4.5 Une application : un modèle probabiliste des mots d'une langue donnée.

Le site `https://fr.wiktionary.org/wiki/Wiktionnaire:Listes_de_fréquence/` fournit des listes des mots les plus fréquents dans plusieurs langues.

On ne considérera que des langues dont un grand nombre de mots n'utilisent que les 26 lettres minuscules de l'alphabet latin

`a,b,c,d,e,f,g,h,i,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x,y,z`

(ce qui exclut en particulier toute forme d'accent). Par exemple : l'anglais, l'allemand, l'espagnol et le néerlandais.

Un premier objectif est de construire un modèle probabiliste pour chacune de ces langues. Dans un second temps, on pourra utiliser ces modèles pour différentes taches de prédiction - prédire la langue d'un mot, prédire la lettre suivant une séquence de lettres connaissant la langue à laquelle le mot appartient, ou prédire la lettre manquante dans un mot d'une langue donnée.

Première tâche : constitution des corpus On créera, pour chaque langue retenue, un fichier constitué des mots écrits sur l'alphabet de 26 lettres choisis parmi les 2000 mots les plus fréquents. J'ai fourni les fichiers `anglais2000` et `allemand2000`. Extension possible
- à l'espagnol et au néerlandais
- aux 4000 mots les plus fréquents.

Seconde tâche : construction de modèles de Markov cachés Pour un nombre d'états n compris entre 1 et un nombre $nbEtatsMax$ à déterminer, construire le HMM le mieux adapté au corpus. Cela peut nécessiter un temps de calcul important.
- Afin d'éviter de faire des itérations inutiles, vous pourrez écrire une variante de `BW2` dans laquelle les itérations s'arrêtent lorsque la log-vraisemblance semble se stabiliser (par exemple : pas d'évolution notable lors des 10 dernières itérations).
- Vous devrez considérer plusieurs HMM d'initialisation aléatoires. Question : la log-vraisemblance atteinte varie t-elle beaucoup selon les initialisations ? Si oui, il faudra considérer plus d'initialisations que si la réponse est négative. Cela varie t-il en fonction de n ?
- Vous devez constater que la log-vraisemblance optimale augmente en fonction de n : est-ce bien le cas ? Pouvez vous l'expliquer ?

Troisième tâche : choix d'un modèle optimal Pour un nombre d'états donné, on sait approcher un modèle optimal. Mais comment savoir quel nombre d'états optimal considérer ? Comme la log-vraisemblance de l'échantillon augmente avec le nombre d'états, il n'est pas possible de se baser sur ce critère pour sélectionner un nombre d'états optimal. En effet, plus le modèle est complexe - et la complexité du modèle croît avec le nombre d'états -, mieux il pourra rendre compte des données sur lesquelles il est entrainé - au risque de se spécialiser sur ces données et de moins bien se comporter sur de nouvelles données. On résoud ce problème expérimentalement par un algorithme de **validation croisée** :

- partitionner les données S en `nbFolds` parties (approximativement) de même taille $S_1, \ldots, S_{nbFolds}$; on notera $\overline{S_i} = S \setminus S_i$.
- pour un nombre d'états n compris entre `nbEtatsMin` et `nbEtatsMax`
  - pour i variant de 1 à `nbFolds`, entraîner un modèle à n états sur $\overline{S_i}$ et calculer la log-vraisemblance de ce modèle $lv_{n,i}$ sur S_i
  - calculer $lv_n = lv_{n,1} + \cdots + lv_{n,nbFolds}$.
- sélectionner le nombre d'états $nbOpt = argmax_n lv_n$.

Le code (non optimal) suivant résoud le problème en `Python` :

```
def xval(nbFolds, S, nbL, nbSMin, nbSMax, nbIter, nbInit):
    n = len(S)
    l = np.random.permutation(n)
    lvOpt = -float('inf')
    for nbS in range(nbSMin, nbSMax):
        lv = 0
        for i in range(1,nbFolds+1):
            f1 = int((i-1)*n/nbFolds)
            f2 = int(i*n/nbFolds)
            learn = [S[l[j]] for j in range(f1)]
            learn += [S[l[j]] for j in range(f2,n)]
            test = [S[l[j]] for j in range(f1,f2)]
            h = BaumWelch3(nbL,nbS,learn,nbIter,nbInit)
        lv += h.logV(test)
        if lv > lvOpt:
            lvOpt = lv
            nbSOpt = nbS
    return lvOpt,nbSOpt
```

Une fois que le nombre d'états optimal `nbEtatsOpt` est sélectionné, on peut considérer que le modèle correspondant construit à l'étape précédente est le modèle optimal. À l'issue de cette tâche, vous disposerez donc d'un modèle unique par langue. Question : le nombre d'états optimal est-il le même pour toutes les langues ?

Quatrième tâche : prédire la langue d'un mot Étudiez les performances des modèles construits dans la section précédente. Pour des langues prises 2 à 2 ou globalement. Exemples de mots les mieux discriminés. Exemples de mots pour lesquels les modèles se trompent. Langues les plus (resp. moins) faciles à différencier.

4.5.1 Projet à rendre

Chaque groupe devra rendre une archive unique comprenant :

- le programme source réparti en (au moins) deux modules : `HMM_Class.py` et `projet.py`
- un fichier de tests : `HMM_unittest.py`
- un fichier par langue, codant le meilleur modèle obtenu (au format défini dans le cours)
- un fichier d'accompagnement décrivant les fonctionnalités implantées, les expériences que vous avez réalisées avec les commentaires associés.

Le programme source devra au moins contenir les fonctionnalités suivantes :

- sauver et charger un fichier représentant un HMM,
- calculer la probabilité d'une séquence relativement à un HMM (méthodes forward et backward),
- calculer le chemin de Viterbi d'une séquence relativement à un HMM,
- prédire le symbole suivant d'une séquence produite par un HMM donné,
- calculer la vraisemblance et log-vraisemblance, d'un échantillon pour un HMM donné,
- générer aléatoirement un HMM dont on donne le nombre de lettres et le nombre d'états,
- l'algorithme de Baum-Welch, calculant un HMM à partir d'un échantillon avec comme paramètres le nombre de lettres, le nombre d'états, le nombre d'itérations maximal, le nombre de points d'initialisation choisis.

Le programme principal devra proposer une exécution standard permettant de tester les différentes fonctionnalités.

Le programme devra être convenablement commenté. Quelques contrôles de saisie seront implantés, avec la gestion des erreurs associée. Un fichier de test devra permettre de tester la plupart des fonctionnalités implantées. Le programme devra passer le test `pep8`.

Chaque participant à un projet doit en connaître la totalité. Le programme sera utilisé pendant l'examen.