

First Assignment Natural Language Understanding

Bertelli Davide
mat. 223718

1 Abstract

In this report it has been expressed the intentions and solutions behind the 1st_Assignment.py python module requested for the Natural Language Understanding course. In particular the first assignment focuses on understanding the principles behind the library **spacy** by implementing custom functions exploiting its architecture.

The assignment requested to implement five functions enabling the user to:

1. Given a sentence, extract the path of relations from the ROOT token to any other one.
2. Given a sentence, the subtree of each of its tokens is extracted.
3. Given a sentence and a list of tokens, checks whether the ordered list of tokens forms a sentence's subtree.
4. Given a list of tokens belonging to a span, it gives back its head.
5. Given a sentence, it extracts tokens, and their spans, which are *nominal subject* or *direct object* or *indirect object*.

2 Code requirements

As already asserted by the Readme file coming with this report, in order to run the code provided, the spacy library is required along with the en_core_web_sm pipeline for english language. Instruction to install them, if needed, are contained into the Readme.

3 Code Implementation

3.1 First Function

The first function "rootToTokenPath()", aims at extract the dependency path that from the ROOT token reaches any other token contained in the provided sentence. For this purpose, a sentence in string format is given as input to the function. Then the english language pipeline is loaded creating the nlp instance of the Spacy Language Class. After this, the sentence is processed by the language class through its pipeline and a spacy.tokens.Doc item is created with the tokens and relations derived from the input sentence and an empty list that will contain the dependencies is initiated. Finally a for-cycle iterating over the tokens inside the Doc element is started and each token it processes enters into a while-cycle and is inserted into the previously empty list then substituted by its head for the next cycle iteration. The while-cycle's stop condition is that the token's dependency relation is the ROOT one

and so the jump from token to token's head has reached the root of the sentence. At the end of the while cycle the last token, that now is the root one, is inserted as first into the list and the function returns to the caller a list of lists containing the tokens that from the root one leads to any other token in the sentence.

3.2 Second Function

The second function, "subtreeOfDependents()", takes as input a sentence in string format and aims at allowing users to extract the subtree for each token in the provided input. To fulfill its task it initiate the spacy language class with the english pipeline and uses it to process the sentence, storing the resulting tokens and dependency relations into the Doc item. After that an empty dictionary that will contain the computed trees is created. Finally a for-cycle is started among all the tokens in the Doc item and an empty list is allocated, that will contain the elements of token's subtree. Then another for-cycle starts iterating all over the elements in the subtree returned by calling the homonym method on the token itself. For all tokens given by the method, we check if they are equal to the one in the outer cycle, avoiding repetitions performed by the subtree method that will return even the root of the subtree. Whenever the check is satisfied the token is added to the previously defined empty list and after the inner cycle is ended the token from which the subtree has been computed is added in front of the list and a new entry in the dictionary is created using the outer cycle token's text as key and the list as value. At the end, the function returns to the user the so populated dictionary.

3.3 Third Function

The third function, "isSubtree()", takes as input a list of tokens and a sentence ; the aim is to understand if the tokens form a subtree of the sentence. Initially the function calls the previous defined "subtreeOfDependents()" to build the subtrees' dictionary using the provided sentence as input. Then a check is made on the list of tokens to verify if it is composed by spacy tokens or by strings. In the latter case they are merged in a sentence that will be tokenized by calling the nlp and Doc objects ensuring to have a list of spacy tokens. After that the text of the first element in the list is checked against the keys of the subtrees' dictionary and whenever a match is found an index constant is set to the index of the token in list after the root one and each element in the subtree is compared with the token at position *index* in the list contained inside the dictionary; if the check is true the constant is incremented. This process keeps repeating and if a mismatch between tokens is found then it is understood as a wrong tree, so no result found with the tokens in the input list. The

function then returns to the user a boolean value: *True* if input tokens and the ones in a subtree matches, *False* otherwise.

3.4 Fourth Function

The fourth function, "headOfSpan", takes as input a list of tokens and returns the head of it. To perform its task, it checks the list's elements type and if they are spacy tokens ones it converts them into strings and merge them together in order to make a sentence. Otherwise the list of strings is simply merged into a sentence. After that the sentence is then processed by the *nlp* and *Doc* items. Then a *span* object is created by getting all the items in the *Doc* item and finally the head is returned to the user as a spacy token by calling the *root* method of the *span* object.

3.5 Fifth Function

The fifth function, "objectsExtractor()", takes a sentence as input and returns the tokens in it having a dependency of *Nominal subject - nsubj*, *Direct object - dobj* or *Indirect object - dative*, together with the span they belong to. In order to perform its task, the sentence is processed by the *nlp* and *Doc* objects. After that an empty dictionary is created having the desired dependencies as keys and an empty list as value. Furthermore, a for-cycle is started on all the tokens in the *Doc* object and whenever a token's dependency matches one of the dictionary's, an empty list for the span is created and filled with the element in the matched token's subtree. After this, the list is appended to the one in the dictionary at the right key value. This function returns to the user a dictionary with dependencies as keys and a list containing the lists of the matched tokens' spans.