NLU First Assignment

Andrea Rigo

April 2021

1 Introduction

The first things I do are:

- Initializing an example sentence
- Initialize the *en_core_web_sm* SpaCy model and parse the sentence
- Use *displacy* to visualize the dependency relations of the sentence, so that I have a reference to better debug the code

2 Extract a path of dependency relations from the ROOT to a token

The *token_path_to_root(token, doc)* function gets as input a token and a doc object and returns the token's path to the root. To do that the code keeps track of the dependency relations while it jumps from the initial token to its head, sets the head as the current token and repeats the process. It goes upward in the dependency graph, until it finds the root:

```
while not current.dep_ == 'ROOT':
        path.insert(0, current.dep_)
        current = current.head
```

Then the *sent_paths_to_root(sent)* function iterates over the tokens of the sentence in input and fetches its path to the root calling the previous function.

3 Extract subtree of dependents given a token

The function iterates over the tokens of the sentence and calls the *token.subtree* property to get its subtree and converts it to a list. The initial plan was to use a dictionary to associate each subtree to its toke, but this way couldn't handle sentences where a word appeared more than one time. The subtree would have been overwritten. So using a list comprehension I packed all the subtrees in list of lists:

```
[list(token.subtree) for token in nlp(sent)]
```

4 Check if a given list of tokens (segment of a sentence) forms a subtree

The *is_subtree(sent, words)* function searches the doc for tokens corresponding to the strings received in input. To do that it compares each doc's token text *tk.text* with each word w in a list comprehension:

```
tokens = sorted([tk for tk in doc for w in words if tk.text == w])
```

If the tokens form a subtree then that subtree has its root in one of the tokens. This means that I have to compute only the subtrees rooted in the input tokens instead of searching in all the subtrees of the doc. The subtrees are computed using the *token.subtree* property. Because a token's subtree is a tree made of the token and all and only its descendants, the subtree I'm looking for will contain all and only the tokens in input. So, the code simply checks if the token list is equal to one of the token's subtrees. To check if two lists are equal, because it's an element-wise comparison, they have to be ordered the same way. So the code orders both of them with *sorted*.

5 Identify head of a span, given its tokens

The *head_of_span(words)* function receives in input a sequence of words, assumed to be in single string (otherwise a simple *string.join(' ')* would be enough), and computes the head of the corresponding span. To convert the sequence into a span the code simply parses it, obtaining a doc object. Then it extracts the first (and only) sentence of the doc using the function *next()*. The sentence is a span object, so the code returns its root using the *span.root* property.

6 Extract sentence subject, direct object and indirect object spans

The function *extract_deps_span(sent)* creates a dictionary of lists, one list for each required dependency. Then it iterates over the required dependencies and each token in the doc obtained by parsing the input sentence. When it finds a token with the right dependency, it uses the *token.subtree* property to get its subtree, which is its span, and adds it the associated list.