# Devoir n°2

## Exercice 1: classification : $k$-moyenne et single pass

On cherche à classer $n = 5$ documents textuels $doc_i$, $i \in [\![1, n]\!]$ 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 $d_i$, $i \in [\![1, n]\!]$ un vecteur de $\mathbb{N}^3$, de composantes $d_{ij}$, $j \in [\![1, p]\!]$, $d_{ij}$ 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, sucessivement, 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 centroide va donc se déplacer légérement).

[1] méthode 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 :** $(d_1, \ldots, d_n)$ les vecteurs des documents, $\theta$ un seuil qui appartient à $\mathbb{R}$.
**Sorties :** $(\mathcal{A}_1, \ldots, \mathcal{A}_j)$ les classes de centres $(c_1, \ldots, c_j)$.

$c_1 \leftarrow d_1$ // 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 $\ell$
    $c_l \leftarrow \frac{1}{|\mathcal{A}_l|} \sum_{d_i \in \mathcal{A}_l} d_i$ // 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 renvoie de la liste des classes s'effectue par indices des documents.

```python
import math as m

def distance(doc1, doc2):
    dist = 0
    for u in range(len(doc1)):
        dist += abs(doc1[u]-doc2[u])
    return dist

def singlepass(docs, s):
    n = len(docs)    # nombre d'objets
    d = len(docs[0]) # dimension objet
    classes = [[0]]
    centroides = [docs[0][:]]
    j = ? # nombre de classes

    for i in range(1,n):
        minimum = m.inf
        for k in range(len(classes)):
            dk = distance(docs[i], centroides[k])
            if dk < minimum :
                minimum = dk
                indice_min = k
        if ? :
            j = j + 1
            classes.append([i])
            centroides.append(docs[i][:])
```

```
27            else :
28                classes[indice_min].append(i) # d_i dans la bonne classe
29                # on met à jour le centroide 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 $\Sigma$ un alphabet fini contenant au moins deux lettres. On note $u = u_0 \cdots u_{n-1}$ un mot sur $\Sigma$, composé de $n$ lettres $u_i \in \Sigma$, $i \in [\![0, n-1]\!]$. La longueur de $u$ est notée $|u|$. Pour $0 \leqslant i < j \leqslant n$, on note $u[i, j]$ le mot $u_i \cdots u_{j-1}$. L'ensemble des mots construits sur $\Sigma$ et contenant le mot vide $\epsilon$ est noté $\Sigma^*$.

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

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

Par convention, le mot vide $\epsilon$ 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 $\{(i, j), 0 \leqslant i < j \leqslant |u|, u[i, j] = \overline{u[i, j]}\}$. 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 naïf [2]

```
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 $|u|$.

[2] pour 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 $i \in [\![0, |u|-1]\!]$, $P[i][i+1] =$ 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 shéma 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.

## Exercice 1: classification : $k$-moyenne et single pass

1. Nous obtenons en respectant l'initialisation :

| itération | $c_1$ | $c_2$ | $\mathcal{A}_1$ | $\mathcal{A}_2$ |
|---|---|---|---|---|
| 1 | $d_2$ | $d_5$ | $\{d_2, d_3\}$ | $\{d_1, d_4, d_5\}$ |
| 2 | $(1, 1, 0)$ | $(1, 2, 5/3)$ | $\{d_2, d_3, d_5\}$ | $\{d_1, d_4\}$ |
| 3 | $(1, 2/3, 1/3)$ | $(1, 3, 2)$ | $\{d_2, d_3, d_5\}$ | $\{d_1, d_4\}$ |

2. On souhaite maintenant traiter ce problème par une méthode dite "single pass" [3].

(a) i. On appliquer l'algorithme single pass avec un seuil $s = 5.0$ aux données de la question 1 : $c_1 = d_1 = (1, 3, 3)$, $\mathcal{A}_1 = \{d_1\}$, $j = 1$ : une classe.
• On regarde $d_2 = (2, 1, 0)$, sa distance avec le (seul pour l'instant) centroïde $c_1$ : $d(d_2, c_1) = 1+2+3 = 6 > s$. $d_2$ est loin des centroïdes existant, nous créons donc une nouvelle classe, $\mathcal{A}_2 = \{d_1\}$, $c_2 = d_2$, $j = 2$.
• On regarde $d_3 = (0, 1, 0)$, sa distance avec les centroïdes

$$d(d_3, c_1) = 6 \qquad d(d_3, c_2) = 2$$

La condition d'éloignement des centroïdes n'est pas réalisée, la classe la plus proche est la classe 2, ainsi $\mathcal{A}_2 = \{d_2, d_3\}$ et la mise à jour du centroïde 2 est $c_2 = (1, 1, 0)$.
• On regarde $d_4 = (1, 3, 1)$, sa distance avec les centroïdes

$$d(d_4, c_1) = 2 \qquad d(d_4, c_2) = 3$$

La condition d'éloignement des centroïdes n'est pas réalisée, la classe la plus proche est la classe 1, ainsi $\mathcal{A}_1 = \{d_1, d_4\}$ et la mise à jour du centroïde 1 est $c_1 = (1, 3, 2)$.

• On regarde pour finir $d_5 = (1, 0, 1)$, sa distance avec les centroïdes

$$d(d_5, c_1) = 4 \qquad d(d_5, c_2) = 2$$

La condition d'éloignement des centroïdes n'est pas réalisée, la classe la plus proche est la classe 2, ainsi $\mathcal{A}_2 = \{d_2, d_3, d_5\}$ et la mise à jour du centroïde 2 est $c_2 = (1, 2/3, 1/3)$.
Nous obtenons ainsi deux classes, avec $\mathcal{A}_1 = \{d_1, d_4\}$ et $\mathcal{A}_2 = \{d_2, d_3, d_5\}$.

ii. On traduit l'algorithme en langage python. 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 renvoie de la liste des classes s'effectue par indices des documents.

```python
import math as m

def distance(doc1, doc2):
    dist = 0
    for u in range(len(doc1)):
        dist += abs(doc1[u]-doc2[u])
    return dist

def singlepass(docs, s):
    n = len(docs)    # nombre d'objets
    d = len(docs[0]) # dimension objet
    classes = [[0]]
    centroides = [docs[0][:]]
    j = 1  # nombre de classes

```

[3] méthode testée pour classer les galaxies, avec 1637 paramètres !

```
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 minimum > s :
24              j = j + 1 # création d'une nouvelle classe
25              classes.append([i]) # on y met le document i
26              centroides.append(docs[i][:]) # on définit cj
27          else :
28              classes[indice_min].append(i) # d_i dans la bonne classe
29              # on met à jour le centroide de la classe indice_min
30              p = len(classes[indice_min])
31              for u in range(d): # nombre de coordonnées
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 : nécessaire pour une copie indépendantes, sinon les documents seront éventuellement modifiés invonlontairement.

Expliquer la ligne 32-33 pour la mise à jour optimisée du centroïde : on se sert de la moyenne précédente : par exemple, si on connait la moyenne de $p = 3$ termes $a, b, c, m = \frac{a+b+c}{3}$, que l'on rajoute une valeur, la nouvelle moyenne est

$$m' = \frac{(a+b+c)+d}{4} = \frac{3m+d}{4}$$

ce qui permet d'économiser quelques calculs.

## Exercice 2 : palindromes dans un mot

1. Nous avons 4 palindromes d'une lettre, 1 de 2 lettres *bb*, 1 de 3 lettres : *bab*, soit au total 6 palindromes.

2. On donne l'algorithme naïf :

```
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
```

Nous avons trois boucles imbriquées dépendantes. Le calcul précis est long est compliqué). Nous majorons grossièrement (ce qui ne change en fait pas l'ordre de grandeur au final mais il faudrait faire ce calcul pour le comprendre vraiment, mais par exemple on rapelle que $1+2\cdots+n=\frac{n(n+1)}{2}\sim\frac{n^2}{2}$ de complexité en $n^2$ en considérant que les trois boucles imbriquées sont de taille $|u|$.
Le corps interne de la triple boucle est en temps constant, nous obtenons une complexité en $|u|\times|u|\times|u|=|u|^3$.
De manière plus précise, il faudrait calculer $\sum_{i=0}^{|u|-1}\sum_{j=i+1}^{|u|}(j-i)$, qui donnera la même chose en complexité.

3. (a) Soit $u[i,j]$ un mot. Le mot $u[i,j]$ est un palindrome si $u_i$ et $u_{j-1}$ sont la même lettre et que soit $u[i+1,j-1]$ est vide (le vide n'est pas un palindrome avec la convention) soit qu'il s'agit d'un palindrome.

   (b) On a donc

$$P[i][j]=\begin{cases}\text{True} & \text{si } u_i=u_{j-1} \text{ et } (u[i+1,j-1] \text{ vide ou } P[i+1][j-1]=\text{True})\\ \text{False} & \text{sinon}\end{cases}$$

On en déduit la dynamique de remplissage du tableau $P$ par diagonales en remontant :
- $P[i][i+1]$ mis à True, $i$ de 0 à $n-1$.
- $P[i][i+2]$, $i$ de 0 à $n-2$ en respectant la formule ci-dessus
- $P[i][i+3]$, $i$ de 0 à $n-3$ en respectant la formule ci-dessus

$\vdots$
- $P[0][n-1]$ et $P[1][n]$
- $P[0][n]$ en dernier.

On peut appliquer cela à l'exemple *babb* (diagonale des rouges, puis des bleus, puis des verts)

| $i$ \ $j$ | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | F | T | F | T | F |
| 1 | F | F | T | F | F |
| 2 | F | F | F | T | T |
| 3 | F | F | F | F | T |

   (c) On complète le code :

```
def palindromes_d(u):
    n = len(u)
    P = [[False]*(n+1) for _ in range(n+1)]
    # les sous-mots d'une lettre sont des palindromes
    for i in range(n):
```

```
6           P[i][i+1] = 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 + P[i][i+d]
16      return nb
```

ligne 2 : $n$ est la longeur du mot $u$.
ligne 3 : on créé un tableau de taille $(n+1) \times (n+1)$ de booléens initiés à FALSE.
ligne 6 : $P[i][i+1]$ est mis à vrai puisque $u[i, i+1]$ est un mot d'une lettre, donc un palindrome.
ligne 9 : la variable $d$ symbolise le numéro de diagonale, la vraie diagonale étant de numéro 0. Nous avons $n-1 = n-2+1$ diagonales à remplir selon le principe décrit dans la question b).
ligne 13-14 : pour avoir un palindrome, les lettres des extrémités doivent être égales (condition $u[i] = u[i+d-1]$) et $(u[i+1, i+d-1]$ doit être un palindrome ou vide, à ne pas oublier parce que vide n'est pas considéré comme un palindrome).
Les palindromes étant comptés petit à petit à l'aide la de variable $nb$, nous avons le total de palindromes à la fin du remplissage des diagonales dans la variable $nb$.

(d) On considère que le corps interne de la double boucle est en temps constant, la double boucle imbriquée balaye les diagonales, soit une complexité en tenant compte de la première boucle for et du remplissage initial du tableau de l'ordre de

$$|u| + |u|^2 + 1 + 2 + \cdots |u| = |u| + |u|^2 + |u|\frac{|u|+1}{2} = \mathcal{O}(|u|^2)$$

(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.

Algorithme 1 : en $n^3$, $n^3 = 10^{12}$ unités de complexité, soit $10^6$ secondes, environ 12 jours.
Algorithme 2 : en $n^2$, $n^2 = 10^8$ unités de complexité, soit 102 secondes, soit environ 1 minutes et demi.
Algorithme 3 : en $n$, $n = 10^4$ unités de complexité, soit $10^{-2}$ secondes, soit un centième de seconde.