Université des Sciences et de la Technologie Houari Boumediene

Faculté d'Electronique et d'Informatique
Département Informatique

Clustering des séquences d'ADN

FOUILLE DE DONNEÉS

Foudili Sirine
BIO-INFORMATIQUE | USTHB 2018-2019

Sommaire

I. Introduction générale :

La classification est un des nombreux domaines de la Fouille de données qui vise à extraire l'information à partir de grands volumes de données en utilisant différentes techniques computationnelles de l'apprentissage, des statistiques et des reconnaissances des formes. On cite les deux méthodes principale supervisée et non supervisée.

La classification non supervisée désigne un corpus de méthodes ayant pour objectif de dresser ou de retrouver une typologie existante caractérisant un ensemble de n observations, à partir de p caractéristiques mesurées sur chacune des observations. Par typologie, on entend que les observations, bien que collectées lors d'une même expérience, ne sont pas toutes issues de la même population homogène, mais plutôt de K populations.

L'appartenance des observations à l'une des K populations n'est pas connue. C'est justement cette appartenance qu'il s'agit de retrouver à partir des p descripteurs disponibles.

Il existe de très nombreuses méthodes de classification non supervisées, seule une sélection est décrite ci-dessous. Cette sélection est opérée en visant des méthodes fréquemment utilisées et appartenant à des types d'algorithmes différents donc complémentaires.

II. Généralités :

II.1.Tâches et techniques :

Deux techniques sont utilisées en classification non supervisé, les techniques hiérarchiques et les techniques de partitionnement.

Les techniques hiérarchiques :

Pour un niveau de précision donné, deux individus peuvent être confondus dans un même groupe, alors qu'à un niveau de précision plus élevé, ils seront distingués et appartiendront à deux sous-groupes différents.

Le résultat d'une classification hiérarchique n'est pas une partition de l'ensemble des individus. C'est une hiérarchie de classes telle que toute classe est non vide et elle est la réunion des classes qui sont incluse dans elle.

L'avantage de cette méthode est qu'elle n'est soumise à aucune initialisation particulière de paramètres ce qui la rend déterministe.

Les techniques de partitionnement :

Aboutissent à la décomposition de l'ensemble de tous les individus en K ensemble disjoints ou classes d'équivalence. Le nombre K de classes est fixé. Le résultat obtenu est alors une partition de l'ensemble des individus, un ensemble de parties, ou classes de l'ensemble I des individus.

II.2.Domaines d'application :

La classification non supervisé est appliquée dans plusieurs domaines, citant par exemple :

-La reconnaissance de formes.

- L'analyse des données spatiales.

-Le traitement d'images.

-Les market Research.

-La recherche d'information :

- Catégorisation de documents ou de termes.

-Web Mining.

Dans le domaine de la bio-informatique le clustering est utilisé pour :

- Identifier les espèces proches.

-Créer un arbre généalogique.

-Regrouper deux par deux les animaux les plus proches.

III. Mesure de similarité :

Afin de définir l'homogénéité d'un groupe d'observations, il est nécessaire de mesurer une ressemblance entre deux observations. On introduit ainsi les notions de dissimilarité et de similarité.

Le choix de la distance est une question primordiale pour les méthodes exploratoires multi variées. En effet, c'est à cette étape qu'il est possible pour l'expérimentateur d'utiliser au mieux l'information a priori dont il dispose, afin de proposer une mesure pertinente de ressemblance entre observations.

Puisque dans ce mini-projet nous allons traiter des séquences d'ADN la mesure de similarité choisie est la distance de Levenshtein.

La distance de Levenshtein :

Repose sur trois opérations d'édition élémentaires qui affectent un caractère à la fois : la substitution, l'insertion et la délétion (suppression).

— substitution : remplacement d'un caractère par un autre.

— insertion : ajout d'un caractère.

— délétion : suppression d'un caractère.

L'application de chacune de ces trois opérations correspond à une erreur entre deux mots.

Définition. Soient U et V deux mots de $\Sigma*$. La distance de Levenshtein entre U et V, notée Lev(U,V), est le nombre minimal d'opérations d'édition nécessaires pour transformer U en V. La suite d'opérations appliquées s'appelle un script d'édition entre U et V.

IV. Les chaines centrales :

Dans les méthodes que nous allons présenter nous avons utilisé deux méthodes pour représenter les chaines centrales :

-La moyenne.

-Le calcul des médoïdes.

La moyenne :

Afin de calculer la distance entre deux classes différentes on fait appel à leurs moyennes (mean).

Celle-ci est composé des caractères plus fréquents du cluster, sa taille sera égale à la taille de la chaine la plus longue de la classe.

Les Medoïds :

Un medoïd est l'objet d'un cluster pour lequel la distance moyenne à tous les autres objets du cluster est minimale l'avantage de son utilisation est qu'il fait partie des séquences du cluster.

V. Implémentation du code :

V.1. K-Means :

Le partitionnement en *k*-moyennes est une méthode de partitionnement de données et un problème d'optimisation combinatoire. Étant donnés des points et un entier *k*, le problème est de diviser les points en *k* groupes, souvent appelés *clusters*, de façon à minimiser une certaine fonction. On considère la distance d'un point à la moyenne des points de son cluster.

Analyse :

La fonction principale a comme entrées un fichier f contenant les séquences d'ADN et un nombre entier k représentant le nombre de classes dont l'utilisateur souhaite classifier ses séquences.

Tout au début, à l'itération 0, un centre est choisi arbitrairement du fichier et le clustering démarre en calculant les distances entre ce dernier et les autres séquences.

A partir du premier cluster obtenu on calcule le centroid et on refait le clustering jusqu'à ce qu'on obtient un même cluster à partir de deux itérations it - 1 et it.

Code :

```
201
202     def k_means(self,f,k):
203         print("="*50,'\n',' '*20,'K-Means',' '*20,'\n',"="*50)
204         global it
205         print('\n itération: 0 \n')
206         ctr=self.centre(f,k)
207         print('centre: ',ctr,'\n')
208         clust1=self.clust(f,ctr,k)
209         print('cluster: ',clust1,'\n\n')
210         ctrd=self.centroid(clust1,f)
211         it=0
212         while(self.comp(ctr,ctrd,k)==0):
213             print('iteration: ',it+1,'\n')
214             ctr=ctrd
215             print('centre: ',ctr,'\n')
216             a=self.clust(f,ctrd,k)
217             print('cluster: ',a,'\n\n')
218             ctrd=self.centroid(a,f)
219             it=it+1
220         print(it)
221         return self.clust(f,ctrd,k)
```

Test :

```
K-means :
Fichier :

{0: 'CC-G--CTGCAT', 1: 'GAGGT--GAAGC', 2: 'GGGCTGCGTT--', 3: 'GAGGT-CAGCC-', 4:
'CCAGACTGGAT-'}
0
1
Cluster {0: [], 1: [0, 1, 2, 3, 4]}
```

La structure de données représentant le cluster est un dictionnaire.

La clé représente le numéro de la classe, et la liste des numéros réfère aux numéros de lignes de chaque séquence.

Exemple :

Pour un fichier contenant 4 lignes, (cluster[1])[4] représente la séquence situé dans la 4ème ligne du fichier, classifié en cluster 1.

Dans notre exemple c'est la séquence 'CCAGACTGGAT-'.

Traitement du fichier :

La représentation des séquences d'ADN par numéro de ligne sert à simplifier la compréhension des clusters surtout quand on a à faire à des fichiers volumineux avec des centaines de séquences.

Pour le faire, avant chaque clustering le fichier est transformé en un dictionnaire où la clé représente le numéro de la ligne et la valeur représente la séquence correspondante.

Code :

```
52      def tri_ADN(self,l):
53          c=''
54          for i in range(39,len(l)):
55              if(l[i] in {'A','C','G','T','-'}):
56                  c=c+l[i]
57          return c
58
59      def trans_fichier(self,fichier):
60          f=open(fichier,'r')
61          i=0
62          seq={}
63          for ligne in f:
64              ligne=f.readline()
65              s=self.tri_ADN(ligne)
66              seq[i]=s
67              i=i+1
68          f.close()
69          return seq
70
```

Test :

Avant :

```
1   EI,    HUMCFVII-DONOR-10218,          AGAACTGGAGGAACCTGATCGCGGTGCTGGGTGGGTACCACTCTCCCCTGTCCGACCGCG
2   EI,    HUMCKMT-DONOR-523,             CAGTGAACGACGGAGGCTGTATCCCCCGAGGTAACAGTGCCTGAGGCGCGGGAGGAGGCG
3   EI,    HUMCKMT-DONOR-1326,            ATGGTGGCTGGAGATGAGGAGACCTATGAGGTAGGGGGTCCCCAGAGTCTCCCTGATGAT
4   EI,    HUMCKMT-DONOR-1625,            AAGCACACCACGGATCTAGATGCCAGTAAAGTGAGTTCAAATATCCCACTTCTGATTTGC
```

Après :

```
Après :
{0: 'CAGTGAACGACGGAGGCTGTATCCCCCGAGGTAACAGTGCCTGAGGCGCGGGAGGAGGCG', 1: 'AAGCACA
CCACGGATCTAGATGCCAGTAAAGTGAGTTCAAATATCCCACTTCTGATTTGC', 2: 'TCGAGACTGGCCAGATGCTC
GTGGAATTTGGTATGAAGCTGCTCATTACCTCTTTTGTCT', 3: 'GTGCACATCAAACTGCCCCTGCTAAGCAAAGTA
AAGGAGTTGTGGGGTTACAGAGGGGTG'}
```

Sélection du centre :

Le centre est aussi représenté sous forme de dictionnaire, le nombre d'items est selon le nombre de clusters k.

Pour k=2, le centre contiendra 2 séquences choisies aléatoirement.

La fonction 'existe' garantie que chaque classe aura un centre unique différent des centres choisis pour les autres classes.

Code :

```
109
110         def existe(self,dic,seq):
111             for valeur in dic.values():
112                 if valeur==seq:
113                     return 1
114                 else:
115                     return 0
116
117         def centre(self,f,k):
118             centre={}
119             for i in range(k):
120                 key = random.choice(list(f))
121                 j=0
122                 while(self.existe(centre,f[key])==1):
123                     key = random.choice(list(f))
124                     j=j+1
125                 centre[i]=f[key]
126             return centre
```

Test :

```
Fichier :
{0: 'CAGTGAACGACGGAGGCTGTATCCCCCGAGGTAACAGTGCCTGAGGCGCGGGAGGAGGCG', 1: 'AAGCACA
CCACGGATCTAGATGCCAGTAAAGTGAGTTCAAATATCCCACTTCTGATTTGC', 2: 'TCGAGACTGGCCAGATGCTC
GTGGAATTTGGTATGAAGCTGCTCATTACCTCTTTTGTCT', 3: 'GTGCACATCAAACTGCCCCTGCTAAGCAAAGTA
AAGGAGTTGTGGGGTTACAGAGGGGTG'}
Centre :
{0: 'CAGTGAACGACGGAGGCTGTATCCCCCGAGGTAACAGTGCCTGAGGCGCGGGAGGAGGCG', 1: 'AAGCACA
CCACGGATCTAGATGCCAGTAAAGTGAGTTCAAATATCCCACTTCTGATTTGC'}
```

Le clustering :

A partir des centres chaque séquence sera dans la classe contenant le centre le plus proche.

Code :

```
183     # Indice du centre le plus proche à l'adn (l'indice du cluster)
184         def plus_proche(self,adn,centres):
185             p=0
186             m=self.levenshtein(adn,centres[0],0,1,1)
187             for i in range(1,len(centres)):
188                 if self.levenshtein(adn,centres[i],0,1,1)<=m:
189                     p=i
190                     m=self.levenshtein(adn,centres[i],0,1,1)
191             return p
192
193         def clust(self,f,centre,k):
194             c={}
195             for i in range(k):
196                 c[i]=[]
197             for i in range(len(f)):
198                 p=self.plus_proche(f[i],centre)
199                 c[p].append(i)
200             return c
201
```

Test :

```
fichier :
 {0: 'CAGTGAACGACGGAGGCTGTATCCCCCGAGGTAACAGTGCCTGAGGCGCGGGAGGAGGCG', 1: 'AAGCACA
CCACGGATCTAGATGCCAGTAAAGTGAGTTCAAATATCCCACTTCTGATTTGC', 2: 'TCGAGACTGGCCAGATGCTC
GTGGAATTTGGTATGAAGCTGCTCATTACCTCTTTTGTCT', 3: 'GTGCACATCAAACTGCCCCTGCTAAGCAAAGTA
AAGGAGTTGTGGGGTTACAGAGGGGTG'}
 Centre :
 {0: 'TCGAGACTGGCCAGATGCTCGTGGAATTTGGTATGAAGCTGCTCATTACCTCTTTTGTCT', 1: 'AAGCACA
CCACGGATCTAGATGCCAGTAAAGTGAGTTCAAATATCCCACTTCTGATTTGC'}
 Cluster :
 {0: [2], 1: [0, 1, 3]}
```

Centroid :

Le calcul de centroid se fait en calculant la moyenne de chaque cluster.

Dans notre cas où la classification est faite sur des chaines de caractères, la moyenne contient les caractères les plus fréquents entre les séquences du cluster.

Code :

```
135         def car_frequent(self,l):
136                 a=l.count('A')
137                 c=l.count('C')
138                 g=l.count('G')
139                 t=l.count('T')
140                 gap=l.count('-')
141                 m=max(a,c,g,t,gap)
142                 if(m==a):
143                         return 'A'
144                 if(m==c):
145                         return 'C'
146                 if(m==g):
147                         return 'G'
148                 if(m==t):
149                         return 'T'
150                 if(m==gap):
151                         return '-'
152
153         def mean(self,l):
154                 mn=''
155                 for p in range(len(self.longest(l))):
156                         c=''
157                         for i in range(len(l)):
158                                 if(p<=len(l[i])-1):
159                                         c=c+(l[i])[p]
160                         mn=mn+self.car_frequent(c)
161                 m=mn
162                 return m
163         def liste_adn(self,l,f):
164                 n=[]
165                 for i in range(len(l)):
166                         n.append(f[l[i]])
167                 return n
168
169         def centroid(self,c,f):
170                 centre={}
171                 for i in range(len(c)):
172                         centre[i]=self.mean(self.liste_adn(c[i],f))
173                 return centre
```

Test :

```
fichier :
 {0: 'CAGTGAACGACGGAGGCTGTATCCCCCGAGGTAACAGTGCCTGAGGCGCGGGAGGAGGCG', 1: 'AAGCACA
CCACGGATCTAGATGCCAGTAAAGTGAGTTCAAATATCCCACTTCTGATTTGC', 2: 'TCGAGACTGGCCAGATGCTC
GTGGAATTTGGTATGAAGCTGCTCATTACCTCTTTTGTCT', 3: 'GTGCACATCAAACTGCCCCTGCTAAGCAAAGTA
AAGGAGTTGTGGGGTTACAGAGGGGTG'}
 Cluster :
 {0: [1, 2], 1: [0, 3]}
 Centroid :
 {0: 'AAGAAAACCACCAAACGAGAGGCCAATAAAGTAAGAACAAACACACCACCTCTGATGTCC', 1: 'CAGCAAA
CCAAACAGCCCCTACCAACCAAAGTAAAAGAGCCGGAGGCGCACAAAGAGGCG'}

C:\Users\sirine\Desktop\Bio Info\S2\Data mining\K-means>
```

V.2.K-Medoïd :

Cet algorithme permet de classer des données de façon plus robuste, c'est-à-dire moins sensible à des valeurs atypiques. Le noyau d'une classe est alors un médoïd. C'est-à-dire l'observation d'une classe qui minimise la moyenne des distances ou dissimilarités aux autres observations de la classe. Une différence majeur avec l'algorithme k-means est qu'un médoïd fait partie des données et permet donc de partitionner des matrices de dissimilarités.

Analyse :

L'initialisation de la fonction est la même que celle de la méthode K-Means, car même dans cette dernière le centre existe réellement dans le fichier (ce n'est pas calculé).

Après avoir calculé le premier cluster, on cherche les medoïd les plus proches aux séquences du cluster.

On refait cette étape jusqu'à arriver à une itération it où les clusters trouvés sont similaires aux clusters de l'itération it-1.

Code :

```
252
253         def k_medoid(self,f,k):
254                 print("="*50,'\n',' '*20,'K-Medoïd',' '*20,'\n',"="*50)
255                 global it
256                 ctr=self.centre(f,k)
257                 clust1=self.clust(f,ctr,k)
258                 print('centre: ',ctr,'\n')
259                 clust1=self.clust(f,ctr,k)
260                 print('cluster: ',clust1,'\n\n')
261                 ctrd=self.medoid(clust1,f)
262                 it=0
263                 while(self.comp(ctr,ctrd,k)==0):
264                         print('itération:',it)
265                         ctr=ctrd
266                         print('centre: ',ctr,'\n')
267                         a=self.clust(f,ctrd,k)
268                         print('cluster: ',a,'\n\n')
269                         ctrd=self.medoid(a,f)
270                         it=it+1
271                 print(it)
272                 return self.clust(f,ctrd,k)
273
```

Test : pour k=2

```
fichier :
 {0: 'CAGTGAACGACGGAGGCTGTATCCCCCGAGGTAACAGTGCCTGAGGCGCGGGAGGAGGCG', 1: 'AAGCACA
CCACGGATCTAGATGCCAGTAAAGTGAGTTCAAATATCCCACTTCTGATTTGC', 2: 'TCGAGACTGGCCAGATGCTC
GTGGAATTTGGTATGAAGCTGCTCATTACCTCTTTTGTCT', 3: 'GTGCACATCAAACTGCCCCTGCTAAGCAAAGTA
AAGGAGTTGTGGGGTTACAGAGGGGTG'>
 Cluster :
 {0: [0], 1: [1, 2, 3]>
```

Calcul des Medoïds :

Pour chaque classe on établit une matrice de distance de laquelle on choisira la séquence la plus proche aux autres séquences de la classe.

Celle-ci sera le medoïd K[i] de la classe i.

Code :

```
230
231         def medo(self,l,f):
232                 distance={}
233                 d=0
234                 c=self.liste_adn(l,f)
235                 for i in range(len(l)):
236                         for j in range(len(l)):
237                                 d=d+self.levenshtein(c[i],c[j],0,1,1)
238                         distance[l[i]]=d
239                 m=min(distance.values())
240                 for c,v in distance.items():
241                         if v==m:
242                                 medo=c
243                 return f[medo]
244
245         def medoid(self,c,f):
246                 med={}
247                 for i in range(len(c)):
248                         med[i]=self.medo(c[i],f)
249                 return med
250
```

Test :

```
fichier :
 {0: 'CAGTGAACGACGGAGGCTGTATCCCCCGAGGTAACAGTGCCTGAGGCGCGGGAGGAGGCG', 1: 'AAGCACA
CCACGGATCTAGATGCCAGTAAAGTGAGTTCAAATATCCCACTTCTGATTTGC', 2: 'TCGAGACTGGCCAGATGCTC
GTGGAATTTGGTATGAAGCTGCTCATTACCTCTTTTGTCT', 3: 'GTGCACATCAAACTGCCCCTGCTAAGCAAAGTA
AAGGAGTTGTGGGGTTACAGAGGGGTG'}
 Cluster :
 {0: [0], 1: [1, 2, 3]}
 Medoid :
 {0: 'CAGTGAACGACGGAGGCTGTATCCCCCGAGGTAACAGTGCCTGAGGCGCGGGAGGAGGCG', 1: 'AAGCACA
CCACGGATCTAGATGCCAGTAAAGTGAGTTCAAATATCCCACTTCTGATTTGC'}
```

V.3.DBSCAN :

Le principe de DBSCAN (Density-based spatial clustering of applications with noise) repose sur la notion de ε-voisinage d'un individu ou point défini comme l'ensemble des points appartenant à la boule de rayon centrée sur ce point. En plus du rayon ε, un autre paramètre est considéré : MinPts qui précise un nombre minimum de points à prendre en compte dans cette boule. L'ensemble des points ou individus se répartit en trois catégories :

Les cœurs (core points), les points atteignables (reachable), les points atypiques (ouliers).

Analyse :

Les paramètres d'entrées :

Un fichier f, un rayon eps définissant la distance minimale entre un point et son voisinage, le nombre de points minpts qui nous permettrons de faire la différence entre un point cœur et les autres points.

Tant que la séquence n'appartient à aucun cluster on la classifie.

Si c'est un point cœur on la classifie dans un cluster avec ses voisins, sinon la séquence est considérée comme du bruit (noise).

Code :

```
306
307         def dbScan(self,f,eps,minpts):
308                 print("="*50,'\n',' '*20,'DBSCAN',' '*20,'\n',"="*50)
309                 global it
310                 it=0
311                 clus={}
312                 noise=[]
313                 for point in f.keys():
314                         print('itération: ',it,'\n')
315                         if self.classifie(clus,point)==0:
316                                 if self.coeur(f,f[point],eps,minpts-1)==1:
317                                         print('Coeur: ',f[point],'\n')
318                                         c=self.voisinage(f,f[point],eps)
319                                         clus[it]=c
320                                         print('Voisinage: ',c,'\n')
321                                         it=it+1
322                                 else:
323                                         noise.append(point)
324                 print('noise: ',noise,'\n')
325                 return clus
326
```

Test :

```
fichier :
 {0: 'CC-G--CTGCAT', 1: 'GAGGT--GAAGC', 2: 'GGGCTGCGTT--', 3: 'GAGGT-CAGCC-', 4:
 'CCAGACTGGAT-'}
 Cluster :
 {0: [0, 1, 2, 3, 4]}
```

Définition des points cœurs et des voisins :

Un point cœur est celui qui a au moins minpts voisins.

Les voisins d'un cœur sont les points loin de lui d'une distance eps.

Code :

```
283
284     #Le point fera partie du voisinage
285             def voisinage(self,f,point,eps):
286                     voisins=[]
287                     for i in range(len(f)):
288                             if self.levenshtein(point,f[i],0,1,1)<=eps:
289                                     voisins.append(i)
290                     return voisins
291
292             def coeur(self,f,point,eps,minpts):
293                     if len(self.voisinage(f,point,eps))>=minpts:
294                             return 1
295                     else:
296                             return 0
297
```

Test :

```
coeur :
CC-G--CTGCAT
son voisinage :
[0, 3, 4]


coeur :
GAGGT--GAAGC
son voisinage :
[1, 3]


coeur :
GGGCTGCGTT--
son voisinage :
[2, 3, 4]


ichier :
{0: 'CC-G--CTGCAT', 1: 'GAGGT--GAAGC', 2: 'GGGCTGCGTT--', 3: 'GAGGT-CAGCC-', 4:
'CCAGACTGGAT-'}
```

V.4.Agnes :

Il s'agit de regrouper itérativement les individus, en commençant par le bas (les deux plus proches) et en construisant progressivement un arbre, ou dendrogramme, regroupant finalement tous les individus en une seule classe, à la racine.

Analyse :

En itération 0 le cluster est initialisé à N classes, chaque classe contenant une seule séquence. Où N est le nombre de séquences à classifier.

Ensuite, Dans chaque itération on choisit les 2 clusters les plus similaires et on les rassemble dans une seule classe.

On retraite la même procédure jusqu'à arriver à une seule classe.

Pour bien visualiser chaque itération j'ai représenté le cluster en dictionnaire de dictionnaire.

Les clés du dictionnaire global représentent le numéro d'itération et chaque itération a son propre cluster constitué de plusieurs classes.

Code :

```
388
389         def agnes(self,f):
390                 global it
391                 it=0
392                 c={}
393                 clust={}
394                 tmp=dict(f)
395                 for i in range(len(tmp)):
396                         c[i]=i
397                         clust[it]=dict(c)
398                 it=it+1
399                 i=0
400                 while len(c)>1 :
401                         m=self.minimum(self.distance(tmp,f))
402                         self.remplacer(m,tmp,f)
403                         self.mis_a_jour(c,m)
404                         clust[it]=dict(c)
405                         i=i+1
406                         it=it+1
407                 return clust
```

Test :

```
fichier :
{0: 'CC-G--CTGCAT', 1: 'GAGGT--GAAGC', 2: 'GGGCTGCGTT--', 3: 'GAGGT-CAGCC-', 4:
'CCAGACTGGAT-'}
Cluster :
{0: {0: 0, 1: 1, 2: 2, 3: 3, 4: 4}, 1: {1: 1, 2: 2, 3: 3, 4: [4, 0]}, 2: {2: 2,
3: [3, 1], 4: [4, 0]}, 3: {3: [3, 1], 4: [4, 0, 2]}, 4: {4: [4, 0, 2, 3, 1]}}
```

Le remplacement des séquences :

A chaque itération on remplace les séquences fusionnées dans une même classe par leur moyenne dans un fichier temporaire pour permettre le calcul de la nouvelle distance.

Code :

```
361
362         def remplacer(self,liste,tmp,f):
363                 tmp[liste[0]]=self.mean(self.liste_adn(liste,f))
364                 del tmp[liste[1]]
365                 return tmp
366
```

Test :

```
Ancien fichier :
{0: 'CC-G--CTGCAT', 1: 'GAGGT--GAAGC', 2: 'GGGCTGCGTT--', 3: 'GAGGT-CAGCC-', 4:
'CCAGACTGGAT-'}
Aprés remplacement :
{0: 'CC-G--CTGCAT', 1: 'GAGGT--GAAGC', 2: 'GGGCTGCGTT--', 3: 'CAAGACCAGAC-'}
```

Mise à jour :

Après avoir repérer les 2 classes les plus similaires on met à jour le cluster en les mettant dans une même classe avec la suppression de leurs anciennes classes.

Code :

```
def mis_a_jour(self,c,liste):
        if(isinstance(c[liste[0]],int)==True) and ( isinstance(c[liste[1]],int)==True ):
                l=[]
                l.append(c[liste[0]])
                l.append(c[liste[1]])
        else:
                if(isinstance(c[liste[0]],int)==False) and ( isinstance(c[liste[1]],int)==False):
                        l=c[liste[0]]+c[liste[1]]
                if(isinstance(c[liste[0]],int)==True):
                        t=[]
                        t.append(c[liste[0]])
                        l=c[liste[1]]+t
                if(isinstance(c[liste[1]],int)==True):
                        t=[]
                        t.append(c[liste[1]])
                        l=c[liste[0]]+t
        c[liste[0]]=l
        del c[liste[1]]
        return c
```

Test :

```
avant:  {0: [1, 2], 1: [5, 6], 2: 19, 3: [4, 0]}
aprés:  {0: [1, 2], 1: [5, 6], 2: [4, 0, 19]}
```

VI. Tests et résultats :

VI.1. Bibliothèques :

Afin de réaliser certaines fonctions, l'utilisation de ces bibliothèques était nécessaire.

```
11
12  import random
13  from random import randint
14  import sys
15  from re import *
16  from PyQt5 import *
17  from PyQt5.Qt import *
18  from PyQt5.QtWidgets import *
19  from PyQt5.QtGui import *
20  from PyQt5.QtCore import *
21  from PyQt5.QtGui import QImage
22  from PyQt5.QtGui import QIcon
23  import matplotlib.pyplot as plt
24  import pandas as pd
25  import numpy as np
26  from time import *
```

Elles peuvent être regroupées en 3 groupes selon leur utilisation.

Les bibliothèques utilisées pour l'interface graphique :

PyQt5, PyQt5.Qt, PyQt5.QtWidgets, PyQt5.QtGui, PyQt5.QtCore, QImage, QIcon.

Les bibliothèques utilisées pour le traçage des graphes :

matplotlib.pyplot, pandas, numpy.

Les bibliothèques utilisées pour calculer le temps d'exécution :

Time, sys.

Les bibliothèques utilisées pour la sélection aléatoire :

Random, randint.

VI.2.Environnement de travail :

L'environnement de travail dans lequel ce mini-projet a été réalisé :

-RAM : 4 GO
-Système d'exploitation : Windows 8.1 x64bits.
- Langage de programmation : Python (Python v 3.5)
Environnement de développement :
-Editeur de texte : Sublime Text.
-Exécution : Terminal.

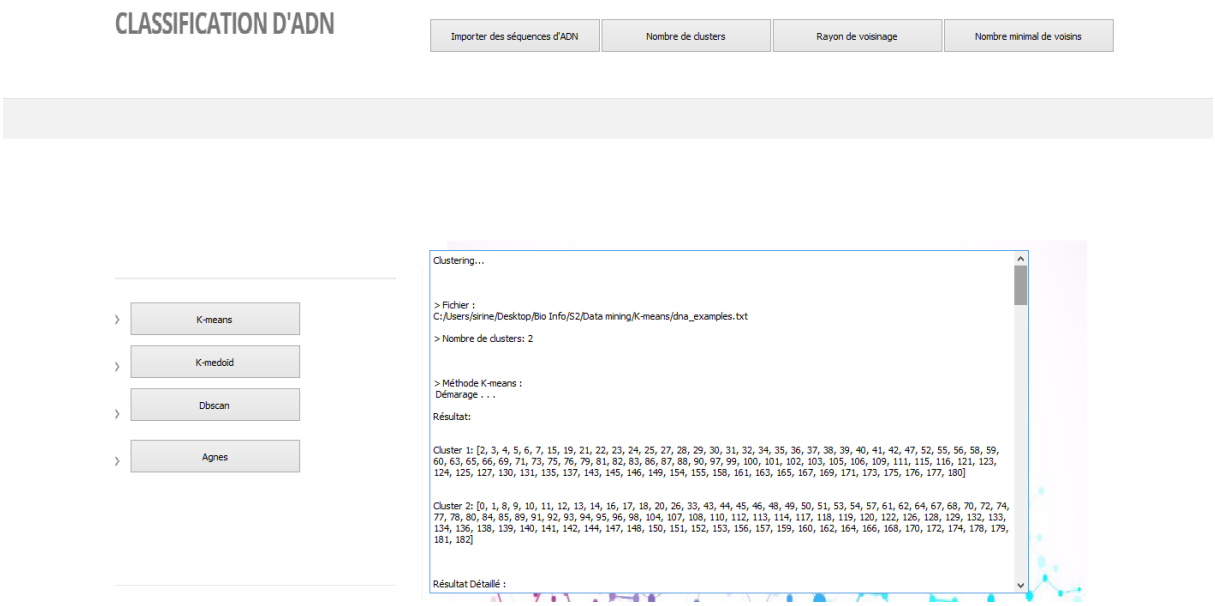
VI.3.Benchmark :

Les données sur lesquelles j'ai testé les méthodes de clustering sont 365 chaines d'ADN regroupées dans un fichier texte.

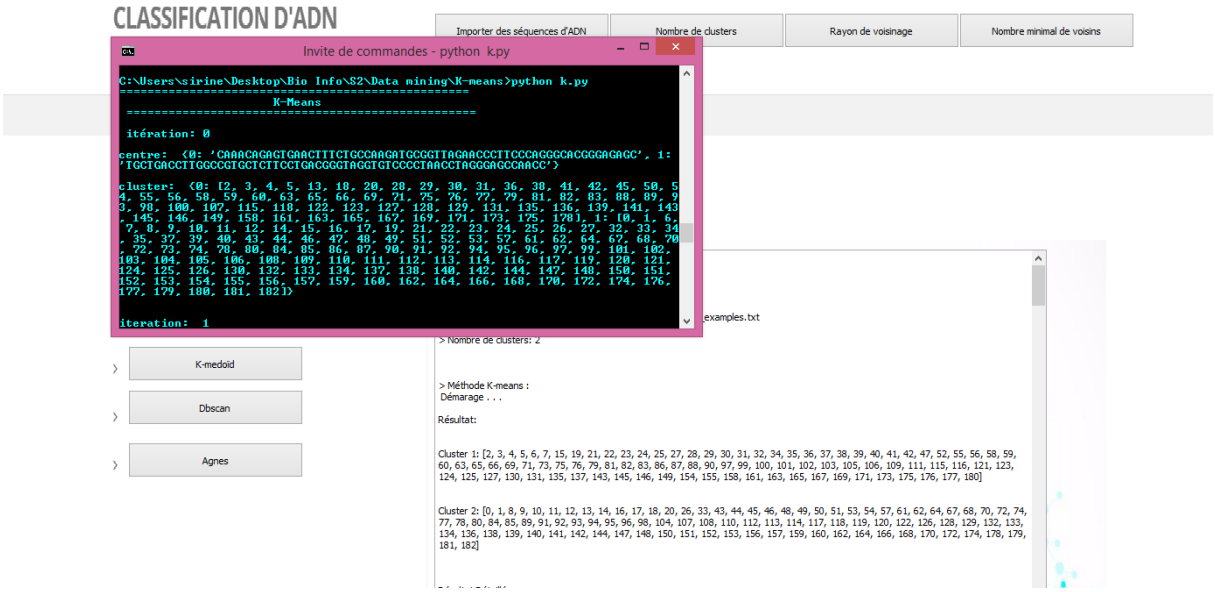
VI.4.Analyse des clusters

VI.4.1.K-Means :

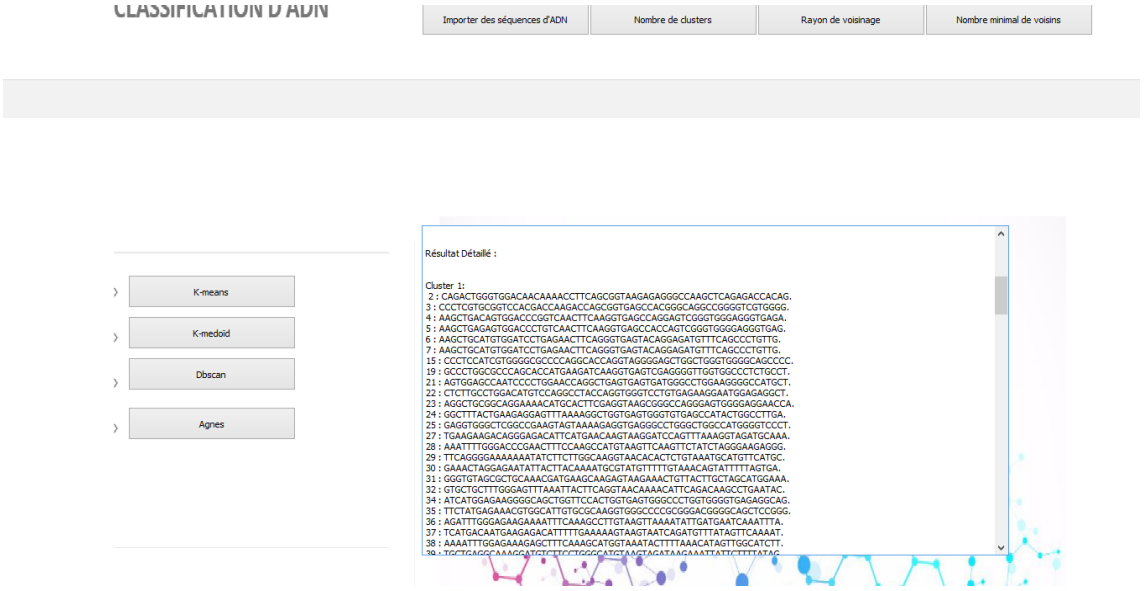
Le résultat de la classification du benchmark précédent avec k=2 :



Aperçu du suivi sur le terminal :



Résultat détaillé :

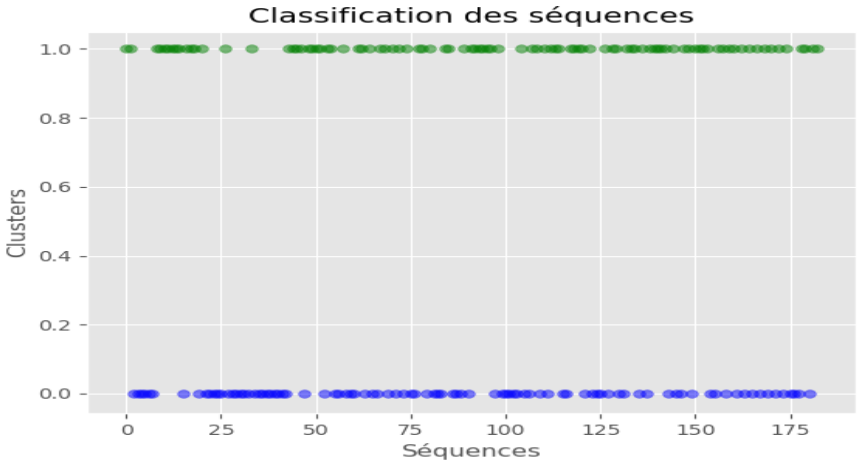


Les clusters obtenus :

Classe 1:
[2, 3, 4, 5, 6, 7, 15, 19, 21, 22, 23, 24, 25, 27, 28, 29, 30, 31, 32, 34, 35, 36, 37, 38, 39, 40, 41, 42, 47, 52, 55, 56, 58, 59, 60, 63, 65, 66, 69, 71, 73, 75, 76, 79, 81, 82, 83, 86, 87, 88, 90, 97, 99, 100, 101, 102, 103, 105, 106, 109, 111, 115, 116, 121, 123, 124, 125, 127, 130, 131, 135, 137, 143, 145, 146, 149, 154, 155, 158, 161, 163, 165, 167, 169, 171, 173, 175, 176, 177, 180]

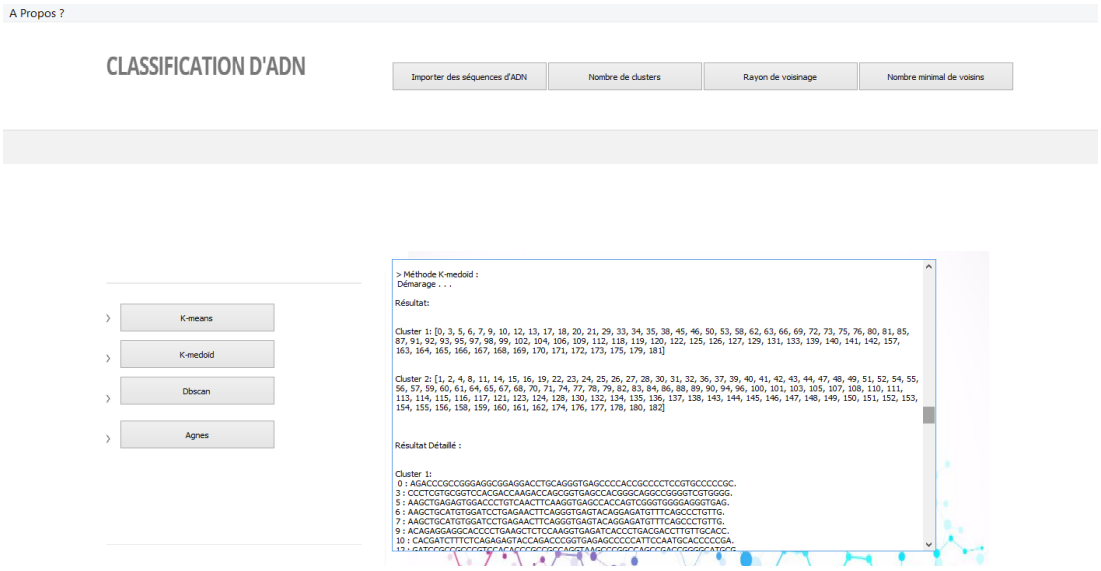
Classe 2:
[0, 1, 8, 9, 10, 11, 12, 13, 14, 16, 17, 18, 20, 26, 33, 43, 44, 45, 46, 48, 49, 50, 51, 53, 54, 57, 61, 62, 64, 67, 68, 70, 72, 74, 77, 78, 80, 84, 85, 89, 91, 92, 93, 94, 95, 96, 98, 104, 107, 108, 110, 112, 113, 114, 117, 118, 119, 120, 122, 126, 128, 129, 132, 133, 134, 136, 138, 139, 140, 141, 142, 144, 147, 148, 150, 151, 152, 153, 156, 157, 159, 160, 162, 164, 166, 168, 170, 172, 174, 178, 179, 181, 182]

Représentation graphique :
L'axe des X représente les numéros de séquences et l'axes des Y représente le numéro de la classe à laquelle elles appartiennent.



VI.4.2.K-Medoïd :

Résultat de la méthode Kmédoid avec k=2 :



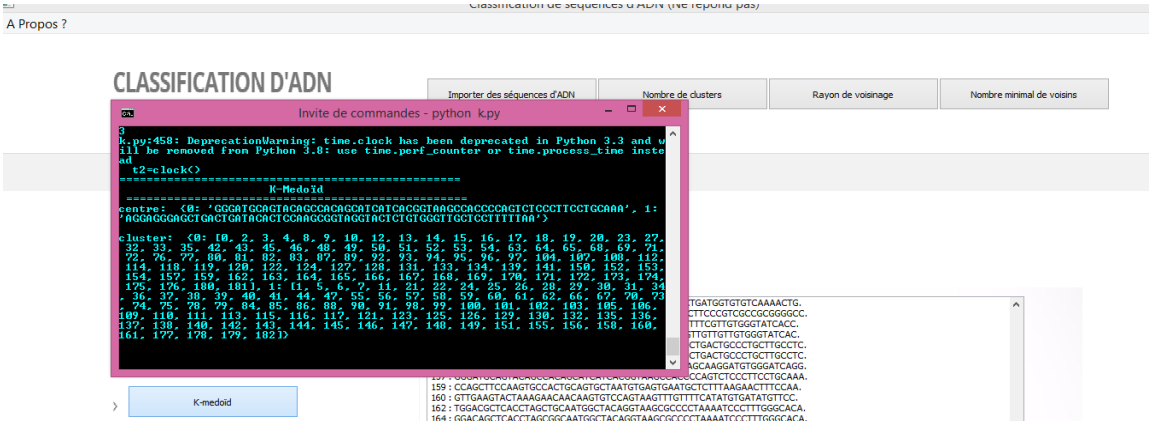
Classe 1 :

[0, 3, 5, 6, 7, 9, 10, 12, 13, 17, 18, 20, 21, 29, 33, 34, 35, 38, 45, 46, 50, 53, 58, 62, 63, 66, 69, 72, 73, 75, 76, 80, 81, 85, 87, 91, 92, 93, 95, 97, 98, 99, 102, 104, 106, 109, 112, 118, 119, 120, 122, 125, 126, 127, 129, 131, 133, 139, 140, 141, 142, 157, 163, 164, 165, 166, 167, 168, 169, 170, 171, 172, 173, 175, 179, 181]

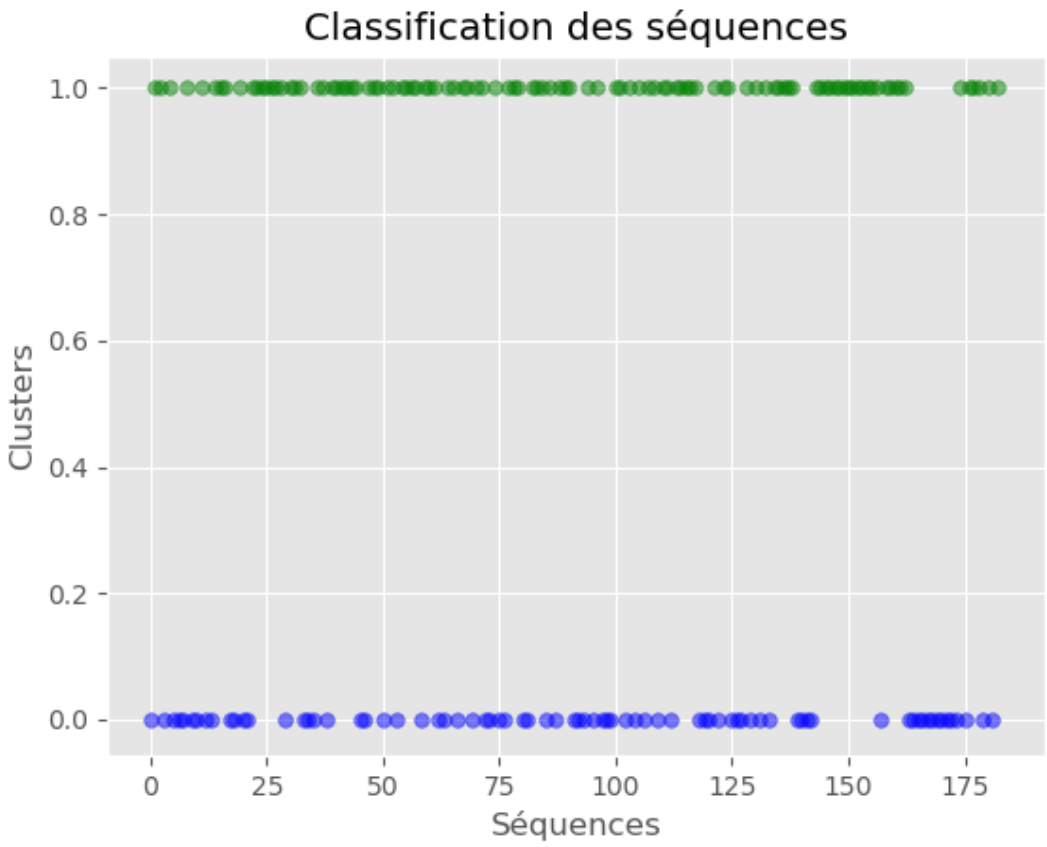
Classe 2 :

[1, 2, 4, 8, 11, 14, 15, 16, 19, 22, 23, 24, 25, 26, 27, 28, 30, 31, 32, 36, 37, 39, 40, 41, 42, 43, 44, 47, 48, 49, 51, 52, 54, 55, 56, 57, 59, 60, 61, 64, 65, 67, 68, 70, 71, 74, 77, 78, 79, 82, 83, 84, 86, 88, 89, 90, 94, 96, 100, 101, 103, 105, 107, 108, 110, 111, 113, 114, 115, 116, 117, 121, 123, 124, 128, 130, 132, 134, 135, 136, 137, 138, 143, 144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 158, 159, 160, 161, 162, 174, 176, 177, 178, 180, 182]

Visualisation sur le terminal :



Représentation graphique :

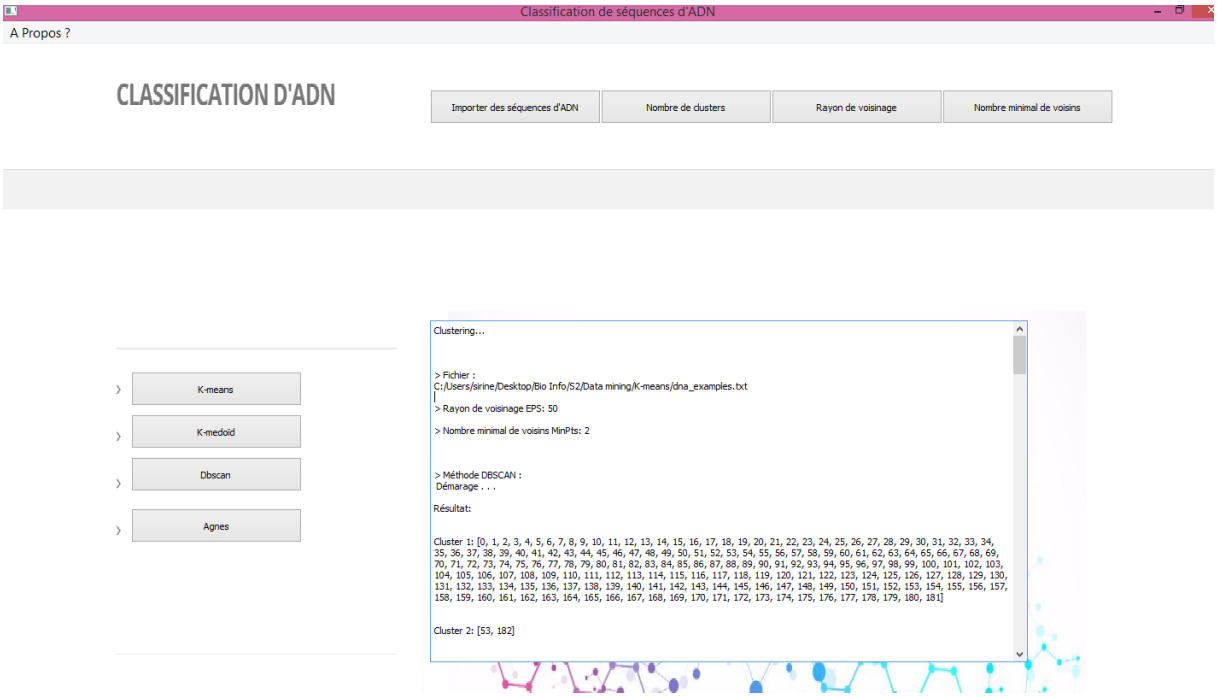


VI.4.3.DBSCAN :

C’est à l’utilisateur de préciser le rayon de voisinage (distance) ainsi que le nombre de voisins.

Le choix sera visualisé pour limiter le taux d’erreur.

Pour minpts=2 et eps=50 on a obtenu 3 classes :



Résultat détaillé :

Classe 1 :

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, 165, 166, 167, 168, 169, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 180, 181]

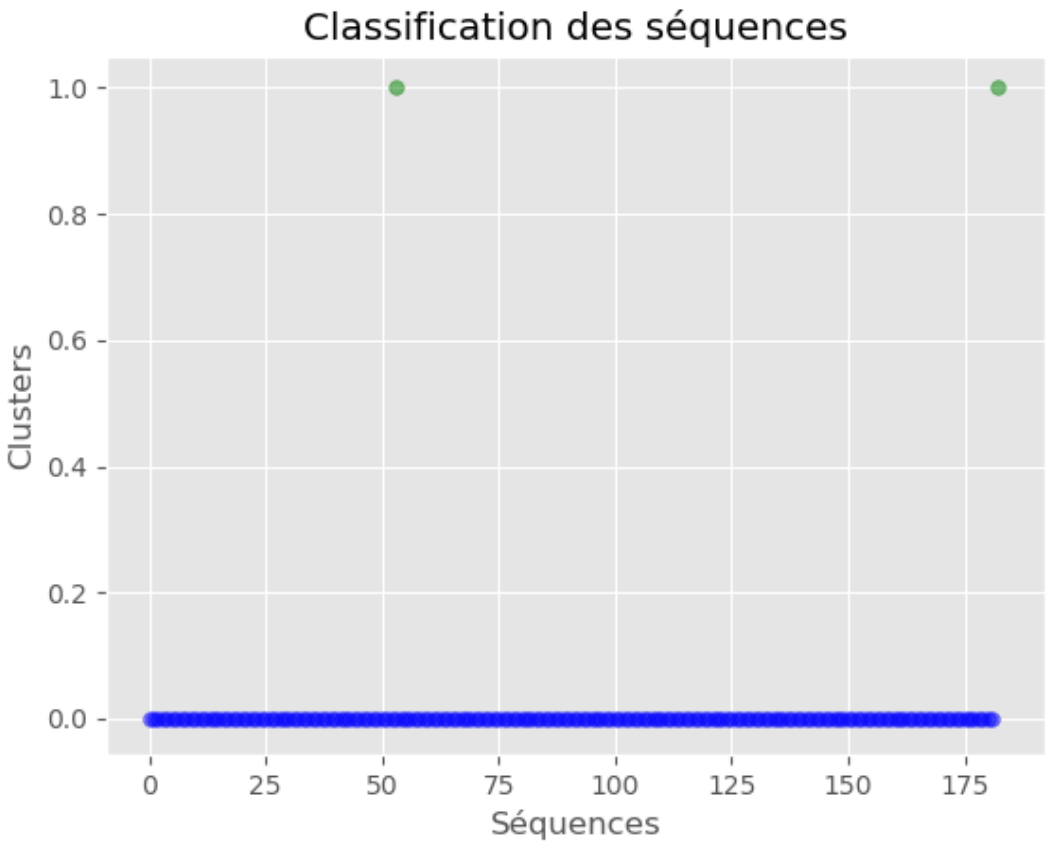
Classe 2 :

[53, 182]

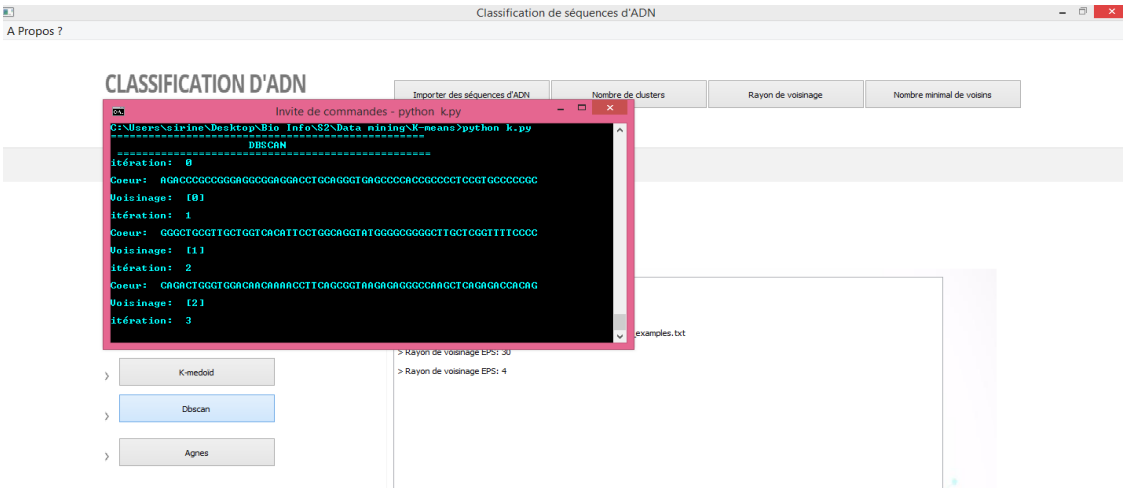
<u>La représentation graphique du résultat :</u>

L'axe des X représente les numéros de séquences et l'axe des y représente le numéro de la classe.

Chaque classe a une couleur différente.

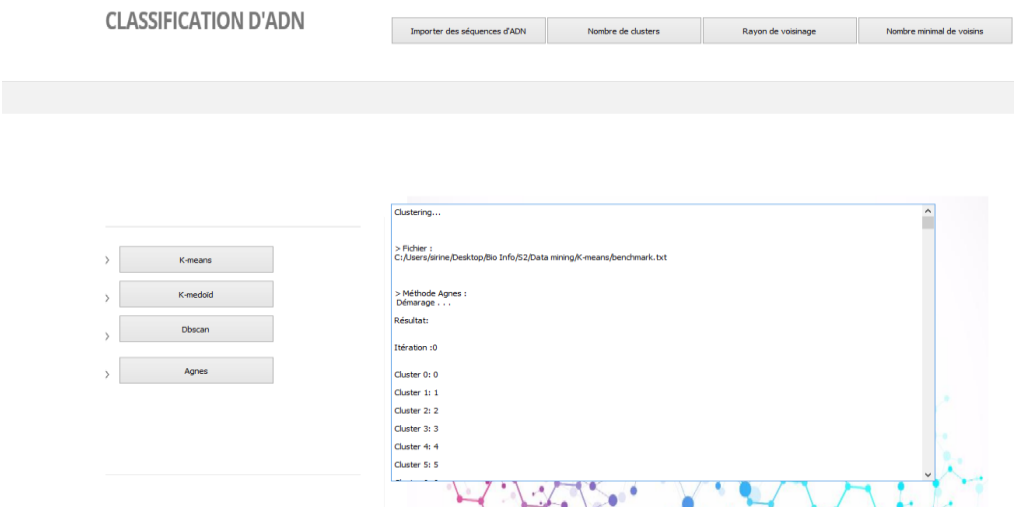


Pendant le calcul le résultat peut suivre les étapes au terminal.

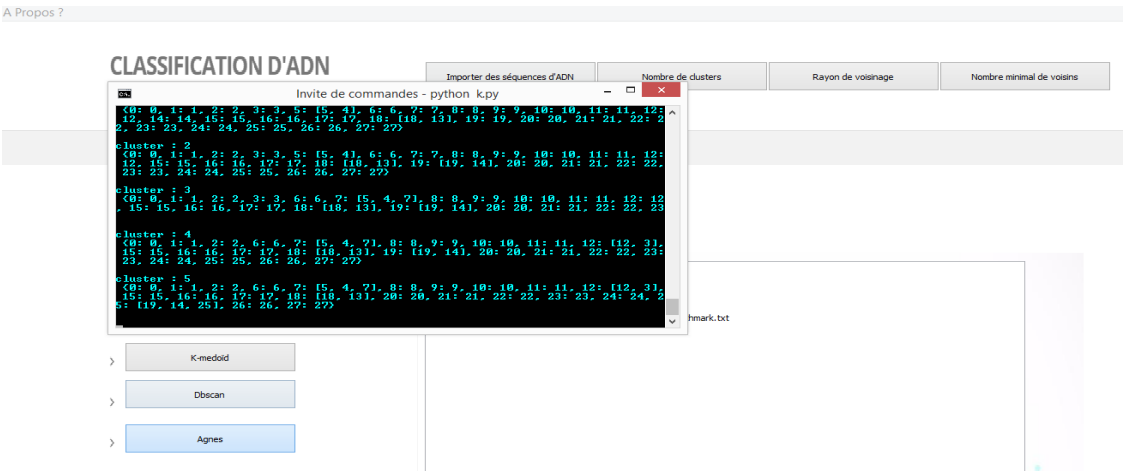


VI.4.4.Agnes :

Résultat :



Visualisation sur le terminal :



Résultat détaillé :

Pour les trois dernières itérations

Itération 25 :

Cluster 17: [12, 3, 17, 5, 4, 7, 16, 6]

Cluster 25: [19, 14, 25, 21, 18, 13, 0, 9, 10, 22, 15]

Cluster 27: [26, 24, 27, 23, 2, 1, 11, 8, 20]

Itération 26 :

Cluster 25: [19, 14, 25, 21, 18, 13, 0, 9, 10, 22, 15, 12, 3, 17, 5, 4, 7, 16, 6]

Cluster 27: [26, 24, 27, 23, 2, 1, 11, 8, 20]

Itération 27 :

Cluster 27: [26, 24, 27, 23, 2, 1, 11, 8, 20, 19, 14, 25, 21, 18, 13, 0, 9, 10, 22, 15, 12, 3, 17, 5, 4, 7, 16, 6]

VI.4.5.Comparasion :

Pour 10 classes et 28 séquences d'ADN en benchmark, en utilisant la distance de levenshtein, nous avons :

Méthode	Inertie inter-classe	Inertie intra-classe	Nombre d'itérations
K-Means	37,9	78	2
K-Medoïd	37,5	78	2
DBSCAN	32 ,3	76,9	12

AGNES :

Itération	Inertie inter-classe	Inertie intra-classe
0	78	78
1	75 ,5	78
2	72,6	78
3	69,9	78
4	67,9	78
5	65,2	78
6	62,9	78
7	60,2	78
8	57,7	78
9	55,3	78
10	52,6	78
11	49 ,6	78
12	46,4	78
13	43,1	78
14	40,6	78
15	37,7	78
16	35	78
17	33	78
18	29,3	78
19	26,9	78
20	23,7	78
21	21,5	78
22	18,1	78
23	14,6	78
24	11 ,8	78
25	8 ,6	78
26	5 ,9	78
27	3,8	78

I. Conclusion :

En conclusion, nous remarquons que les différentes méthodes de classification pour les mêmes données peuvent conduire à des résultats différents et que même les performances diffèrent d'une méthode à une autre.

Dans notre cas, en utilisant un nombre restreint de séquences d'ADN, nous avons constaté que la méthode DBSCAN est la plus performante en termes d'inertie intra-classe et inter-classe, tout de même il y a une légère supériorité de la méthode k-Means pour l'inertie inter-classe par rapport à la méthode K-Medoïd.

Néanmoins, les algorithmes hiérarchiques, AGNES dans notre cas, restent les meilleurs pour mieux comparer les performances dans chaque itération et en sélectionner la meilleure classification.