Report on Assignment 1 - NLU - Dependency Parsing

Giovana Meloni Craveiro

This report briefly describes the logic behind each function in the dependency graph file.

The code consists of the call of five functions which deal with dependency parsing using spaCy, a python library for Natural Language Processing tasks.
The first step is to load an english pipeline, containing all of the useful resources needed and to define the segments of sentence that will be processed.
In this case, I used the sentence 'Lola and Anna gave me a birthday cake' and the segments 'difficult tasks' and 'a birthday cake'. Then all of the functions are called and the outputs are printed, along with the labels of each function.

All of the functions take a sentence and/or a segment of sentence as input and parse it, transforming it into a Doc object of spaCy, before actually executing its role.

1 - *extract a path of dependency relations from the ROOT to a token*

The first function - extract_path_root_token(sentence) - receives only the example sentence, which is composed by tokens. For each token, it extracts the path from the root to that token and prints it. To do so, a double loop was used. The external loop passes by each token of the sentence to make sure all tokens will have its path to the root calculated. The internal loop iterates through all of the token's ancestors, using the spaCy function ancestor, which returns a generator containing all of the tokens that have dependency relations to the current token, until it reaches the root, a common ancestor to all tokens.
However, since the ancestors generator iteration would normally start with the token and work its way to the root, it was necessary to use the sorted() function to iterate through it contrariwise. Inside both loops, the current token's text and dependency relation is printed.

2 - *extract subtree of a dependents given a token*

The second function - extract_subtree(sentence) - also takes as an input only one sentence and then outputs the subtree of each token in that sentence. A double loop is used in this case too.
While the external loop iterates through the tokens in the Doc object(sentence), the internal loop iterates through the descendants of the current token, which are found in the subtree of the token by the subtree() function of spaCy. All of the subtrees are printed.

3 - *check if a given list of tokens (segment of a sentence) forms a subtree*

The third function - check_tokens_form_subtree(sentence, segment) - receives a sentence and a segment of a sentence as inputs. It verifies if the segment of sentence given consists of a subtree present in the sentence given and outputs true or false.

Firstly, it parses both spans. It then transforms the segment span into a list of words (token.text) with a for loop through the tokens of the segment span. It is necessary to transform it to be able to compare the segment members with the members of each subtree formed in the sentence. Since the subtree() function returns a generator type and the span was of type Span object, they are both transformed into lists of strings, each string representing a token, before comparison.

Then it iterates through the sentences and the tokens, and inside both, it restarts the subtree list to make sure it is empty before adding the members of the current subtree.

It is then possible to iterate through the descendants of the token, adding them to the subtree list.

Finally, it becomes possible to compare the list containing the members of the current subtree with the list containing the words of the segment.

If that comparison turns out equal, it means that those tokens do form a subtree of that sentence. The result is printed.

4 - *`identify head of a span, given its tokens`*

The fourth function - identify_head(sequence) - receives as an input a segment of a sentence, which could also be a sentence, and outputs the head of this fragment.

It parses the fragment and iterates through its tokens (tokens in the Doc object) with a single for loop. In each token, it checks whether this token's head is the token itself.

Since the function head() points to the closest ancestor of the token, if the token points to itself, it means it has no ancestors and therefore, it is the root/head of that fragment.

5 - *`extract sentence subject, direct object and indirect object spans`*

The fifth function - extract_constituents(sentence) - extracts the subject, direct object(dobj) and indirect object(iobj) of the sentence, printing the span that corresponds to them, if they exist, or "None" if they don't.

It receives only the sentence as input, parses it and creates three empty lists for the subject, direct object and indirect object.

It contains a single loop, that iterates through the tokens in the Doc object of the sentence, checking if the grammatical role of the token in the sentence corresponds to either subject, dobj or iobj.

In case it does, all of the elements present in its subtree, including itself, are added to the corresponding list, with a for loop though the subtree.

The function used to check the grammatical role of each token is dep_(), which returns a string containing the abbreviation ("nsubj", "dobj" and "dative" are used in this case).