April 19, 2021

Trono Francesco – Student no. 221723

Assignment 1 - NLU 2020/21 UniTrento

The assignment is implemented in Python using the NLP tool spaCy. All 5 .py modules include a **MAIN** section with one or more default sentences, the call to the implemented function and the printing of the function return items to the standard output.

1. Task 1 - Extract a path of dependency relations from the ROOT to a token

For task 1, two functions have been implemented:

- *depmap(sentence)* – main function
- *traceback(head, path)* – helper function for recursion

depmap is the main function. It takes as input a sentence in string format. The sentence is parsed using spaCy English model. Then, for each token in the sentence the following operations are performed:

- An empty list called "path" is defined.
- If the current token is not the ROOT of the sentence (condition checked on token.dep_), the helper ***traceback*** function is called, passing as input parameters the head of the current token (token.head) and the empty list "path".
- The ***traceback*** helper function performs the following operations:
    - It appends to the obtained empty "path" list a sublist made of:
        - the head token obtained as input parameter, and
        - its dependency relation label.
    - If this head is not the ROOT of the sentence, it calls itself through recursion, passing each time as parameters the head of the current head (head.head) and the partial "path" list.
    - The recursion goes on until the ROOT is reached. At that point, the full "path" list of lists is returned to the caller ***depmap***.
- ***depmap*** stores the path obtained through ***traceback*** and reverses it, in order to have the dependency path start from "ROOT" and not from the last token.
- The reversed path is stored in the dictionary "dep_map" using the current token as key.
- The above procedure is repeated for each token in the input sentence. The full dictionary containing the dependency path for each token is returned as output.

2. Task 2 - Extract subtree of dependents given a token

For task 2, ***depsubtree*** is the implemented function. It takes as input a sentence in string format and, if needed, a nullable parameter called "info": this parameter is an addition that, if set to True, allows to obtain not only the list of dependent tokens but also the dependency relation label for each dependent.

The sentence obtained as input is parsed using spaCy English model. Then, the following operations are performed:

- For each token in the sentence, the attribute "token.subtree" is called, in order to obtain the subtree of dependents, cast to list format and already ordered w.r.t the sentence order.
- Only if the "info" parameter is enabled, for all tokens in the subtree also the dependency relation label is stored.
- The subtree list obtained is stored in a dictionary using the current token as a key.
- The procedure is repeated for all tokens, then the full dictionary is returned as output.

3. Task 3 - Check if a given list of tokens (segments of a sentence) forms a subtree

For task 3, ***is_subtree*** is the implemented function. It takes as input a full sentence and a test segment of a sentence, both in string format.

Both the main sentence and the test segment obtained as input are parsed using spaCy English model. Then, the following operations are performed:

- Each token in the test sentence is converted to string and saved into a list "test_txt". This will represent the subtree to be found.
- For each token in the test sentence:
    - The function iterates across tokens in the main sentence looking for a matching word (main token.text must be equal to test token.text).

- If a match is found, token.subtree is called on the token in the main sentence.
- Each token in the subtree is converted to string and stored in a list “sbt_txt”.
- If “test_txt” (ordered words in sentence) matches with “sbt_txt” (ordered words in subtree), the function returns True and completes its work, otherwise it will loop across the remaining tokens in the test sentence.

4. Task 4 - Identify the head of a span, given its tokens

For task 4, ***span_root*** is the implemented function. It accepts as input a sentence which may come in different formats: a string sentence, a list of strings, a sentence in spaCy Doc format, a sentence in spaCy Span format, a list of spaCy tokens. The MAIN section of the .py module provides already items in all these formats to test the function.

span_root executes the following operations on the basis of a typecheck:

- If input is a Span, the built-in method “span.root” is directly returned.
- If input is a Doc sentence, it is already parsed, so it is converted to Span and span.root is returned.
- If input is a list of strings or tokens, it is converted first to a list of strings (if tokens) and then to one single string through join(). Then the string is passed to the parser, converted to a Span and span.root is returned.
- If input is already a string sentence, it is passed directly to the parser, converted to a Span and span.root is returned.

5. Task 5 - Extract sentence subject, direct object and indirect object spans

For task 5, ***key_spans*** is the implemented function. It takes as input a sentence in string format and a parameter “out” which accepts one of the following two values:

- ‘*subtree*’: each returned span will coincide with the subtree of dependents of the token which has the required dependency label;
- ‘*chunk*’: each returned span will coincide with the noun_chunk, meaning the base noun phrase whose head is the token with the required dependency label. The noun chunk will be generally shorter than the subtree, since it consists only of the noun plus the words describing the noun[1].

The sentence is parsed using spaCy English model. Then, the following operations are performed:

- A list of key dependency labels is defined. The following labels[2] are considered:
    - **Subject**: *nsubj* (nominal subject), *nsubjpass* (nominal subject of passive verbs), *csubj* (clausal subject, for the cases in which the subject of a clause is itself a clause), *csubjpass* (clausal subject of passive verbs).
    - **Direct object**: *dobj*.
    - **Indirect object**: *dative* (indirect object is defined[3] as the person or thing that the action of a verb is performed for or directed to).
- A loop looks for all tokens (or noun_chunks, depending on the chosen value of the “out” arg) in the sentence that have one of these labels as token.dep_ (or chunk.root.dep_).
- When a match is found:
    - If “out=’*chunk*’”, the noun chunk is cast to a list and stored in a dictionary using the head dependency label as key.
    - If “out=‘*subtree*’”, the current token’s subtree is obtained with token.subtree, then cast to a list and stored in a dictionary using the head dependency label as key.
- Once all tokens (or chunks) have been analysed, the dictionary with all identified lists is returned.

[1] Documentation: http://spacy.io/usage/linguistic-features#dependency-parse

[2] Labels available at: https://spacy.io/models/en#en_core_web_sm . Explanations obtained using spacy.explain().

[3] Definitions by Merriam-Webster:
- Indirect object -> https://www.merriam-webster.com/dictionary/indirect%20object ,
- Dative -> https://www.merriam-webster.com/dictionary/dative