Natural Language Understanding - Assignment 1

Vincenzo Marco De Luca
Università degli Studi di Trento
vincenzomarco.deluca@studenti.unitn.it
Matricola: 224463

***Abstract*—The objective of the assignment is to learn how to work with dependency graphs by defining functions.**

***Index Terms*—Spacy, dependency graph, Doc, Token, subtree, head, NLTK, Transition-Based Dependency Parser, Classifier**

I. Working with Dependency Graphs

A. Utility function

For every mandatory exercise, it is necessary to import the Spacy library which will allow us to generate a *Doc* that is a sequence of *Token*. Therefore, in order to improve modularity, it has been defined a utility function: *get_doc(sentence)* which receives as input a string (*sentence*) and returns as output a Doc. First of all, it is necessary to load a *Language* instance by using the method *load* which is a build-in loader to load the package by name (in our case the English language: *"en_core_web_sm"*)

After having defined the instance of the language, by passing an input sentence as its argument, we will produce the desired Doc instance.

B. Exercise 1

Exercise: *Extract a path of dependency relations from the ROOT to a token.*

In order to solve the first exercise, I have defined the function: *path_of_dependencies(sentence)*, it receives as input a string (*sentence*) and it returns as output a list of list. The input of the function is a string (*sentence*) representing a sentence and each unit of this sentence represents a token and the output is a list of lists of strings. The reason why I preferred to use a list of list instead of a dictionary is due to the possibility of having more than one token corresponding to the same word in the input sentence which would have caused some inconsistency in the path associated to the key, which in this case is a token, in the dictionary. After having processed the Doc, we need to iterate over it, at each iteration we will have a Token from the input sentence, so we need to append in a temporary list the string describing the dependency relation of the current token by using the attribute of Token: *dep_*, then we need to proceed the path from the current token to the root node and we can do that by exploiting the member *head* of the current token and for each token we iterate until the root token (the token which is equal to its head) and at each iteration, we need to push at the top of that specific list, the label of the current arc, at the end of the internal iterative cycle, we append the current list of dependency relations to a list of lists of dependency relations containing the list of dependency relations for all tokens, then we need to repeat the iterative cycle again for the successive token within a sentence. After having repeated this process for every token from itself to the ROOT token, we will return the list of lists of strings where each string corresponds to the label of the current arc.

C. Exercise 2

Exercise: *Extract subtree of a dependents given a token*

As in the previous exercise, the input of the function *sorted_path_of_dependencies* is a string *sentence* and the output is a list of lists of strings where the i-th list represents the subtree associated to the i-th token in the sentence and each subtree is described as a list of string where each string represents a dependency relation.

As in Exercise 1, the first operation is to process the Doc, then iterate over all the token in the sentence, at each iteration, we need to append to the list (which will be returned as output) the subtree of the current token, this operation is performed by resorting to the Token's member *subtree* which returns a sequence containing the token and all the token's syntactic descendants, another important aspect about this function is that it provides a list of ordered token with respect to the order of the tokens within the sentence. Finally, after having repeated this process for every token in the sentence, we will return the list of lists of strings where the i-th list contains an ordered sequence of tokens in the subtree of the i-th token in the sentence.

D. Exercise 3

Exercise: *Check if a given list of tokens (segment of a sentence) forms a subtree*

In this case, the function has two different inputs: the first one is the sentence and the second one is a list of strings *path* and the output is a boolean which is *True* if *path* is a subtree of the sentence otherwise it is *False*

As in the previous exercises, the first operation to do is to process the Doc, then for each token in the sentence we need to temporarily store its subtree, in order to do that we need to use the Token's member *subtree*, but we need to take into consideration that the second input of the function is a list of strings, so we have to convert the subtree member (which is a sequence of Token instances) into a list of strings and it is possible by using an auxiliary function that I have defined: *gen_to_str*, after this conversion we simply need to use the overloaded Python equality operator == to compare the two strings representing the list of words in input and the subtree of the current token.

We need to iterate it until we will find a subtree equal to the input string and in this case, the function will return *True*, otherwise (the function iterates over all tokens in the sentence but it does not find any corresponding subtree) it will return *False*

E. Exercise 4

Exercise: *Identify head of a span, given its tokens*

The function *get_head* receives a string (*sentence*) as input and returns a Token. As in the previous exercises, the first operation to perform is the Doc processing, then we need to store in a local variable the Span contained in the Doc and finally return the *root* attribute of the Span representing the input sentence.

F. Exercise 5

Exercise: *Extract sentence subject, direct object and indirect object spans*

The function *extract_soi* receives a string (*sentence*) as input and returns a dictionary of lists as output. As all the previous exercises, the first thing to do is to define the processed doc, then we need to declare a dictionary which, we will use to store the token of interest, made up of three different keys: *subj*, *dobj* and *dative*, where the first key represents the subject, the second one the direct object and the third one the indirect object. After this declaration, we can iterate over all tokens in the input sentence, then by using the *dep_* member of the current instance of Token, we can check if the current token is a subject, an object, an indirect object or none of them, in the first three case we will append the current Token to the list associated the key correspondent to its label. Finally, after having repeated this process for all tokens, we can return the dictionary of lists.

II. Training Transition-Based Dependency Parser

A. Configuration class

Exercise: *Modify NLTK Transition parser's Configuration class to use better features.*

The original NLTK Transition parser's Configuration class uses as features: word, lemma and tag of the token on the top of the stack (*stack[len(stack)-1]*) and, if there is at least another element in the stack, it uses also the element following the one on the top of the stack (*stack[len(stack)-2]* from which it takes: the tag, information from the left-most and right-most dependency of the element at the end of the stack (*stack[0]*), then it takes similar information from the buffer but here the indexes are inverted (e.g. if in the stack it was the last element: *stack[len(stack)-1]*, in the buffer it is the first element: *buffer[0]*) but, for the buffer, it takes also the feat information from *buffer[0]* and the tag from *buffer[2]* and *buffer[3]* (if they exist).

Starting from the original code, the first feature I have added is the Levenshtein distance [1]) [2] between the word on the top of the stack and the last word in the buffer [3].

I have also added the left child and the right child of two given nodes [1] (the one at the top of the stack and the last in the buffer) , the index of the element considered and the length of the buffer.

B. Performance comparison

Exercise:*Evaluate the features comparing performance to the original*

In order to simplify the testing and increase the code flexibility, I have added a further argument to the function *train* specifying if the user wants to train the model with the original features or with the modified ones.

The results in terms of performance, by still using the same classifier (SVM) have been resumed in Table I.

TABLE I
PERFORMANCE - FEATURES

Features	Accuracy	Training time
Modified Features	0.899	2 min
Original Features	0.868	2 min

It is important to underlie that we pass the same parameters in both cases: (*dependency_treebank.parsed_sents()[:300]*) to the train function and $dependency\_treebank.parsed\_sents()[-30\ :]$ to the parse function and to the evaluator function.

Then I have also tried to train them even on by passing to the train function (*dependency_treebank.parsed_sents()[:500]*) and to the the parse function and to the evaluator function $dependency\_treebank.parsed\_sents()[-50\ :]$. By trying this setting, the accuracy has been equal to 0.914 by using the modified features and 0.901 by using the original ones.

Moreover in the source code, I have listed other features which could be added but they still require more experiments, that is the reason why I left them as a comment and I have not described them in detail in this document.

C. Classifier

Exercise:*Replace SVM classifier with an alternative of your choice*

I have added another parameter to the train function in order to specify the classifier to be used during training, the possible classifier and their results have been resumed in Table II.

TABLE II
PERFORMANCE - CLASSIFIERS

Model	Original Features	Modified Features
SVM	0.914	0.901
Random Forest	0.891	0.812
MLP	0.836	0.726
Decision Tree	0.828	0.783

It is important to underlie that the models in Table II have all been trained with the same training and testing set size (500,-50) , but they require different time to complete train (e.g. Random Forest and Decision Tree require just one minute, while MLP four minutes and SVM more than six minutes).

References

[1] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions and reversals," 1965.

[2] D. Jurafsky and J. H.Martin, *Speech and Language Processing - An introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*, 2019.

[3] S. Kubler, R. McDonald, and J. Nivre, *Dependency Parsing - Synthesis Lectures on Human Language Technologies*, 2009.