ተቋም национален שפה 文化 شرقية

inalco

Institut national
des langues
et civilisations orientales

TALB530B - Programmation itérative et récursive - INALCO 2023/24

RAPPORT DU GROUPE 1 : l'étiquetage morpho-syntaxique

Sandra JAGODZIŃSKA
Valentina OSETROV

Introduction

Ce rapport présente le travail réalisé par l'équipe 1, dont l'objectif était de fournir un module de traitement pour l'étiquetage morpho-syntaxique en partie du discours (Part-Of-Speech tagging). Nous y détaillerons les différentes étapes de notre travail, nos choix technologiques ainsi que les contraintes rencontrées.

Choix technologiques

Pour implémenter un module de POS tagging, nous avons considéré plusieurs outils : NLTK, spaCy et TreeTagger. Nous avons évalué principalement leur adaptabilité aux besoins de l'ensemble des équipes travaillant sur le projet.

Malgré l'indisponibilité d'étiqueteur NLTK pour le français, il reste utilisable en téléchargeant des étiqueteurs complémentaires développés par Stanford, écrits en Java. Cependant, nous avons rencontré une difficulté due au manque de clarté dans la documentation du système d'annotation, notamment en ce qui concerne les étiquettes utilisées. De ce fait, l'utilisation de NLTK (et du Stanford POS tagger) aurait nécessité un mapping pour assurer l'uniformité des étiquettes entre les différentes équipes, ce qui aurait généré des problèmes inutiles.

L'intégration de TreeTagger avec les autres modules utilisés dans le projet aurait également demandé un effort supplémentaire, puisqu'il est limité à l'étiquetage en partie du discours.

Finalement nous avons choisi d'utiliser la bibliothèque python spaCy en raison de sa qualité de production, de sa disponibilité en français et de sa compatibilité très facile avec les modules traités par les autres équipes. De plus, nous avons utilisé spaCy régulièrement et il fonctionne bien sur nos ordinateurs. Ce module n'est pas parfait, mais tout bien considéré, c'est le meilleur choix.

Défis et solutions

Version 1 du code

La première version du code traitait des données d'entrée (le corpus sous forme de fichiers txt) de manière à supprimer tous les sauts de ligne et à ne laisser que le texte nécessaire à l'annotation. Ensuite, l'élément "doc" de la bibliothèque spaCy a été initialisé dans la fonction récursive et a été passé en argument à chaque appel. Le résultat d'annotation était présenté sous la forme d'une liste des tuples contenant le token et son POS tag.

Exemple : [('le','DET'),('bon','ADJ'),('docteur','NOUN')]

Cependant, cette approche nous a fait perdre beaucoup d'informations sur la forme du texte. Après délibérations avec les autres groupes, l'équipe n°5 nous a demandé de stocker les annotations sous la forme d'un dictionnaire, où les clés sont des numéros id uniques à chaque token, reflétant leur position dans une phrase, et les valeurs sont des dictionnaires représentant le token lui-même et son POS tag.

Exemple : {1 : {'le':'DET'}, 2 : {'bon':'ADJ'}, 3 : {'docteur':'NOUN'}}

Une difficulté résidait dans le fait de maintenir l'ordre des tokens afin de garder la structure logique des positions grammaticales. C'est pour cela que la sortie n'est plus une liste mais un dictionnaire des dictionnaires où la clé (du "grand dictionnaire") est l'index attribué aux tokens et est augmentée à chaque itération.

Version 2 du code

Étant donné que l'objet "doc" de la bibliothèque spaCy a été initialisé dans notre fonction récursive, il était inaccessible pour les autres groupes. Vu qu'ils ont choisi le module spaCy et avaient également besoin de cet objet pour leurs annotations, il était crucial de trouver un moyen de le rendre. Afin de ne pas ralentir l'intégralité du code final après des discussions avec les autres groupes, nous avons décidé d'ajouter une autre fonction recursive *process_gp1* qui initialise la pipeline de la bibliothèque spaCy et renvoie un objet doc. Cette approche assure la plus grande réutilisation possible du code.

Pour aller plus loin, afin de rendre l'ensemble du processus plus facile, nous avons ajouté une autre fonction récursive *preprocess_gp1*, qui lit un fichier ligne par ligne, supprime des saut de lignes et renvoie une liste contenant des lignes de texte d'un fichier donné en entrée. Grâce à cela, les autres groupes n'auront pas besoin de s'occuper de la lecture des fichiers et de l'initialisation de l'élément doc et peuvent utiliser notre chaîne de traitement en utilisant les fonctions preprocess et process ensemble.

Enfin, le traitement en lui-même se passe avec deux fonctions récursives. La fonction *recursive_objet_doc* parcourt l'ensemble des objets *doc*. Elle appelle une autre fonction récursive, *recursive_tokens_pos*, pour traiter chaque token du document. Pendant l'annotation des tokens récursive, un dictionnaire vide est initialisé pour stocker les POS tags et le document est parcouru token par token jusqu'à la fin du document, en ajoutant les nouveaux tags dans ce dictionnaire sous la forme imposée (c'est-à-dire {token:pos}) et en tenant compte de l'ordonnement des mots.

Mesures de complexité

Mesure de complexité empirique en espace

Nous avons mesuré la complexité empirique en temps et en espace de notre algorithme. Notre première version du code calculait l'espace maximal alloué ainsi que l'espacé alloué par le résultat de notre algorithme, afin de bien visualiser la quantité d'espace nécessaire à l'exécution de l'algorithme. La différence entre les deux est qu'une fois que le code s'arrête, toutes les variables inutiles sont supprimées par le garbage collector, et seulement le résultat final est gardé en mémoire. Avec les modifications de notre code et les versions ultérieures, le calcul d'espace maximal résultait en ajoutant des masses de lignes de code inutiles. Comme ce calcul n'était pas l'objectif principal de notre expérience, nous avons abandonné cette idée pour avoir le code le plus clair et simple mais aussi bien axé sur notre objectif.

Nous avons calculé l'espace mémoire alloué par notre résultat final. Pour cela nous avons construit un dictionnaire Python. Ce dictionnaire associe le nom d'élément, en tant que clé, à une liste contenant sa longueur, en tant que valeur. Par la suite, nous ajoutons à cette liste des longueurs des annotations successives. Il est important de noter que nos annotations sont des dictionnaires contenant des paires constitués d'un token et de son étiquette. Pour déterminer la consommation mémoire totale, nous calculons la somme des longueurs de ces annotations. Cette somme équivaut à la taille du corpus, mais vu que chaque élément de chaque dictionnaire contient deux éléments (token(clé) et tag(valeur)), nous voyons que l'espace alloué réel est égal à 2n où n représente la longueur de corpus exprimée en tokens.

Selon nos observations empiriques, l'espace mémoire alloué par le résultat d'algorithme présente une complexité O(*n*). Ce résultat est attendu, car l'ajout d'une étiquette à chaque token double effectivement la mémoire utilisée.

Remarque code : Cette partie du code a initialement été calculée à la fin du script (dans if *"__name__" = "__main__"*). Cependant, après la consultation finale avec le groupe 5, qui a besoin d'accéder à nos résultats, une fonction dédiée spécifiquement au calcul de la complexité en espace (*get_complexity_space*) a été créée, et est appelée à la fin.

Mesure empirique du temps

Nous avons également évalué empiriquement la complexité temporelle de notre algorithme. À travers plusieurs expériences, nous avons pu observer que le temps de calcul augmente proportionnellement à la taille des données. Comme l'algorithme traite chaque token, nous pouvons conclure que la complexité temporelle empirique de notre algorithme équivaut O(*n*). Cette observation est conforme à la théorie, car notre algorithme doit étiqueter chaque unité de texte de l'input.

Cette linéarité est une caractéristique inhérente à notre traitement, car chaque token est individuellement analysé, sans opérations supplémentaires qui soient coûteuses en termes de temps de calcul.

Remarque code : Tout comme pour le calcul de complexité en espace, nous avons dû réorganiser notre code afin de permettre l'accès à la variable de calcul de complexité en temps au groupe 5. La fonction "*get_complexity_time*" est donc dédiée au calcul de complexité du temps du processing. Ensuite, le résultat (le fichier processé) est enregistré dans un fichier json.

Complexité O(*n*)

La notation Big O est une notation mathématique qui décrit le comportement asymptotique d'une fonction lorsque l'argument tend vers une valeur particulière ou l'infini. Alors, la notation Big O caractérise les fonctions en termes de leur taux de croissance. Ainsi, différentes fonctions ayant le même taux de croissance peuvent être représentées par la même

notation Big O. En informatique, cette notation est particulièrement utilisée dans l'analyse de la complexité des algorithmes.[1]

Dans notre étude, nous avons constaté que les deux complexités sont équivalentes à O(n). Notre fonction récursive *"recursive_objet_doc"* parcourt l'ensemble des objets *doc*. Ensuite, cette fonction récursive appelle une autre fonction récursive *"recursive_tokens_pos*", pour traiter chaque token du document. Étant donné que chaque token est traité exactement une fois, la complexité temporelle de cet algorithme est linéaire, soit O(n), où n représente la longueur de tous les documents exprimée en nombre de tokens. De plus, comme chaque token est traité une fois et qu'une étiquette lui est assignée, la complexité en espace est également linéaire, soit O(n).

Ces résultats sont cohérents avec la théorie de la notation Big O et avec les complexités théoriques des algorithmes similaires, où il est nécessaire de parcourir et traiter l'ensemble des données d'entrée. La linéarité de la complexité temporelle et spatiale s'explique par le fait que chaque élément des données est manipulé une fois au cours de processus de traitement. Nos expériences attestent cette analyse théorique, renforçant ainsi notre conclusion.

Bibliographie :

1. "*Big O notation*." Wikipedia, https://en.wikipedia.org/wiki/Big_O_notation. Visitée le 21 mai 2024.
2. Shen Huang, "*What is Big O Notation Explained: Space and Time Complexity*", https://www.freecodecamp.org/news/big-o-notation-why-it-matters-and-why-it-doesnt-1674cfa8a23c/ . Visitée le 20 mai 2024
3. Mehdi Zed, "Comprendre la notation Big O en 7 minutes", https://www.jesuisundev.com/comprendre-la-notation-big-o-en-7-minutes/. Visitée le 21 mai 2024.
4. "Big O Notation: A Complete Overview With Real-World Applications" https://www.simplilearn.com/big-o-notation-in-data-structure-article. Visitée le 19 mai 2024.

[1] "Big O notation", Wikipedia, https://en.wikipedia.org/wiki/Big_O_notation visité le 21 mai 2024

5. “Time Complexity analysis in Data Structures and Algorithms” https://www.enjoyalgorithms.com/blog/time-complexity-analysis-in-data-structure-and-algorithms. Visitée le 21 mai 2024.
6. CFAURY, Complexité d’un algorithme – l’Informatique, c’est fantastique ! 3 Nov. 2019, https://info.blaisepascal.fr/nsi-complexite-dun-algorithme/
7. “Language Processing Pipelines · spaCy Usage Documentation.” *Language Processing Pipelines*, https://spacy.io/usage/processing-pipelines
8. *The Stanford Natural Language Processing Group*. https://nlp.stanford.edu/software/tagger.shtml