Assignment 1

Luigi Riz - Mat. 223823

To complete the first assignment, consisting in the creation of some functions based on the spaCy library, I worked on a Jupyter Notebook, that helped me in subdividing my work into chunks. For each method I provide both inline comments as documentation and an example of the use of them. While developing, I read spaCy online documentation and tried to understand how the provided datatypes and functions could help me in maintaining the code simple and clear. To have a better visualization of the Dependency Trees provided by spaCy I used a submodule named displaCy, that plots a `Span` or `Document` in a more readable way.

The main methods I implemented in my code are the following:

1 dep_paths

The first method inside the Notebook has the following signature:

```
def dep_paths (sentence, nlp = spacy.load('en_core_web_sm'))
```

It takes as input a `string`, representing the sentence to be processed, and a spaCy `Language` object, the model used to perform dependency parsing.
It returns a `dict`, mapping every spaCy `Token` in the sentence, to a list of `string`s, constituting the path of dependency relations, starting from the `ROOT` and reaching the `Token` itself. This key-value mapping is in the form:

```
'TOKEN'    ['ROOT', 'TOKEN_i'.dep_, ..., 'TOKEN_j'.dep_, 'TOKEN'.dep_]
```

meaning that to reach `Token` 'TOKEN' following the Dependency Tree, we have to start from the `ROOT` node and run through the sequence of archs 'TOKEN_i'.dep_, ..., 'TOKEN_j'.dep_, 'TOKEN'.dep_.

The internal logic of the function is rather simple: first of all the sentence is parsed into a spaCy `Doc`, then, for each `Token`, the list of ancestors is scanned and the list of dependencies is consequently updated, adding time by time the `token.dep_`. Once the `ROOT` is reached, the `list` is added to the `dict` and the following `Token` is analyzed.

2 subtrees

This function is represented by the following signature:

```
def subtrees (sentence, nlp = spacy.load('en_core_web_sm'))
```

Similarly to the previous case, it requires a `string` sentence to be processed and a spaCy `Language` model to parse it. The output of the function is a `dict` in which:

- the *key* is a `Token` in the parsed sentence,
- the *value* is a `list` of `Token`s, representing the subtree of the related key. The `Token`s are inserted in the `list`, following their ordering in the original sentence.

So, every mapping in the output `dict` is in the form:

```
'TOKEN'    ['TOKEN_i', ..., 'TOKEN_j']
```

Note that the `list` is never empty since at least 'TOKEN' itself will always be contained in it.

Also in this case the logic is not so complex. In fact, after parsing the `string` into a `Doc` object, all the `Tokens` are scanned and their `token.subtree` paramenter is added to the output `dict`.

3 check_subtree

The third method has the signature:

```
def check_subtree(sentence, subsentence, nlp = spacy.load('en_core_web_sm')):
```

In this case the required inputs are a `string` sentence to be parsed, an ordered `list` of words (strings) to be checked and, as previously, a spaCy `Language` model to perform parsing. The function's output is:

- `True`, if the ordered `list` of words represents a subtree for at least one of the `Token`s contained in the sentence;
- `False`, if the *subsentence* is not a subtree or if the words contained in it may form a subtree, but they are in the wrong order.

Note that the order in which the words appear in *subsentence* is meaningful.

The functioning of this method is based on `list` equality. For each `Token` in the input sentence, the subtree is retrieved. At this point, the `Token.text` of each of the `Token`s inside the tree is added into a `list`. Finally, this `list` is compared with the input `list` of strings. The output boolean value depends on this equality test.

4 span_root

This function is represented by the following signature:

```
def span_root (sentence, span_start, span_end, nlp = spacy.load('en_core_web_sm')):
```

It takes as input a `string` sentence, two `integer`s, representing the start and the end of the span we want to extract from the sentence, and a spaCy `Language` model to perform parsing. The method provides as output a spaCy `Token`, representing the *root* of the `Span`. Following spaCy documentation, the returned object is *"The token with the shortest path to the root of the sentence (or the root itself). If multiple tokens are equally high in the tree, the first token is taken."*

Trying to develop a method which is independent from a sentence, also another function is provided:

```
def span_root_nosent (span, nlp = spacy.load('en_core_web_sm')):
```

In this case the user has only to provide a `string` span, out of the context of a more general sentence, and it will receive as output the `root` of it.

The behaviour of the two methods is rather similar: they both parse the input string into a spaCy `Doc`, extract from it the corresponding `Span` and finally retrieve its `Token` root.

5 extract

The last method has the following definition:

```
def extract (sentence, nlp = spacy.load('en_core_web_sm')):
```

The inputs to provide are a `string` sentence and a spaCy `Language` model to parse it. The function will extract from the sentence three different `list`s of `Token`s, each of them representing a span. The structure provided as output is a `dict`, associating the *keys* `'nsubj'`, `'dobj'` and `'iobj'` to the relative span. In this way the sentence subject, the direct object and the indirect object spans of the sentence are available. Note that if the sentence does not contain one of the spans, it will return a `dict` with an empty list to the related key.

The internal logic of this function is based on the subtrees. Scanning all the `Tokens` in the sentence, the equality between their `dep_` field and the predefined strings is checked. If the equality is verified, the list of all `Tokens` in actual `Token`'s subtree is added to the corresponding key in the output `dict`, otherwise the `list` is leaved empty.