tp_l8dn003_20_04_cor_q3

March 29, 2020

TP n°4: distance de Levenshtein

La distance de Levenshtein, ou distance d'édition, est une façon de mesurer à quel point deux mots se ressemblent. L'idée de base est d'attribuer un poids à chacune des opérations qui peuvent permettre de passer d'un mot à un autre (suppression ou insertion d'une lettre, interversion de deux lettres, remplacement d'une lettre par une autre…). La distance est calculée à partir de ce nombre d'opérations (éventuellement pondérées). Cette notion est très utilisée pour faire de la correction orthographique, pour gérer les tokens inconnus dans de nombreuses applications de TAL, pour faire de la morphologie computationnelle, *etc*. L'article Wikipedia https://fr.wikipedia.org/wiki/Distance_de_Levenshtein est une bonne source d'information. On y donne même une version de l'algorithme, mais je propose plutôt une approche progressive pour mieux appréhender les aspects algorithmiques du problème: on va implémenter différentes fonctions de mesure de distance qui en faisant différentes hypothèses simplificatrices.

3. Implémentation de la distance de Levenshtein standard

Les deux questions précédentes permettent de se faire une (petite) idée de la complexité de la version complète de l'algorithme. La troisième partie du TP va consister à implémenter l'algorithme, en partant de la version proposé dans la page wikipedia. C'est un algorithme de programmation dynamique: l'idée est de stocker le plus possible de résultats intermédiaires pour ne les faire qu'une seule fois. La base de l'algorithme est donc la table qui stocke ces informations.

1. Essayer de faire tourner l'algo sur quelques exemples à la main pour comprendre son esprit général. Les exemples de la page peuvent aider.
2. Adapter en python l'algorithme de la page. On trouve facilement sur internet des bouts de code qui implémentent l'algorithme, mais j'espère que vous avez compris le peu d'intérêt qu'il y a à recopier du code si on ne comprend pas ce qu'il fait.
3. Installer (avec anaconda ou pip) le package python-levenshtein, ce qui va vous permettre de comparer votre version à l'implémentation fournie par cette bibliothèque.

On part directement du pseudo-code donné dans la cellule suivante

[31]:
```
# entier DistanceDeLevenshtein(caractere chaine1[1..longueurChaine1],
#                               caractere chaine2[1..longueurChaine2])
#   // d est un tableau de longueurChaine1+1 rangées et longueurChaine2+1 colonnes
#   // d est indexé à partir de 0, les chaînes à partir de 1
#   déclarer entier d[0..longueurChaine1, 0..longueurChaine2]
#   // i et j itèrent sur chaine1 et chaine2
#   déclarer entier i, j, coûtSubstitution
#
#   pour i de 0 à longueurChaine1
#       d[i, 0] := i
#   pour j de 0 à longueurChaine2
#       d[0, j] := j
#
#   pour i de 1 à longueurChaine1
#      pour j de 1 à longueurChaine2
#          si chaine1[i-1] = chaine2[j-1] alors coûtSubstitution := 0
#                                         sinon coûtSubstitution := 1
#          d[i, j] := minimum(
#                                 d[i-1, j  ] + 1,                 // effacement du␣
 ↪nouveau caractère de chaine1
#                                 d[i,   j-1] + 1,                 // insertion dans␣
 ↪chaine2 du nouveau caractère de chaine1
#                                 d[i-1, j-1] + coûtSubstitution   // substitution
#                             )
#
#   renvoyer d[longueurChaine1, longueurChaine2]
```

[32]:
```
# Pour coder la table, on peut utiliser une liste de listes
# Cette fonction crée une table entière initialisée à 0
# Pour plus d'efficacité on peut utiliser des arrays (numpy)
def init_table(k,l):
    table = []
    for i in range(k):
        x = []
        for j in range(l):
            x.append(0)
        table.append(x)
    return table

def maLevenshtein(s,t):
    table = init_table(len(s)+1,len(t)+1)
    for i in range(len(s)+1):
        table[i][0] = i
    for j in range(len(t)+1):
        table[0][j] = j
    for i in range(1,len(s)+1):
        for j in range(1,len(t)+1):
            cout = 0 if s[i-1] == t[j-1] else 1
            table[i][j] = min(table[i-1][j]+1,table[i][j-1]+1,table[i-1][j-1]+cout)
    return table[len(s)][len(t)]
```

[24]:
```
# Petite vérification avec quelques paires de mots
paires_test = [("mamaison","mison"),("bateau","toto"),("petard","petard")]
import Levenshtein as lev
for (x,y) in paires_test:
    print(x,y,maLevenshtein(x,y),lev.distance(x,y))
```

```
mamaison mison 3 3
bateau toto 5 5
petard petard 0 0
```

[27]:
```
# Test avec des mots piochés dans un texte (utilise la fonction charge_lexique())
# Ce script n'affiche que les cas où notre algorithme
# donnerait une valeur différente de celle de la bibliothèque
formes = charge_lexique("pensees_pascal_utf8.txt")
formes = formes[:200]
for i in range(len(formes)):
    for j in range(i,len(formes)):
        if maLevenshtein(x,y) != lev.distance(x,y):
            print(x,y)
```

[50]:
```
# Petite séquence pour comparer l'efficacité de notre version
formes = charge_lexique("pensees_pascal_utf8.txt")
formes = formes[:600]
# Mesure du temps (de calcul)
import time
t1 = time.process_time()
for i in range(len(formes)):
    for j in range(i,len(formes)):
        x,y = formes[i], formes[j]
        lev.distance(x,y)
t2 = time.process_time()
t = t2-t1
for i in range(len(formes)):
    for j in range(i,len(formes)):
        x,y = formes[i], formes[j]
        maLevenshtein(x,y)
t3 = time.process_time()
print("version bibliothèque: %f, version maison: %f (%.0f fois moins vite)"␣
 ↪%(t,t3-t2,(t3-t2)/t))
```

```
version bibliothèque: 0.159544, version maison: 10.453290 (66 fois moins vite)
```