re.finditer('chunk-"NP" (lem="be" [raw@"POSITIVE"  & !raw="than"]* raw="than")
    chunk-"NP"' , extended_pyrata_data, lexicons = lexicons).group().group(2)
[{'raw': 'is', 'pos': 'VBZ', 'lem': 'be'}, {'raw': 'cooler', 'pos': 'JJR', 'lem': 'cooler'
    }, {'raw': 'than', 'pos': 'IN', 'lem': 'than'}]
```

Figure 7: Defining groups and lexicons in PyRATA syntax.

ous types of information (one at a time) as well as be cascaded.

The *CLIPS pattern.search*[12] module (De Smedt and Daelemans, 2012) has a pattern matching system similar to regular expressions, that can be used to search a string by syntax (word POS and chunk tags) or by lexical (word surface, lemmata and semantical categories) constraints (e.g. "`DT? JJ?+ NN`"). *CLIPS pattern.search* is actually the best choice in terms of pattern language expressiveness. Unfortunately, the language and the engine are not well separated from the clips internal data structure which prevents from using it with external processing.

The *spaCy*[13] module features a rule-matching engine that operates over tokens, similar to regular expressions. The rules can refer to token annotations and flags, and matches support callbacks to accept, modify and/or act on the match. spaCy is written in Python and Cython. Despite its astonishing performance, the use of the spaCy rule-matching engine is not easy out of the box. Indeed, it is strongly attached to the spaCy internal data structure and without entering the code, it cannot go further than addressing simple chunk extractions. In addition, the matches cannot serve as input of new patterns.

Table 2 gives the time performance of searching similar patterns on each module. Due to the expressiveness differences, only a minimal pattern expression can be used to compare the modules. This is the role played by the first line dedicated to each module in the table. This minimal noun phrase definition assumes that an NP is ending by a noun which can be preceded by an optional determiner and zero or more adjectives in this order. Figure 8 depicts the results for this four systems. Other NP definition variants allow to consider various adjective and noun forms as well as nouns as alternative to adjectives. These patterns illustrate the syntax of each module. To address the comparison, we used the Brown corpus as data (1,161,192 words) and measure the average time over 10 runs to process the n first words with n varying from 10,000 to 1,000,000. The experience was performed on a processor i7-4600U running at 2.10GHz. For *nltk chunker* and PyRATA we used the original Brown corpus word tokenization and the default nltk POS tagger in English. In order to use *clip.pattern* and *spaCy*, we joined the Brown words in a single string and performed the tokenization and the pos

[12] `https://www.clips.uantwerpen.be/pages/pattern-search` and `https://github.com/clips/pattern`, python 2.6, BSD-3

[13] `https://spacy.io/docs/usage/rule-based-matching`Â and `https://github.com/explosion/spaCy`, Python 3, MIT

	PyRATA	clips.pattern	nltk_chunk_regexp	spaCy
Python version	3	2	2 and 3	2 and 3
Token type	no restriction	word/chunk token	word/chunk token	word token
Data structure dependency	Python native	dependent	Python native	dependent
Feature type	no restriction	restricted to some syntax and semantic	no restriction but a single one at a time	some predefined, extensible with code
Pattern language	declarative	declarative	declarative	declarative + code
Wildcard	yes	yes	yes	
Quantifiers	yes	yes	yes	yes
Element alternative	yes	yes	yes	no
Group	yes	yes	no	no
Group alternative	yes	no	no	no
Match offsets	yes	no	no	yes
Cascade matcher	yes	no	yes	no
Find operations	search, match, findall	search, match, findall	findall	findall
Edit operations	sub, update, extend	no	extend	no

Table 1: Module functionalities comparison.

noun phrase patterns	10k	50k	100k	200k	300k	500k	750k	1,000k
CLIPS pattern.search								
`DT? JJ?+ NN+`	0.145	0.714	1.510	3.113	4.484	7.585	11.700	15.515
`DT? JJ|NN?+ NN|NNS`	0.212	1.008	2.110	4.675	6.328	10.825	16.712	22.224
`DT? JJ|NN?+ NN|NNS+`	0.220	1.033	2.170	4.493	6.538	11.069	17.929	22.879
nltk chunker								
`<DT>?<JJ>*<NN>+`	0.037	0.521	1.842	7.146	15.194	43.481	97.216	169.396
`<DT>?<JJ|NN>*<NN|NNS>`	0.043	0.652	2.386	9.574	20.130	58.745	127.489	228.397
`<DT>?<JJ|NN>*<NN.*>`	0.062	1.137	4.190	15.502	32.720	89.219	200.388	334.670
PyRATA								
`pos="DT"? pos="JJ"* pos="NN"+`	1.681	8.350	17.089	33.773	53.506	91.840	131.540	176.272
`pos="DT"? [pos="NN" | pos="JJ"]* [pos="NN" | pos="NNS"]`	2.132	10.662	21.413	42.731	65.062	114.885	167.478	222.817
`pos="DT"? [pos~ "NN|JJ"]* pos~"NN.*"`	1.854	9.695	18.872	40.288	58.423	101.974	141.782	189.163
spaCy								
`[{POS:"DET",'OP':"?"}, {POS:"ADJ",'OP':"*"},{POS:"NOUN",'OP':"+"}]`	0.002	0.004	0.008	0.020	0.0266	0.048	0.070	0.086

Table 2: Time performance (in seconds) for recognizing various Noun Phrases patterns on the first n Brown corpus words.

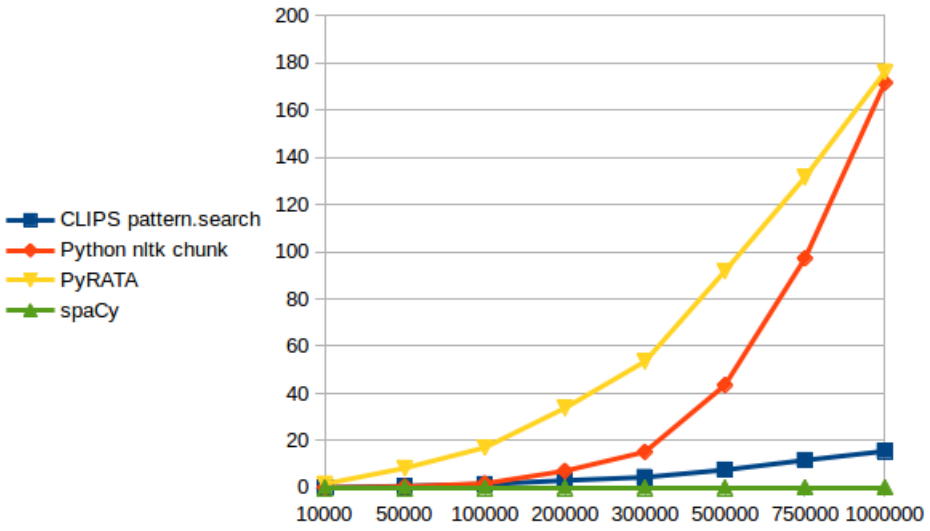

Figure 8: Selection of some system performances from Table 2. Only assumed identical patterns (first line of each module) are presented. Time (in seconds) vs quantity of processed words.

tagging processes offered by these modules. We used the `en_core_web_sm-1.2.0` model for *spaCy*. We could have hacked the *spaCy* tokenizer by making it segment only on the whitespaces and consequently format the Brown corpus so that *spaCy* would eventually obtain the same tokenization as the one already present in nltk. Since the resulting amount of tokens was pretty much the same we did not follow this path. As a result, we observe that there is nothing compare to the time performance of spaCy and right now we can say the same about the PyRATA language expressiveness. Nevertheless, if a few minutes to process one million words can largely be acceptable depending on the use cases it is a fact that the PyRATA holds the worse position in term of time performance. Surprisingly, we also observe that the *nltk chunker* time processing increases and converges toward the PyRATA one. We did not experiment with more data.

4. Conclusion

PyRATA is *fun and easy to use* to explore data for research or pedagogical motivations, define machine learning features, pre/post process machine learning based analysis, formulate expert knowledge in a declarative way.

Various directions exist to optimize further the engine code such as parallel programming, or caching the lists of states corresponding to effectively explored and possibly recurrent DFA (avoiding the cost of future computation repetitions). The language can also be extended like for instance the management of back-references.

Acknowledgments

The current work was supported by the ANR 2016 PASTEL[14]. The authors would like to thank the anonymous reviewers for their valuable comments.

5. Bibliographical References

Bird, S. (2006). Nltk: The natural language toolkit. In *Proceedings of the COLING/ACL on Interactive Presentation Sessions*, COLING-ACL '06, pages 69–72, Stroudsburg, PA, USA. Association for Computational Linguistics.

Christ, O. (1994). A modular and flexible architecture for an integrated corpus query system. In *Proceedings of COMPLEX'94: 3rd Conference on Computational Lexicography and Text Research*, pages 23–32, Budapest, Hungary. tt cmp-lg: tt 9408005.

Cunningham, H., Cunningham, H., and Tablan, V. (1999). Jape: a java annotation patterns engine.

De Smedt, T. and Daelemans, W. (2012). Pattern for python. *Journal of Machine Learning Research*, 13:2063–2067.

Kluegl, P., Toepfer, M., Beck, P.-D., Fette, G., and Puppe, F. (2016). Uima ruta: Rapid development of rule-based information extraction applications. *Natural Language Engineering*, 22:1–40, 1.

Manning, C. D., (2011). *Part-of-Speech Tagging from 97% to 100%: Is It Time for Some Linguistics?*, pages 171–189. Springer Berlin Heidelberg, Berlin, Heidelberg.

Paumier, S., Nakamura, T., and Voyatzi, S. (2009). UNITEX, a Corpus Processing System with Multi-Lingual Linguistic Resources. In *eLexicography in the 21st century: new challenges, new applications (eLEX'09)*, pages 173–175, October.

Silberztein, M. (2005). Nooj: A linguistic annotation system for corpus processing. In *Proceedings of HLT/EMNLP on Interactive Demonstrations*, HLT-Demo '05, pages 10–11, Stroudsburg, PA, USA. Association for Computational Linguistics.

Thompson, K. (1968). Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, June.

[14] `http://www.agence-nationale-recherche.fr/?Projet=ANR-16-CE33-0007`