TP : implémentation clustering hiérarchique agglomératif, single-link

Le projet consiste à implémenter l'algorithme de clustering hiérarchique agglomératif, avec similarité entre clusters de type single-link (similarité des objets les plus proches).

1 Les données

Le programme prend en entrée un fichier contenant la matrice de similarité entre objets, dans un format de type « thésaurus », avec :

- un objet par ligne
- sur chaque ligne, l'objet X, une tabulation, puis la liste des paires Y:similarité(X,Y), séparées par des tabulations (et en général ordonnées par similarité décroissante à X, mais ce n'est pas requis par le format).

On donne pour s'entraîner un exemple jouet (toy-thesaurus.txt, cf. l'exemple en section 2).

2 Questions préparatoires à l'implémentation ...

Dans la phase d'implémentation, on travaille avec un mini-exemple.
Soient les objets à clusteriser A,B,C,D et E, dont les similarités sont :

A	B:0.2	C:0.1	D:0.04	E:0.01
B	A:0.2	D:0.15	E:0.15	C:0.1
C	D:0.4	E:0.175	A:0.1	B:0.1
D	C:0.4	B:0.15	A:0.04	E:0.02
E	C:0.175	B:0.15	D:0.02	A:0.01

(**Attention**: il ne s'agit pas d'une matrice, cf. sur la ligne de l'objet X, les objets apparaissent par ordre décroissant de similarité avec l'objet X.)

Algorithme:
On utilise en plus de la (demi-) matrice de similarité, une liste des plus proches cluster de chaque cluster : PPC(i) = j signifie j est le plus proche cluster de i
==> cela permet de trouver plus efficacement la paire de clusters à fusionner.
Question: Si on fusionne x et y, et si on avait PPC(z)=x, que vaut après fusion PPC(z) ?

Déroulé de l'algorithme

Ecrivez la demi-matrice de similarité correspondante.
Déroulez l'algo HAC single-link de manière à dessiner le dendrogramme résultant.
Donnez à chaque étape la matrice mise à jour, et la liste des PPC (plus proche cluster) de chaque cluster mise à jour.

Etudiez le déroulé de l'algorithme : Quelles sont les cellules à mettre à jour ? Quels sont les ppc à mettre à jour ? Mêmes questions pour l'algo complete-link.

Pseudo-code

Pour le code, les clusters sont identifiés par des **indices** et pas par le nom des objets qu'ils contiennent, i.e. on aura au départ :
cluster 0 = cluster singleton contenant A cluster 1 = cluster singleton contenant B
cluster 2 = cluster singleton contenant C etc…

On utilise les structures de données suivantes :

- Clusters : la liste indice = id de cluster, valeur = les objets contenus ds le cluster
- sim_matrix : matrice carrée de similarité entre clusters, remplie pour les cases i < j
- ppc : tableau indice = cluster i ,
  valeur = paire (similarité(i,k), cluster k le plus proche de i)

Choix d'implémentation: lorsque l'on fusionne 2 clusters, d'indices i et j, avec i < j, le nouveau cluster aura l'indice i et on efface les lignes et colonnes d'indice j, ce qui décale tous les indices > j
(Une autre possibilité serait de maintenir une liste des clusters "actifs")

On suppose disposer des méthodes :
sim_matrix.initialize(objets_et_sim) : initialisation de la matrice de similarité, étant donnée la liste d'objets et leurs similarités
sim_matrix.get_sim(i,j) : renvoie la similarité des clusters i et j, **peu importe que i<j ou i>j**
sim_matrix.set_sim(i,j, val) : met la valeur val à la case (i,j) ou (j,i) selon que i<j ou j>i
sim_matrix.delete_row_and_column(i) : suppression ligne et colonne i
sim_matrix.size() : renvoie la taille de la matrice carrée

NB : get_sim et set_sim permettent de ne pas se préoccuper de l'ordre i<j ou i>j …

Ecrire le pseudo-code pour l'algo single-link, **avec une gestion la plus efficace possible de la mise à jour des PPC (plus proches clusters) et de la matrice**.

Comparez le déroulé de l'algo si on utilise la similarité **complete-link**.

3 Structures de données et canevas

Ci-dessous on utilise « **ppc** » pour « plus proche cluster ». On donne deux classes :

class Dendrogram:
  """ Un dendrogramme : utilisé pour représenter tout cluster obtenu par clustering hiérarchique
  Contient
  - soit un seul objet (cluster singleton), dans self.child1 (et child2 vaut None)
  - soit une paire de dendrogrammes (les 2 noeuds fils) et leur similarité
  Membres :
  - **child1** : string (si singleton) ou instance de Dendrogram
  - **child2** : None (si singleton) ou instance de Dendrogram

```
   - sim : similarité entre child1 et child2
   - members : la liste à plat des objets contenus dans le dendrogramme (cf. pb de récursion trop profonde)
   """
```

```
class HAC:
   """
   Classe implémentant le single-link hierarchical agglomerative clustering
   Membres:
   * clusters : liste python de clusters, de type Dendrogram
                (éventuellement réduits à un seul objet)
                IMPORTANT: le rang dans cette liste sert d'id de cluster dans les
                autres membres (matrice de sim, et closest_clusters)
   * sim_matrix : matrice de similarité entre clusters (type numpy.ndarray)
   * ppc : les ppc de chaque cluster
        Plus précisément : une list de paires :
                pour chaque cluster, identifié par son rang,
                stocke une paire (similarité avec le ppc, ppc)
   """
```

La lecture du thésaurus et sa conversion vers une matrice de similarité est fournie (thesaurus2simmatrix), et **rend un ndarray numpy**.

Il reste à se concentrer sur l'initialisation de l'algo, et la boucle de fusions des clusters.

Prenez connaissance du canevas fourni hac-canevas.py
et **remplissez** la méthode suivante de la classe HAC :

```
def clusterize(self, object_names, sim_matrix, nbclusters=1):
        """ Calcule le clustering étant donnés :
        object_names = une liste de noms d'objets (dont le rang constitue l'id)
        sim_matrix   = une matrice de similarite entre ces objets
        nbclusters   = le nb de clusters voulus """
```

4 Un peu de python

4.1 Type Decimal

Type float python : représenté comme des sommes de fractions en base 2, ce qui n'est pas très intuitif, et de plus amène à des approximations pour des nombres rationnels (sommes de fractions en base 10 : 0.125 = 1/10 + 2/100 + 5/1000)

Par exemple 0.15 est approximé, ce qui donne des effets surprenants :

```
>>> a = 0.15
>>> a*3
0.44999999999999996
>>> a*5
0.75
>>> a*6
```



```
0.89999999999999991
```

Pour retrouver les nombres rationnels que l'on connait bien, on peut utiliser le module **Decimal** (http://docs.python.org/library/decimal.html#module-decimal)

```
>>> from decimal import Decimal
>>> a = Decimal('0.15')
>>> a*2
Decimal('0.30')
>>> a*3
Decimal('0.45')
```

4.2 Tableaux multidimensionnels avec numpy

Pour la matrice de similarité nxn, au lieu de faire n listes de n objets, on choisit d'utiliser le type ndarray du module numpy.
Voir mémo numpy

En particulier ici:

```
# rappels sur les slices : x:y signifie de l'indice x inclus à l'indice y exclus
# soit x une liste par ex: x = range(5)
# x[1:3] = les valeurs de l'indice 1 inclus à l'indice 3 exclus

# donc dans un ndarray, cela donne :
#     la première ligne (indice 0) : a[ 0, : ]
#     la 3eme colonne (indice 2) : a[ : , 2]

# numpy.delete rend des tableaux tronqués
# exemple : supprimer des lignes d'un tableau à 2 dims
#  (ou plus généralement, supprimer un hyperplan de n-1 dims d'un tableau à n dims)
#  c = numpy.delete( a, numpy.s_[1], axis=0 )   #=> rend une matrice
                                                #où la ligne d'indice 1 a été supprimée
#  c = numpy.delete( a, numpy.s_[3], axis=1 )       #=> supprime la colonne d'indice 3
```

5 Applications

L'algo peut être appliqué à des éléments pour lesquels on a déjà les similarités 2 à 2 (par exemple un thésaurus distributionnel : des mots et leurs similarités à d'autres mots, calculées en utilisant les contextes communs aux mots).

Ou bien à des objets représentés par un vecteur, en utilisant par ex. cosinus pour la similarité entre objets.