
Travaux Pratiques : Probabilistic parser, CYK algorithm and PCFG model

Thibault Cordier
ENS Paris Saclay - Master MVA
thibault.cordier@ens-paris-saclay.fr

1 Présentation du travail

Dans cette section, nous présentons notre travail, le fonctionnement des composantes de notre solution et la logique derrière laquelle se cache les choix que nous avons réalisés.

1.1 Gestion des données

Nous avons tout d'abord pensé astucieusement la gestion des données à travers la classe `ManageData` qui permet de rendre la lecture et l'écriture de fichiers plus aisées. En particulier, les données sont stockées dans un dictionnaire `data` hiérarchisées selon une première clé correspondant à la catégorie des données (`train`, `valid`, `test` etc) et une seconde correspondant à la nature des données (phrases brutes, phrases annotées, arbres etc).

De plus, la classe a été pensée pour pouvoir appliquer des fonctions sur l'ensemble des lignes d'un ensemble de données avec la méthode `process_data`, facilitant ainsi la conversion de données d'une nature donnée à une autre (ex : conversion de phrases annotées en arbres avec la fonction `line2tree`). Naturellement, nous avons trouvé utile de définir l'ensemble de toutes les fonctions de conversion d'une nature à l'autre (`line2tree`, `tree2line`, `tree2sent`, `gf2cnf`, `cnf2gf`).

Aussi, pour éviter de générer systématiquement la grammaire et le lexique de notre modèle (voir 1.2) chaque fois que nous réalisons un *parsing*, nous avons décidé de garder en mémoire toutes les informations ainsi générées une unique fois grâce à la librairie `pickle` et d'y accéder rapidement par la suite.

1.2 Module PCFG

Le premier module principal de la solution contient la classe PCFG permettant d'extraire un *probabilistic context-free grammar* et un *probabilistic lexicon* à partir d'un corpus d'entraînement (composés d'arbres sous forme générale extrait des phrases annotées grâce à la classe `ManageData`).

La constitution de la grammaire N , du lexique W et des règles de productions grammaticales R et lexicales T de notre modèle s'opère par mise à jour successive lors de la visite des arbres relatives aux phrases annotées de notre corpus.

Au cours des visites, nous prenons le temps de transformer les arbres sous forme générale (`gf`) en arbres sous forme normal de Chomsky (`cnf`) anticipant l'utilisation de l'algorithme CYK mais permettant aussi de prendre en compte la grammaire auxiliaire ainsi générée. En effet, il est important d'opérer cette transformation avant de constituer les règles de productions pour ne pas les oublier lors de cette étape.

Nous prenons soin aussi de compter les occurrences des éléments de $\# \in \{N, W, R, T\}$ dans `#idx2count` - pour prévoir le calcul des probabilités dans `Ridx2prob` et `Tidx2prob` par la méthode d'approximation fréquentielle - et d'indexer leur apparition dans `#idx2sentidx` pour prévenir les `debugs` si nécessaire.

1.3 Module OOV

Le second module contient la classe OOV dont le but principal est d'assigner un unique *part-of-speech* à tout *token* qui ne serait pas inclu dans le lexique grâce à la méthode `assign_POS` - il s'agit de lui attribuer un mot "similaire" du lexique dont nous connaissons des règles lexicales.

Nous avons donc implanté un algorithme de similarité combinant la similarité "formelle" (pour traiter les erreurs d'orthographe) avec la similarité d'*embedding* (pour gérer les mots inconnus) qui, par défaut, est mesurée par la similarité cosinus.

- (1) Si le mot *word* est inclu dans le lexique : Le mot *word* est donc un vrai mot. Renvoyer *word*.
- (2) Sinon, si le mot *word* est inclu dans l'*embedding* (l'ensemble des mots dont nous connaissons un *embedding*) : Le mot *word* est un vrai mot mais dont nous ignorons la nature lexicale. Renvoyer le mot de l'*embedding* le plus proche de *word* inclu dans le lexique.
- (3) Sinon, chercher des mots du lexique le plus proche de *word* d'après la similarité "formelle". Si il en existe un : Le mot *word* a été potentiellement mal orthographié. Renvoyer le corrigé.
- (4) Sinon, chercher des mots de l'*embedding* le plus proche de *word* d'après la similarité "formelle" inclu dans le lexique. Si il en existe un : Le mot *word* a été potentiellement mal orthographié mais dont, après correction, nous ignorons toujours la nature lexicale. Renvoyer le meilleur mot similaire des mot de l'*embedding* les plus proches de *word*.
- (5) Sinon, aucun mot n'a été trouvé.

Pour le calcul de la similarité "formelle", nous avons choisi d'opter pour la distance de Damerau Levenshtein puisqu'elle prend en compte les transformations les plus fréquentes à savoir l'insertion, la suppression ou la substitution d'un simple caractère, ou la transposition de deux caractères adjacents.

1.4 Module CYK

Le dernier module proposé contient l'implantation de l'algorithme CYK, un *parser* qui prend en entrée une phrase *tokenised* et retourne en sortie la phrase annotée selon le format imposé par le corpus.

Pour éviter de manipuler des grandeurs extrêmement faibles et aboutir à une non-convergence de l'algorithme, nous avons décidé de calculer les log-probabilités au lieu des probabilités directement.

Dans la phase d'initialisation des log-probabilités, nous avons décidé d'imposer un prior uniforme sur les POS si un mot de la phrase et ses mots "similaires" étaient inconnus du lexique.

2 Analyse de la solution

Dans cette seconde section, nous présentons une brève analyse de notre solution, ses points forts et faibles ainsi que quelques suggestions d'amélioration.

2.1 Points forts de la solution

Nous avons tous mis en oeuvre pour que la solution soit *user-friendly*, avec des classes qui s'initialisent sans assistance (ex : PCFG) et qui soient gérables avec peu de méthodes. Après validation de notre solution, nous avons obtenu des résultats toutefois satisfaisants (avec presque 80% d'*accuracy*). A la lecture de quelques exemples que renvoie le *parser*, les résultats semblent grammaticalement satisfaisants dans la majeure partie des cas. Enfin, le module OOV est un point fort de la méthode qui permet de faire face, dans certains cas, à des mots inconnus ou mal orthographiés.

2.2 Points faibles de la solution

Cependant, notre solution faiblit par sa complexité temporelle qui peut s'avérer pénalisante lorsque nous considérons de longues phrases. Qui plus est, elle n'est pas infaillible face aux mots rares qui peuvent passer au travers du filet du module OOV. Et sans surprise, même si le *parser* annote correctement bien les phrases, il n'est pas garanti qu'il trouve le *parsing* désiré dû à des ambiguïtés grammaticales - dans le cas où une phrase peut avoir plusieurs interprétations.

2.3 Suggestions d'amélioration

- Mettre à jour le lexique pour tout nouveaux mots rencontrés.
- Améliorer l'algorithme d'assignation d'un POS. En situation (5), réaliser un traitement sur le mot (décomposition du mot, recherche de préfixe-radical-suffixe, mise en relation des mots de même famille, prise en compte des différents formats des dates/nombres, standardisation des formats (singulier/pluriel, majuscule/minuscule, conjugaison ...)).
- Améliorer la fonction de conversion *gf2cnf* pour mieux gérer la génération de nouvelles tags auxiliaires, plus sophistiquées.
- Prendre en compte les labels fonctionnels que nous avons mis de côté selon les conseils du TP.
- Améliorer/Nettoyer le corpus et l'*embedding* pour couvrir plus de mots.