Enrichissements du corpus avec des analyseurs en dépendances syntaxiques

Mot d'introduction

Après un changement de groupe, il vous faudra maintenant vous familiariser avec le code laissé par le groupe précédent et continuer le travail. La première étape sera donc de relire et et vérifier que les attentes de la semaine précédente sont bien remplies (à commenter dans le journal de bord). Il faut vérifier que le code de départ est capable de lire et filtrer l'arborescence de fichiers RSS et de sauvegarder le corpus filtré sur le disque.

Pour cette semaine, le nouvel objectif sera d'enrichir le corpus avec les sorties de différents analyseurs en dépendances syntaxiques. Il faudra commencer par prendre en main les différents outils (ne pas hésiter à consulter leurs documentations !)

Le code obtenu en fin de semaine devra pouvoir recharger un corpus, ajouter les analyses d'un des trois analyseurs proposés dans un format unifié et de sauvegarder le résultat de l'analyse.

Pour rappel

— un nouveau groupe gitlab vous a été attribué aléatoirement pour la semaine. Vous y trouverez un dépôt à cloner.
— la branche **main** sert à l'avancée du projet et des exercices, mais elle ne doit contenir que du code finalisé. il faut y pousser le moins souvent possible.
— une branche **doc** sert au rendu du journal de bord (un fichier markdown *différent* par semaine),
— chaque semaine, vous devrez créer des branches individuelles réservées au travail de chaque membre du groupe.
— un tag xxx-fin doit être utilisé pour indiquer qu'un exercice et terminé et un tag xxx-relu indiquera qu'il a été relu par un tiers et est prêt à être fusionné (**merge**). Un dernier tag indiquera que le travail sur la branche **main** est terminé.
— pour rappel, les **xxx** d'un tag seront à remplacer par **xy-sTrN**, où xy sont vos initiales, T le numéro de la séance et N votre rôle.

Exercice 1 Prise en main d'outils

La première étape sera donc de prendre en main différents outils d'analyses. Vous devez avoir déjà installé **spacy**, **stanza** et **trankit**.

Chaque membre du groupe travaillera de façon autonome à prendre en main un des trois outils.

Vous pouvez consulter leurs documentations respectives :

spacy `https://spacy.io/`

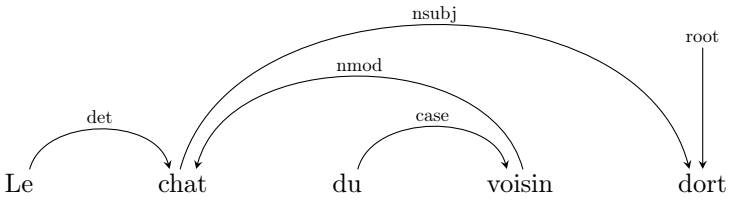
stanza `https://stanfordnlp.github.io/stanza/`

trankit `https://trankit.readthedocs.io/en/latest/`

Vous devez pouvoir lancer l'analyse d'un petit texte et expliquer comment récupérer pour chaque texte :
— la liste des phrases d'un Item, une phrase sera une liste de tokens,
— il faudra ajouter à chaque token l'information de son gouverneur/ses dépendants et de la nature du lien.

Il faudra peut-être modifier le modèle ou la façon de charger le modèle en fonction de l'approche adoptée par le groupe précédent. Les outils sont généralement bien documentés de ce côté là.

Pour rappel, par simplicité de représentation, les outils utilisent généralement la représentation suivante en dépendances qui permet d'inclure l'information de l'arc en dépendance simplement sur chaque token :



Exercice 2 Script de démonstration indépendant

Dans un premier temps, vous pouvez écrire un petit script de démonstration de l'outil, **demo_deps_xxx.py** (où xxx est spacy, stanza ou trankit). Ce script vous servira à appréhender les interfaces des outils (méthodes, structures de données, logique générale de l'utilisation).

Exercice 3 Intégration des dépendances syntaxiques

Modifiez le fichier **analyzers.py**, afin que la ou les fonctions d'analyse enrichissent les `Token` de l'analyse en dépendances syntaxiques que peut fournir chaque outil.

Vous devrez modifier le fichier **datastructures.py** afin que la classe `Token` intègre les informations de l'analyse syntaxique en dépendances. Cette analyse étant faite à l'échelle d'une phrase, il faudra sans doute faire évoluer vos *dataclasses* pour rendre compte de cette structure, de même que le code de sérialization.

Proposez une fonction principale (**main**) dans le fichier **analyzers.py** afin que celui-ci puisse être utilisé comme une **commande bash** qui charge un corpus précédemment sauvegardé, l'analyse avec votre outils et sauvegarde le résultat. Les fonctions de (dé)sérialisation devront être mises à jour pour intégrer les analyses (les lire ou les écrire quand elles sont présentes).

Dans un premier temps, proposez en groupe une représentation pour intégrer ces nouvelles informations au niveau des *dataclasses*. Une fois cette base commune décidée, le travail devra se répartir de la façon suivante :

r1 intégrera les dépendances syntaxiques fournies par `Spacy` et mettra à jour la sérialization `XML`

r2 intégrera les dépendances syntaxiques fournies par `Stanza` et mettra à jour la sérialization `json`

r3 intégrera les dépendances syntaxiques fournies par `trankit` et mettra à jour la sérialization `pickle`

Dans la mesure du possible, utilisez un outil différent de celui que vous avez utilisé la semaine dernière.

Exercice 4 Extraction des premiers patrons

Dans le fichier `patterns.py`, vous écrirez les fonctions relatives à l'extraction de ces patrons. Nous intégrerons dans un premier temps l'extraction de trois patrons :

r1 le patron de dépendances `verbe --obj-> nom` pour recueillir les verbes et les noms qu'ils ont pour objet

r2 le patron de dépendances `verbe --suj-> nom` pour recueillir les verbes et les noms qu'ils ont pour sujet

r3 le patron de dépendances `nom --nmod-> nom` pour recueillir les compléments du nom

Proposez une fonction principale (**main**) dans le fichier **patterns.py** afin que celui-ci puisse être utilisé comme une **commande bash** qui charge un corpus précédemment sauvegardé, fait l'extraction des patrons et sauvegarde le résultat. Pour chacun de ces patrons, on récupérera leurs instances (les lemmes retrouvés avec ce patron) ainsi que leurs comptes. Le résultat sera à écrire sous forme tabulaire dans un fichier qu'on donnera en argument (ou sur le terminal sinon). Les informations à écrire dans le fichier tabulaire seront le patron, l'instance et le compte.

Exercice 5 Mise en production

Comme pour les semaines précédentes, validez le code d'un(e) de vos camarade et ajoutez un tag **xxx-relu** quand le résultat est satisfaisant.

Finalement, fusionnez vos travaux et proposez une version finale combinant les différentes contributions sur la branche **main**.

La fonction principale et les arguments proposés avec **argparse** doivent être adaptés en conséquence pour permettre à l'utilisateur de choisir un des trois outils d'analyse.

Indiquez que le travail du groupe est terminé au moyen d'un tag.

Exercice 6 Mise à jour du journal de bord

Pour ce travail, chaque membre renseigne sa partie du journal de bord, qui sera hébergé sur la branche **doc**. Commentez :

1. vos difficultés
2. vos solutions
3. les choix lors des *merges*

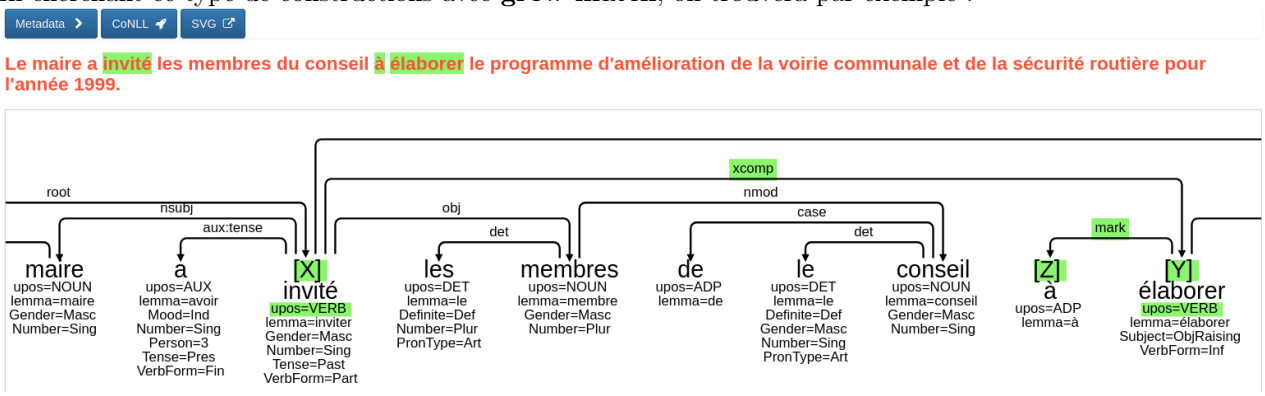
N'hésitez pas à ajouter quelques indications et conseils pour le groupe qui reprendra votre code la semaine suivante !

Exercice 7 Question bonus 1 (préparer la suite)

Si l'on pense à notre objectif final et qu'on observe les données fournies par *Les voisins de Le Monde*, on constate que pour certains mots sont "recollés" à une préposition ou un mot subordonnant qui leur est souvent associé. On peut consulter par exemple les tableaux pour *regarder (avec/vers/dans)* ou *inviter (à)* qui sont considérés au même niveau que les relations plus directes d'*obj* ou *suj*.

`http://redac.univ-tlse2.fr/voisinsdelemonde/lemme.jsp?toDo=chercherLemme&lemme=regarder`

En cherchant ce type de constructions avec **grew match**, on trouvera par exemple :



`https://universal.grew.fr/?custom=65f15552d2ea9`

Pour préparer la suite de travail

— proposez (en groupe) une solution pour avoir des patrons capables de capturer ce type de constructions plus complexes

— cherchez d'autres constructions qui demandent ainsi un *parcours* plus complexe de l'arbre syntaxique

Vous exposerez vos réflexions dans le journal, et une proposition d'implémentation dans **patterns.py**

Exercice 8 Question bonus 2 (améliorer l'existant)

Nous avons maintenant deux fichiers éxecutables comme des scripts shell : **main.py** et **analysers.py**.

Le rôle de **main.py** n'est donc plus très clair. Il faudrait mieux le dédier à la lecture du corpus d'origine et le renommer en **read_corpus.py**.

Une fois cette modification faite, il serait intéressant de pouvoir combiner les deux dans une *pipeline bash* pour combiner les deux tâches, qui ressemblerait à

```
python read_corpus.py <...> 2024 | python analysers.py <...>
```