NLU Second Assignment

Andrea Rigo

April 2021

1 Loading the dataset

I used the file text.txt loaded using conll.py's function *read_corpus_conll(file)*. To have a more friendly interface to access the data I wrote two very simple classes in my_tokens.py: Sentence and NamedEntity. These classes simply take as input a row of conll data as a string, and split it so that I can access each token's properties through NamedEntity's attributes, *text, pos, chunk_tag* and *ent_tag* which correspond to CoNLL's *text, part-of-speech tag, syntactic chunk tag* and *named entity tag* respectively. The class Sentence allows me to merge the texts of each token and get the sentence as a string by simply doing *str(sentence)*.

Once loaded the data, I filtered out all the lines containing *DOCSTART* since they are not relevant.

2 Evaluate spaCy NER on CoNLL 2003 data

Here I passed each sentence (by merging all the tokens with space characters) one at a time to the spaCy model. Since each sentence is independent of the others, I split this task into four processes so that it completes a bit faster.

2.1 Custom tokenizer

SpaCy and CoNLL use two different tokenizers, so spaCy will tokenize the sentences in a different way and its predictions will have different tokens, rendering impossible to compute any classification metric. Because of this I defined a custom tokenizer and inserted it in the spaCy pipeline so that spaCy will use it instead of its own tokenizer. This way sentences will be tokenized the way I want them to, producing the same tokens of CoNLL. The custom tokenizer is the function *tokenizer_(sent)* which gets as input a sentence and splits it using white spaces, exactly like CoNLL.

2.2 Labels conversion

SpaCy and CoNLL also use two different types of labels, so a conversion is needed to compare predictions to ground truths. This function uses a dictionary to associate each spaCy class to the correct CoNLL label, and merges the IOB tag and the named entity tag since in CoNLL those are placed in a single string. After reading CoNLL annotation description I came up with the conversion reported in table 1.

spaCy	CoNLL
PERSON	PER
NORP	MISC
LOC	LOC
FAC	LOC
GPE	LOC
ORG	ORG
PRODUCT	MISC
EVENT	MISC
LANGUAGE	MISC
Everything else	-

Table 1: Label conversion map. '-' means that the corresponding named entity will be converted to a simple token, removing the entity tag and setting its IOB to O.

2.3 Evaluation and results

Here first I organized both the predictions (with converted labels) and the test set in pairs *(token text, token tag)*. Then I used sklearn's classification report to compute the token-level total accuracy, getting *90.9%*, and precision, recall, and f1 for each class. To obtain the chunk-level metrics I used conll.py's *evaluate* function. All the results are reported in the notebook.

3 Grouping entities

The function *group_by_noun_chunks(sent_doc)* takes a spaCy's Doc object and creates a list of groups, for now just noun chunks, using *doc.noun_chunks* and filtering the tokens so that only named entities are added to the list, since this the objective of the exercise. After that, for each token of the sentence, the code checks if it is already in a group with *is_in_a_group(token, groups)* and if its not and it also belongs to a named entity, *insert_in_group(token, groups)* is called. This function adds to the list a group with only one element, the token, in sentence order. Due to the fact that each group is a sub-list the token's index can't be used to position it in sentence order: we need the group position. So the function searches for the previous word in the groups, so the one with the target token position minus one, and then adds the token to the right of that group. If no previous token is found, it means the token to add is the first, so it will be added in position 0.

To check if a token belongs to a group I compared each group's token position in the sentence with the new token's position. If they have the same position they are the same token. I made this choice because I couldn't find any equivalence function in spaCy's Token class source code, so I wasn't sure a direct comparison would have worked in case of duplicate words in a sentence.

After having grouped all named entities, I used a simple dictionary to count their occurrences. Results are provided in the notebook.

4 Fix segmentation errors using the compound dependency relation

The code iterates over each token in every named entity found in *sentence.ents*. Then it searches for a child of the token which has a compound dependency relation with its parent but it is also not part of another entity (the child iob tag has to be O). Then if it is adjacent, on the right or on the left, the code expands the entity span accordingly by defining a new span with the new start and end indexes and adding it using *Doc.set_ents*.

Lets suppose that a named entity has a compound, which in turn has a compound and so on, forming a sort of compound chain. To expand the span to include the entire chain I used three flags to control the main loop:

```
while keep_expanding and updated:
        updated = False
        keep_expanding = recursive
```

The *updated* flag is set to true if at least one named entity has been expanded in the current iteration. The *recursive* flag is used to enable or disable the inclusion of the entire compound chain. If it is set to false, *keep_expanding* will be true only the first iteration, so only the adjacent compounds will be included, not the entire chain. After the first iteration *keep_expanding* will be false, so the *updated* flag will be ignored. *keep_expanding* is needed due to the lack of a do-while loop in python. If *recursive* is true, *keep_expanding* will always be true, which means that now *updated* is controlling the while loop. Because *updated* is set to true if a span gets expanded, the entire process will be repeated but now the adjacent compounds will be the next level in the chain (the code included the previous level in the last loop) and thus they will be included.

4.1 Test post-processing

After having included the whole compound chain in each named entity's span, I computed again all the performance metrics used in section 2.3. All the results are included in the notebook. The post-processing worsened performance, accuracy and total chunk-level measures are reported in table 2.

	Original	**After post-processing**
Accuracy	90.9%	89.8%
Precision	68.7%	64%
Recall	52.4%	48.8%
F1	59.4%	55.4%

Table 2: Performance before and after fixing segmentation errors.