Master 2 - Mathématiques, Vision et Apprentissage

Homework 2

Speech and NLP

Semestre 2 - 2020

Chadebec Clément

1 The work

Organization of the project The project I conducted is based on 6 distinct modules: 1) *preprocessing.py* 2) *cpfg.py* 3) *grammaire.py* 4) *cyk_parser.py* 5) *train.py* 6) *main.py*

Preprocessing This module basically requires an embedding such as *polyglot-fr.pkl* to be built. Such embedding will then be used to find the closest word of an input sentence in case it is Out Of Vocabulary i.e. this word does not appear in the learned lexicon from the training set. In such a case, the *Part-Of-Speech* that will be used is the one of the "closest" word in our lexicon. The choice I made in the implementation is to 1) check if the work is not in the lexicon we got from the training set; 2) find the closest word in terms of Levenshtein distance in the dictionary from *polyglot-fr.pkl* dictionary (this is performed to handle spelling errors); 3) once this substitute is found I try to find its k neighbors in the dictionary using $l2$-norm on their embeddings and 4) finally I compute the Levenshtein distance between the k neighbors and the words in the lexicon and return the closest one. All of these operations are finally performed in a single function *find_closest* and encapsulated in a *Processor* instance. In this section, only the three normalization functions were taken from the Pyglot embedding tutorial.

PCFG modules This module handles the transformation of the train, test and eval sets along with the grammar computation and PCFG derivation. First of all, the corpus *sequoia* is split into three sets (train (80%), test (10 %) & eval (10 %)). Then, each sentence is processed to be transformed into a *Tree* instance and put in Chomsky Normal Form. The grammar is extracted from the training trees using the *induce_pcfg* module from *nltk*. The lexicon over the training set is computed in this module as well and list any terminal token observed in the training set. All functions are encapsulated in a *CPFG_CNF* class.

Grammaire This module was added to facilitate the grammar handling on my end. It is composed by an instance *Regle* that stores both the probability derived form the *PCFG* module and the left and right symbols composing the rule. The *Grammaire* instance stores each and every rule of the grammar along with the observed lexicon. This was created to group all the corpus elements needed to perform the parsing.

CYK Parser This module is the one where the parser is built. An other instance *Node* was created as well to store 1) the start symbol of a given tree 2) its left and right sub-trees (children) and 3) the probability of such a tree. This is useful in the main parsing function. The *CYK_PROBA* instance is only built from a *Grammaire* instance and a *Processor* instance. Then, the main function *parse* is called to perform the parsing itself. First, the input sentence is preprocessed using the *Processor* instance to remove any misspelled word or any OOV word. As soon as this preprocessing operation is done, the main loop can begin. I will not go to much into the details as the main pseudo-code algorithm can be found easily on wikipedia for example. Nonetheless, in order to make it probabilistic I used the Node instance and at each iteration I compute recursively the probability of the next tree by multiplying the probability of the current observed rule by the probability of the right and left sub-trees (i.e. right and left Nodes). The best tree is then returned and displayed using the *get_best_tree* and *build_tree* functions.

Train This module first performs the construction of the different sets for the training and testing (train trees, test trees). Then, the computation of the grammar is done along with the construction of the *Processor* instance. These are feeded to the *CYK_Proba* instance to build the parser. The parser is trained and everything is then stored using pickle format. The following lines basically only build the *evaluation_data.parser_output* file and perform the scoring on it using *evalb*.

Main This script is the one that is called in the shell file. It allows the user to enter the following command line:

```
python main.py -s 'path_to_sentences' -p 'path_to_parser' -f 'path_to_store
```

where 1) -s requires the path to a *.txt* file containing the sentences to be parsed (in string format with exactly 1 whitespace between each token); 2) -p requires the path to the cyk parser to be used to perform the parsing; 3) -f is the path to a *.txt* file where the parsed sentences will be stored.

2 Performances

In terms of number of sentences handled by the parser, it performed quite poorly as only 56.4% (175 out of 310) of the sentences in the testing set were actually parsed. This is mainly due to the variety of grammatical sentences we have, some of which can reveal to be quite complex to handle. One example of unparsed sentences can be seen below:

- Les hommes politiques -clés de la capitale comme Jean Tiberi , Michel Roussin ou Jacques Chirac ont bénéficié de non- lieu , de vices de forme ou de protection statutaire .

It is composed by a list of proper names which may correspond to a grammatical structure either weakly probable or unseen by the parser during the training. Others like: *Liens externes* do not reveal any contextual grammar and are simple phrases but remain hard to parse.

Nevertheless, on the sentences the parser can actually decode, it seems to perform quite well with a tagging accuracy of 82.24% over the 175 parsed sentences. In order to increase the parser accuracy, one may try to build a comprehensive training set that comprises several samples of such aforementioned adversarial sentences. Moreover, we can think of a language model to enhance the parser accuracy such as a bi-gram model for example. Nonetheless, it can noted that the bigram approximation can be quite hard to compute given the flexibility of the French language which allows many contextual propositions and does not really pay attention to word order. Moreover, the OOV approximation scheme can surely be enhanced by using either other embeddings or trying other ways to combine Levenshtein distance and $l2$-norm. For example, we can try to find the closest word in the *polyglot-fr.pkl* dictionary to each word in the lexicon using Levenshtein distance in order to get an embedding for each word in the lexicon. Then, we do the same for any OOV word in a given sentence and find the closest word in the lexicon using $l2$-distance and the embeddings. Finally, in order to improve the code speed, one may think of multithreading which seems pretty useful in such a situation.