Algorithms for Speech and Natural Language Processing - TP2

Cheng ZHANG

In this report, we explain the technical details for developing a basic probabilistic parser for French. It mainly contains three modules which are: Probabilistic Context-Free Grammar (PCFG) module, Out-of-Vocabulary (OoV) module and Probabilistic Cocke-Younger-Kasami parser (PCYK) module. We will explain these three modules in Section 1 and some error analysis in Section 2.

1 Implementation Details

Data Preprocessing Before implementing the three modules, we first do some data preprocessing. We first eliminate the functional labels in the raw data as demanded and then we split the whole dataset (3099 sentences) to 2479 training sentences and 309 validation sentences and 311 test sentences by a random split. We also get raw sentences without tags for these three subsets. The raw sentences of the training set are used to get Bigram transition matrix which is used to handle Out-of-Vocabulary problem. And the raw sentences of the validation set and test set are considered as input of the final parser to evaluate the performance. These preprocessed data are stored in the `data` folder.

Probabilistic Context-Free Grammar The PCFG module aims to define the grammar rules for the language (here French) from the training corpus so that we can assign a part-of-speech (pos) tag for each word in a sentence according to the grammar rules. Specifically, the PCFG module contains two parts: (i) assign a probability for each transition between tags and the transition terminates at pos tags; (ii) assign a probability for each couple of (token, pos tag) as demanded, which is different from the standard (pos tag, token) version.

To do so, we create a class `Grammar` which stores all rules in the following format:

- for rules `ltag` $\rightarrow$ `rtag`, we store them in a dictionary which takes the couple `(ltag, rtag)` as key and the probability of this rule as value. We also create an auxiliary dictionary which takes `ltag` as key and the list of all possible `rtag`s for this `ltag` as value in order to store all possible rules so that we can track the probability for a specific rule.
- for rules `tag` $\rightarrow$ `token`, we store them in a dictionary which takes the couple `(token, tag)` as key and the probability that the `token` is generated from the `tag` as value. As before, we also use an auxiliary dictionary to track all possible tags for a token.

We store these rules in python dictionary so that we can access to these rules with a time complexity $O(1)$. We define a function `create_pcfg` which takes the training file path as argument and stores all rules in those dictionaries. In this function, we first utilise `nltk` python package to create a tree from each line of training corpus, then we transform the tree to Chomsky normal form (which should be useful for the implementation of CYK algorithm). Specifically, the built-in function `chomsky_normal_form` in `nltk` package transforms the rules to those which contains at most two right-hand-side tags. Then we traverse the tree from the root node and store the number of all `ltag`s and the number of all rules `ltag` $\rightarrow$ `rtag`, and the number of all `token`s and `tag` $\rightarrow$ `token`. After iterating over all training sentences, we can get the probability of each rule by dividing the corresponding number.

Out of Vocabulary module Before applying the CYK algorithm for parsing, we need to first handle the Out-of-Vocabulary problem, which is how should we assign a pos tag to an unseen token while testing. The idea is to find a similar word among the tokens that we already know and then use the rule of the similar word for this unseen word.

To do so, we utilise the Polyglot embedding lexicon for French[1] to measure the similarity between the unseen word and each word among all tokens. We first adapted the Polyglot embedding tutorial[2] to check if a word can be found in the embedding vocabulary. There are two cases to handle with:

[1] `https://sites.google.com/site/rmyeid/projects/polyglot`

[2] `https://nbviewer.jupyter.org/gist/aboSamoor/6046170`

- If the unseen word can be found in the embedding vocabulary, we compute the cosine similarity between the unseen word and each word among all tokens which can also be found in the embedding vocabulary. Then we also generate some candidates among all tokens which are within distance 2 considering the Damerau-Levenshtein distance to the unseen word, in order to handle spelling error. For these candidates, we compute the Bigram probability of the previous word and the next word to each candidate and use this Bigram probability as a part of similarity (a hyperparameter λ is used to adjust the importance of these two similarities). The idea behind this is that even though a word has its embedding, it is also possible that the word has spelling error, so we also need to consider the Bigram probability to make the result robust.
- If the unseen word can not be found in the embedding vocabulary, it is probable that the word has spelling error. In this case, we generate candidates according the Damerau-Levenshtein distance. If there are candidates within distance 2, we only consider these candidates. Otherwise, we consider all tokens as candidates. Then we compute the Bigram probability of each candidate as before and take the Bigram probability as similarity.

The idea behind Bigram probability is that if two words appear in the same context (previous word and next word), they are probable to be similar and have the same tag. Finally, we assign the token which has the largest similarity with the unseen word to this unseen word and utilise the rules associated with the token as the rules for this unseen word.

Probabilistic Cocke-Younger-Kasami parser After handling the Out-of-vocabulary issue, we can finally implement the CYK parser. Our implementation of CYK algorithm refers to Chapter 12 of *Speech and Language Processing*[3] and we modify it slightly to adapt to our `(token, tag)` rule and to handle the unary rule. We start the algorithm by storing the probability of each tag for each token. So firstly if we encounter an unseen word, we assign a similar token to the word by the OoV module mentioned previously. Then instead of considering the probability $p(\texttt{tag} \rightarrow \texttt{token})$, we consider $p(\texttt{token} \rightarrow \texttt{tag})$ to conform with our PCFG implementation. During the finding tags phase, apart from all the two non-terminal rules, we also consider all the unary rules and store the most possible parse results. The experiments show that without considering the unary rule, there is a much greater number of sentences for which we can not find a parse.

2 Error Analysis

We evaluate the performance of different ideas on the validation set and we find that maintaining the unary rule and adapt the CYK algorithm by considering the unary rule is necessary to have a satisfactory performance. We have tried to transform the unary rule to two non-terminals rule (*e.g.* rule $A \rightarrow B$ and $B \rightarrow CD$ will be transformed to $A \rightarrow CD$ and $B \rightarrow CD$). However, with this formulation, the back track chain is more probable to be broken and we can not even find a parser for a certain number of sentences. And we improve the performance by expliciting the unary rule.

We have also tried different ideas for OoV issue (*e.g.* first generate some candidates according to Damerau-Levenshtein distance and only consider these candidates afterwards, which can reduce time consumption as we do not compare all tokens.). However, the final OoV module that we previously explained gives a better performance without augmenting greatly the time complexity.

We use the open-source EVALB package[4] to evaluate our performance and we get a tagging accuracy of 95.90% on the validation set and there are 7 sentences that we are not able to find a parse. By analysing those error sentences, we find that this is because at a certain step of CYK algorithm, we can no more find any corresponding rules. There may be two main reasons: (i) the test sentence really has a structure that we have never seen in the training set; (ii) we may not assign a good similar word which leads to bad tagging at the beginning and thus afterwards. Our system finally get a tagging accuracy of 95.68% with 20 error sentences on the test set.

For further improvement of our system, we may focus on the two reasons. For the first problem, we can take a larger training set. Or we may carry out a k-fold cross validation to fine-tune the best solution and finally train on the whole training set and validation set. For the second problem, we may take a bigger embedding vocabulary and consider more complex N-gram probability to measure the similarity. Or we may take several similar tokens and do the parsing based on all possible rules of all these similar tokens.

[3] `https://web.stanford.edu/~jurafsky/slp3/12.pdf`

[4] `https://nlp.cs.nyu.edu/evalb/`