ENSAI – Traitement Automatique du Langage

[TP1] Chaîne de traitement et représentation des mots

Le but de ce TP est d'une part de se familiariser avec l'environnement `spaCy`, suite logicielle en `python` pour le TAL, et, d'autre part, de mieux appréhender les notions fondamentales vues en cours, notamment les notions de mot, de lexeme, de *token*, et de vecteur de mot. Dans une deuxième partie, on abordera la notion de représentation vectorielle de documents.

De manière pratique, les séances de travaux pratiques se déroulent sous Google colab. Avant de démarrer le TP, regardez les instructions sur moodle pour charger les modèles et fichiers dont vous aurez besoin par la suite.

Fondements et pipelines `spaCy`

Pour l'ensemble des TP, nous nous appuierons sur le logiciel `spaCy`. Il est donc important de se familiariser avec son fonctionnement et sa logique. Il existe de nombreux autres *framework* compatibles avec `python`, *e.g.*, NLTK[1], gensim[2], AlenNLP[3] : cependant, `spaCy` présente l'avantage d'être orienté vers le développement d'applications plutôt que vers la recherche académique, ce qui facilite son utilisation pour le développement rapide d'applications comme nous le verrons. Son fonctionnement est assez similaire, bien que moins souple, aux autres *framework* standards.

La documentation complète de `spaCy` est consultable en ligne sur le site `https://v2.spacy.io`. On pourra aventageusement consulter le cours d'initiation sur `https://v2.spacy.io/usage/spacy-101`[4].

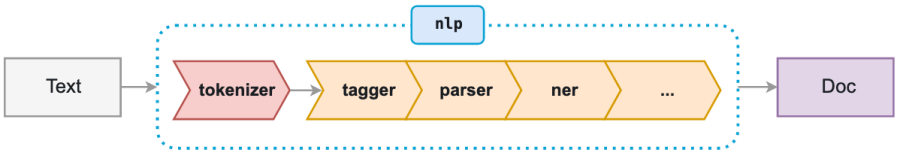

Figure 1: La notion de *pipeline* `spaCy` (source : `https://v2.spacy.io/usage/spacy-101`)

Nous reprenons ici les notions principales au cœur de `spaCy`, illustrées par la figure 1 :

- *Pipeline* ou chaîne de traitement : un *pipeline* permet d'enchainer un ensemble de traitements sur un texte, *e.g.*, tokenisation, lemmatisation, étiquettage en partie du discours (POS tagging), analyse en dépendance, détection d'entités, *etc.* Le logiciel fournit des chaînes de traitement prédéfinies pour plusieurs langues, dont l'anglais et le français. Chaque chaîne inclut les modèles nécessaires aux différentes étapes du traitement. Elle comporte notamment pour certaines d'entre elles une représentation vectorielle des lexemes du vocabulaire calculée à partir de GloVe.

  Le chargement d'une chaîne de traitement se fait *via* les instructions suivantes :

```
> import spacy
> process = spacy.load("en_core_web_md")   # load English pipeline en_core_web_md
> print(process.pipeline)                  # shows pipeline components (in order)
```

[1] `https://www.nltk.org`
[2] `https://radimrehurek.com/gensim`
[3] `https://allennlp.org/`
[4] Par défaut, Google colab importe la version 2 de `spaCy` alors que la documentation sur le site `https://spacy.io` concerne la version 3. La plupart des *features* de `spaCy` que nous allons utiliser sont maintenus dans la version 3.

- *Document* : l'application d'un *pipeline* sur un texte (chaîne de caractère pyton ou tableau de chaînes) produit un objet de type *Document*. Ce dernier regroupe l'ensemble des résultats de l'application de la chaîne de traitement sur le texte d'entrée. En particulier, un document contient la liste des *Tokens* résultant de la tokenisation du texte, chaque *token* étant accompagné d'informations tels que la forme graphique correspondante, le lemme, l'étiquette POS, l'éventuelle dépendance syntaxique, *etc.* Ces informations dépendent des éléments présents dans la chaîne de traitement. Le *document* contient aussi la liste des entités détectées si la chaîne de traitement inclue la détection des entités nommées.

  Les instructions suivantes permettent de visualiser une partie des informations contenues dans un *document* issu ici de l'analyse du texte contenu dans la variable `text` :

```
text = 'This is a $2 test sentence to test a U.K. spaCy pipeline"

doc = process(text)        # -> apply pipeline on text

for token in doc:          # -> iterate over tokens in doc
    print(token.text, token.lemma_, token.pos_, token.tag_, token.dep_)

for entity in doc.ents:    # -> iterate over the entities detected
    print(entity.text, entity.start_char, entity.end_char, entity.label_)

# visualisation of the dependency parse
spacy.displacy.render(doc, style="dep", jupyter=True)
```

- *Vocab* : le vocabulaire est une sorte de dictionnaire des lexèmes rencontrés, présents soit dans les modèles, soit dans les *tokens* du document. Les lexèmes du vocabulaire sont porteurs d'informations sur leur forme graphique et de nombreux indicateurs de leur formes de surface (`is_alpha`, `is_digit`, `is_alnum`, `is_stop`, *etc.*[5]). Les lexèmes du vocabulaire contiennent aussi leur représentation vectorielle (*word embedding*) lorsque celle-ci est présente dans le modèle.

  On accède aux principaux éléments des lexèmes du vocabulaire comme suit :

```
# iterate over the lexemes (too many to list)
for lexeme in process.vocab:
    print(lexeme.text, lexeme.norm_, lexeme.lower_, lexeme.has_vector)
    # if lexeme.has_vector:    # commented because too verbose
    #    print(lexeme.vector)

# direct access via indexing
lexeme = vocab['apple']
print(lexeme.has_vector, lexeme.is_stop, lexeme.is_alpha)
```

À faire :

1. En vous appuyant sur les instructions ci-dessus et sur les détails de la documentation `spaCy` mettre en place la chaîne de traitement `en_core_web_md`. Visualiser ses composantes ainsi que quelques lexemes du vocabulaire. Le modèle comporte-t-il des *word embeddings* ? Si oui, de quelle dimension ?

2. Appliquer la chaîne de traitement sur quelques phrases de votre choix, en visualisant pour chaque phrase les différents éléments du document résultant de l'analyse par `spaCy` de la phrase : *tokens*, lemmes, POS tags, arbre de dépendance, entités, *etc.* Comment sont traitées les ponctuations par la tokenisation ?

[5] *cf.* liste complète sur `https://v2.spacy.io/api/lexeme`

Représentations vectorielles de mots

Nous souhaitons maintenant mieux comprendre les vecteurs de mots, en commançant par ceux présents dans le modèle `en_core_web_md` que vous venez d'utiliser. En particulier, `spaCy` permet de calculer la similarité entre deux (vecteurs de) lexèmes de la manière suivante :

```
lexeme1 = process.vocab["apple"]     # get two lexemes from the vocabulary
lexeme2 = process.vocab["tree"]
d = lexeme1.similarity(lexeme2)  # get their similarity
```

À faire :

3. Choisisez quelques paires de lexèmes et regarder leur similarité. Vérifier que cette dernière correspond bien à la *cosine similarity* entre les deux vecteurs correspondants.

Pour bien appréhender les propriétés de ces vecteurs, nous allons choisir une centaine de mots sur lesquels nous pourrons faire les opérations suivantes :

- Visualiser en 2D les vecteurs des mots à l'aide de la méthode tSNE. On utilisera pour cela le module correspondant de `scikit-learn` pour obtenir une représentation `Y` 2D de chacun des vecteurs à partir de la matrice des distances `X` (matrice symétrique où X[i,j] est la distance entre le mot i et le mot j)[6] :

```
from sklearn.manifold import TSNE
Y = TSNE(n_components=2, metric="precomputed",
  random_state=0, method="exact").fit_transform(X)
```

- Mesurer la corrélation entre la proximité de deux vecteurs dans l'espace vectoriel et la proximité sémantique des mots correspondants telle que donnée dans le fichier `simlex-1.csv`[7]

À faire :

4. Chargez les données de similarité lexicale du fichier `simlex-1.csv` en ne gardant que les mots en minuscules (`islower()`) et qui ne sont pas des chiffres (`isdigit()`). On établira une liste de ces mots. Assurez-vous que tous les mots présents dans cette liste ont un vecteur dans le modèle `spaCy` chargé.

5. Visualisez les vecteurs pour ces mots en 2D à l'aide de tSNE. Identifiez des *clusters* lexicaux évidents et naturels.

6. Écrivez une fonction qui calcule la similarité entre vecteurs pour les paires de mots de `simlex-1.csv` et qui affiche la corrélation (Pearson) et la corrélation de rang (Spearman) avec les valeurs de référence issues des jugements humains. On pourra pour cela utiliser les fonctions du module `stats` de `scipy`.

Représentation des documents

On s'intéresse maintenant à la représentation vectorielle d'un document par la moyenne des vecteurs des mots du document. Pour étudier cette représentation, nous utiliserons les documents contenus dans le fichier `imdb-toy.json`[8]. Le fichier est un extrait de la base de données *The Internet Movie*

[6] On prendra comme distance 1 - cosine similarity.

[7] `https://people.irisa.fr/Guillaume.Gravier/teaching/ENSAI/data/simlex-1.csv` – *cf.* transparents du cours pour un exemple du contenu du fichier simlex issu de WordSim.

[8] `https://people.irisa.fr/Guillaume.Gravier/teaching/ENSAI/data/imdb-toy.json`

DataBase (IMDB) constituée de commentaires de films, en anglais, accompagnés de leur polarité positive ou négative. Cette base de données servira de support au TP suivant. Nous nous contentons ici d'un court extrait de 200 commentaires : les 100 premiers sont positifs, les 100 derniers négatifs. Pour chaque article, le premier champ indique la polarité du commentaire (`pos` ou `neg`) et le second champ contient le texte.

À faire :

11. Chargez les données et regardez quelques articles pour vous familiariser avec ces contenus.

12. En vous appuyant sur un modèle `spaCy`, écrire une fonction `avg_w2v` qui prend en entrée un article et qui retourne un *embedding* moyen pour le document en calculant la moyenne des *embeddings* des *lemmes* du document. On se limitera aux mots pleins (*e.g.*, adjectifs, noms et verbes ou juste adjectifs et verbes).

13. En vous aidant de la fonction `NearestNeighbor` vue précédemment, prendre quelques articles de la base de données et recherchez leur 5 plus proches voisins. Le résultat vous paraît-il satisfaisant ? Si on étend ce principe à l'ensemble des documents de la base pour faire une classification au sens des k plus proches voisins, quel taux de classification obtient-on ?

    On utilisera pour cela le module de recherche des plus proches voisins de `scikit-learn`. Par exemple, si la matrice X contient la liste des vecteurs de documents (une ligne = un échantillon du corpus), la recherche de plus proches voisins se fait par :

    ```
    from sklearn.neighbors import NearestNeighbors
    neighbors = NearestNeighbors(n_neighbors=5, p=2).fit(X)
    dist, idx = neighbors.kneighbors(vector)
    ```

    où `dist` contient les distances aux plus proches voisins de `vector` et `idx` la liste des indices correspondant dans `X`. On peut aussi calculer d'un coup l'ensemble des voisins de chaque document en appelant `dist, idx = neighbors.kneighbors(X)`, auquel cas `dist` et `idx` sont des matrices dont la permière dimension correspond à celle de X.