TAL: Reconnaissance de langues

version
2023

Reconnaissance de Langues

Crédits : Gaël Lejeune, Carlos Gonzales, Andrea Briglia, Antoine Lebrun, Sorbonne

Objectifs

- Traiter de gros corpus
- Factoriser des traitements
- Calculer et évaluer des modèles

Nous savons par la loi de Zipf que l'effectif des mots d'une langue suivait certaines lois de distribution. Nous pouvons exploiter ces propriétés pour calculer des "modèles de langue". Ces modèles de langue permettent d'identifier la langue d'un texte inconnu.

Nous allons utiliser le **corpus parallèle** multilingue *corpus_multi.zip* Décompressez cette archive dans un dossier dédié. Il y a un sous-dossier pour chaque langue (22 langues en tout) et pour chaque langue un corpus d'apprentissage ("appr") et un corpus de test ("test"). Dans un nouveau script Python (notebook ou autre), testez le code suivant (en vous plaçant au bon endroit):

```
import glob
liste_fichiers = glob.glob("corpus_multi/*/*/*")
print("Nombre de fichiers : %i"%len(liste_fichiers)
for chemin in liste_fichiers:
  print(chemin)
  print(chemin.split("\\\\"))#Sur Linux/Mac: chemin.split("/")
  break#stoppe le code (provisoire)
```

Observez le résultat du second print qui doit ressembler à :

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

Ceci nous permettra d'exploiter la structure des dossiers pour faire des traitements efficaces.

Exercice 1 : Créer le modèle de langue (jeu de données appr)

Nous allons traiter simplement le jeu de données d'apprentissage en changeant le démarrage de la boucle for :

```
import glob
dic_langues = {}#pour recevoir les modèles de langue
liste_fichiers_appr = glob.glob("corpus_multi/*/appr/*")
print("Nombre de fichiers : %i"%len(liste_fichiers_appr)
for chemin in liste_fichiers_appr:
  print(chemin)#le commenter apres cette etape
  dossiers = chemin.split("\\\\")#Sur Linux/Mac: chemin.split("/")
```

```
  langue = dossiers[1]
  ## on calcule les mots les plus frequents du texte:
  if langue not in dic_langues:
  ##on crée un sous-dictionnaire pour une nouvelle langue
    dic_langues[langue] = {}
  #on réutilise notre outillage de découpage en mots :
  chaine = lire_fichier(chemin)
  mots = chaine.split()
  # et on stocke les effectifs :
  for m in mots:
    if m not in dic_langues[langue]:
      dic_langues[langue][m] = 1
    else:
      dic_langues[langue][m] += 1
  print(dic_langues)#Ligne à supprimer par la suite sinon affichgae tro
  1/0#On pourrait aussi mettre un break pour sortir de la boucle
```

Pour chaque langue du corpus, utilisez l'ensemble des textes figurant dans le dossier " appr " pour calculer un modèle de langue. Ce modèle de langue sera constitué des n mots les plus fréquents identifiés dans ce sous-corpus.

Pour l'étape suivante, on commente les deux print et la condition d'arrêt (1/0). Après la boucle on va pouvoir calculer les modèles de langue de la façon suivante :

```
dic_modeles = {}
for langue, dic_effectifs in dic_langues.items():
  paires=[[effectif, mot] for mot, effectif in  dic_effectifs.items()]
  liste_tri = sorted(paires)[-10:]#les 10 mots fréquents
  dic_modeles[langue] = [mot for effectif, mot in liste_tri]
print(dic_modeles)#à retirer dès que posible
```

Enlèvez le dernier print puis sauvegardez tous ces modèles dans un unique fichier au format Json :

```
import json
with open("models.json", "w", encoding="utf-8") as w:#ici on change dem
  w.write(json.dumps(dic_modeles, indent = 2))
```

Consultez ce fichier `models.json` pour comprendre le modèle.

Exercice 2 : Appliquer les modèles de langue

Nous allons maintenant regarder si ces modèles fonctionnent bien. Pour chaque langue nous allons nous intéresser aux fichiers contenus dans le dossier "test". Nous allons comparer les 10 mots les plus fréquents de chacun de ces fichiers avec nos modèles de langue. Nous allons ainsi pouvoir établir un diagnostic de langue pour chaque texte.

Il nous faut donc obtenir pour chaque texte du dossier "test" la même **représentation** afin de pouvoir le comparer aux modèles que nous avons calculé.

Dans une nouvelle cellule, on va tout d'abord charger les modèles :

```
with open("models.json", "r", encoding="utf-8") as f:
  dic_modeles = json.load(f)
```

Puis nous allons parcourir les fichiers de test, les ouvrir et compter l'effectif des mots de ce texte :

```
liste_fichiers_test = glob.glob("corpus_multi/*/test/*")
print("Nombre de fichiers : %i"%len(liste_fichiers_test)
for chemin in liste_fichiers_test:
  dossiers = chemin.split("\\\\")#Sur Linux/Mac: chemin.split("/")
  langue = dossiers[1]
  chaine = lire_fichier(chemin)
  mots = chaine.split()
  dic_freq_texte = {}
  for m in mots:
    if m not in dic_freq_texte:
      dic_freq_texte[m] = 1
    else:
      dic_freq_texte[m] += 1
  print(dic_freq_texte)
  # dic_freq_texte contient les effectifs des mots du texte
  1/0
```

Maintenant on enlève la condition d'arrêt et on calcule les dix mots les plus fréquents du texte (attention, il y a de l'indentation, cela signifie que l'on doit toujours être dans la boucle créée ci-dessus):

```
  paires=[[effectif, mot] for mot, effectif in  dic_freq_texte.items()]
  liste_tri = sorted(paires)[-10:]#les 10 mots fréquents
  plus_frequents = set([mot for effectif, mot in liste_tri])
  print(plus_frequents)
  1/0
```

A nouveau, on enlève la condition d'arrêt, le print et on passe à la suite. Ici on va calculer l'**intersection** (les mots en commun) entre les mots fréquents du texte et ceux de chaque modèle de langue (comme précédemment, l'indentation est présente on est toujours dans la boucle):

```
  print("Document en %s"%langue)
  for langue_ref, model in dic_models.items():
    mots_communs = set(model).intersection(plus_frequents)
    print("%i mots en commun avec le modèle (%s):"
%(len(mots_communs), langue_ref))
    print(mots_communs)
  1/0
```

Maintenant on va chercher automatiquement l'intersection la plus grande et vérifier si ça correspond à notre intuition (toujours dans la boucle):

```
  liste_predictions = []
  print("Document en %s"%langue)
  for langue_ref, model in dic_models.items():
    mots_communs = set(model).intersection(plus_frequents)
    NB_mots_communs = len(mots_communs)
    liste_predictions.append([NB_mots_communs, langue_ref])
    print(sorted(liste_predictions))
```

A vous maintenant de faire la suite : extraire la prédiction majoritaire puis passer à la section suivante : l'évaluation.

Exercice 3 : Evaluer le diagnostic

Nous devons évaluer l'efficacité de notre système : pour chaque langue d'une part et pour l'ensemble des langues d'autre part. Nous allons considérer que notre but est de trouver la bonne classe (i.e. la bonne langue) pour chaque document.

Dans un premier temps nous utiliserons l'**exactitude** c'est à dire la proportion de bonnes réponses. Il suffira pour chaque texte du jeu de `test` de comparer le diagnostic de langue donné par votre programme et de regarder s'il est conforme à la vérité. Voici les étapes :

- On crée une variable qui va stocker le nombre de réponses (c'est donc un entier qui vaudra zéro au démarrage), appelons la `NB_bonnes_reponses`
- Pour chaque texte du jeu de `test`, on prédit la langue du texte
- On compare la langue prédite avec la langue réelle :
  - Si c'est pareil, alors `NB_bonnes_reponses` est **incrémenté** de 1
  - Sinon il ne se passe rien
- À la fin on calcule la proportion de bonnes réponses : `NB_bonnes_reponses`/ Le nombre de documents du jeu de test.

Dans un deuxième temps, nous utiliserons le Rappel qui permet de vérifier que l'on a bien classé tous les documents d'une même langue ainsi que la Précision qui évalue combien de documents sont classés dans la bonne langue. Il s'agit donc de calculer le nombre de Vrais Positifs (VP), Faux Positifs (FP) et Faux Négatifs (FN).

Pour chaque document traité à l'exercice précédent nous allons comptabiliser nos VP, nos FP et nos FN. Nous pourrons ainsi pour chaque langue calculer :

- le rappel : VP/(VP+FN)
- la précision : VP/(VP+FP)
- la F_1-mesure (moyenne harmonique du rappel et de la précision) : (2*Rappel*Précision) / (Précision+Rappel)

Ensuite, vous comparerez les résultats selon le nombre n de mots fréquents pris en considération et vous identifierez les erreurs les plus fréquentes: couples de langues qui posent problème et documents pour lesquelles la prédiction en plus d'être fausse est étrange (normalement vous avez un document en bulgare pour lequel la prédiction est "anglais", il faudra aller voir pourquoi).

Vous verrez que pour se faciliter la tâche, on aura intérêt à en faire des fonctions :

- Une qui prend en entrée un texte et retourne la prédiction
- Une qui prend en entrée une liste de prédictions de langues et la liste des langues réelles et qui les compare pour calculer exactitude, rappel, précision et F-mesure

Déposez avant le 22/10 minuit votre code qui réalise cette étape là.

Exercice 4 : Et en caractères?

- Reprenez les 3 exercices précédents en utilisant non plus des mots mais des n-grammes de caractères.
- Testez pour des valeurs de n de 1 à 4 en créant une fonction qui prend en paramètre le nombre de caractères (le N)
- Regardez si cela fonctionne mieux ou moins bien que la méthode en mots

L'évaluation de cette partie du travail sera réalisée en cours le 28/10 (présence obligatoire).

Vous rédigerez 2 pages décrivant vos résultats à l'issue de la séance.

Exercice 5 : Exercice bonus : avec du *Machine Learning*

En apprentissage automatique on cherche le bon algorithme qui permet d'associer des exemples (ici des textes) à des classes ou étiquettes (ici des langues). Il s'agit 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 va stocker 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`). Nous exploitons le vectoriseur CountVectorizer [1], les classifieur bayésiens naïfs [2] et les rapports de classification [3].

I: Préparation des données

On va stocker les textes et les langues dans des structures de données adaptées :

```
from sklearn.feature_extraction.text import CountVectorizer
import glob
import re
textes = {"appr":[], "test":[]}
classes= {"appr":[], "test":[]}

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

  textes[corpus].append(chaine)
```

A noter que l'on ne prend que les 1500 premiers exemples seulement pour faire fonctionner le programme sur un échantillon.

[1] https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html

[2] https://scikit-learn.org/stable/modules/naive_bayes.html

[3] https://scikit-learn.org/stable/modules/generated/sklearn.metrics.classification_report.html

II: Vectorisation et stockage des étiquettes

Par simplicité, 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(textes["appr"]).toarray()
X_test  = vectorizer.transform(textes["test"]).toarray()
y_train = classes["appr"]
y_test  = classes["test"]
```

III: Classification et évaluation

```
from sklearn.naive_bayes import GaussianNB
gnb = GaussianNB()
y_pred = gnb.fit(X_train, y_train).predict(X_test)
print("Erreurs d'étiquetage sur %d textes : %d"% (X_test.shape[0], (y_t

from sklearn.metrics import classification_report
print(classification_report(y_test, y_pred))
```

Votre travail consiste à faire varier les paramètres de CountVectorizer :

- Tester différentes valeurs pour le nombre de caractéristiques : `max_features`[4]
- Tester en vectorisant avec des N-grammes de caractères : `analyzer="char"`
- Faire varier les valeurs min et max de N: `ngram_range=(min,max)`
- Quel est la configuration la plus efficace ?
- A votre avis pourquoi ?

[4]Pas de valeur trop grande pour ne pas surcharger la mémoire