Dans ce devoir, 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.

> Solution :
>
> ```python
> def hamming(s, t):
>     n = len(s)
>     d = 0
>     for i in range(n):
>         if s[i] != t[i]:
>             d += 1
>     return d
> ```

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 $j \geq 0$;
- $d_{i,0}(s, t) = i$ pour tout $i \geq 0$;
- $d_{i,j}(s, t) = d_{i-1,j-1}$ si $s_i = t_j$;
- $d_{i,j}(s, t) = \min(d_{i-1,j-1}, d_{i-1,j}, d_{i,j-1}) + 1$ 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.

> Solution :
>
> ```python
> def d(i, j, s, t):
>     if i == 0:
>         return j
>     if j == 0:
>         return i
>     if s[i] == t[j]:
>         return d(i-1, j-1, s, t)
>     return min(d(i-1, j-1, s, t), d(i-1, j, s, t), d(i, j-1, s, t)) + 1
> ```

3. Expliquer pourquoi cette fonction `d` n'est pas efficace.

> Solution : Il peut y avoir plusieurs appels récursifs à la fonction `d` avec les mêmes arguments, ce qui est inefficace et donne une complexité exponentielle.

4. Écrire une fonction `d(s, t)` renvoyant $d(s,t)$ par mémoïsation, en utilisant un dictionnaire pour stocker les valeurs déjà calculées.

> Solution :
>
> ```python
> def d(s, t):
>     memo = {} # dictionnaire pour stocker les valeurs déjà calculées
>     def aux(i, j):
>         if (i, j) in memo:
>             return memo[(i, j)]
>         if i == 0:
>             return j
>         if j == 0:
>             return i
>         if s[i] == t[j]:
>             memo[(i, j)] = aux(i-1, j-1)
>         else:
>             memo[(i, j)] = min(aux(i-1, j-1), aux(i-1, j), aux(i, j-1)) + 1
>         return memo[(i, j)]
>     return aux(len(s) - 1, len(t) - 1)
> ```

5. Quelle est la complexité de la fonction `d` précédente ?

> Solution : Soit n la taille de `s` et p la taille de `t`. Soit $i \in [\![0, n-1]\!]$ et $j \in [\![0, p-1]\!]$.
> Comptons le nombre d'appels récursifs à `aux(i, j)`. Seuls `aux(i+1, j)`, `aux(i+1, j+1)` et `aux(i, j+1)` peuvent appeler `aux(i, j)`. Donc il y a au plus 3 appels récursifs à `aux(i, j)`.
> Le nombre total d'appels récursifs pour calculer `aux(i, j)` pour $i \in [\![0, n-1]\!]$ et $j \in [\![0, p-1]\!]$ est donc au plus $3np = \boxed{\mathrm{O}(np)}$.

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).

> Solution : 1ère possibilité (avec slicing) :
>
> ```python
> def split(L):
>     n = len(L)
>     return L[:n//2], L[n//2:]
> ```
>
> 2ème possibilité :
>
> ```python
> def split(L):
>     n = len(L)
>     L1, L2 = [], []
>     for i in range(n):
>         if i % 2 == 0:
>             L1.append(L[i])
>         else:
>             L2.append(L[i])
>     return L1, L2
> ```

7. Expliquer quel est l'intérêt de séparer les données en deux ensembles avant d'utiliser un algorithme d'apprentissage.

> Solution : L'algorithme est censé être utilisé sur de nouvelles données (que l'on ne connaît pas encore). Tester l'algorithme sur des données qu'il a déjà vues ne permet pas de savoir s'il est efficace sur de nouvelles données.

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.

   > Solution :
   >
   > ```python
   > def plus_frequent(L):
   >     d = {}
   >     for x in L:
   >         if x in d:
   >             d[x] += 1
   >         else:
   >             d[x] = 1
   >     return max(d, key=d.get)
   > ```
   >
   > On peut aussi remplacer `return max(d, key=d.get)` par :
   >
   > ```python
   >     kmin, vmin = 0, 0
   >     for k in d:
   >         if d[k] > vmin:
   >             kmin, vmin = k, d[k]
   >     return kmin
   > ```

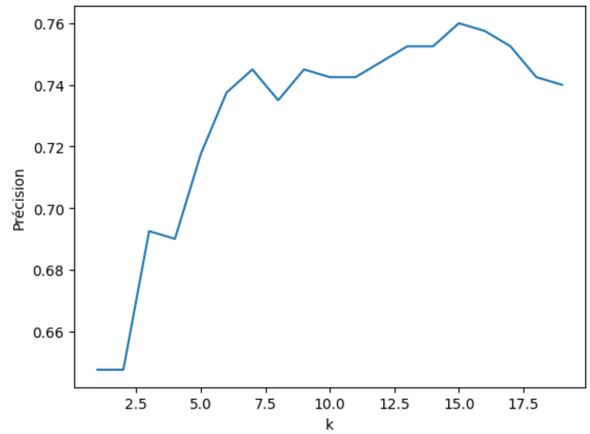
9. En déduire une fonction `knn(x, k)` qui renvoie la langue majoritaire parmi les k mots les plus proches de `x` dans `X_train`.

   > Solution :
   >
   > ```python
   > def knn(x, k):
   >     L = voisins(x, X_train, k)
   >     return plus_frequent([y_train[i] for i in L])
   > ```

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`).

    > Solution :
    >
    > ```python
    > def precision(k):
    >     n = len(X_test)
    >     nb_correct = 0
    >     for i in range(n):
    >         if knn(X_test[i], k) == y_test[i]:
    >             nb_correct += 1
    >     return nb_correct / n
    > ```

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.

> Solution : Graphiquement, la précision maximum semble être 0.76. Donc l'erreur minimum est $1 - 0.76 = 0.24$.

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.

> Solution : 1ère solution :
>
> ```
> def meilleur_k(precisions):
>     return max(precisions, key=precisions.get)
> ```
>
> 2ème solution :
>
> ```
> def meilleur_k(precisions):
>     kmax = 1
>     for k in precisions:
>         if precisions[k] > precisions[kmax]:
>             kmax = k
>     return kmax
> ```

13. On applique l'algorithme des plus proches voisins avec deux langues : anglais (donné par l'entier 0 dans `y_train`) et français (donné par l'entier 1 dans `y_train`). On obtient la matrice de confusion $\begin{pmatrix} 72 & 33 \\ 30 & 65 \end{pmatrix}$. Dire à quoi correspond chacun des nombres de cette matrice. Quelle est la précision correspondante ?

> Solution : 72 est le nombre de fois où l'algorithme a prédit anglais et où la langue était bien anglais. 33 est le nombre de fois où l'algorithme a prédit anglais et où la langue était français. 30 est le nombre de fois où l'algorithme a prédit français et où la langue était anglais. 65 est le nombre de fois où l'algorithme a prédit français et où la langue était français. La précision est donc $137/200 = 0.685$.

III Distance entre deux phrases

Nous avons précédement 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*.

14. 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}`.

> Solution :

```python
def bag(s):
    d = {}
    for mot in s.split():
        if mot in d:
            d[mot] += 1
        else:
            d[mot] = 1
    return d
```

On peut voir `bag(s)` comme un vecteur de $\mathbb{R}^n$ 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.

15. 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`).

Solution :

```python
def d_bag(s1, s2):
    d1, d2, d = bag(s1), bag(s2), 0
    for mot in d1:
        if mot in d2:
            d += abs(d1[mot] - d2[mot])
        else:
            d += d1[mot]
    for mot in d2:
        if mot not in d1:
            d += d2[mot]
    return d
```

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"`.

16. 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.

Solution :

```python
def bag_ngram(s, n):
    d = {}
    words = s.split()
    for i in range(len(words) - n + 1):
        ngram = words[i]
        for j in range(1, n):
            ngram += " " + words[i + j]
        if ngram in d:
            d[ngram] += 1
        else:
            d[ngram] = 1
    return d
```

Une modification immédiate `d_bag` permet alors de calculer la distance entre deux phrases en utilisant des n-grams.

IV Exercice bonus : plus grand sous-mot palindrome

Exercice à faire seulement si vous avez terminé le reste.
Un palindrome est un mot qui se lit de la même façon dans les deux sens. Par exemple, `"radar"` est un palindrome. On dit que t est un sous-mot de s si les lettres de t apparaissent dans le même ordre dans s, mais pas forcément de façon consécutive.

17. Écrire une fonction `pal(s)` qui renvoie, en complexité quadratique en la taille de `s`, la plus grande longueur d'un sous-mot de `s` qui est un palindrome.

Solution : Soit $p_{i,j}$ la longueur maximum d'un sous-mot de $s[i:j]$ qui soit un palindrome. Alors $p_{i,j} = p_{i+1,j-1} + 2$ si $s[i] = s[j-1]$ et $p_{i,j} = \max(p_{i+1,j}, p_{i,j-1})$ sinon. On peut donc calculer $p_{i,j}$ par programmation dynamique :

```python
def pal(s):
    n = len(s)
    p = [[0] * n for _ in range(n)]
    for i in range(n): # cas de base
        p[i][i] = 1
    for l in range(2, n + 1):
        for i in range(n - l + 1):
            j = i + l
            if s[i] == s[j - 1]:
                p[i][j] = p[i + 1][j - 1] + 2
            else:
                p[i][j] = max(p[i + 1][j], p[i][j - 1])
    return p[0][n - 1]
```