ITC - PC

devoir libre

Exercice 1

Dans cet exercice, on souhaite classifier des mots dans différentes langues. Étant donné un mot, l'objectif est donc de savoir à quelle langue il appartient. Pour cela, on dispose d'un ensemble de mots appartenant à chaque langue. Pour déterminer la langue d'un mot, on va utiliser l'algorithme des plus proches voisins, ce qui nécessite d'abord de définir une distance entre deux mots.

I - Distances sur les mots

I.1 - Distance de Hamming

La **distance de Hamming** entre deux mots de même longueur est le nombre de positions où les deux mots sont différents.
Par exemple, la distance de Hamming entre *arbre* et *arche* est 2 car il y a deux différences :
à l'indice 2 (b $\neq$ c) et à l'indice 3 (r $\neq$ h).
On rappelle qu'on peut accéder à la i-ème lettre d'une chaîne de caractères s avec `s[i]` et qu'on peut connaître sa taille avec `len(s)`.

1. Écrire une fonction `hamming(s, t)` qui calcule la distance de Hamming entre deux mots de même longueur. Par exemple, `hamming("arbre", "arche")` doit renvoyer 2.

I.2 - Distance de Levenshtein

Si s et t sont deux chaînes de caractères de tailles n et p, la **distance de Levenshtein** $d(s,t)$ entre s et t est le nombre minimum de modifications de s pour obtenir t, où chaque modification est une insertion, une suppression ou une substitution d'un caractère.
Par exemple, $d(chat, chien) = 3$ car, à partir de $chat$, on peut obtenir $chien$ en remplaçant a par i puis t par e et en insérant n à la fin.
On pose $s = s_1 ... s_n$ et $t = t_1 ... t_p$ les lettres de s et t (ainsi, s_1 est la première lettre de s, par exemple) et on définit $d_{i,j}(s,t) = d(s_1 ... s_i, t_1 ... t_j)$ (distance de Levenshtein entre les i premières lettres de s et les j premières lettres de t). On admet la relation de récurrence suivante :

- $d_{0,j}(s,t) = j$ pour tout $j \leqslant 0$;
- $d_{i,0}(s,t) = i$ pour tout $i \leqslant 0$;
- $d_{i,j}(s,t) = d_{i-1,j-1}(s,t)$ si $s_i = t_j$;
- $d_{i,j}(s,t) = min(d_{i-1,j-1}(s,t),\ d_{i-1,j}(s,t),\ d_{i,j-1}(s,t)) + 1$ sinon (correspondant aux trois possibilités de modification).

2. Écrire une fonction récursive `d(i, j, s, t)` qui calcule $d_{i,j}(s,t)$, en utilisant directement la relation de récurrence.
3. Expliquer pourquoi cette fonction `d` n'est pas efficace.
4. Écrire une fonction `d(s, t)` renvoyant `d(s, t)` par mémoïsation, en utilisant un dictionnaire pour stocker les valeurs déjà calculées
5. Quelle est la complexité de la fonction `d` précédente ?

II - Plus proches voisins

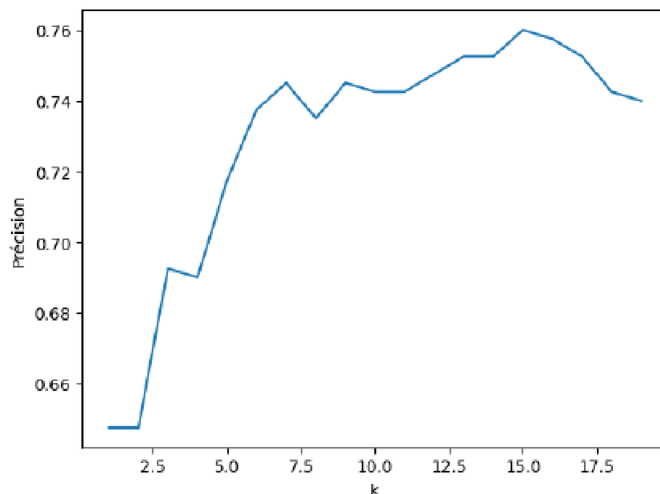
On suppose avoir une liste `X` de mots dont les langues sont données par `y` (`y[i]` est la langue du mot `X[i]`). On commence par séparer `X` en deux ensembles `X_train` et `X_test` (et les langues correspondantes `y_train` et `y_test`).

6. Écrire une fonction `split(L)` renvoyant deux listes `L1` et `L2` séparant `L` en deux listes de même taille (à ± 1 près).
7. Expliquer quel est l'intérêt de séparer les données en deux ensembles avant d'utiliser un algorithme d'apprentissage.

On suppose l'existence d'une fonction `voisins(x, k)` permettant de trouver les indices des `k` plus proches voisins d'un mot `x` dans la liste de mots `X_train` (en utilisant, par exemple, la distance de Levenshtein). Ainsi, si `L = voisins(x, k)` alors `L[0]` est l'indice du mot de le plus proche de `x` dans `X_train`, et `X_train[L[0]]` est le mot correspondant (le mot le plus proche de `x` dans `X_train`).

8. Écrire une fonction `plus_frequent(L)` renvoyant l'élément le plus fréquent d'une liste `L`.
   Par exemple, `plus_frequent([3, 4, 1, 1, 4, 3, 1])` doit renvoyer 1.
   On essaiera d'avoir la meilleure complexité possible.
9. En déduire une fonction `knn(x, k)` qui renvoie la langue majoritaire parmi les `k` mots les plus proches de `x` dans `X_train`.
10. Écrire une fonction `precision(k)` qui renvoie la précision de l'algorithme knn pour une valeur de `k` donnée, en utilisant les données de test `(X_test)`.

En calculant la précision pour différentes valeurs de `k`, on obtient la courbe suivante :



11. Donner (approximativement) l'erreur minimum que l'on peut obtenir avec l'algorithme des plus proches voisins.
12. On suppose avoir stocké les valeurs de précision dans un dictionnaire `precisions` dont les clés sont des valeurs de `k` et les valeurs les précisions correspondantes.
    Écrire une fonction `meilleur_k(precisions)` qui renvoie la valeur de k qui donne la meilleure précision.

III - Distance entre deux phrases

Nous avons précédemment défini une distance entre deux mots. Il peut également être intéressant de définir une distance entre deux phrases. Une possibilité est le *bag of words*.

13. Écrire une fonction `bag(s)` qui renvoie un dictionnaire dont les clés sont les mots de la phrase `s` et les valeurs le nombre de fois où le mot apparaît dans `s`. On utilisera `s.split()` pour obtenir

la liste des mots de `s`.
Par exemple, `bag("bonne année bonne santé")` doit renvoyer `{"bonne": 2, "année": 1, "santé": 1}`.

On peut voir `bag(s)` comme un vecteur de R^n où n est le nombre de mots différents dans la langue, ce qui permet de définir une distance entre deux phrases en calculant la distance entre les vecteurs correspondants.

14. Écrire une fonction `d_bag(s1, s2)` qui renvoie la distance de Manhattan (en valeur absolue) entre `bag(s1)` et `bag(s2)`.
Par exemple, `d_bag("bonne année bonne santé", "très bonne année")` doit renvoyer 3 (il y a 3 différences : "santé" et "très" apparaissent dans une phrase mais pas dans l'autre, et "bonne" apparaît une fois de plus dans `s1`).

Pour prendre en compte l'ordre des mots, on peut aussi utiliser des *n-grams*. Un n-gram est une suite de n mots consécutifs.
Par exemple, les 2-grams de la phrase "bonne année bonne santé" sont "bonne année", "année bonne", et "bonne santé".

15. Écrire une fonction `bag_ngram(s, n)` qui renvoie un dictionnaire dont les clés sont les n-grams de la phrase `s` et les valeurs le nombre de fois où le n-gram apparaît dans `s`. On pourra utiliser `s1 + s2` pour concaténer deux chaînes de caractères.

Exercice 2

On souhaite utiliser une base de données pour stocker les résultats obtenus par une communauté de joueurs en ligne. On suppose qu'on dispose d'une base de données comportant deux tables : `Joueurs(id_j, nom, pays)` et `Parties(id_p, date, duree, score, id_joueur)` où :

- `id_j` de type entier, est la clef primaire de la table `Joueurs` ;
- `nom` est une chaîne de caractères donnant le nom du joueur ;
- `pays` est une chaîne de caractères donnant le pays du joueur ;

- `id_p` de type entier, est la clef primaire de la table `Parties` ;
- `date` est la date (AAAAMMJJ) de la partie ;
- `duree` de type entier, est la durée en secondes de la partie ;
- `score` de type entier, est le nombre de points marqués au cours de la partie ;
- `id_joueur` est un entier qui identifie le joueur de la partie.

1. Rédiger une requête SQL qui renvoie la date, la durée et le score de toutes les parties jouées par Alice, listées par ordre chronologique.
2. Alice vient de réaliser un score de n points. Rédiger une requête SQL qui renvoie la position qu'aura le score n dans le classement des parties par ordre de score (on suppose que la partie que vient de jouer Alice n'est pas encore insérée dans la base de données). En cas d'ex aequo pour le score n le rang sera le même pour tous les joueurs ayant ce score.
3. Rédiger une requête SQL qui renvoie le record de France pour ce jeu.
4. Rédiger une requête SQL qui renvoie le rang d'Alice, c'est-à-dire sa position dans le classement des joueurs par ordre de leur meilleur score.