April 30, 2021

Trono Francesco – Student no. 221723

Assignment 2 - NLU 2020/21 UniTrento

The assignment is implemented in Python using the NLP tool **spaCy**, the **ScikitLearn** library (to compute the classification report) and the **Pandas** library (to render dataframes for better visualization). All these libraries must be installed on the machine.

For the statistical evaluation part, the dataset used is "*conll_2003*" (available in folder 'src'), which is downloadable from the Github '**data**' folder. In the Github '**code**' folder, a Python module called "*conll.py*" is also provided, which is used to perform accuracy evaluation using the conll_2003 dataset.

For all the assignment questions, a **MAIN** section is provided, which includes one or more default sentences, the call to the implemented function and the printing of the function return items to the standard output.

<u>IMPORTANT NOTE:</u> all the .py modules provided, the file 'conll.py' and the 'src' folder must be placed together <u>in the same directory</u>. For all tasks, the file '*./src/conll2003/<u>dev.txt</u>*' has been used. The code has been developed and executed on Linux Mint.

1. Task 1 - Evaluate spaCy NER on CoNLL 2003

Considering that the same functions prepared for task 1 will be used for both **task 1** and **task 3**, two separate .py modules have been prepared, one with the ***functions*** and the other with the ***main***.

Functions.py

In the ***functions*** module, global variables provided are the spaCy English model load and a standard dictionary "labels" containing a default format for named entity ("NE") labels frequency count. The following functions are implemented.

a) conll_to_str()

The function converts the conll_2003 dataset into a list of sentences in string format. It takes as input the **directory** of the *conll_2003/dev.txt* original text file and returns 5 outputs:

- A list of string sentences *sent_ls*
- The total count of tokens *tok_count* in sent_ls;
- A dictionary *ner_conll* containing the frequency count for all identified named entity labels;
- A *refs* array containing the NE label of each token in sent_ls (for the classification_report);
- A *tup_refs* array of lists of tuples, where each inner list is a sentence and each tuple is a pair ('token', 'NE label').

The function opens the file in the directory provided as arg and duplicates the standard dictionary "labels". Then, it checks whether each line in the file is not empty and does not contain only a "-DOCSTART-" word: if the line is good, it splits it into words, stores the first word in the buffer list "words", stores it again as tuple together with its NE label in another buffer list "buf" and finally updates tok_count and the refs array. When an empty line is reached, this marks the end of a sentence: therefore, the "words" buffer with the tokens is converted to a single string sentence, saved into the "sent_ls" array and then emptied, while the "buf" buffer with the tuples if stored as inner list into the "tup_refs" array and emptied.

b) export_to_file() *(note: needed for Task 2)*

This function simply exports into a .txt file the list of string sentences returned by *conll_to_str()*. It accepts as input this list and the desired file name – no directory or extension, just the file name word(s) as string.

c) sent_retok()

Companion function for the next *spacy_retok()* function. It takes as input a single sentence in doc format and produced as output the retokenized sentence in doc format. Note that the "a" index variable defaults at a high value (10000) when no span is under construction.

To retokenize, the method *doc.retokenize()* is used, which simply adapts the span of a token according to the indications received. The retokenization is triggered by checking the *whitespace_* attribute of each token: the "a" index is set as the position of the first token which, in the original string sentence, was not followed by a space (*whitespace_* = ''). The "b" index is accumulated until one of the next tokens returns a whitespace_ = ' ' (meaning: it was followed by a space in the original string sentence). Once such token is found (or if the end of the sentence is reached), the two indexes are reset and the *doc.retokenizer.merge()* method is launched on the span identified by a and b.

d) spacy_retok()

This function is the outer layer of *sent_retok()*. It takes as input the list of string sentences returned by *conll_to_str()*: each string sentence in the list is parsed with spaCy, converted to doc format and passed as input to the companion function *sent_retok()*. The returned retokenized doc sentence is stored in the list *sent_ret*. Once all sentences have been processed, *sent_ret* is returned as output together with the count *ret_count* of all tokens in it.

e) Labels_remap()

The function takes as input the *sent_ret* list of retokenized doc sentences returned by *spacy.retok()*. It returns as output:

- A dictionary *ner_spacy* containing the frequency count for all identified NE labels realigned to the conll_2003 format;
- A *hyps* array containing the remapped NE label of each token (for the classification_report);
- A *tup_hyps* array of lists of tuples, where each inner list is a sentence and each tuple is a pair ('token', 'remapped NE label').

The function remaps spaCy's "*ent_type_*" for each token by cross-checking a legend stored into the dictionary "remap", which is built by hand on the bases of the labels available in spaCy documentation.[1] A string is concatenated using spaCy's "*ent_iob_*" and the conll NE type correspondent to spaCy's original "ent_type_", then stored into the *ner_spacy* dictionary (together with the label occurrences count) and into the "hyps" array and "tup_hyps" array, using the came procedure seen in *conll_to_str()*.

Main.py

The ***main.py*** file simply calls, in order, all the functions imported from the ***functions.py*** file. The conll text file used is ***dev.txt***. In particular, the token count is checked (and printed, together with the two *ner_conll* and *ner_spacy* dicts of label frequency counts) for both the *conll_to_str()* output and the *spacy_retok()* output, in order to ensure the full alignment and obtain a quick frequency comparison.

For **Task 1.1** (token-level and class-level evaluation), ScikitLearn's classification_report() metric is used, passing as input the two arrays of aligned labels occurrences *refs* and *hyps*.

For **Task 1.2** (chunk-level evaluation), the "*evaluate()*" function provided into the *conll.py* module is imported and the two arrays of list of tuples *tup_refs* and *tup_hyps* are passed as input.

2. Task 2 – Grouping of entities

The starting input source is the text document already exported in Task 1 through the *export_to_file()* function, whose assigned name is *"./conll_str.txt"*. Three are the functions implemented in this .py module.

a) *group_entities()*

This is the companion function to *groups_freqcount()*. It takes as input a sentence in doc format. It stores the chunks identified by spaCy's *noun_chunks* method into a waiting list called "*chunks*". Then, it iterates through the entities identified in the sentences spaCy's *doc.ents* iterator. For each entity: if chunks are available in the waiting list, it takes the first chunk and stores its start and end indexes. It then compares these indexes with the current entity start and end indexes, to understand whether the NE belongs to a chunk or it is a singleton. If it is a singleton, its spaCy label ("*ent_type_*") is stored into the *"out"* output list as a list; if it is instead part of a chunk, its label is provisionally stored into a buffer list "*buf*" until the whole chunk has been analysed, then the buffer is appended into the "out" list and emptied and the chunk is removed from the waiting list. The full "out" list is returned, which contains, in sentence order, all the identified NE labels as singleton lists or inner lists (for chunks).

b) *groups_freqcount()*

The function is the outer layer of *group_entities()*. It takes as input a file path in string format (which is the *"./conll_str.txt"* file exported in Task 1). It opens the file, then parses each line through spaCy nlp and passes the obtained parsed doc sentence as input to *group_entities()*. The returned "out" list is scanned to count each identified entity group's occurrences (including singletons). The dictionary with the full count for all entity groups is returned.

c) *nbest()*

Function used during lab session to return n max occurrences from a dictionary.

main

In the main section, first the group_entities function is tested on a single sentence. Then, the full *groups.freqcount()* function is called on the *"./conll_str.txt"* file exported in Task 1. From the returned frequency dictionary, only the 12 highest frequencies are extracted and shown in a Pandas dataframe as absolute and normalized frequencies.

3. Task 3 – Expand NE to include compounds

The ***functions.py*** module for Task 1 is imported here, in order to re-use the functions already implemented. The Task 3 module contains 3 functions.

a) *sent_extend()*

This is the companion function for *ne_extend()*. It takes as input a sentence in doc format. It makes use of a series of boolean vars to keep track of the various situations. Also in this case, the var "a" defaults at a high value (1000) when no span is under construction. It stores the list of compounds (token having the "compound" relationship as "dep_") and the list of NEs. Then, for each compound, the steps of the algorithm are the following:

[1] "Label scheme" in https://spacy.io/models/en .

1. What is a NE? The current token or its head?
   - Check if the current compound is a NE (store to related bool). If it is a NE, store the full NE as "orig_ent".
   - Check if the head ("head") of the compound is a NE (store to related bool). If it is a NE, store the full NE as "orig_ent" overwriting the saving in point 1.
2. To what NE does the current "head" refers? Is there any risk that the current token and its head refer to some *different NE* from any span that is already under construction from a previous iteration?
   - Check if the "gran" (head of "head") is a compound.
   - If yes, check if "gran" is a NE.
   - If yes: if "head" was a NE of a different type than "gran", OR if the current token was a NE of a different type than "gran". If one of these, a boolean is created to mark that the head of the head of the current compound actually refers to a different NE. This will be useful to understand where the extended NE span must end.
3. What label should be assigned to the extended NE, if any?
   - Check if a target label has already been found in a previous iteration (if a span is already under construction). If no label was already assigned:
     - If both the current token and its head are NEs of the same type, take this type as target label;
     - If only one among the current token and the head token is a NE, take that type as target label.
4. *Early stop case*: check if you are in a situation in which the current token and its head are not NEs but a span is still under construction from a previous iteration. In this case, *unlock merge* if ALL these conditions hold:
   - a span is under construction (check made on the value of a, which must be different in this case from the default of 1000);
   - both the current compound and its head are NOT NEs;
   - the "gran" (head of the head) of the current token is NOT a different NE (meaning: no risk that the current token and its head refers to some different NE from the span already accumulated).
5. If a target label has been assigned, go ahead with the *span construction*.
   - If no span is currently under construction, start to accumulate the new span from the current compound position in the sentence. The key assumption, indeed, is that any compound is always at the left of what it refers to (regardless of what token is a NE).
   - Checks for setting the end index of the new span:
     - Check the neighbour token on the right of the current compound: if it is not a compound, set the end index of the new span at the position of this neighbour token (it is clear that the compound refers only to it) and unlock merge.
     - Check "diff_gran": if the head of the head of the current compound is a different NE, the end index of the new span must be set at the current token index instead. Then, unlock merge.
     - Check if "*unlock merge*" had already been called on the "early stop case" analysis. If yes, set the end index of the new span at the current token index.
   - If merge is unlocked, prepare the new span and save it as "new_ent".
   - Check if the new entity span ("new_span") will actually be longer than the already existing entity ("old_span" set at the beginning of the iteration), in order not to cut it up.
     - If the check is successful, finally set the new, extended entity on the identified span using spaCy's "set_ents" method, using "default=unmodified" as parameter in order not to change any other label in the sentence.
   - Finally, if the new span has been set as new entity or has been discarded (because shorter than the original entity), reset all variables to start with a new span.

The function returns the sentence in doc format with the updated entity labels (ent_type_ and ent_iob_).

b) ne_extend()

This is the outer layer of *sent_extend()*. It takes as input a list of sentence in doc format (the retokenized list of doc sentences *conll_ret* from Task 1) and simply launches *sent_extend()* on each sentence in the list. It returns the list *ext_ls* of doc sentences with the updated NE labels.

c) label_visualizer()

This function is used to obtain, from the sentence obtained as input:

- a list of [token, NE label (ent_iob_ + ent_type_)] lists,
- a list of [token, token.head, token.dep_ label] lists for its compounds, and
- a list of the NEs in a sentence.

It is used to do the before / after print screen checks on test sentences in the main.

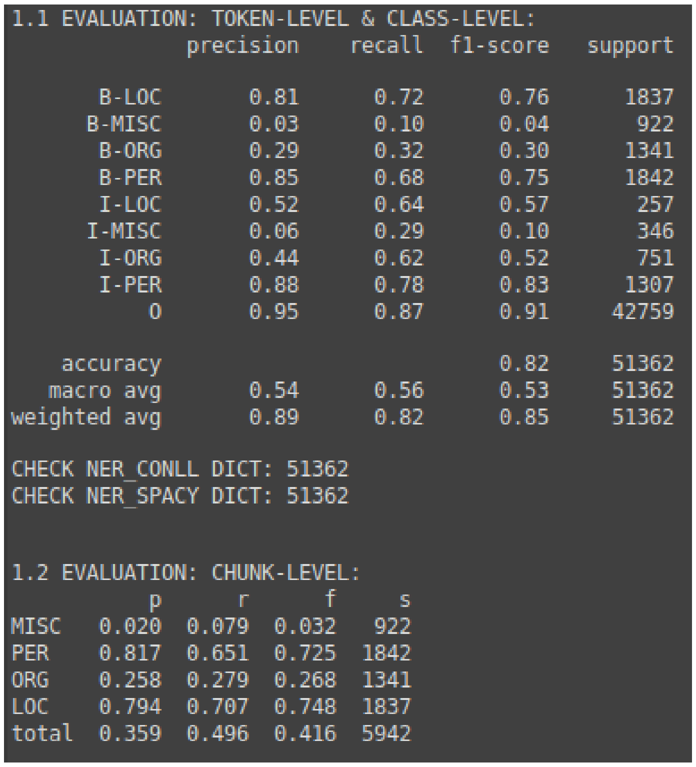
main

In the ***main***, a series of sample sentences are provided, on which *sent_extend()* is called and *label_visualizer()* is called before and after, in order to check the results of the NE extension.

Then the steps taken in Task 1 are repeated, calling, in order, the functions imported from the ***functions.py*** file. The conll text file used is ***dev.txt***. The *dev.txt* file is converted to string list, retokenized (the token count is printed on std output to verify tokenization alignment), extended using the new *ne_extend()* function, then NE labels are remapped using *labels_remap()* and finally the *conll.evaluate()* function is called to evaluate accuracy.

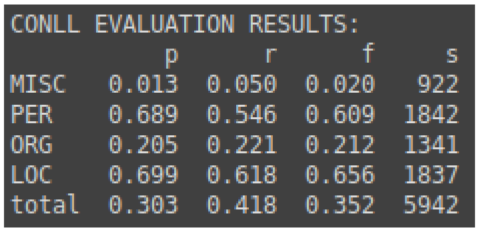
A decrease in accuracy is noted after NE extension, compared to the evaluation done in task 1.

Task 1:



```
1.1 EVALUATION: TOKEN-LEVEL & CLASS-LEVEL:
              precision    recall  f1-score   support

       B-LOC       0.81      0.72      0.76      1837
      B-MISC       0.03      0.10      0.04       922
       B-ORG       0.29      0.32      0.30      1341
       B-PER       0.85      0.68      0.75      1842
       I-LOC       0.52      0.64      0.57       257
      I-MISC       0.06      0.29      0.10       346
       I-ORG       0.44      0.62      0.52       751
       I-PER       0.88      0.78      0.83      1307
           O       0.95      0.87      0.91     42759

    accuracy                           0.82     51362
   macro avg       0.54      0.56      0.53     51362
weighted avg       0.89      0.82      0.85     51362

CHECK NER_CONLL DICT: 51362
CHECK NER_SPACY DICT: 51362


1.2 EVALUATION: CHUNK-LEVEL:
           p      r      f     s
MISC   0.020  0.079  0.032   922
PER    0.817  0.651  0.725  1842
ORG    0.258  0.279  0.268  1341
LOC    0.794  0.707  0.748  1837
total  0.359  0.496  0.416  5942
```

Task 3:



```
CONLL EVALUATION RESULTS:
           p      r      f     s
MISC   0.013  0.050  0.020   922
PER    0.689  0.546  0.609  1842
ORG    0.205  0.221  0.212  1341
LOC    0.699  0.618  0.656  1837
total  0.303  0.418  0.352  5942
```