TP3 : IDL_06 Language identification

Prérequis :

- gestion des chaînes de caractères
- listes et dictionnaires
- ouverture et fermeture de fichiers.

Objectifs :

- traitement de gros corpus
- factorisation des traitements
- évaluation des modèles

Ce TP s'attaque à une tâche de classification "en langue" de documents (textes).
L'objectif est d'identifer correctement la langue d'un texte.

Nous utiliserons trois méthodes ici :

1. modèle de langue simple granularité mot
2. modèle de langue simple granularité n-gram de caractères
3. entraînement d'un classifieur supervisé

Vous disposez d'un corpus parallèle multilingue de 22 langues. Pour chaque langue le corpus a été séparé en un jeu d'entraînement (train) et de test (test).

Le code suivant vous permet d'afficher chaque fichier et ses caractéristiques :

```
import glob

liste_fichiers = glob.glob("corpus_multi/*/*/*")

for chemin in liste_fichiers :
        print(chemin)
        print(chemin.split("/"))
```

par exemple, pour un des fichiers de test du corpus de l'anglais :

`['corpus_multi', 'en', 'test', '2009-10-23_celexIP-09-1574.en.html ']`

Notez que la manière dont ont été structurés les fichiers présente ici un intérêt : on exploite directement la structure hiérarchique pour extraire pour chaque document des caractéristiques propres (langue et type).

Exercice 1 : créer un modèle de langue

1. En prenant exemple sur `files_list`, créez deux variables `train_files_list` et `test_files_list` contenant respectivement les fichiers de train et de test de l'ensemble du corpus. Combien y a-t-il de fichiers de chaque type ?

On utilise dans cette section uniquement les fichiers du jeu de données appr pour chaque langue.

2. Implémentez la fonction read_file qui prend en argument un chemin et qui renvoie la chaîne de caractères correspondant au contenu du fichier pointé par le chemin.
3. Implémentez la fonction `language_wc` (`wc` pour *word count*) qui prend pour argument une liste de fichiers, et qui renvoie un dictionnaire donc les clés sont les langues `l` et dont les valeurs sont des dictionnaires associant à chaque mot rencontré dans le corpus d'apprentissage de la langue `l` son effectif.
   En guise d'approximation, on utilisera ici la fonction `split()` pour tokéniser tous les langages.

Résultat attendu : (lt = lithuanien)

```
{'lt': {
        'IP/09/': 1,
        '1197': 1,
        'Briuselis,': 1,
        '2009': 3,
        'm.': 6,
        'liepos': 1,
        '28': 3,
        'd.': 3,
        'Antimonopolinė': 1,
        'politika.': 1,
        'Komisija': 7,
        'pradeda': 1,
        'viešąsias': 1,
        'konsultacijas': 1,
        'dėl': 8
        ...
```

4. Implémentez la fonction `create_lmodels_wc` qui prend en paramètre le dictionnaire d'effectifs par langue créé précédemment, et qui renvoie un dictionnaire qui à chaque langue `l` associe la liste des 10 mots les plus fréquents dans cette langue. Stockez le dictionnaire renvoyé dans une variable `lmodels_wc`.

Résultat attendu :

```
{'lt': [
        'y.',
        'yra',
        'platinimo',
        'm.',
        'taisyklių',
        'Komisija',
        'į',
        'dėl',
        'rinkos',
        'ir'
        ], ...
```

Sauvegardez ces modèles dans un fichier `models.json` à l'aide de la fonction `dic_to_json(dic, json_file)`.

Exercice 2 : utiliser le modèle de langue

Implémentez-donc la fonction lpredict qui prend en argument un fichier de test et qui prédit sa langue :

1. Commencez par calculer les 10 mots les plus fréquents du texte et stockez-les dans une variable `most_frequent_test`.
2. Calculez l'intersection entre cette liste de mots et les 10 mots les plus fréquents associés à chacun des modèles de langue existant (`lg`). Vous pourrez stocker la taille de cette intersection dans une liste lprediction en vous inspirant par exemple du code :

```
lprediction = []
for lg, model in lmodels_wc.items():
            common_words = set(model).intersection(most_frequent_test)
            common_wc = len(common_words)
            lprediction.append([common_wc, lg])
            lprediction = sorted(lprediction, reverse=True)[0][1]
```

À ce stade, la prédiction réalisée correspond au premier élément de la liste `lprediction`.

Exercice 3 : évaluez la méthode

Une fois la prédiction établie pour chaque fichier, vous pouvez calculer la performance de votre méthode **par langue** et sur l'ensemble du corpus.

Stockez dans une variable `pred_results` (pour prediction results) la liste des couples `[language, predicted_language]` pour chacun des fichiers de test.

Créez ensuite une fonction `evaluate_wc_model` qui prend en paramètre une liste de liste de résultats `pred_results` et qui renvoie un dictionnaire `results_dic` qui associe à chaque langue le nombre d'identification correctes et incorrectes réalisées par le modèle. Vous pourrez vous inspirer du code suivant :

```
def evaluate(pred_results):
    correct = 0
    for language, predicted_language in pred_results:
        results_dic.setdefault(langue, {"VP": int(), "FP": int(), "FN": int()})
        results_dic.setdefault(langue_pred, {"VP": int(), "FP": int(), "FN": int()})
        if langue == langue_pred:
            results_dic[language]["VP"] += 1  # langue bien détectée
            correct += 1
        # TO FILL

    return dic_resultats, NB_bonnes_reponses
```

Parmi les métriques vues en cours, utilisez celles qui vous paraissent les plus adaptées pour procéder à l'évaluation. Vous pourrez créer une nouvelle liste qui pour chaque langue renvoie la valeur obtenue pour chacune des métriques.

Exercice 4 : et en caractères ?

1. Reprenez le code précédent en utilisant non plus les mots mais les n-grams de caractères : testez avec n allant de 1 à 4.
2. Proposez une comparaison entre les différentes méthodes testées. Vos observations

feront l'objet d'un commentaire dans le README associé à votre rendu.

Exercice 5 : apprentissage

Il s'agit ici de vectoriser les exemples et d'entraîner un classifieur à trouver la relation entre le contenu des exemples (souvent regroupés dans une matrice nommée X) et les classes (une liste nommée y). On stocke séparément ces élements pour l'apprentissage (ou train)
et le test.

L'exemple donné ci-dessous exploite la bibliothèque sklearn (à éventuellement installer via la commande `pip install sklearn`). Comme expliqué en cours, nous exploitons le vectoriseur `CountVectorizer`, les classifieur bayésiens naïfs, ainsi que les rapports de classification.

Pour les sous-sections 1, 2 et 3, vous n'avez rien à faire à part comprendre et reproduire le code proposé.
Les questions sont dans la section 4.

1. Préparation des données

Le code ci-dessous stocke les textes (en entrée des classifieurs) et les langues (classe à prédire) dans des structures de données adaptées :

```
from sklearn.feature_extraction.text import CountVectorizer
import glob

import re

texts = {"appr":[], "test":[]}
classes= {"appr":[], "test":[]}

for path in glob.glob("corpus_multi/*/*/*")[:1500]:
    _, lg, corpus, filename = re.split("/", path)
    classes[corpus].append(lg)
    with open(path, encoding="utf-8") as f:
        fstring = f.read()
        texts[corpus].append(fstring)
```

2. Vectorisation données

Pour simplifier ici, on utilise les 1 000 caractéristiques (*features*) les plus fréquentes sur tout le corpus (et donc pas les plus fréquentes pour chaque langue).

```
vectorizer = CountVectorizer(max_features=1000)
# Pour travailler avec des caractères : analyzer="char"
# spécifier la taille des n-grammes : ngram_range=(min,max))
X_train = vectorizer.fit_transform(texts["appr"]).toarray()
X_test = vectorizer.transform(texts["test"]).toarray()
y_train = classes["appr"]
y_test = classes["test"]
```

X_train correspond à la représentation matricielle des données d'entraînement. Elle est associée à y_train qui correspond à l'étiquetage *de référence* pour ces données là.

3. Classification et évaluation

La classification se fait ici "simplement" en utilisant la fonction `fit` qui crée le modèle à partir des données d'entraînement puis la fonction `predict` appliquée au corpus de test. L'application du modèle renvoie sa prédiction, la classe la plus probable, sous forme d'un vecteur `y_pred`.

```
from sklearn.naive_bayes import GaussianNB

gnb = GaussianNB()

# Calcul de la prédiction :
y_pred = gnb.fit(X_train, y_train).predict(X_test)

# comparaison prédiction et attendu
NB_textes = X_test.shape[0]
NB_erreurs = (y_test != y_pred).sum()
print("Erreurs d'e«tiquetage sur %d textes : %d" % (NB_textes, NB_erreurs))
```

4. Faire jouer les paramètres

Votre travail consiste à faire varier les paramètres de `CountVectorizer` :

- Tester différentes valeurs pour le nombre de caractéristiques : `max_features`
- Tester en vectorisant avec des n-grammes de caractères : `analyzer="char"`
- Faire varier les valeurs min et max de N : `ngram range=(min,max)`
- Quelle est la configuration la plus efficace ?
- A votre avis pourquoi ?