RAPPORT DE TP

Module :

Fouille de données

Membres du binôme :

AKKOUCHE Adnane Aymen
SALHI Omar

Dans ce TP on nous a demandé d'implémenter l'algorithme **KMeans** qui fait du clustering (algorithme non supervisé), et d'essayer d'automatiser le choix du K (nombre de clusters). Dans ce qui suit on utilise le dataset iris dans un notebook jupyter (python).

Lecture des données

Comme dit en page de garde on utilisera la dataset iris pour cette démonstration donc on commencera par l'ouvrir.

Entrée [2]:
```
don = pd.read_csv('iris.csv')
don.head()
```

Out[2]:

	X1	X2	X3	X4	Y
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa

Se débarrasser de la colonne cible

Étant un algorithme non-supervisé **KMeans** se passera des étiquettes, on enlevé donc la colonne cible et on en profite pour visualiser le résultat qu'on souhaite obtenir de notre clustering.

Entrée [3]:
```
Y = np.zeros(don.values[:,4].shape)
Y = Y + (don.values[:,4]=='Iris-versicolor')
Y = Y + 2*(don.values[:,4]=='Iris-virginica')
Y
```

Out[3]:
```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1.,
       1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
       1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
       1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 2., 2.,
       2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2.,
       2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2.,
       2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2., 2.])
```

Entrée [4]:
```
data = don.values[:,:-1].astype('float32')
data[:5]
```

Out[4]:
```
array([[5.1, 3.5, 1.4, 0.2],
       [4.9, 3. , 1.4, 0.2],
       [4.7, 3.2, 1.3, 0.2],
       [4.6, 3.1, 1.5, 0.2],
       [5. , 3.6, 1.4, 0.2]], dtype=float32)
```

Pré-traitement

Les données de nos colonnes variant avec des écarts différents (pas énormément différents certes mais assez pour être remarquable) on préférera travailler sur les données centrées réduites.

Entrée [5]:
```
data = (data - data.mean(axis=0))/data.std(axis=0)
data[:5]
```

Out[5]:
```
array([[-0.9006812 ,  1.0320569 , -1.3412726 , -1.3129768 ],
       [-1.1430167 , -0.12495793, -1.3412726 , -1.3129768 ],
       [-1.3853527 ,  0.33784813, -1.3981384 , -1.3129768 ],
       [-1.5065205 ,  0.10644482, -1.284407  , -1.3129768 ],
       [-1.0218489 ,  1.2634597 , -1.3412726 , -1.3129768 ]],
      dtype=float32)
```

Métriques

Il nous faut l'implémentation de quelques métriques vues en cours, ainsi que le calcule de l'inertie pour la méthode du coude.

Entrée [124]:
```
def dstMoyCentres(cent): #distance moyenne interclasse, difference entre centroides
    DSTs = []
    for i in range(cent.shape[0]):
        for j in range(i+1,cent.shape[0]):
            DSTs = DSTs + [(cent[i]-cent[j]).sum()**2]
    return (np.mean(DSTs))
def dstMoyIntra(data,cent,clus,k): #difference moyenne intraclasse, difference entre chaque valeur et son centroid
    DSTs = []
    for c in range(k):
        DSTs = DSTs + [abs((data[clus==c]-cent[c]).sum())]
    return (np.mean(DSTs))
def inertie(data,cent,clus):
    dist = 0
    for i in range(data.shape[0]):
        dist = dist + (np.power(data[i]-cent[int(clus[i])],2).sum())
    return (dist)
```

Implémentation de KMeans

Entrée [101]:
```
def kmeans(X, k):
    diff = True
    cluster = np.zeros(X.shape[0])
    centroids = data[:k]
    while diff:
        for i, row in enumerate(X):
            mn_dist = float('inf')
            for idx, centroid in enumerate(centroids):
                d = np.sqrt(np.power(centroid-row,2).sum())
                if mn_dist > d:
                    mn_dist = d
                    cluster[i] = idx
        new_centroids = pd.DataFrame(X,dtype=float).groupby(by=cluster).mean().values
        if np.count_nonzero(centroids-new_centroids) == 0:
            diff = False
        else:
            centroids = new_centroids
    return centroids, cluster
```

Au lieu de choisir K centroids arbitrairement on a opté pour prendre les K premiers du dataset (un résultat possible pour le choix aléatoire) pour que dans ce qui suit seul le K influera sur l'algorithme et pas le choix des centroids initiaux.

Varier la valeur de K

Dans cette étape on fais varier la valeur du K et on calcule nos métriques précédemment implémentées, et on utilisera nos résultats pour prendre des décisions automatiques sur la valeur de K a choisir.

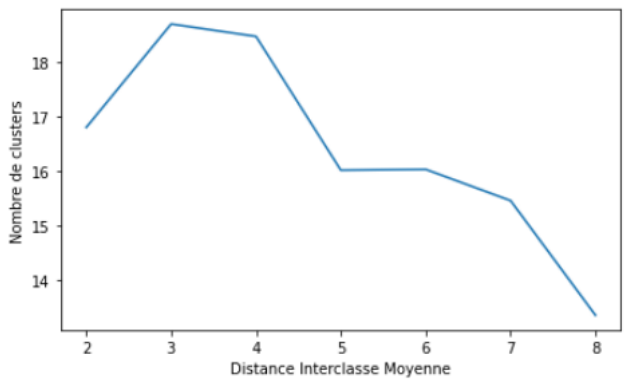
Entrée [180]:
```
maxClusN = 8
minClusN = 2
Ks = range(minClusN,maxClusN+1)
dst = []
dst2 = []
ine = []
for k in Ks:
    cent,clus = kmeans(data,k)
    dst = dst + [dstMoyCentres(cent)]
    dst2 = dst2 + [dstMoyIntra(data,cent,clus,k)]
    ine = ine + [inertie(data,cent,clus)]
print("Les distances interclasse trouvées sont : ",np.round(dst,4))
print("Les distances intraclasse trouvées sont : ",np.round(np.array(dst2)*1000000000000000,4),"/10^15")
print("Les inerties trouvées sont : ",np.round(ine,4))
```

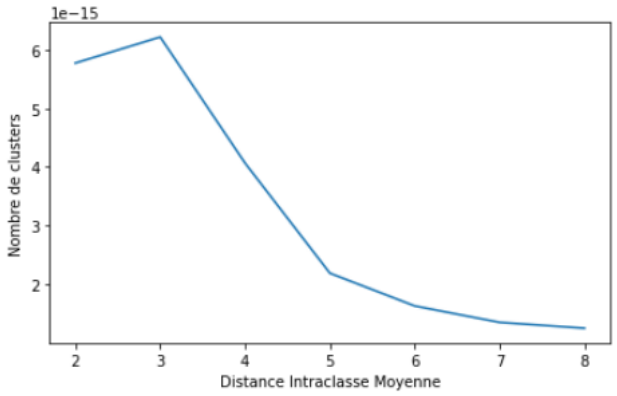
```
Les distances interclasse trouvées sont :  [16.797  18.6924 18.4652 16.0115 16.0264 15.4542 13.3526]
Les distances intraclasse trouvées sont :  [5.7732 6.2172 4.0662 2.1871 1.6283 1.3481 1.249 ] /10^15
Les inerties trouvées sont :  [223.732  141.1542 114.6155 105.3245 103.7241 102.0623 101.8744]
```

Distances inter/intra classe moyennes

Comme on peut le voir sur les graphes, pour K=3 (et potentiellement K=4) on a la meilleure valeur pour la distance inter-classe moyenne. On pourrais presque dire la même chose avec la distance intra-classe moyenne ne serait-ce pour les valeurs bien trop petites de cette dernière (dans les 10^{15})

```
Il suffit donc de prendre le nombre de clusters qui donne le maximum de distance interclasse!
On choisi donc  3
Les distance intraclasse par contre sont trop petites
```



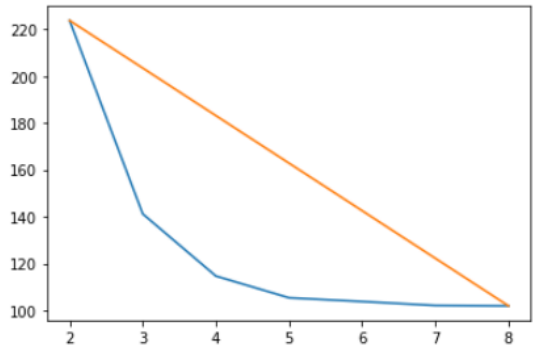


La méthode du coude

Le "coude" est sur le graphe ci-bas formé par les inerties calculées précédemment. Avec cette méthode on va choisir comme K celui dont l'inertie est la plus éloignée de la ligne qui relie le maximum d'inertie au minimum.

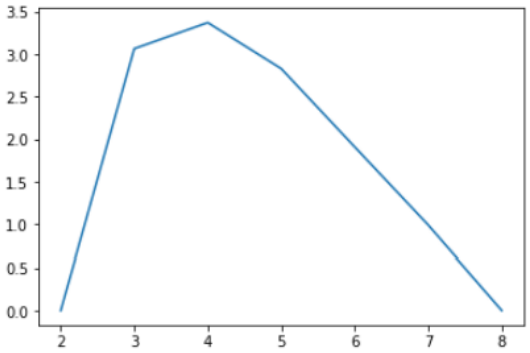
Entrée [182]:
```
plt.plot(Ks,ine)
plt.plot([min(Ks),max(Ks)],[ine[min(Ks)-minClusN],ine[max(Ks)-minClusN]])
```

Out[182]: [<matplotlib.lines.Line2D at 0x14807c58>]



Entrée [188]:
```
def calcDst(x,y,a,b,c):
    return (abs(a*x+b*y+c)/np.sqrt(a*a+b*b))
a = ine[max(Ks)-minClusN] - ine[min(Ks)-minClusN]
b = -(max(Ks)-min(Ks))
c = -min(Ks)*ine[max(Ks)-minClusN]+max(Ks)*ine[min(Ks)-minClusN]
DSTS = []
for k in range(maxClusN-minClusN+1):
    DSTS = DSTS + [calcDst(Ks[k],ine[k],a,b,c)]
plt.plot(Ks,DSTS)
print("Les distances trouvées sont : ",np.round(DSTS,4))
print("Donc avec la méthode du coude on choisi ",Ks[np.argmax(DSTS)]," clusters.")
```

```
Les distances trouvées sont :  [0.     3.0622 3.3686 2.8267 1.9066 0.9896 0.    ]
Donc avec la méthode du coude on choisi  4  clusters.
```



Avec cette méthode on trouve que K=4 est le meilleur choix avec K=3 un alternative presque tout aussi bonne.

Résultat de MeanShift

Contrairement à KMeans l'algorithme MeanShift trouve le nombre de clusters lui même, on peut donc songer a en copier le nombre de clusters et l'appliquer avec KMeans si on juge que notre implémentation est meilleure que celle de MeanShift.

Entrée [193]:
```
from sklearn.cluster import MeanShift, estimate_bandwidth

bandwidth = estimate_bandwidth(data, quantile=0.26, n_jobs=-1)
ms = MeanShift(bandwidth=bandwidth, bin_seeding=False, n_jobs=-1, max_iter=500)
ms.fit(data)

print(f"Number of estimated clusters : {len(np.unique(ms.labels_))}")
```

```
Number of estimated clusters : 3
```

MeanShift aussi semble indiquer K=3.

Conclusion

On a trouvé des résultats assez similaires dans les trois cas mais aucune méthode n'est parfaite, les deux premières requièrent toujours d'appliquer KMeans pour un certain nombre de K et utiliser MeanShift ressemble plus a de la triche qu'autre chose. On dira donc qu'automatiser le choix du K n'est pas aussi facile que ça.