NLU Assignment 1

Dimitri Vinci (223960)
University of Trento
Via Sommarive, 9, 38123 Povo,Trento TN
dimitri.vinci@studenti.unitn.it

Abstract

This is the report for the first assignment of NLU. The document is divided in a mandatory part containing the descriptions of the functions implemented. The Advanced part accounts for the implementation of the advanced part of the assignment. In conclusion it will be reported the comparisons regarding old and new implementations and some observations about the mandatory and advanced part.

1 Mandatory part

the first function (esericizio1) take as input

- a phrase in string format

the output is a dictionary that contains for each token extracted from the sentence

- as key the text of the the token itself or if there are more than one token with the same text, the token text followed by the position of the token in the original document
- as value the list of the token dependency of the ancestors from the root to the token itself

the implementation is the following:

1. parse the phrase using spacy
2. iterate through all the tokens of the parsed document
3. create the key of the dictionary (also checking if the key is already present to create an alternative key)
4. for each token in the loop extract the generator of the ancestors with the method token.ancestors
5. iterate over all the ancestors saving the value of dependency (token.dep_) in a list and then reverse the order of the list
6. save the list as value of the dictionary with, as key, the corresponding token in the cycle

the second function take as input

- a phrase in string format

the output is a dictionary that contains for each token extracted from the sentecne

- as key the text of the the token itself or if there are more than one tokens with the same text, the token text followed by the position of the tokens in the original document
- as value the list of the token text of the descendants ordered with the order of the original document

the implementation is the following:

1. parse the phrase using spacy
2. iterate through all the tokens of the parsed document
3. create the key of the dictionary (also checking if the key is already present to create an alternative key)
4. for each token in the loop extract the generator of the descendants with the method token.subtree
5. iterate over all the descendant saving them in a list
6. save the list as value of the dictionary with, as key, the corresponding token in the cycle

the third function take as input

- a parsed phrase
- a wordlist taken from the parsed phrase

the output is a bolean stating if the sequence forms a subtree (indipendently of the order of the words in the sequence).
The implementation is the following

1. sort the wordlist
2. iterate on each token of the document and extract the generator object of the subtree
3. obtain all the element of the subtree in a list and order it
4. compare the wordlist and the list of the subtree

the fourth function take as input

- a sequence in string or wordlist format rapresenting a pattern span
- a parsed document (optional) forming the context from witch span was extracted

the output is

- if all argument were setted, a dictionary having as key the match id of the pattern founded in the document and as value the root of theese matches
- if only the wordlist was setted only the root of the span as a token

the implementation is the following:

1. if sequence is a list convert it to string
2. check if context was setted

then if it was not set

3. parse the phrase
4. put in a span
5. extract the root if it was set
3. create a phrasematcher using as pattern the
wordlist
4. call the matcher on the context
5. extract the matches id start and and and to
construct the result dictionary
6. output the dicitionary

the fifth function take as input a phrase to parse
the output is a dictionary containing

- as a key the string identifier of the dependencies dobj, iobj, nsubj
- as value the list of span that satisifies this relations

the implementation is the following

1. parse the document and create the dictionary (and also the keys)
2. create a matcher and inside it add three different patterns, one for each of the dependencies that we want to extract
3. find the matches with the matcher. The matches are automatically adedd to the correct key of the dictionaries using inside defined functions as

callbacks in the case of the matching of one of the patterns(in particular we have three different callbacks for the three different dependencies that we want to retrieve)

2 Advanced part

The first point of the advanced part was faced trying add and combine different sets of feature and comparing results as described in the following chapter. The sets of features (I will use the notation of Nivre et all. in [3]) were composed with

- The original features
- Extend features also to subsequent elements of the stack and buffer (for example FORM, LEMMA, POSTAGS, FEATS and DEPREL on STK[1], BUF[1], BUF[2], BUF[3])
- Increasing elements on buffer and stack (increasing to STK[3] and BUF[4], BUF[5])
- Trying composed type of features (as advocated by Jurafksy et all.[2] and Zhang et all. [4])
- Adding some new features (for example the depth of the farthest child of the w on left and right)

The second part lead to the substitution of support vector machine classifier with a multinomial logistic regression as pointed out by Jurafsky[2] as one of the standard classifier used in the topic because of the good performance on sparse features.

3 Conclusions and observations

Regarding the mandatory part some observation could be done

- Firstly in the implementations I was intentionally longwinded to make more clear the logic inside it
- Regarding first function: the function must iterate in a double cycle to obtain the results so the complexity (ignoring spacy functions complexity) is O(n^2). Other implementations can possibly be done in a breadth first fashion or without using ancestors . In the second case you simply follow the link so the complexity is approximately the same because you have to cycle from the token to the root . You could also do a search in all tree from the root to the tokens with a breadth first search (recursive or iterative) and in this case the complexity is exponential in the worst case but you have not to order at the last step. The usage of the dictionary offers the possibility to navigate the results between tokens but other options could be preferred in case of spatial constraints. Reverse swaps the list in place accounting for more efficiency if the list is big.
- Regarding second function: considerations are similar to the first function. If you want a different order you can sort with an inside function that extract a value for the ordering, for example using the i attribute of token object(it was left in the comment of the function). Given the fact that we want to present the subtree in the order of the original sentence we do not have to order.
- Regarding third function: the complexity is approximately O(n^2) due to the double cycle neglecting the spacy functions and the sorting. it is possible to check if the list was composed of elements in the same order of the three that we are try to find (so without consider all possible permutation of the list) just removing the sorting actions inside the functions. In case we do not want the sorting and maintaining the same functionalities we could probably resort to a breadth first fashion with exponential time
- Regarding fourth function: this function is in the worst case (all parameter setted) linear excluding the complexity of the matcher. We could also find the pattern manually using a double cycle and setting manually the limits of the span but the complexity is quadratic
- Regarding fifth function: the usage of the callback could be avoided in favor of a cycle in which the element matched are assigned when you iterate over them but you have to design mechanism to recognise them because the matcher only match the element and do not do nothing else but the callback is probably more efficient and make the code more readable.

Regarding advanced part

- The first point was compared and trained using dependency treebank corpus for training and testing with DependencyEvaluator class of NLTK. The custom parser, with new sets of features, show a slightly improvement in respect to the old one in both LAS and UAS when increasing the number of training samples from 100 to 200 (0.827 of the custom parser against 0.816 against the old one). Surely the evaluation could be better done increasing training and testing samples and changing to different corpora.
- The second point was compared with the two previous classifiers scoring worse on both LAS and UAS (0.73)

References

[1] Bird, S., Klein, E., Loper, E. (2009), *Natural Language Processing with Python*

[2] Jurafsky, D., Marin, J.H.(2000), *Speech and Language Processing*

[3] Kubler, R., McDonald, S., Nivre, J. (2009), *Dependency Parsing*

[4] Zhang, Y., Nivre, J.(2011*) Transition-based Dependency Parsing with Rich Non-Local Features*