MVA - Algorithms for Speech and NLP: **TD 2**

Souhaib Attaiki

Mars 2019

Objectif du TP L'objectif de ce TP est de créer un parser probabiliste pour l'étiquetage morpho-syntaxique pour la langue française, basé sur SEQUOIA treebank v6.0, en utilisant le modèle PCFG, et en implémentant l'algorithme CYK probabiliste et un module pour gérer les mots qui n'apparaissent pas dans le training dataset. Nos différents implémentations sont basées sur le chapitre 12 de l'ouvrage [1].

1 Apprentissage du Parser

Nous avons utilisé *Python 3* pour implémenter les différentes briques. Il est à noter qu'en dehors de *Numpy*, nous n'avons pas utilisé de bibliothèques externes, et nous avons implémenté tout *"from scratch"* (*graphviz* est utilisé pour dessiner l'arbre, mais il n'est pas nécessaire pour le fonctionnement du parser.). Pour ce faire, nous avons procédé en cinq étapes :

Manipulation de la treebank C'est la partie qui nous a pris le plus de temps. Le but de cette dernière est de créer un arbre à partir d'une ligne avec des parenthèses. Notre méthode consiste à transformer la ligne en plusieurs listes imbriquées,

`( (SENT (NP (NPP Gutenberg)))) → [[SENT, [NP, [NPP, Gutenberg]]]]`

et d'utiliser une fonction récursive pour extraire l'arbre puisque les différents niveaux de cette dernière sont facile à manipuler (voir `parser/tree.py`)

Forme normale de Chomsky à partir de l'arbre créé, nous extrayons les différentes règles de la grammaire qui correspondent simplement aux arêtes sortantes de chaque noeud. Puis nous les transformons dans la forme normale de Chomsky, d'abord, par l'élimination des règles unitaires en remplaçant les règles de la forme $\alpha \rightarrow \beta \rightarrow \gamma$ par la règle $\alpha \rightarrow \gamma$, et deuxièmement, par la suppression des membres droits avec plus de deux symboles en utilisant une factorisation à droite : $\alpha \rightarrow \beta|\gamma|\sigma|\zeta$ devient $\alpha \rightarrow \beta|\gamma\_\sigma\_\zeta$, $\gamma\_\sigma\_\zeta \rightarrow \gamma|\sigma\_\zeta$ et $\sigma\_\zeta \rightarrow \sigma|\zeta$. Ces règles sont ensuite stockées dans des dictionnaires qui, pour chaque règle, associent son nombre d'occurrences (voir `parser/tree.py`). On a pas utilisé les autres règles de normalisation parce qu'elles ne correspondent pas à notre cas.

Extraire les règles PCFG Cela correspond au calcul des différents probabilités. Pour cela, on se base sur les nombre d'occurrences des différentes règles, et sur la formule:

$$P(\alpha \rightarrow \beta) = \frac{Count(\alpha \rightarrow \beta)}{\sum_\gamma Count(\alpha \rightarrow \gamma)} = \frac{Count(\alpha \rightarrow \beta)}{Count(\alpha)}$$

(voir `parser/pcfg.py`)

Implémentation du CYK probabilisé Nous nous sommes basés sur la formulation de l'algorithme contenu dans le chapitre 12 de [1]. Pour rendre notre implémentation plus efficace, nous avons créé des dictionnaires (qui ont un temps de recherche constant) pour stocker les différentes règles de grammaire et probabilités qui seront nécessaires pendant l'exécution. Par exemple, un dictionnaire, qui à chaque *token* associe POS tel que $POS \rightarrow token$, ou un dictionnaire, qui pour chaque membre droit BC, associe tous les membres gauche A qui le génèrent $A \rightarrow BC$. Nous avons aussi implémenté une fonction qui inverse l'effet de la normalisation de Chomsky après CYK pour avoir un parsing similaire au données d'entraînement.

Gestion des OOV On a utilisé 90% du dataset pour l'entraînement, et 10% pour l'évaluation. Lorsqu'il s'agit d'un mot OOV, la piste typo est d'abord considérée et on lui assigne un mot avec une distance de Levenshtein de moins de deux. Si aucun mot de la base de données ne correspond à ce critère, on lui attribue le mot le plus proche en termes de similitude cosinus en utilisant des embeddings pyglot.

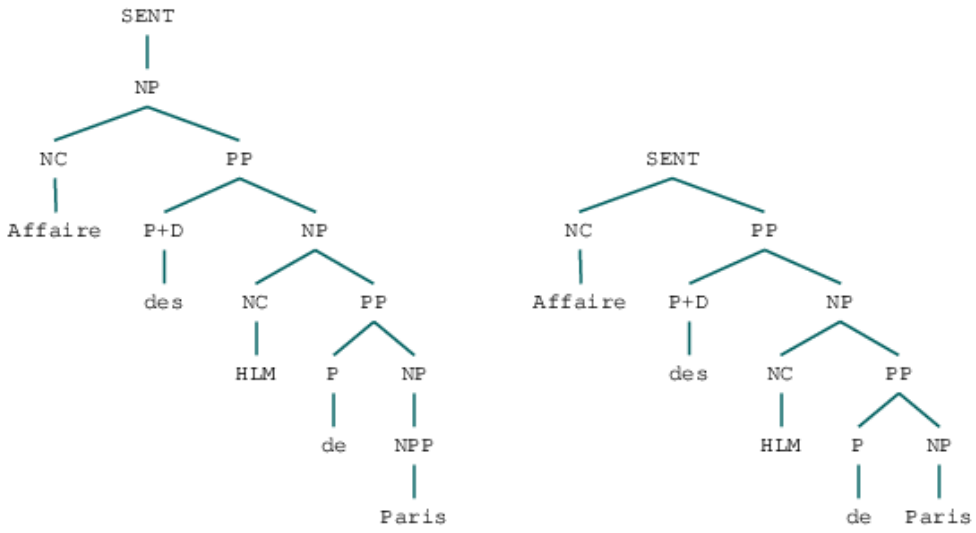
2 Analyse des résultats

Pour évaluer notre parser, on a utilisé *"evalb"*. Dans la plupart des cas, notre implémentation tourne en une seconde, et ce temps s'allonge si nous avons affaire à de très longues phrases.

Tout d'abord, nous avons testé notre implémentation sur 190 phrases de l'ensemble d'entraînement pour voir si elle fonctionne correctement. Une précision de 86 % a été obtenue. Nous remarquons que notre parser parse correctement les phrases courtes, et se trompe dans quelques phrases longues. On constate aussi que nous ne pouvons pas récupérer les règles de la forme $\alpha \rightarrow \beta \rightarrow \gamma$ parce qu'elles ont été retirées de la base pendant l'entraînement. Un exemple de parsing est illustré à la figure 1 (les arbres sont dessinés à l'aide de NLTK).

Cependant, lors de l'évaluation sur l'ensemble de test, nous avons été confrontés à des cas où nous n'avons pas trouvé de parsing commençant par *"SENT"*, auquel cas nous avons choisi de ne rien envoyer au lieu d'envoyer un parsing qui ne commence pas par "SENT". On a obtenu une précision de 67 %.

Une idée pour améliorer le système est de ne pas supprimer les règles unités pendant la normalisation, mais de les remplacer par des règles qui permettent de les récupérer après le parsing. Une autre idée est d'utiliser un ensemble d'entraînement plus grand, pour avoir d'une part un vocabulaire plus grand et donc moins d'OOV, et d'autres part, pour améliorer la qualité de la grammaire.



(a) (b)

Figure 1: Gauche: vérité terrain — Droite: notre prédiction

References

[1] James H. Martin Daniel Jurafsky. Speech and language processing: An introduction to natural language processing, computational linguistics, and speech recognition. 2018.