td1a_correction_session4_5_jaccard

October 13, 2019

1 1A.algo - distance de Jaccard (dictionnaires) - correction

De la distance de Jaccard à la distance de Levenshtein.

[1]:
```
from jyquickhelper import add_notebook_menu
add_notebook_menu()
```

[1]:
```
<IPython.core.display.HTML object>
```

1.1 Exercice 1 : Constuire l'ensemble des lettres supprimées et ajoutées

Deux mots n'ont pas forcément la même longueur, il est donc difficile de les comparer lettre à lettre. De même, on ne doit pas tenir compte de l'ordre des lettres dans chaque mot. Cette contrainte peut paraître plus difficile à mettre en oeuvre. Pourtant, si on construit un résultat intermédiaire : le décompte de chaque lettre dans un mot.

[2]:
```
def compte_lettre(mot):
    d = {}
    for c in mot:
        d[c] = d.get(c,0) + 1
    return d

compte_lettre("lettre"), compte_lettre("etre")
```

[2]:
```
({'e': 2, 'l': 1, 'r': 1, 't': 2}, {'e': 2, 'r': 1, 't': 1})
```

Une lettre est ajoutée ou supprimée entre deux mots si son décompe est différent dans les deux dictionnaires. On s'en sert pour contruire la liste des lettres supprimées et ajoutées. Cela revient à construire la différence entre deux dictionnaires.

[3]:
```
mot1 = "lettre"
mot2 = "etres"

d1 = compte_lettre(mot1)
d2 = compte_lettre(mot2)

suppression = {}
for l in d1:
    c1 = d1[l]
    c2 = d2.get(l, 0)  # la lettre l n'appartient pas forcément au second mot
    if c2 != c1:
        suppression[l] = c2 - c1
```

```
ajout = {}
for l in d2:
    if l not in d1:
        c1 = 0
        c2 = d2[l]
        if c2 != c1:
            ajout[l] = c2 - c1
    else:
        # on a déjà compté les lettres présentes dans les deux mots
        # lors de la première boucle
        pass

suppression, ajout
```

[3]:
```
({'l': -1, 't': -1}, {'s': 1})
```

[4]:
```

```

1.2 Exercice 2 : écrire une fonction qui calcule la distance de Jaccard

On copie et on colle le code précédent pour créer la distance.

[5]:
```
def jaccard(mot1, mot2):
    d1 = compte_lettre(mot1)
    d2 = compte_lettre(mot2)

    suppression = {}
    for l in d1:
        c1 = d1[l]
        c2 = d2.get(l, 0)  # la lettre l n'appartient pas forcément au second mot
        if c2 != c1:
            suppression[l] = c2 - c1

    ajout = {}
    for l in d2:
        if l not in d1:
            c1 = 0
            c2 = d2[l]
            if c2 != c1:
                ajout[l] = c2 - c1
        else:
            # on a déjà compté les lettres présentes dans les deux mots
            # lors de la première boucle
            pass

    dist = sum(abs(x) for x in suppression.values()) + sum(abs(x) for x in ajout.
 ↪values())
    return dist

jaccard("lettre", "etre"), jaccard("lettre", "etres")
```

[5]:
```
(2, 3)
```

[6]:
```
jaccard("marie", "aimer")
```

[6]:
```
0
```

[7]:
```
jaccard("hôpital", "hospital")
```

[7]:
```
3
```

1.3 Exercice 3 : calculer la matrice des distances

Tout d'abord, on découpe un texte en mot. On remplace les tirets en espaces.

[8]:
```
texte = "à combien sont ces six saucissons-ci, ces six saucisson-ci sont à six sous".
↪replace("-", " ").split()
texte
```

[8]:
```
['à',
 'combien',
 'sont',
 'ces',
 'six',
 'saucissons',
 'ci,',
 'ces',
 'six',
 'saucisson',
 'ci',
 'sont',
 'à',
 'six',
 'sous']
```

1.3.1 liste de listes

[9]:
```
distl = [ [ jaccard(texte[i], texte[j]) for i in range(len(texte))] for j in␣
↪range(len(texte))]
distl
```

[9]:
```
[[0, 8, 5, 4, 4, 11, 4, 4, 4, 10, 3, 5, 0, 4, 5],
 [8, 0, 7, 6, 8, 9, 6, 6, 8, 8, 5, 7, 8, 8, 9],
 [5, 7, 0, 5, 5, 8, 7, 5, 5, 7, 6, 0, 5, 5, 4],
 [4, 6, 5, 0, 4, 9, 4, 0, 4, 8, 3, 5, 4, 4, 5],
 [4, 8, 5, 4, 0, 9, 4, 4, 0, 8, 3, 5, 4, 0, 5],
 [11, 9, 8, 9, 9, 0, 9, 9, 9, 1, 8, 8, 11, 9, 6],
 [4, 6, 7, 4, 4, 9, 0, 4, 4, 8, 1, 7, 4, 4, 7],
 [4, 6, 5, 0, 4, 9, 4, 0, 4, 8, 3, 5, 4, 4, 5],
 [4, 8, 5, 4, 0, 9, 4, 4, 0, 8, 3, 5, 4, 0, 5],
 [10, 8, 7, 8, 8, 1, 8, 8, 8, 0, 7, 7, 10, 8, 5],
```

```
 [3, 5, 6, 3, 3, 8, 1, 3, 3, 7, 0, 6, 3, 3, 6],
 [5, 7, 0, 5, 5, 8, 7, 5, 5, 7, 6, 0, 5, 5, 4],
 [0, 8, 5, 4, 4, 11, 4, 4, 4, 10, 3, 5, 0, 4, 5],
 [4, 8, 5, 4, 0, 9, 4, 4, 0, 8, 3, 5, 4, 0, 5],
 [5, 9, 4, 5, 5, 6, 7, 5, 5, 5, 6, 4, 5, 5, 0]]
```

1.3.2 dictionnaire

[10]:
```
distd = { (w,y): jaccard(w,y) for w in texte for y in texte}
distd
```

[10]:
```
{('ces', 'ces'): 0,
 ('ces', 'ci'): 3,
 ('ces', 'ci,'): 4,
 ('ces', 'combien'): 6,
 ('ces', 'saucisson'): 8,
 ('ces', 'saucissons'): 9,
 ('ces', 'six'): 4,
 ('ces', 'sont'): 5,
 ('ces', 'sous'): 5,
 ('ces', 'à'): 4,
 ('ci', 'ces'): 3,
 ('ci', 'ci'): 0,
 ('ci', 'ci,'): 1,
 ('ci', 'combien'): 5,
 ('ci', 'saucisson'): 7,
 ('ci', 'saucissons'): 8,
 ('ci', 'six'): 3,
 ('ci', 'sont'): 6,
 ('ci', 'sous'): 6,
 ('ci', 'à'): 3,
 ('ci,', 'ces'): 4,
 ('ci,', 'ci'): 1,
 ('ci,', 'ci,'): 0,
 ('ci,', 'combien'): 6,
 ('ci,', 'saucisson'): 8,
 ('ci,', 'saucissons'): 9,
 ('ci,', 'six'): 4,
 ('ci,', 'sont'): 7,
 ('ci,', 'sous'): 7,
 ('ci,', 'à'): 4,
 ('combien', 'ces'): 6,
 ('combien', 'ci'): 5,
 ('combien', 'ci,'): 6,
 ('combien', 'combien'): 0,
 ('combien', 'saucisson'): 8,
 ('combien', 'saucissons'): 9,
 ('combien', 'six'): 8,
 ('combien', 'sont'): 7,
 ('combien', 'sous'): 9,
 ('combien', 'à'): 8,
 ('saucisson', 'ces'): 8,
```

```
('saucisson', 'ci'): 7,
('saucisson', 'ci,'): 8,
('saucisson', 'combien'): 8,
('saucisson', 'saucisson'): 0,
('saucisson', 'saucissons'): 1,
('saucisson', 'six'): 8,
('saucisson', 'sont'): 7,
('saucisson', 'sous'): 5,
('saucisson', 'à'): 10,
('saucissons', 'ces'): 9,
('saucissons', 'ci'): 8,
('saucissons', 'ci,'): 9,
('saucissons', 'combien'): 9,
('saucissons', 'saucisson'): 1,
('saucissons', 'saucissons'): 0,
('saucissons', 'six'): 9,
('saucissons', 'sont'): 8,
('saucissons', 'sous'): 6,
('saucissons', 'à'): 11,
('six', 'ces'): 4,
('six', 'ci'): 3,
('six', 'ci,'): 4,
('six', 'combien'): 8,
('six', 'saucisson'): 8,
('six', 'saucissons'): 9,
('six', 'six'): 0,
('six', 'sont'): 5,
('six', 'sous'): 5,
('six', 'à'): 4,
('sont', 'ces'): 5,
('sont', 'ci'): 6,
('sont', 'ci,'): 7,
('sont', 'combien'): 7,
('sont', 'saucisson'): 7,
('sont', 'saucissons'): 8,
('sont', 'six'): 5,
('sont', 'sont'): 0,
('sont', 'sous'): 4,
('sont', 'à'): 5,
('sous', 'ces'): 5,
('sous', 'ci'): 6,
('sous', 'ci,'): 7,
('sous', 'combien'): 9,
('sous', 'saucisson'): 5,
('sous', 'saucissons'): 6,
('sous', 'six'): 5,
('sous', 'sont'): 4,
('sous', 'sous'): 0,
('sous', 'à'): 5,
('à', 'ces'): 4,
('à', 'ci'): 3,
('à', 'ci,'): 4,
('à', 'combien'): 8,
```

```
 ('à', 'saucisson'): 10,
 ('à', 'saucissons'): 11,
 ('à', 'six'): 4,
 ('à', 'sont'): 5,
 ('à', 'sous'): 5,
 ('à', 'à'): 0}
```

On compare le nombre de coefficients :

[11]:
```
sum(len(_) for _ in distl)
```

[11]: 225

[12]:
```
len(distd)
```

[12]: 100

Le dictionnaire paraît une meilleure option en terme de nombre de coefficients. Mais en terme d'espace mémoire occupé, le dictionnaire est largement plus conséquent.

[13]:
```
import sys
sys.getsizeof(distl), sys.getsizeof(distd)
```

[13]: (192, 6240)

Pour ce petit exemple, la matrice paraît une bonne option mais lorsque les mots sont répétés un grand nombre de fois :

[14]:
```
texte_grand = texte * 40

distl = [ [ jaccard(texte_grand[i], texte_grand[j]) for i in range(len(texte_grand))]␣
 ↪for j in range(len(texte_grand))]
distd = { (w,y): jaccard(w,y) for w in texte_grand for y in texte_grand}
sum(len(_) for _ in distl), len(distd)
```

[14]: (360000, 100)

[15]:
```
sys.getsizeof(distl), sys.getsizeof(distd)
```

[15]: (5496, 6240)

Le dictionnaire permet également de vérifier aisément qu'un calcul a déjà été effectué afin de gagner du temps et de ne pas le refaire.

[16]:
```
distd = {}
for w in texte_grand:
    for y in texte_grand:
        if (w,y) not in distd:
            distd[w,y] = jaccard(w,y)
            distd[y,w] = distd[w,y]
len(distd)
```

[16]: 100

[17]: