tp_l8dn003_20_04_q1

March 26, 2020

0.1 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, inter-version 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.

0.1.1 Prise en compte (simplifiée) des substitutions

1. Ecrire une fonction `nb_subst(u,v)` qui renvoie le nombre de substitutions d'un caractère qui distinguent u de v. Par hypothèse les deux chaînes sont de même longueur (sinon on renvoie -1), et la seule opération mesurée est le remplacement d'un caractère par un autre.
2. Cette valeur dépend bien sûr du nombre de lettres du mot. On peut essayer de prendre cela en compte en renvoyant le ratio de caractères modifiés par rapport au nombre total de caractères: la distance maximale est alors à 1 (tous les caractères sont différents). Implémenter cette fonction `dist_subst(u,v)`.
3. Cette fonction est une fonction *partielle* : elle ne permet pas de définir une distance entre deux chaînes quelconques, mais seulement entre deux chaînes de même longueur. Pour passer à une mesure totale, on peut proposer que si les deux chaînes ne sont pas de même longueur, on fait comme si on rallongeait artificiellement la plus courte en ajoutant des caractères toujours différents de leur correspondant dans la chaîne la plus longue. Le nombre d'opérations peut être interprété comme *le nombre de caractères à modifier ou à supprimer pour passer de la chaîne la plus longue à la plus courte.* Comme précédemment, on va renvoyer un ratio plutôt qu'un nombre brut. Implémenter cette fonction `subst_totale(u,v)`.
4. Pour tester les fonctions :
    1. Faites un jeu d'essai (paires de mots) qui correspondent aux différents cas de figure que vous pouvez anticiper. Augmentez systématiquement ce jeu d'essai, en vérifiant bien sûr que vos prédictions sont correctes.
    2. Prennez plusieurs centaines de mots-formes dans un texte (pas trop quand-même: 200 c'est bien) de la façon dont on a procédé dans les TP précédents, et calculez les distances 2 à 2: regardez quels sont les mots les plus proches (mais quand-même différents), etc.

Une fois vos tests réalisés, vous constaterez que, sans surprise, les mots les plus proches (mais différents) sont les paires de mots les plus longs qui ne diffèrent que d'une seule lettre. En effet, à cause de la normalisation effectuée, une différence d'un seul caractère est comptée comme plus petite si le mot a plus de lettres. Cela peut correspondre à une certaine intuition, mais dans la pratique la distance proposée par Levenshtein ne procède pas à une normalisation.

0.1.2 Autre approche : prise en compte de séquences absentes

Ecrire une fonction qui va s'appliquer à deux chaînes de caractères telles que la seconde est obtenue à partie de la première en ôtant une (seule) portion *contiguë*. La fonction `coupe(u,v)` doit renvoyer la taille de la coupe qui a été pratiquée dans u pour obtenir v. Formellement: $\exists a, b, w$ t.q. $u = awb$ & $v = ab$, où a, b et w sont des chaînes de caractères non vides. La fonction doit renvoyer -1 si la condition n'est pas vérifiée (0 signifie que u==v).

Indice algorithmique: il faut chercher d'une part le préfixe commun, d'autre part le suffixe commun: si ces deux sous-chaînes sont non vides, il ne reste plus qu'à compter la taille de w.

0.1.3 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.

Nombre de substitutions entre chaînes de même longueur.

[1]:
```python
def nb_subst(orig,dest):
    if len(orig) != len(dest):
        return -1
    c = 0
    for i in range(len(orig)):
        if orig[i] != dest[i]:
            c += 1
    return c
```

[2]:
```python
# Une première fonction de test, avec 10 mots différents
def test_dix_mots(f):
    t = "le plus noir des nuages a toujours sa frange d'or".split()
    for i in range(len(t)):
        for j in range(i,len(t)):
            x,y = t[i], t[j]
```

```
            mesure = f(x,y)
            if mesure > 0: print("%10s %10s %f" % (x,y,mesure))
```

[3]:
```
test_dix_mots(nb_subst)
```

```
        le         sa 2.000000
      plus       noir 4.000000
      plus       d'or 4.000000
      noir       d'or 3.000000
    nuages     frange 5.000000
```

Nombre de substitutions de chaînes de même longueur, rapporté à la longueur (→ distance).

[4]:
```
# C'est le même code que précédemment, avec simplement une fraction comme␣
 ↪valeur de retour
def dist_subst(orig,dest):
    if len(orig) != len(dest):
        return -1
    c = 0
    for i in range(len(orig)):
        if orig[i] != dest[i]:
            c += 1
    return c/len(orig)
```

[5]:
```
test_dix_mots(dist_subst)
```

```
        le         sa 1.000000
      plus       noir 1.000000
      plus       d'or 1.000000
      noir       d'or 0.750000
    nuages     frange 0.833333
```

[6]:
```
# Deuxième fonction de test avec plus de mots
# Toutes les paires de mots testés sont stockées dans une liste qui est␣
 ↪renvoyée.
def test_vingt_mots(f):
    t = "montagne montague montera monticule portage protège pylone police␣
 ↪pilote pilate pratique pirate patate lien lieu lois lions loup loupe poule".
 ↪split()
    l = []
    for i in range(len(t)):
        for j in range(i,len(t)):
            x,y = t[i], t[j]
            l.append((x,y,f(x,y)))
    return l
```

[7]:
```
# Récupération de toutes les mesures 2 à 2
results = test_vingt_mots(dist_subst)
# Récupération des mesures >0
results_pos = []
for c in results:
    if c[2] > 0:
        results_pos.append(c)
# Tri des résultats >0
results_pos_tries = sorted(results_pos, key=lambda x: x[2])
# Affichage tabulé des résultats positifs triés
for (x,y,m) in results_pos_tries:
    print("%10s %10s %f" % (x,y,m))
```

```
 montagne   montague 0.125000
   pilote     pilate 0.166667
   pilate     pirate 0.166667
     lien       lieu 0.250000
   pylone     pilote 0.333333
   pilote     pirate 0.333333
   pilate     patate 0.333333
   pirate     patate 0.333333
    loupe      poule 0.400000
  portage    protège 0.428571
   pylone     police 0.500000
   pylone     pilate 0.500000
   police     pilote 0.500000
   police     pilate 0.500000
   pilote     patate 0.500000
     lois       loup 0.500000
 montague   pratique 0.625000
   pylone     pirate 0.666667
   pylone     patate 0.666667
   police     pirate 0.666667
   police     patate 0.666667
  montera    portage 0.714286
 montagne   pratique 0.750000
     lien       lois 0.750000
     lien       loup 0.750000
     lieu       lois 0.750000
     lieu       loup 0.750000
    lions      loupe 0.800000
  montera    protège 0.857143
    lions      poule 1.000000
```

Distance de substitution comme fonction totale On définit la métrique pour faire en sorte que la distance est définie entre toutes les paires de mots.

[8]:
```
# Il faut commencer par trouver le mot le plus petit, pour obtenir
# (1) le nombre de tours de boucles (= la longueur du plus petit), et
# (2) le nombre de lettres en plus (la différence entre les longueurs)
# On charge une fonction de s'occuper juste de ça:
# elle renvoie les longueurs dans l'ordre (petit,grand)
def longueurs_comparees(s,t):
    if len(s) < len(t):
        return (len(s), len(t))
    return (len(t), len(s))

# le compteur c est initialisé au nombre de lettres en plus
def subst_totale(orig,dest):
    (borne, lgmax) = longueurs_comparees(orig,dest)
    c = lgmax - borne
    for i in range(borne):
        if orig[i] != dest[i]:
            c += 1
    return c/lgmax
```

[9]:
```
# Récupération de toutes les mesures 2 à 2
results = test_vingt_mots(subst_totale)
# Récupération des mesures >0
results_pos = []
for c in results:
    if c[2] > 0:
        results_pos.append(c)
# Tri des résultats >0
results_pos_tries = sorted(results_pos, key=lambda x: x[2])
# Affichage tabulé des résultats positifs triés
# (je n'affiche que les 20 premiers pour la lisibilité)
for (x,y,m) in results_pos_tries[:20]:
    print("%10s %10s %f" % (x,y,m))
```

```
 montagne   montague 0.125000
   pilote     pilate 0.166667
   pilate     pirate 0.166667
     loup      loupe 0.200000
     lien       lieu 0.250000
   pylone     pilote 0.333333
   pilote     pirate 0.333333
   pilate     patate 0.333333
   pirate     patate 0.333333
     lien      lions 0.400000
    loupe      poule 0.400000
  portage    protège 0.428571
 montague  monticule 0.444444
 montagne    montera 0.500000
```

```
montagne    portage 0.500000
montague    montera 0.500000
montague    portage 0.500000
  pylone     police 0.500000
  pylone     pilate 0.500000
  police     pilote 0.500000
```

Tests en prenant des mots dans un texte. On réutilise les fonctions déjà définies dans d'autres TP

[10]:
```
def charge_fichier(nf):
    lgns = []
    with open(nf, "r", encoding="utf8") as f:
        for ligne in f:
            lgns.append(ligne.strip())
    return lgns
def charge_lexique(nf):
    liste_lignes = charge_fichier(nf)
    lexique = []
    for l in liste_lignes:
        for w in l.split():
            if w not in lexique:
                lexique.append(w)
    return lexique
```

[11]:
```
# On récupère les mots-formes du texte de Pascal
formes = charge_lexique("pensees_pascal_utf8.txt")
# Les 200 premiers suffiront pour faire le test
formes = formes[:200]
```

[12]:
```
# Variante des fonctions de test précédentes
# Les paramètres sont une liste de mots-formes,
# et la fonction de distance à tester
# La liste de toutes les comparaisons est retournée.
def test_lexique(liste,f):
    l = []
    for i in range(len(liste)):
        for j in range(i,len(liste)):
            x,y = liste[i], liste[j]
            l.append((x,y,f(x,y)))
    return l
```

[13]:
```
# Récupération de toutes les mesures 2 à 2
results = test_lexique(formes,subst_totale)
# Récupération des mesures >0
results_pos = []
for c in results:
```

```
    if c[2] > 0:
        results_pos.append(c)
# Tri des résultats >0
results_pos_tries = sorted(results_pos, key=lambda x: x[2])
# Affichage tabulé des résultats positifs triés
# (je n'affiche que les 20 premiers pour la lisibilité)
for (x,y,m) in results_pos_tries[:20]:
    print("%15s %15s %f" % (x,y,m))
```

```
 raisonnement.   raisonnement, 0.076923
 Jésus-Christ.    Jésus-Christ 0.076923
  contrariétés    Contrariétés 0.083333
     véritable       Véritable 0.111111
      Religion       Religion. 0.111111
      religion        Religion 0.125000
      l'homme,       l'homme. 0.125000
      l'homme,        l'homme 0.125000
      l'homme.        l'homme 0.125000
        Pascal        Pascal, 0.142857
        autres        autres. 0.142857
       Preuves        preuves 0.142857
         écrite        écrites 0.142857
   Chrétienne.    Chrétiennes. 0.166667
        Juifs.           Juifs 0.166667
        XVIII.         XXIII. 0.166667
        XXIII.         XXVII. 0.166667
        XXIII.         XXXII. 0.166667
        XXVII.         XXXII. 0.166667
         n'est          s'est 0.200000
```