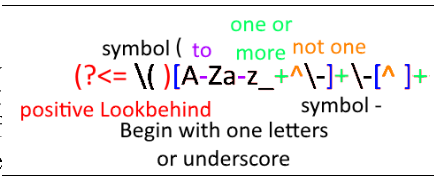
Juliette RENGOT 18 Mars 2019

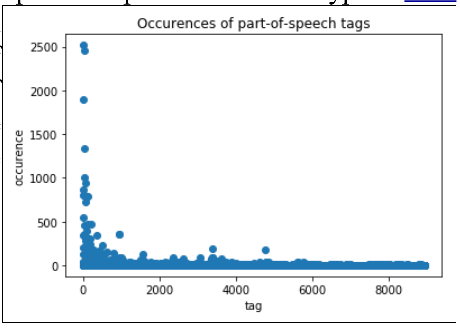
Algorithms for Speech and NLP TD 2

PART 1 : Implementation explanations

First, I load and prepare the corpus, stored in a parenthesis-delineated parse tree (file *prepare_data.py*). Functional labels have to be ignored to avoid sparsity issues. If there is '-' in a non-terminal name, I reject it and everything that follows. To do that, I compile a regular expression pattern (see figure). If there is a match, I keep only what is before '-'. Then, I split the corpus : the first sentences (80%) make the training set, the middle ones (10%) are used for validation and the last ones (10%) compose the testing set. The function *BracketParseCorpusReader* gives the natural sentences. The goal is to be able to find again the parenthesis-delineated parse tree of the test set from these natural sentences.



Afterwards, a Probabilistic Context Free Grammars (PCFG) is extracted from the training set (file *extract_pcfg.py*). I look over the training tree-bank. For each tree, I convert it into a Chomsky Normal Form (the right side of each production rule is either two non terminals or one terminal) to reduce the possible production rule types. This conversion is always possible. I study statistics in different cases : terminal rule (the left side is a part-of-speech tag), unary rule (the left side is made of one symbol) or binary rule (the left side is made of two symbols). The set of part-of-speech tags is imbalanced : some of them are really infrequent (see figure). Not to damage the performances by learning rules from too rare examples, I group all words that occurs less than two times in a '<UNKNOWN>' tag. Finally, the probabilities are obtained by normalising the computed frequencies. A dictionary associating each right symbol to its corresponding left symbols in production rules is also created in order to ease the parent lookup in CYK algorithm.



Next, I focus on the handling of Out Of Vocabulary words (file *OOV.py*). The testing vocabulary contains 823 words that don't appear in the training set (like « éparpillés », « Février » or « G.P.S »). One solution is to replace these words by another one in the training vocabulary. For example, I could replace all of them by the '<UNKNOWN>' tag. A more sophisticated approach could consist in replacing an unknown word by it closest word in the training vocabulary. To decide which word is the closest, two metrics are used :

- The Levenshtein distance (LD) handles spelling mistakes. For more efficiency, I implement it with an iterative approach (bottom-up dynamic programming). The costs for insertions, deletions or substitutions equal to 1. To compute the LD between *word1* of length *M* and *word2* of length *N*, I construct a matrix *L* of size *(M, N)* whose coefficient *(i, j)* is the LD between *word1[:i+1]* and *word2[:j+1]*. The matrix is filled from the upper left corner to the lower right corner. The first row and column are easy to fill. Then, *L[i, j]=min(L[i-1,j]+1, L[i,j-1]+1, L[i-1,j-1]+cost)* with cost equal to 0 if the last characters of *word1[:i+1]* and *word2[:j+1]* are the same of and 1 otherwise. I add the Damereau extension (option *reversal_use*) to allow for adjacent characters swap. Another amelioration can be done by tuning the cost according to the considered letters (option *prob_use*). I compute the probability of apply an operation to a group of letters thanks to confusion matrices of Kernighan, Church and Gale (file *spelling_error_proba.py*). I fix the cost to *1-P* if the probability *P* is computable and 1 otherwise. The last proposed improvement is to take into account the previous word in the probability computation (option *context_use*). It give contextual information that makes the language model more precise. I use a linear interpolation with uni-gram model to avoid unseen bi-grams. It it done with the maximum likelihood estimate (MLE): $P_{\text{linear\_interpolation}}(w_k|w_{k-1}) = \lambda\ P_{\text{uni-gram}}(w_k) + (1-\lambda)\ P_{\text{MLE}}(w_k|w_{k-1})$ with $P_{\text{MLE}}(w_k|w_{k-1}) = \text{cardinal}\{w_k|w_{k-1}\}\ /\ \text{cardinal}\{w_{k-1}\}$. λ is a parameter chosen to be equal to 0,6. To obtain a candidate list of words, I keep all words in the training vocabulary that could be obtained with at most two operations. The distance associated is the computed cost.

- The cosine similarity (CS) handles genuine unknown words. For each word, I look for the closest one in the training set according to the polyglot embedding. This closest word is added to the candidate list.

If the obtained candidate list is empty, the tag becomes '<UNKNOWN>'. Otherwise, the chosen word is the one that minimises *α LD + β CS* where *$\alpha=10^7$* and *$\beta=1$* are coefficients chosen by experimentation. This approach doesn't hold all possible mistakes. I add a preprocessing of the unknown word before computing LD and CS. I try some basic modifications and I check the word is still not in de training vocabulary. Thus, I write it in lower case ('Février'→'février'), compute all groups '*word[:i] word[i:]*' and '*word[:i]-word[i:]*' for *i∊[1, length of word-1]* to allow for space and hyphen addition ('unchat'→'un chat') and split according to '-' character ('député-maire'→'député maire'). The corrected text is saved in *sequoia_test_corrected.txt*.

Finally, I apply the Cocke–Younger–Kasami (CYK) parser (*CYK_parser_class.py*). To transform the original recogniser into a parser, I record the productions (and not just non-terminal nodes) with the positions and lengths of the right side symbols. The parse tree is obtained by looking at the productions from left to right. The result can be evaluated (*eval.py*) by computing precision, recall, F-score and tag accuracy. The following table shows some obtained results and proves the interest of a sophisticated handling of unknown words.

Model	Precision	Recall	F-score	Tag accuracy
No OOV processing	0.57	0.71	0.63	0.75
Non-probabilistic Levenshtein distance	0,92	0,69	0,77	0,7
Probabilistic Damerau-Levenshtein distance (unigram)	0.9	0.7	0.78	0.71
Probabilistic Damerau-Levenshtein distance (mle)	0.93	0,7	0.79	0.72

PART 2 : Error analysis

The obtained trees are almost always correct if all words belong to the training vocabulary. Little mistakes like misspelling in frequent words, error in conventions or conjugations don't disturb the tree construction. One missing letter or one incorrectly added character seem not to be an issue if it results in an unknown word in the vocabulary. However, if it results in an existing word in the vocabulary (for example 'lièvre' becoming 'lèvre'), the error won't be detected and it can prevent the tree construction to be correct. The problem is that there is no meaning analysis of the considered sentence. A more sophisticated language model could detect absurd word in natural language sentence to help the parser to correct these mistakes. To do that, I should analyse the whole sentence whereas or treatment is done at word scale. This modification should allow for other improvements. Thus, there is nothing done for missing word or redundant word. If one writes 'Je vais au au marché', the word 'au' will appear twice in the tree whereas the second one should be deleted. I could imagine to look at each word in the considered sentence and suppress each word such as the previous one is the same. Not to suppress correct stuff, I should check in a bi-gram model that this specific word couldn't be repeated. Thus 'aïe aïe aïe' won't be modified. Next, I can note that the kind of word is not taken into account in the closest candidate generation. For example, the word 'institut' has one candidate word that is 'instituant' (find with LD measure). That makes sense because these words belong to the same family. Otherwise, the first one is a noun whereas the other one is a present participle. This kind difference is an issue for the tree construction. A grammatical model could be added to favour replacement of words if they are of the same kind.