IDL 14 Word Embeddings (`plongements lexicaux`)

- aborder la notion d'embeddings
- entraîner des embeddings sur un corpus de discours en français
- interpréter ces embeddings pour analyser le discours

Conception : G. Lejeune et J-B. Tanguy

Les versions des librairies requises pour ce TP sont données dans le fichier `requirements.txt`

Exercice 0 : mise en place d'un environnement virtuel adapté

Dans le dossier du TP, exécutez :

```
python3 -m venv env_idl_14
source env-idl_14/bin/activate
pip install -r requirements.txt
```

De cette manière vous aurez toutes les librairies à jour dans la version attendue pour ce TP.

Exercice 1 : Constitution des sous-corpus

La plateforme du logiciel Hyperbase met à disposition de nombreux textes organisés en corpus et sous-corpus. Nous proposons de travailler sur les discours politiques français de la Vème république, organisés en deux sous-corpus : gauche et droite. Pour cela, vous devrez :

1. observer le fichier HYPERBASE_Droite_VS_Gauche.txt et les métadonnées qui s'y trouvent (comment distingue-t-on les discours de droite de ceux de gauche ?) ;
2. ouvrir un flux de lecture pour récupérer les données ;
3. ouvrir deux flux d'écriture, le premier rassemblant les discours de gauche, le second les discours de droite.

Ci-dessous, un exemple d'ouverture-fermeture d'un flux de lecture et d'un flux d'écriture.

```
import io

# 1. Flux de lecture file_path = './path/to/file.txt'
# io.open() :modes : 'r', 'w', 'a', 'x'
inFile = io.open(file_path, mode='r', encoding='utf-8')
lines = inFile.readlines() # lines : liste des lignes
inFile.close()

# 2. Flux d'ecriture
s = 'Ceci est un enonce.'
out_file_path ='./path/to/file.txt'
```

```
outFile = io.open(out_file_path, mode='w', encoding='utf-8')
outFile.write(s) # Ecriture de s dans le fichier
outFile.close()
```

Exercice 2 : Définition d'une classe `Embeddings`

Nous définissons une classe `Embeddings` (voir le fichier `embeddings.py`) rassemblant les méthodes et fonctions suivantes :

- `init` (constructeur)
- `learn` (apprentissage des `embeddings`), - `load` (chargement d'un modèle déjà appris),
- `most_similar` (retourne le vecteur le plus proche d'un mot donné en paramètre),
- `most_similar_analogy` (retourne le vecteur le plus proche d'une analogie donnée en paramètre),
- `similarity` (retourne la valeur de la similarité cosinus de deux vecteurs),
- `get_vocab` (retourne le vocabulaire du modèle),
- `get_vector` (retourne le vecteur d'un mot donné en paramètre), et
- `get_model` (retourne le modèle lui-même).

Observez la méthode `learn` et les fonctions qu'elle appelle.

- Quelle librairie est utilisée pour procéder à l'apprentissage des `embeddings` ?
- Quel modèle en particulier ?
- Quelles limitations identifiez-vous dans le processus général de construction de ces embeddings ? Proposez, sans les réaliser, deux améliorations possibles.

Exercice 3 : Apprentissage des Embeddings sur nos corpus

Notre objectif est de comparer les sous-corpus `Gauche` et `Droite` constitués en début de TD, en utilisant les `embeddings`. Le code ci-dessous montre comment réaliser l'apprentissage d'`embeddings` pour un fichier texte appelé `subCorpus_path`. Le modèle (qui contient l'ensemble des `embeddings`) sera sauvé à `model_path`. L'apprentissage dure une dizaine de secondes.

```
subCorpus_path = './path/to/subCorpus.txt'
model_path = './path/to/model.W2Vmodel'
embeddings = Embeddings(subCorpus_path, model_path)
embeddings.learn() #l'apprentissage est réalisé ici
```

Une fois les `embeddings` appris, pour toute utilisation du modèle, il suffira de charger les `embeddings` (cela évite de les ré-apprendre à chaque fois) :

```
subCorpus_path = './path/to/subCorpus.txt'
model_path = './path/to/model.W2Vmodel'
embeddings = Embeddings(subCorpus_path, model_path)
embeddings.load()
```

Dans un fichier `ex3_train_embeddings.py`, réalisez l'apprentissage des `embeddings` pour les deux sous-corpus `Gauche` et `Droite`. Vous devrez disposer de deux modèles, qu'on pourra appeler : `GAUCHE.W2Vmodel` et `DROITE.W2Vmodel`, stockés dans un répertoire `model`

Après chaque apprentissage, affichez un message à l'écran du type : `Model ./models/DROITE.W2Vmodel saved.`

Dans un fichier `ex3_test_embeddings.py`, chargez les modèles dans un dictionnaire ayant pour clés "DROITE" ou "GAUCHE" et pour valeurs les modèles correspondants, et affichez, pour chacun des modèles :

- la taille du modèle en nombre de mots (longueur du vocabulaire)
- le vecteur du mot "patrie".

Les deux vecteurs pour le mot patrie sont-ils égaux ? Pourquoi ?

Quelle est la taille des vecteurs que vous observez ? Il s'agit de la taille par défaut. Où celle-ci peut-elle être ajustée à votre avis ?

Vous utiliserez pour cela les fonctions déjà définies dans la classe `Embeddings`.

Exercice 4 : Utilisation des Embeddings

Dans un fichier `ex4_use_embeddings.py`, chargez comme précédemment vos deux modèles.

Étude des vecteurs proches et éloignés

La fonction ci-dessous vous permet d'obtenir la similarité cosinus de deux vecteurs `vec1` et `vec2` :

```
import numpy as np

def cosine_similarity(vec1, vec2):
    dot = np.dot(vec1, vec2) # produit scalaire
    norm1 = np.linalg.norm(vec1) # norme du vec1
    norm2 = np.linalg.norm(vec2) # norme du vec2
    cos = dot / (norm1 * norm2)
    return cos # type : float
```

La fonction `get_vocab` de la classe `Embeddings` vous permet de récupérer le vocabulaire du modèle (l'ensemble des mots pour lesquels un vecteur a été appris). Pour tous les mots appartenant aux deux vocabulaires (celui du sous-corpus `GAUCHE` et celui du sous-corpus `DROITE`), calculez la similarité cosinus des deux vecteurs (le vecteur de `mot` appris sur le sous-corpus `GAUCHE` et le vecteur de `mot` appris sur le sous-corpus `DROITE`).

Le code permettant d'afficher les 30 mots ayant les similarités les plus proches est fourni. Commentez le (dans le code) et rajoutez des messages pour produire un affichage lisible du type :

```
*** Vecteurs proches : `sera, avons, tout, temps, me, nos, etc.
*** Vecteurs éloignés : Nous, encore, nous, leurs, du, etc.`
```

```
NB_TO_SHOW = 30
emb1, emb2 = embeddings['DROITE'],
embeddings['GAUCHE'] vocab1, vocab2 = emb1.get_vocab(),
emb2.get_vocab()
sims = {}
for word in vocab1:
    if word in vocab2:
        sims[word] = cosine_similarity(emb1.get_vector(word),
emb2.get_vector(word))

# Tri des dictionnaires : ordres croissant et décroissant
sims_up = k: v for k, v in sorted(sims.items(), key=lambda
    item: item[1], reverse=True)
sims_down = k: v for k, v in sorted(sims.items(), key=lambda item:item[1])
i = 0 while i < NB_TO_SHOW:
    w = list(sims_up.keys())[i]
    i += 1

i = 0 while i < NB_TO_SHOW:
    w = list(sims_down.keys())[i]
    i += 1
```

Que pouvez-vous interpréter à propos des mots `Droite` et `Gauche` ? Des mots "vides" (`stop-words`) ?

2. Observation de vecteurs similaires

Rassemblez dans une liste un petit ensemble de mots (par exemple : gauche, droite, homme, action...). Pour chacun de ces mots, affichez les 10 mots les plus proches (en terme de similarité cosinus) en utilisant la fonction `most_similar` de la classe `Embeddings`.

Utilisez le code suivant et des boucles adaptées pour produire un affichage du type :

`gauche (DROITE) droite extrême Blum barrage défaite bouche Jaurès coalition mort idéologie`

```
# embeddings : un objet de la classe Embeddings
# word : un mot qui vous intéresse
# topN : un entier correspondant au nombre de mots similaires à afficher
sims = ' '.join([w for w,s in embeddings.most_similar(word, topN)])
print('{}\t({})\t{ }'.format(word, corpus, sims))
```

Les rapprochements de mots affichés vous semblent-ils pertinents ? Y a-t-il des différences dans l'interprétation des mots similaires, selon qu'on observe le sous-corpus `Gauche` ou le sous-corpus `Droite` ?

3. Analogies

Les `embeddings` étant des vecteurs, il est possible de rapprocher non plus seulement deux vecteurs par la similarité cosinus mais bien un calcul sur les vecteurs. Par exemple : quel est le mot le plus proche de : Roi -

Homme + Femme ? Entendons, quel vecteur est le plus proche du vecteur résultat des opérations d'addition et de soustraction entre les mots Roi, Homme et Femme. Un résultat satisfaisant pourrait être Reine.

Voici un exemple d'utilisation de la fonction `most_similar_analogy` de la classe `Embeddings` :

```
# roi - homme + femme
# + roi - homme + femme
# + roi + femme - homme
pos, neg = ['roi', 'femme'], ['homme'] most_sim =
embeddings.most_similar_analogy(pos, neg, topn=5)
for w_s in most_sim:
    word, sim = w_s
    print('{}\t{}'.format(word, sim))
```

Testez quelques analogies sur les deux sous-corpus. Pour commencez :

- président - homme + femme
- candidat - homme + femme
- député - candidat + candidats

Qu'observez-vous ?

La méthode `tsne_plot` (voir ici) permet de représenter, par analyse en composantes principales (ACP en français, PCA en anglais), des `embeddings` sur un graphique. Elle prend en entrée `plot_title`, une chaîne de caractère qui constituera le titre du graphique et `model` un modèle Word2Vec.

Regardez le code `ex5_plot.ipynb`.

On y relance un apprentissage d'`embeddings` selon une méthode plus restrictive (méthode `learn_restrictive` de la classe `Embeddings`) en terme de fréquence minimale d'apparition d'un mot, sans quoi trop de vecteurs apparaîtront sur le graphique, lequel sera illisible. Testez plusieurs valeurs de `min_freq` (paramètre de la fonction `learn_restrictive`) pour trouver un bon compromis entre un nombre trop faible de vecteurs représentés et un nombre trop important rendant le graphique illisible. Vous pouvez utiliser la liste de `stop-words` fournie par la librairie `nltk` pour ne garder que les mots "pleins" -- si vous le jugez pertinent -- en modifiant par exemple le début de la méthode `tsne_plot`.

- Deux apprentissages d'`embeddings` sur un même corpus entraînent-ils des vecteurs strictement équivalents ? Pourquoi ?
- Comment cela s'exprime à travers les graphiques générés par la méthode `tsne_plot` ?