NLU first assignment

Azzolin Steve

University of Trento
Department of Information Engineering and Computer Science (DISI)

1 Introduction

This document describes the 5 exercises comprising the first assignment of the course *Natural Language Understanding*. The code is available at *https://github.com/steveazzolin/NLU-first-assignment*. Overall, the input sentences are assumed to be passed as Python strings, whereas the output depends on the specific function.

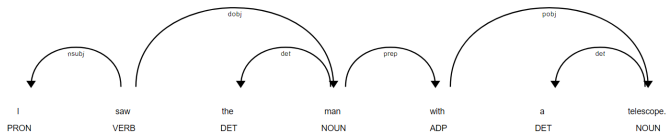

Figure 1: Dependency graph for a test sentence

2 Exercise 01

extract a path of dependency relations from the ROOT to a token

- *input is a sentence, you parse it and get a Doc object of spaCy*
- *for each token the path will be a list of dependency relations, where first element is ROOT*

My solution to this exercise is built around the method *ancestors* of spaCy. It simply returns the list of ancestors, and to get them in the requested order, I simply inverted that list. Since we are interested in just the dependency relations, the output is build concatenating just *token.dep_*. The result for the test sentence is the following, where each element of the array correspond to a token in the input sentence:

```
[['ROOT', 'nsubj'],
 ['ROOT'],
 ['ROOT', 'dobj', 'det'],
 ['ROOT', 'dobj'],
 ['ROOT', 'dobj', 'prep'],
 ['ROOT', 'dobj', 'prep', 'pobj', 'det'],
 ['ROOT', 'dobj', 'prep', 'pobj'],
 ['ROOT', 'punct']]
```

Figure 2: Output of exercise 01

3 Exercise 02

extract subtree of a dependents given a token

- *input is a sentence, you parse it and get a Doc object of spaCy*
- *for each token in Doc objects you extract a subtree of its dependents as a list (ordered w.r.t. sentence order)*

This function is nearly the same as the first one, with the only difference that instead of returning the dependency relation I return the subtree of each token (by means of *token.subtree*). The output is also structured in a similar way to the previous function.

```
[['I'],
 ['I', 'saw', 'the', 'man', 'with', 'a', 'telescope', '.'],
 ['the'],
 ['the', 'man', 'with', 'a', 'telescope'],
 ['with', 'a', 'telescope'],
 ['a'],
 ['a', 'telescope'],
 ['.']]
```

Figure 3: Output of exercise 02

4 Exercise 03

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

- *input is a sentence, you parse it and get a Doc object of spaCy*
- *for each token in Doc objects you extract a subtree of its dependents as a list (ordered w.r.t. sentence order)*

The aim of this exercise was to test whether a list of words in input constitutes a subtree of a parsed sentence. The input is thus a Python string for the sentence to parse, and a list of words for the ”query” subtree. The output is True/False depending if the query forms a subtree. Recalling that a subtree of a tree T is a tree consisting of a node in T and all of its descendants in T, the output for the test sentence is:

- query=[”the”, ”man”, ”with”, ”a”, ”telescope”] –> True
- query=[”man”, ”with”, ”a”, ”telescope”] –> False

Note that the second query is evaluated to False since the subtree originating in ”man” must include also ”the”.

5 Exercise 04

identify head of a span, given its tokens

- *input is a sequence of words (not necessarily a sentence)*
- *output is the head of the span (single word)*

This function could be resolved by simply parsing the sentence and taking the root of the span composing the spaCy Doc object (*spacy_doc[:].root*). The input to the function is again a Python string, and the output is just a string.

```
saw
```

Figure 4: Output of exercise 04

6 Exercise 05

extract sentence subject, direct object and indirect object spans

- *input is a sentence, you parse it and get a Doc object of spaCy*
- *output is lists of words that form a span (not a single word) for subject, direct object, and indirect object (if present of course, otherwise empty). Dict of lists, is better*

First of all, the input Python string is parsed with spaCy. Then I keep track of the subtree of every token of the sentence that matches one of the dependencies we are looking for (respectively *nsubj, dobj* and *dative*). The output is depicted in the following figure. Note that the output is made, for each key of the dict, of a list of strings and not a list of Span objects, since from the comments on Piazza emerged that the word "span" in the exercise text should be intended with its generic meaning.

```
{'dative': [], 'dobj': ['the man with a telescope'], 'nsubj': ['I']}
```

Figure 5: Output of exercise 05

7 Advanced extra point

- *Modify NLTK Transition parser's Configuration class to use better features. Evaluate the features comparing performance to the original, and replace SVM classifier with an alternative of your choice.*

In order to change the features extracted by the Transition parser I forked the official repository of NLTK[1]. First, I tried to concatenate to the existing features the GloVe embeddings[2] of the token under analysis to try to capture the high-level meaning of words. Then, following the suggestions of Sandra Kubler, Ryan McDonal and Joakim Nivre (2009) I computed also the number of left and right children of the first element of both the stack and the buffer. The resulting performances are depicted in Table (1). These scores are computed training on the first 100 samples and testing on the last 20 samples of the dependency treebank provided by NLTK. We can immediately see that the GloVe embeddings don't increase the expressive power of the model, but on the contrary, by increasing considerably the dimensionality of the problem (at least by 50 dimensions), they bring down the performances. On the other hand, adding the number of left/right children for a particular token seems a good starting point for the development of better features.

Score	Default	GloVe	L/R children
LAS	0.76	0.09	0.88

Table 1: LAS score for: the default implementation, default implementation + the addition of GloVe embeddings, and for the default implementation + the addition of the number of children. Only LAS scores are reported since equal to UAS.

At this point the goal was to experiment with different classifiers. I noticed that the SVM used by NLTK relies on libsvm, which is an optimization library with a complexity at least quadratically with the number of samples. Indeed, training with more than 1000 samples takes a lot of time. So, instead of focusing blindly on the maximization of the performances, I took into consideration also the fitting time, since a lower fitting time would allow the training on more samples, and consequently to better generalization performances. Figure (6) illustrates a comparison that takes into account both performances and training time for three models: the SVM provided by the default implementation of NLTK, DecisionTree, and RandomForest.

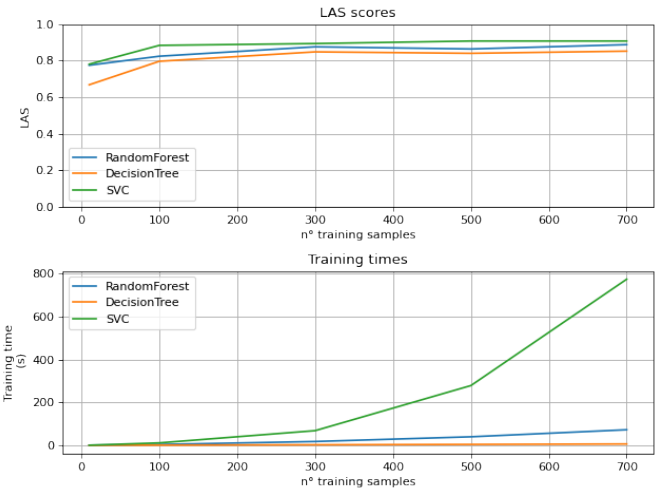

Figure 6: Benchmark of three different MachineLearning models

Overall, I believe that RandomForest is the best model taken into consideration, since the drop in performance is very limited, but the gain in fitting time is considerable.

[1]The fork can be found at https://github.com/steveazzolin/nltk
[2]https://nlp.stanford.edu/projects/glove