Année universitaire	**2023-2024**		
Département	**Informatique**	**Année**	**M2 IA & BioInfo**
Matière	**Techniques d'Apprentissage Automatique**		
Intitulé TD/TP :	**Atelier 2 : Traitement de données Textuelles**		
Contenu	• **Préparation de données textuelles** • **Vectorisation de données textuelles** • **TFIDF, LSA** • **Word2vec** • **Apprentissage multi-label sur des données textuelles**		

Dans cet atelier pratique, l'objectif dans cette partie est de faire une étude comparative entre plusieurs algorithmes d'apprentissage supervisé multi-label sur un jeu de données textuelles avec le langage **Python**.

Pour lancer le notebook Python, il faut taper la commande **jupyter notebook** dans votre dossier de travail. Une fenêtre va se lancer dans votre navigateur pour ouvrir l'application Jupyter. Créer un nouveau notebook Python et taper le code suivant dans une nouvelle cellule :

```
import numpy as np
np.set_printoptions(threshold=10000, suppress = True)
import pandas as pd
import warnings
import matplotlib.pyplot as plt
warnings.filterwarnings('ignore')
```

L'objectif dans cette partie est d'apprendre sur un jeu de données de *commentaires* du fichier **"PubMed-multi-label-dataset.csv"** en suivant les étapes suivantes. *Votre code doit bien être factoriser en proposant plusieurs mini-fonctions pour chaque traitement proposé.*

- **Importer** ce jeu de données avec la librairie pandas (**c.f. read_csv**)
- **Analyser** votre jeu de données, essentiellement la target. Chaque texte est labélisée par un ensemble de labels parmi les 14 labels suivants : Anatomy [**A**], Organisms [**B**], Diseases [**C**], Chemicals and Drugs [**D**], Analytical, Diagnostic and Therapeutic Techniques, and Equipment [**E**], Psychiatry and Psychology [**F**],Phenomena and Processes [**G**], Disciplines and Occupations [**H**], Anthropology, Education, Sociology, and Social Phenomena [**I**], Technology, Industry, and Agriculture [**J**], Information Science [**L**], Named Groups [**M**], Health Care [**N**], Geographicals [**Z**]
- **Modéliser** le problème d'apprentissage supervisé sur ces données.
- **Traiter** vos données textuelles en supprimant les bruits dans les textes et en les normalisant. Vous pouvez vous inspirer par exemple du code par ici[1] (en l'améliorant s'il le faut) si vous utiliserez la librairie **NLTK**. Vous pouvez aussi utiliser d'autres librairies (A vos recherches). N'oubliez pas que cette **étape de pré-traitement (preprocessing)** dépend de vos données et du problème traité. Quelques traitements à faire sont :
  - Suppression des ponctuations comme **. , ! $( ) * % @**
  - Suppression des URLs
  - Suppression des Stop words
  - Transformation de tout le texte en minuscule.
  - Tokenisation de vos textes
  - Racinisation (Stemming)
  - Lemmatisation (lemmatization)
  - Etc.

[1] https://www.kdnuggets.com/2018/03/text-data-preprocessing-walkthrough-python.html

- **Séparer** les données en jeu de données d'**apprentissage** et jeu de données de **test (50-50)**.
- **Proposer** une fonction **run_models** permettant de comparer plusieurs modèles d'apprentissage (en fonction de votre modélisation) sur ces données. Pour les approches multi-label, vous utiliserez l'approche **EnsembleClassifierChain** et **MultiOutputCLassifier** du package **sklearn.multioutput**. Ces classifieurs nécessite un classifieur de base (**base_estimator**) dont vous avez le libre choix d'utilisation. Votre évaluation se basera sur les deux mesures **hamming_loss** et **zero_one_loss**.
- **Proposer** une première **vectorisation** de vos données textuelles par une représentation **TF-IDF**.
- **Exécuter** ensuite votre fonction **run_models** sur vos données et interpréter les résultats obtenus. Vous pouvez toujours appliquer votre fonction sur les **données pré-traitées** et les **données non pré-traitées** afin d'analyser l'apport de la partie pré-traitement de données.
- **Appliquer** la méthode **SVD** de **réduction de dimensions** (**TruncatedSVD**) afin de construire des "**concepts**" liés aux documents et aux termes. Elle permettra entre autres de résoudre les problèmes de **synonymie** (plusieurs mots avec un seul sens) et de **polysémie** (un seul mot avec plusieurs sens). La fonction suivante vous aidera à comprendre les concepts en affichant leurs mots les plus pertinents.

```
def print_top_words(model, feature_names, n_top_words):
    for topic_idx, topic in enumerate(model.components_):
        message = "Concept #%d: " % topic_idx
        message += " ".join([feature_names[i] for i in topic.argsort()[:-n_top_words - 1:-1]])
        print(message)
    print()

print_top_words(SVD, vocabulaire(), 10) # pour afficher les 10 mots les plus pertinents par concept
```

- **Exécuter** ensuite votre fonction **run_models** sur vos données et interpréter les résultats obtenus. Vous pouvez toujours appliquer votre fonction sur les **données pré-traitées** et les **données non pré-traitées** afin d'analyser l'apport de la partie pré-traitement de données.
- **Dans un nouveau notebook (inspiré de celui sur moodle Word2Vec_creation.ipynb)**, **proposer** un code qui permettra d'apprendre votre propre modèle de plongement lexical **Word2Vec** sur vos données textuelles. **Évaluer visuellement et numériquement** sur quelques mots clés votre nouveau modèle de vectorisation (*Embedding*).
- Exploiter votre modèle **Word2Vec** pour la vectorisation de vos textes (avec deux méthodes utilisant ou non le TF-IDF des mots). Exécuter ensuite à nouveau votre fonction **run_models** sur vos données vectorisées par **Word2Vec** et interpréter les résultats obtenus en les comparant à ceux obtenus aux étapes précédentes. Vous pouvez vous inspirer de la fonction suivante :

```
## get word2vec for each sentence by using average word embeddings
def word2vec_generator(texts,model,vector_size):
    dict_word2vec = {}
    for index, word_list in enumerate(texts):
        arr = np.array([0.0 for i in range(0, vector_size)])
        nb_word=0
        for word in word_list:
            try:
                arr += model[word]
                nb_word=nb_word+1
            except KeyError:
                continue
        if(len(word_list) == 0):
            dict_word2vec[index] = arr
        else:
            dict_word2vec[index] = arr / nb_word
    df_word2vec = pd.DataFrame(dict_word2vec).T
    return df_word2vec
```

- Idem en utilisant le **modèle Word2Vec pré-entrainé de Google**[2] **ou un autre**.
- Exploiter votre modèle **Word2Vec** en entrée de la couche d'embedding d'un modèle à base de réseaux de neurones récurrents de type LSTM permettant :
    - Dans un premier temps d'optimiser la ***0/1 loss***.
    - Dans un deuxième temps d'optimiser la ***hamming loss***.

Pour le réseau LSTM, vous pouvez vous inspirer du code suivant :

```
from keras.preprocessing.sequence import pad_sequences
def to_sequence(index, text):
  indexes = [index[word] for word in text if word in index]
  return indexes

word2idx = {word: idx for idx, word in enumerate(model_wv_google.index_to_key)}
X_train_sequences = [to_sequence(word2idx, x) for x in train_tokenized]
X_test_sequences = [to_sequence(word2idx, x) for x in test_tokenized]

MAX_SEQ_LENGHT=50 #A définir
N_FEATURES = len(model_wv_google.index_to_key)
X_train_sequences = pad_sequences(X_train_sequences, maxlen=MAX_SEQ_LENGHT, value=N_FEATURES)
X_test_sequences = pad_sequences(X_test_sequences, maxlen=MAX_SEQ_LENGHT, value=N_FEATURES)

EMBEDDINGS_LEN = model_wv_google.vector_size
embeddings_index = np.zeros((len(model_wv_google.index_to_key)+1, EMBEDDINGS_LEN))

for word, idx in word2idx.items():
  try:
    embedding = model_wv_google[word]
    embeddings_index[idx] = embedding
  except:
    pass

from keras.models import Sequential
from keras.layers import Embedding, Dense,LSTM

model_lstm = Sequential()
model_lstm.add(Embedding(len(model_wv_google.key_to_index)+1,
            EMBEDDINGS_LEN,  # Embedding size
            weights=[embeddings_index],
            trainable=False))

model_lstm.add(LSTM(300, dropout=0.2))
model_lstm.add(Dense(1))
model_lstm.compile(loss='A définir', optimizer='adam', metrics=['A définir'])
model_lstm.summary()
```

Notez qu'il est possible également d'exploiter la sortie de la couche LSTM de votre modèle en entrée de votre fonction run_models.

- ***Pipeline*** **:** Automatiser l'enchainement de votre meilleur traitement dans une fonction ou un pipeline.

[2] https://drive.google.com/file/d/0B7XkCwpI5KDYNlNUTTlSS21pQmM/edit?usp=sharing