

Devoir n°2

Exercice 1: classification : k -moyenne et single pass

On cherche à classer $n = 5$ documents textuels $doc_i, i \in \llbracket 1, n \rrbracket$ dans lesquels les termes T_1, T_2 et T_3 apparaissent un certain nombre de fois, ces occurrences étant décrites dans le tableau suivant :

	doc1	doc2	doc3	doc4	doc5
T1	1	2	0	1	1
T2	3	1	1	3	0
T3	3	0	0	1	1

Chaque document doc_i est donc représenté par un ensemble de $p = 3$ valeurs. On notera $d_i, i \in \llbracket 1, n \rrbracket$ un vecteur de \mathbb{N}^3 , de composantes $d_{ij}, j \in \llbracket 1, p \rrbracket, d_{ij}$ indiquant le nombre d'occurrences du terme T_j dans le document doc_i .

On cherche à voir si des textes traitent des mêmes thématiques, en faisant l'hypothèse que des textes sont sémantiquement proches si des termes communs apparaissent.

1. Recopier et remplir le tableau suivant en appliquant l'algorithme des k -moyennes avec $k = 2$ et d_2 et d_5 comme centres de classe initiaux. On utilisera la distance

$$\delta(d_i, d_j) = \sum_{\ell=1}^p |d_{i\ell} - d_{j\ell}|.$$

On notera de plus c_i les centres de classe et \mathcal{A}_i les classes correspondantes.

itération	c_1	c_2	\mathcal{A}_1	\mathcal{A}_2
1	d_2	d_5		
2				
3				

2. On souhaite maintenant traiter ce problème par une méthode dite "single pass"¹.
 - (a) On propose une seconde méthode de classification, méthode « single pass », qui en une passe (un unique balayage des données) permettant de classer les données en plusieurs classes, ce nombre de classes étant déterminé par la méthode elle-même.

Le principe consiste à :

– initialisation : on choisit le premier objet, qui constitue seul dans un premier temps la première classe, le centroïde étant alors l'objet lui-même.

Ce nombre de classes va éventuellement augmenter au fil de l'avancement de l'algorithme.

– on considère alors les autres objets un par un, successivement, en effectuant les tâches suivantes : on détermine les distances aux centroïdes ; puis

- si **toutes** ces distances sont supérieures à un seuil s fixé au départ, on considère que le nouvel objet fait partie d'une autre classe, que l'on initie avec cet objet, le centroïde associé étant l'objet lui-même.
- sinon on attribue à l'objet la classe dont le centroïde est le plus proche, on ajoute cet objet à cette classe, et l'on met à jour le centroïde associé en prenant en compte donc l'ajout de cet objet (le centroïde va donc se déplacer légèrement).

¹méthode testée pour classer les galaxies, avec 1637 paramètres !

- i. Appliquer l'algorithme single pass avec un seuil $s = 5.0$ aux données de la question 1. La structure algorithmique est aussi donnée dans la question suivante.
- ii. On donne l'algorithme sous la forme algorithmique :

```

*****
Entrées :  $(d_1, \dots, d_n)$  les vecteurs des documents,  $\theta$  un seuil qui appartient à  $\mathbb{R}$ .
Sorties :  $(\mathcal{A}_1, \dots, \mathcal{A}_j)$  les classes de centres  $(c_1, \dots, c_j)$ .
 $c_1 \leftarrow d_1$  // Initialisation
 $\mathcal{A}_1 \leftarrow \{d_1\}$ 
 $j \leftarrow 1$ 
pour  $i$  de 2 à  $n$  faire
  pour  $k$  de 1 à  $j$  faire
    Étape (i) Calculer  $\delta(d_i, c_k)$ 
  fin pour
  si Étape (ii)  $\delta(d_i, c_k) > \theta \forall c_k$  alors
     $j \leftarrow j + 1$  // Création d'une nouvelle classe
     $\mathcal{A}_j \leftarrow \{d_i\}$ 
     $c_j \leftarrow d_i$ 
  sinon
    // Indice du centre de classe le plus proche de  $d_i$  au sens de  $\delta$ 
    Étape (iii)  $l \leftarrow \arg \min_{1 \leq k < j} (\delta(d_i, c_k))$ 
     $\mathcal{A}_l \leftarrow \mathcal{A}_l \cup \{d_i\}$  // Affectation de  $d_i$  à la classe  $l$ 
     $c_l \leftarrow \frac{1}{|\mathcal{A}_l|} \sum_{d_i \in \mathcal{A}_l} d_i$  // Recalcul de  $c_l$ 
  fin si
fin pour
*****

```

Traduire l'algorithme en langage python. On donne la structure à compléter code et commentaires. Certaines optimisations sont effectuées par rapport à l'algorithme donné ci-dessus : le calcul du minimum est effectué en simultané du calcul des distances, et la mise à jour du centroïde est minimaliste. Le renvoi de la liste des classes s'effectue par indices des documents.

```

1 import math as m
2
3 def distance(doc1, doc2):
4     dist = 0
5     for u in range(len(doc1)):
6         dist += abs(doc1[u]-doc2[u])
7     return dist
8
9 def singlepass(docs, s):
10    n = len(docs) # nombre d'objets
11    d = len(docs[0]) # dimension objet
12    classes = [[0]]
13    centroides = [docs[0][:]]
14    j = ? # nombre de classes
15
16    for i in range(1,n):
17        minimum = m.inf
18        for k in range(len(classes)):
19            dk = distance(docs[i], centroides[k])
20            if dk < minimum :
21                minimum = dk
22                indice_min = k
23        if ? :
24            j = j + 1
25            classes.append([i])
26            centroides.append(docs[i][:])

```

```

27     else :
28         classes[indice_min].append(i) # d_i dans la bonne classe
29         # on met à jour le centroïde de la classe indice_min
30         p = len(classes[indice_min])
31         for u in range(?):
32             centroides[indice_min][u] =
33                 (centroides[indice_min][u] * (p-1) + docs[i][u]) / p
34
35     return classes

```

Expliquer le `[:]` des lignes 13 et 26.

Expliquer la ligne 32-33 pour la mise à jour optimisée du centroïde.

Exercice 2 : palindromes dans un mot

En langue française, "ressasser" est le mot palindrome le plus long, tandis qu'il semble que "saippuakauppias" soit le plus long mot palindrome au monde, désignant un marchand de savon en Finlande.

L'objet de cette partie est de compter le nombre de palindromes présents dans un mot donné.

Soit Σ un alphabet fini contenant au moins deux lettres. On note $u = u_0 \cdots u_{n-1}$ un mot sur Σ , composé de n lettres $u_i \in \Sigma$, $i \in \llbracket 0, n-1 \rrbracket$. La longueur de u est notée $|u|$. Pour $0 \leq i < j \leq n$, on note $u[i, j]$ le mot $u_i \cdots u_{j-1}$. L'ensemble des mots construits sur Σ et contenant le mot vide ϵ est noté Σ^* .

Définition 1 (Miroir)

Soit $u \in \Sigma^*$. Le miroir de $u = u_0 \cdots u_{n-1}$, noté \bar{u} , est le mot $\bar{u} = u_{n-1} \cdots u_0$. Par convention $\bar{\epsilon} = \epsilon$.

Définition 2 (Palindrome) $u \in \sigma^*$ est un palindrome si et seulement si $u = \bar{u}$.

Par convention, le mot vide ϵ n'est pas considéré comme un palindrome.

On dira qu'un palindrome u est pair (respectivement impair) lorsque sa longueur $|u|$ est paire (resp. impaire).

On recherche donc dans cette partie le nombre de palindromes facteurs d'un mot $u \in \Sigma^*$ (comptés avec les multiplicités éventuelles), soit le cardinal de l'ensemble $\{(i, j), 0 \leq i < j \leq |u|, u[i, j] = \overline{u[i, j]}\}$. Dit autrement, on recherche le nombre de palindromes contenus dans u .

1. Si $\Sigma = \{a, b\}$, donner le nombre de palindromes contenus dans le mot $u = babb$.
2. On donne l'algorithme naïf²

```

def palindromes(u):
    l = len(u)
    nb = 0
    for i in range(l):
        for j in range(i+1, l+1):
            estPalindrome = True # drapeau
            for k in range(i, j): # test si u[i, j] est un palindrome
                if u[k] != u[j-1+i-k]:
                    estPalindrome = False
                    break
            if estPalindrome:
                nb = nb + 1
    return nb

```

Évaluer la complexité en domination au pire des cas de cet algorithme en fonction de $|u|$.

²pour rappel, en devoir surveillé, il y avait une petite optimisation de la boucle en k

3. On souhaite bien sûr améliorer cette première idée. Pour ce faire, on utilise tout d'abord le paradigme de la programmation dynamique.

Pour $u \in \Sigma^*$, on définit un tableau de booléens P de taille $(|u|+1) \times (|u|+1)$, $P[i][j]$ étant vrai si $u[i, j]$ est un palindrome. On a donc pour tout $i \in \llbracket 0, |u|-1 \rrbracket$, $P[i][i+1] = \text{True}$.

- (a) Soit $u[i, j]$ un mot. À quelles conditions sur u_i , u_{j-1} et $u[i+1, j-1]$ le mot $u[i, j]$ est-il un palindrome ?
- (b) En déduire une relation de récurrence vérifiée par les coefficients de P . Expliquer sur un schéma la dynamique de remplissage du tableau T .
- (c) Écrire une fonction python d'argument un mot u de programmation dynamique qui renvoie le nombres de palindromes contenu dans u . On donne le code incomplet suivant :

```

1 def palindromes_d(u) :
2     n = ?
3     P = [[False]*(n+1) for _ in range(n+1)]
4     # les sous-mots d'une lettre sont des palindromes
5     for i in range(n) :
6         ? = True
7     nb = n
8
9     for d in range(2, n+1) :
10
11         for i in range(n+1-d) :
12
13             P[i][i+d] = (u[i] == u[i+d-1]) and
14                 (P[i+1][i+d-1] or i+1 == i+d-1)
15             nb = nb + ?
16     return nb

```

Compléter le code et expliquer les lignes 3, 9 ainsi que 13-14.

- (d) Évaluer sa complexité en domination selon $|u|$.
- (e) Il existe un algorithme de complexité $\mathcal{O}(n)$, dit algorithme de Manacher, mais c'est une autre histoire. Comparez ces 3 algorithmes avec $n = 10000$ en supposant que chaque unité de complexité est en millionième de seconde.

