CR TP RCP208

TP K-Means

Générez 500 données suivant une distribution uniforme dans [0,1)3[0,1)3. Appliquez sur ces données K-means avec n_clusters=5 et initialisation aléatoire (random) et examinez la stabilité des résultats en utilisant l'indice de Rand

```
def stability(z):
    max_k = len(z)
    r = []
    for i in range(0, max_k):
        for j in range(i + 1, max_k - 1):
            r.append(metrics.adjusted_rand_score(z[i].labels_, z[j].labels_))
    return np.asarray(r)
# Générez 500 données suivant une distribution uniforme dans [0,1).
d = np.random.uniform(0, 1, (500, 3))

kmeans = KMeans(n_clusters = 5, n_init = 10, init = 'random')
kmeans.fit(d[:, :3])

fig = plt.figure()
ax = fig.add_subplot(111, projection = '3d')
ax.scatter(d[:, 0], d[:, 1], d[:, 2], c = kmeans.labels_)
plt.show()

# Appliquez sur ces données K-means avec n_clusters=5 et initialisation
aléatoire (random),
# et examinez la stabilité des résultats en utilisant l'indice de Rand.
max_k = 30
z = []
for k in range(0, max_k):
    z.append(KMeans(n_clusters = 5, n_init = 10, init='random').fit(d[:,:3]))

r = stability(z)

from scipy import stats
print("MEAN = ", r.mean())
print("STD = ", r.std())
```

```
MEAN =  0.888423178049
STD =  0.0937534297232
```

Après plusieurs classifications, on trouve, que l'indice de Rand est en moyenne égale à 0.9 et un écart-type égal à 0.1. Cela montre que la classification est globalement stable. Les éléments sont classés dans un groupe puis dans un autre entre deux classifications différentes.

Appliquez sur ces mêmes données K-means avec toujours n_clusters=5 mais initialisation k-means++, examinez la stabilité des résultats.

```
z = []
for k in range(0, max_k):
    z.append(KMeans(n_clusters = 5, n_init = 10, init='k-means++').fit(d[:,:3]))
```

```
r = stability(z)

print("MEAN = ", r.mean())
print("STD = ", r.std())

MEAN =  0.908037891359
STD =  0.0669934091903
```

Globalement, le classement est plus stable avec une initialisation avec k-means. On observe des classements proches et moins de variations.

Observez-vous des différences par rapport aux résultats obtenus sur les données générées avec np.random.randn ? Expliquez.

```
Cas génération avec une loi normale, initialisation des centres aléatoires :
MEAN =  0.703571157983
STD =  0.149479079965

Cas génération avec une loi normale, initialisation avec k-means++ :
MEAN =  0.73547691986
STD =  0.136989731019
```

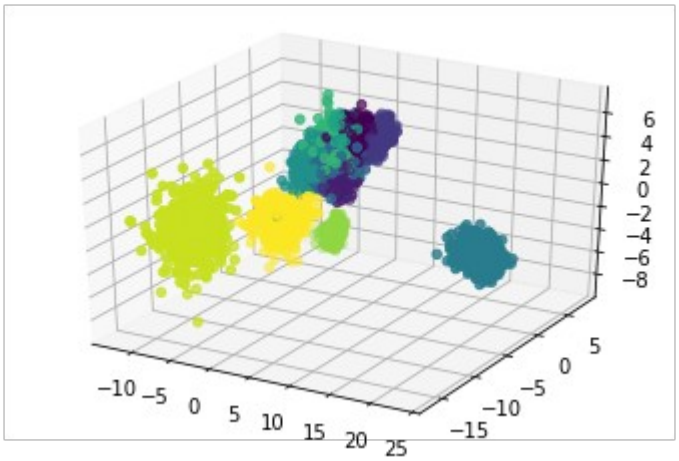
Dans le cas où les données sont générées selon une loi normale, la moyenne de l'indice de Rand ajusté est proche de 0.7 et l'écart-type à 0.15. Cela indique une forte variation entre différentes exécutions de l'algorithme de classification.

En moyenne, l'indice de Rand est plus fort en moyenne pour les données uniformes car celles-ci sont générées de manière moins « compacte » que via une loi normale et donc plus facilement séparables en classes.

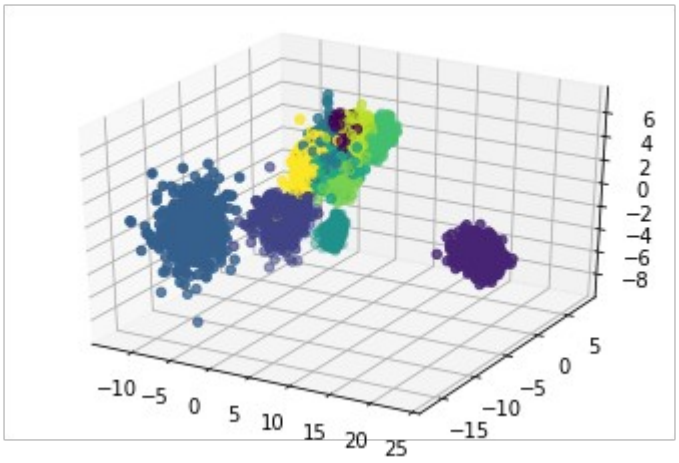
Cependant, dans les deux cas, l'indice est très variable. Cela est dû au fait d'une absence de groupes dans le jeu de données. Le premier jeu de données, généré à partir de 5 échantillons issues d'une distribution normale, contient lui 5 groupes, permettant une bonne stabilité de l'indice de Rand.

Appliquez l'analyse discriminante à ces données et appliquez de nouveau K-means avec n_clusters = 11 aux données projetées dans l'espace discriminant.

```
texture = np.loadtxt('texture.dat')
np.random.shuffle(texture)
kmeans = KMeans(n_clusters=11).fit(texture[:,:40])
print(metrics.adjusted_rand_score(kmeans.labels_, texture[:,40]))
0.46198856078637229

from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
lda = LinearDiscriminantAnalysis()
tlda = lda.fit_transform(texture[:,:40],texture[:,40])
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(tlda[:, 0], tlda[:, 1], tlda[:, 2], c = texture[:,40])
plt.show()
```



```
kmeans = KMeans(n_clusters=11).fit(tlda)

fig = plt.figure()
ax = fig.add_subplot(111, projection = '3d')
ax.scatter(tlda[:, 0], tlda[:, 1], tlda[:, 2], c = kmeans.labels_)
plt.show()
```



```
metrics.adjusted_rand_score(kmeans.labels_, texture[:,40])
```

0.99007579314959171

Que constatez-vous?

On observe de très bons résultats (passant de 0.44 à 0.99 (soit une classification quasi-parfaite)).

Expliquez.

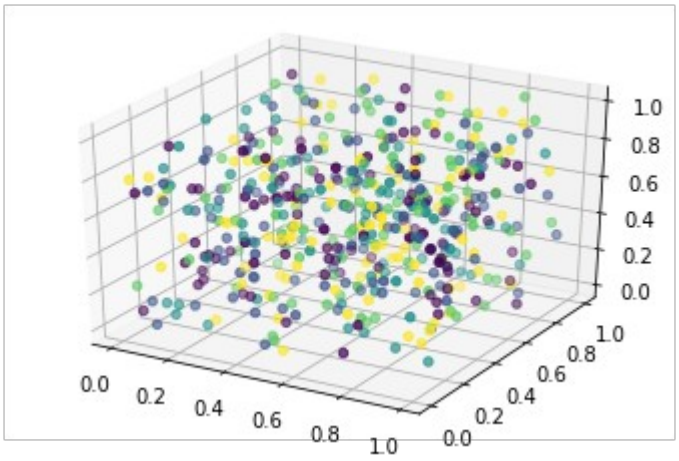
Grâce à l'analyse discriminante, nous avons trouvé les facteurs séparant au mieux les données, simplifiant les variables et permettant d'avoir des données plus sphériques, comme on peut le voir sur le graphique en 3D. Cela aide K-Means à mieux trouver les groupes et donc, obtient un meilleur indice de Rand. De plus, on fournit le bon nombre de classes (11 au total), qui est un hyper-paramètre important pour la classification via K-Means

TP Classification spectrale

Sur les 500 données générées comme au TP précédent (sur K-means) suivant une distribution uniforme dans [0,1)^3, appliquez la classification spectrale avec toujours n_clusters=5 et visualisez les résultats. Examinez la stabilité (en utilisant l'indice de Rand) des partitionnements obtenus. Observez-vous des différences significatives par rapport aux résultats obtenus lors du TP sur K-means ? Expliquez.

```
d = np.random.uniform(0, 1, (500, 3))
max_k = 30
z = []
for k in range(0, max_k):
    z.append(SpectralClustering(n_clusters = 5, eigen_solver='arpack',
                affinity='nearest_neighbors').fit(d))
r = stability(z)
from scipy import stats
print("MEAN = {}".format(r.mean()))
print("STD = {}".format(r.std()))

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(d[:,0], d[:,1], d[:,2], c=spectral.labels_)
plt.show()
```

```
MEAN = 0.9703396800555109
STD = 0.03874301431652687
```



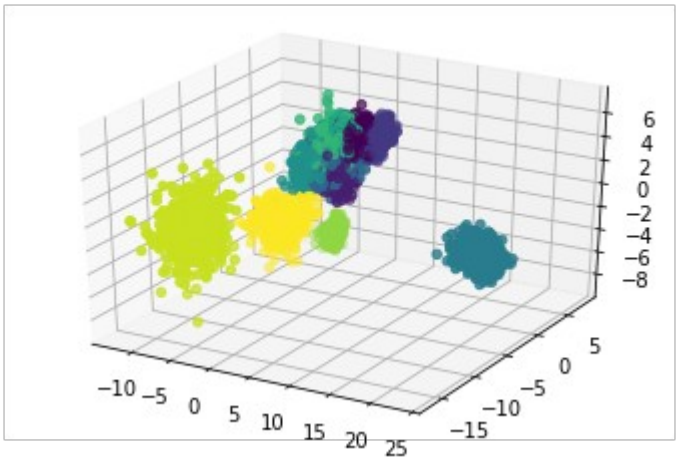
La classification est bien meilleure que celle de effectuée par K-Means. On trouve quasiment toujours la même répartition. La variation vient de K-Means à la fin. Cependant, la préparation est très efficace.

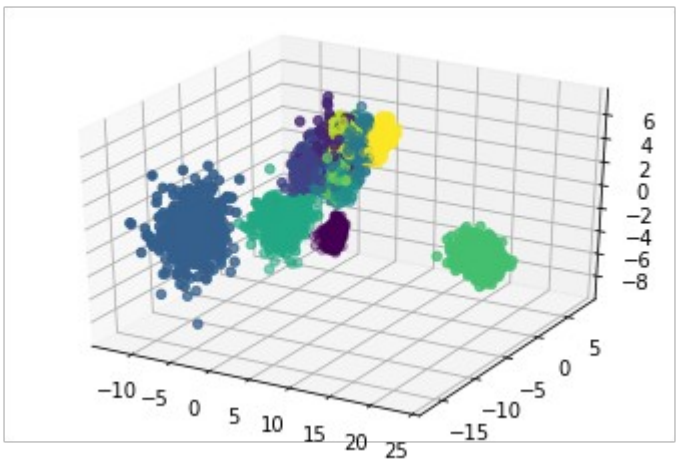
Après une analyse discriminante de ces données, appliquez la classification spectrale avec n_clusters = 11 aux données projetées dans l'espace discriminant. Que constatez-vous ? Expliquez. Visualisez les résultats.

```
texture = np.loadtxt('texture.dat')
np.random.shuffle(texture)
spectral = SpectralClustering(n_clusters=11, eigen_solver='arpack',
                affinity='nearest_neighbors').fit(texture[:,:40])
print(metrics.adjusted_rand_score(spectral.labels_, texture[:,40]))
```

```
0.581451089715
```

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
lda = LinearDiscriminantAnalysis()

tlda = lda.fit_transform(texture, texture[:,40])
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(tlda[:, 0], tlda[:, 1], tlda[:, 2], c = texture[:,40])
plt.show()

spectral = SpectralClustering(n_clusters=11, eigen_solver='arpack',
               affinity='nearest_neighbors').fit(tlda)

fig = plt.figure()
ax = fig.add_subplot(111, projection = '3d')
ax.scatter(tlda[:, 0], tlda[:, 1], tlda[:, 2], c = spectral.labels_)
plt.show()

metrics.adjusted_rand_score(spectral.labels_, texture[:,40])
```





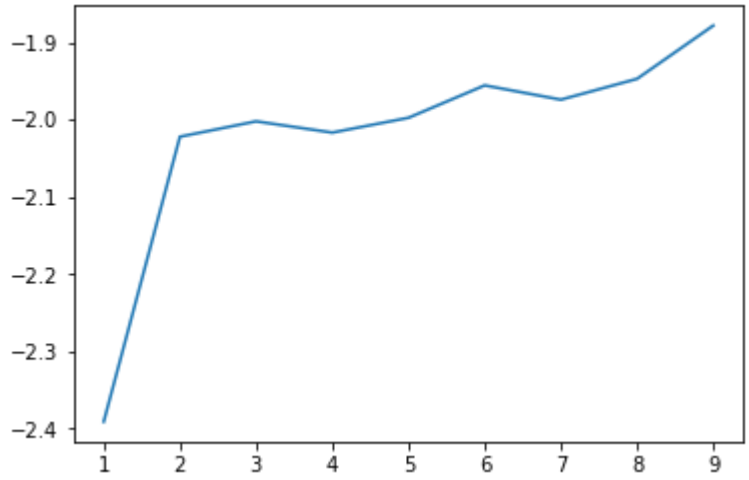

```
0.87982991298013347
```

On constate que l'analyse discriminante améliore les résultats de la classification. Cependant, le résultat est moins stable que par rapport à K-Means entre plusieurs exécutions.

TP Modèle de mélange

Faites varier le nombre de composantes et examinez visuellement les résultats. Regardez les pondérations des composantes et les moyennes de ces composantes. Examinez de quelle manière évolue la valeur finale atteinte par la log-vraisemblance avec le nombre de composantes.

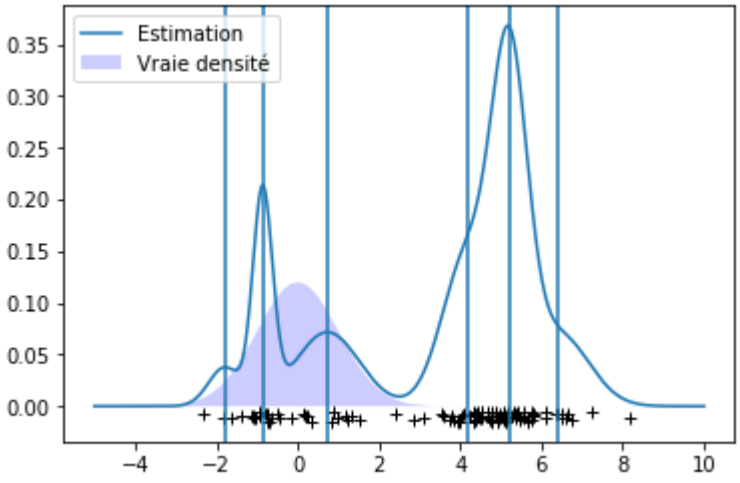
```
lower_bounds = []

for i in range(1, 10):
    gmm = GaussianMixture(n_components=i,n_init=3).fit(X)
    lower_bounds.append(gmm.lower_bound_)

plt.plot(range(1, 10), lower_bounds,  '-', label="Evolution du log-vraisemblance
en fonction du nombre de composantes")
plt.show()
```

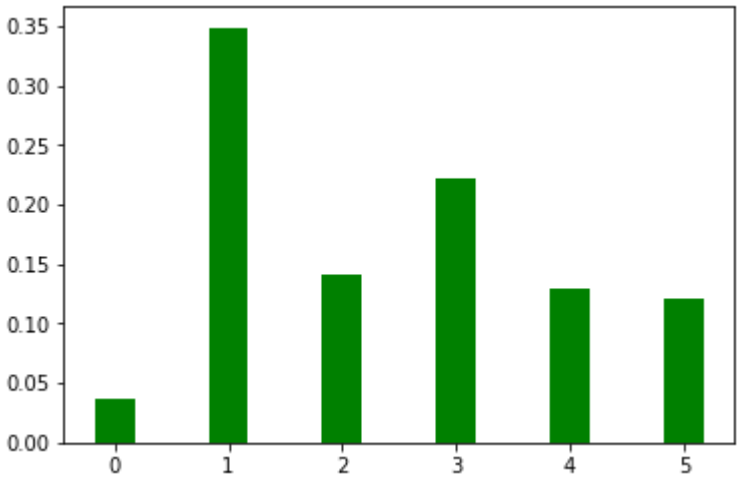


On voit que à partir d'un certain nombre de composantes, le log-vraisemblance augmente, mais de manière plus lente.

Examinons la répartition des poids et des composantes sur un exemple ainsi que la densité estimée:

```
gmm = GaussianMixture(n_components=6,n_init=3).fit(X)

# calcul de la densité pour les données de X_plot
density = np.exp(gmm.score_samples(X_plot))

# affichage : vraie densité et estimation
fig, ax = plt.subplots()
ax.fill(X_plot[:,0], true_density, fc='b', alpha=0.2, label='Vraie densité')
ax.plot(X_plot[:,0], density, '-', label="Estimation")
ax.plot(X[:, 0], -0.005 - 0.01 * np.random.random(X.shape[0]), '+k')
for mean in gmm.means_:
    ax.axvline(x=mean)
ax.legend(loc='upper left')
plt.show()

plt.bar(np.arange(6), gmm.weights_, 0.35, color='g')
plt.show()
```



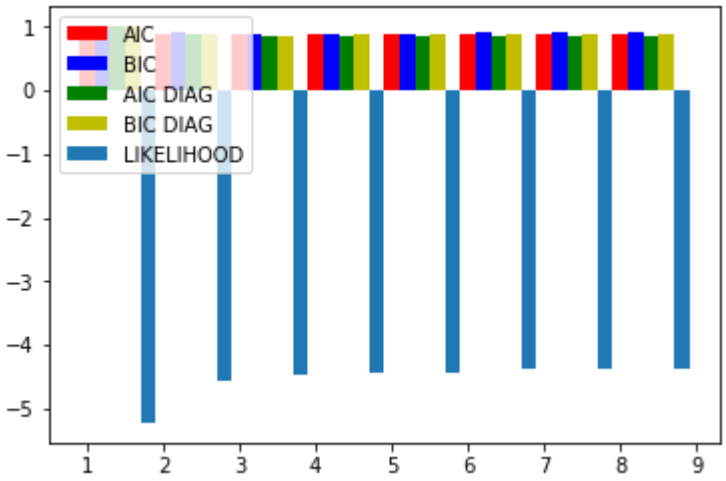


Ainsi, quand il y a trop de composantes, l'estimateur apprend les particularités de l'échantillon et n'arrive pas à converger.

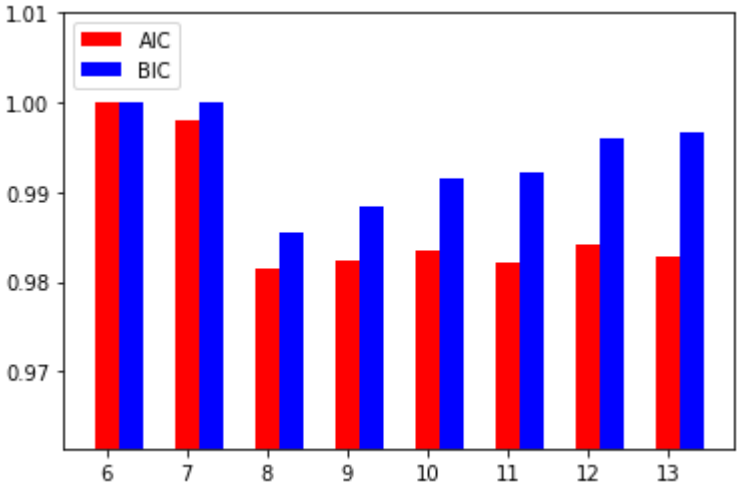
Réalisez la même comparaison pour ces données avec des matrices de covariances 'diag'.

```
n_max = 8    # nombre de valeurs pour n_components
n_components_range = np.arange(n_max)+1
aic_diag = []
bic_diag = []
likelihoods = []

# construction des modèles et calcul des critères
for n_comp in n_components_range:
    gmm = GaussianMixture(n_components=n_comp, covariance_type='diag').fit(md)
    aic_diag.append(gmm.aic(md))
    bic_diag.append(gmm.bic(md))
    likelihoods.append(gmm.lower_bound_)

# normalisation des résultats obtenus pour les critères
raic_diag = aic_diag/np.max(aic_diag)
rbic_diag = bic_diag/np.max(bic_diag)

# affichage
xpos = np.arange(n_max)+1  # localisation des barres
largeur = 0.2              # largeur des barres
#plt.ylim([min(np.concatenate((rbic,raic)))-0.1, 1.1])
```

```
plt.bar(xpos, raic, largeur, color='r', label="AIC")
plt.bar(xpos+largeur, rbic, largeur, color='b', label="BIC")
plt.bar(xpos+2*largeur, raic_diag, largeur, color='g', label="AIC DIAG")
plt.bar(xpos+3*largeur, rbic_diag, largeur, color='y', label="BIC DIAG")
plt.bar(xpos+4*largeur, likelihoods, largeur, label="LIKELIHOOD")
plt.legend(loc='upper left')
plt.show()
```

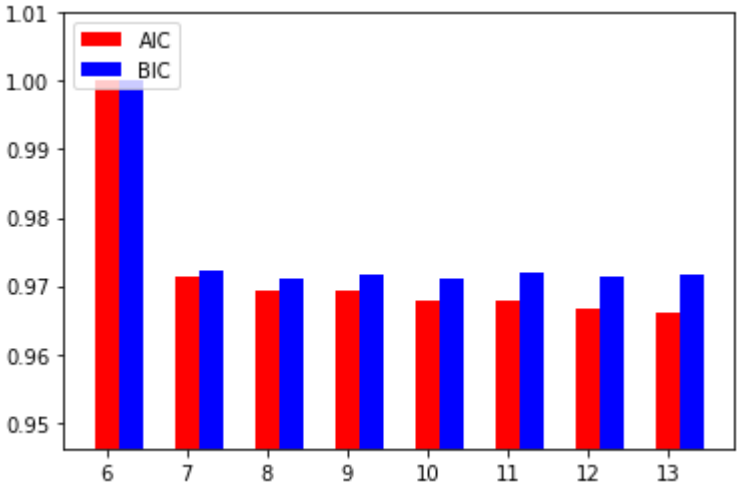


Comment expliquez-vous que la valeur optimale obtenue avec BIC pour n_components soit inférieure au nombre de classes de textures (qui est de 11) ?



Le nombre de classes est de 11. Cela indique que certaines classes peuvent être regroupées

Réalisez la même étude en utilisant plutôt les projections des données « textures » sur les deux premiers axes discriminants. Comment expliquez-vous la différence par rapport au cas précédent ?



Dans le cas de l'ACP, on cherche à maximiser la variance sans prendre en compte la présence de classes. Les données seront globalement mélangées. Dans le cas de l'AFD, on cherche à séparer au mieux les données. Ce qui fait qu'une nouvelle composante diminuera la variance globale.