**TELECOM ParisTech** **MDI343**
**2015/2016** **Alexandre Gramfort, Joseph Salmon**

---

**TP n° 6 : Clustering**

---

- Introduction -

Le but de cette séance est la mise en œuvre d'algorithmes de *clustering*. Le *clustering* est un problème non-supervisé. Étant donné un ensemble d'apprentissage $\mathbf{x}_1, \dots, \mathbf{x}_n$ de points dans $\mathbb{R}^d$, que l'on regroupe potentiellement dans une matrice $X \in \mathbb{R}^{n \times p}$, le but est de regrouper les points par groupes (si possible homogène), ou *clusters*. Notez qu'à la différence d'un problème supervisé il n'y a pas d'étiquettes associées dans ce contexte. Nous n'aborderons ici que quelques méthodes, mais un panorama plus exhaustif est disponible sur la page : http://scikit-learn.org/stable/modules/clustering.html.

- Mise en œuvre -

Les algorithmes de *clustering* de `scikit-learn` sont disponibles dans le module *sklearn.cluster*. Les modèles de mélanges Gaussiens (*Gaussian Mixture Models* ou GMM) peuvent être vus comme des estimateurs de densité et sont disponibles dans le module `sklearn.mixture`, mais ils peuvent aussi servir pour faire du *clustering*.

## $K$-Means

L'algorithme $K$-Means partitionne les points en $K$ groupes disjoints $\{\mathcal{C}_1, \dots, \mathcal{C}_K\}$ en minimisant la variance intra-cluster. Le critère minimisé est appelé l'*inertie*, qui est définie comme suit :

$$I_K = \sum_{k \in [\![1,K]\!]} \sum_{i \in \mathcal{C}_k} \|\mathbf{x}_i - \mu_k\|_2^2$$

où les $\mu_k \in \mathbb{R}^d$ sont les centroïdes des $K$ classes ($|\mathcal{C}_k|$ le cardinal de chaque classe) :

$$\mu_k = \frac{1}{|\mathcal{C}_k|} \sum_{i \in \mathcal{C}_k} \mathbf{x}_i, \quad \forall k \in [\![1, K]\!] \ ,$$

et où la $i^{\text{e}}$ observation a pour classe celle du centroïde le plus proche, *i.e.,* $i \in \mathcal{C}_{k_o}$ si et seulement si :

$$k_o = \underset{k \in [\![1,K]\!]}{\arg\min} \|\mathbf{x}_i - \mu_k\|_2 = \underset{k \in [\![1,K]\!]}{\arg\min} \|\mathbf{x}_i - \mu_k\|_2^2 \quad ,$$

et si besoin, les ex-æquo sont départagés aléatoirement.

L'apprentissage des *clusters* se fait en alternant itérativement les deux étapes suivantes :

I) une étape d'assignation où, connaissant les $(\mu_k)_{k=1,\dots,K}$, on détermine les labels de chaque point.

II) une étape de mise à jour des centroïdes, où connaissant les labels, on détermine les centres de classe.

On arrête l'algorithme quand l'inertie ne décroît plus beaucoup (il est à noter que sans critère d'arrêt l'algorithme $K$-Means termine de toute manière en un nombre fini d'étape *cf.* [BMDG05]).

L'inertie est un critère non-convexe, la solution trouvée dépend donc de l'initialisation, comme souvent pour de tels problèmes. En conséquence, l'algorithme est souvent lancé plusieurs fois avec des initialisations différentes, et l'on ne garde alors que la solution dont l'inertie est la plus faible.

Pour plus d'information sur l'algorithme $K$-means et ses généralisations on pourra consulter [BMDG05], ainsi que la variante (en terme d'initialisation) $K$-means++ [AV07].

1) En vous basant sur le squelette de code de la page suivante (extrait de `TP_clustering_kmeans.py`), implémentez l'algorithme $K$-Means. Seule la fonction *compute_ inertia_ centroids* est à compléter.

2) L'inertie décroît-elle bien au cours des itérations ?

3) En faisant varier l'initialisation du générateur de nombres aléatoires observez que la solution trouvée n'est pas toujours la même.

4) Comparez le résultat avec celui fourni par l'implémentation de `scikit-learn`.

```
from sklearn import cluster
kmeans = cluster.KMeans(n_clusters=3, n_init=10)
kmeans.fit(X)
labels = kmeans.labels_
```

```
# type here missing imports
import numpy as np

def generate_data(n, centroids, sigma=1., random_state=42):
    """Generate sample data

    Parameters
    ----------
    n : int
        The number of samples
    centroids : array, shape=(k, p)
        The centroids
    sigma : float
        The standard deviation in each class

    Returns
    -------
    X : array, shape=(n, p)
        The samples
    y : array, shape=(n,)
        The labels
    """
    rng = np.random.RandomState(random_state)
    k, p = centroids.shape
    X = np.empty((n, p))
    y = np.empty(n)
    for i in range(k):
        X[i::k] = centroids[i] + sigma * rng.randn(len(X[i::k]), p)
        y[i::k] = i
    order = rng.permutation(n)
    X = X[order]
    y = y[order]
    return X, y


def compute_labels(X, centroids):
    """Compute labels

    Parameters
    ----------
    X : array, shape=(n, p)
        The samples

    Returns
    -------
    labels : array, shape=(n,)
        The labels of each sample
```

```
    """
    dist = spatial.distance.cdist(X, centroids, metric='euclidean')
    return dist.argmin(axis=1)


def compute_inertia_centroids(X, labels):
    """Compute inertia and centroids

    Parameters
    ----------
    X : array, shape=(n, p)
        The samples
    labels : array, shape=(n,)
        The labels of each sample

    Returns
    -------
    inertia: float
        The inertia
    centroids: array, shape=(k, p)
        The estimated centroids
    """
    # insert code here
    return inertia, centroids


def kmeans(X, n_clusters, n_iter=100, tol=1e-7, random_state=42):
    """K-Means : Estimate position of centroids and labels

    Parameters
    ----------
    X : array, shape=(n, p)
        The samples
    n_clusters : int
        The desired number of clusters
    tol : float
        The tolerance to check convergence

    Returns
    -------
    centroids: array, shape=(k, p)
        The estimated centroids
    labels: array, shape=(n,)
        The estimated labels
    """
    # initialize centroids with random samples
    rng = np.random.RandomState(random_state)
    centroids = X[rng.permutation(len(X))[:n_clusters]]
    labels = compute_labels(X, centroids)
    old_inertia = np.inf
    for k in xrange(n_iter):
        inertia, centroids = compute_inertia_centroids(X, labels)
        if abs(inertia - old_inertia) < tol:
            print 'Converged !'
            break
        old_inertia = inertia
        labels = compute_labels(X, centroids)
```

```
        print inertia
    else:
        print 'Dit not converge...'
    return centroids, labels


if __name__ == '__main__':
    n = 1000
    centroids = np.array([[0., 0.], [2., 2.], [0., 3.]])
    X, y = generate_data(n, centroids)

    centroids, labels = kmeans(X, n_clusters=3, random_state=42)
    # plotting code
    import matplotlib.pyplot as plt
    plt.close('all')
    plt.figure()
    plt.plot(X[y == 0, 0], X[y == 0, 1], 'or')
    plt.plot(X[y == 1, 0], X[y == 1, 1], 'ob')
    plt.plot(X[y == 2, 0], X[y == 2, 1], 'og')
    plt.title('Ground truth')

    plt.figure()
    plt.plot(X[labels == 0, 0], X[labels == 0, 1], 'or')
    plt.plot(X[labels == 1, 0], X[labels == 1, 1], 'ob')
    plt.plot(X[labels == 2, 0], X[labels == 2, 1], 'og')
    plt.title('Estimated')
    plt.show()
```

## Calcul optimal du nombre de clusters (Partie optionnelle)

Dans la section précédente, nous avons supposé que le nombre de clusters $K$ était fixé par l'utilisateur. Néanmoins, il peut être intéressant en pratique de déterminer ce nombre en fonction des données. Nous proposons de définir une méthode pour le calculer, basée sur une "statistique de saut" (en : *gap statistic*) [TWH01].

On s'intéresse toujours à un jeu de données $X \in \mathbb{R}^{n\times p}$ ($n$ points dans $\mathbb{R}^p$). On note $X^{(b)}, t \in [\![1, B]\!]$, une matrice de taille $n \times p$, obtenues en tirant uniformément et indépendamment $n$ vecteurs de $\mathbb{R}^p$ dans l'ensemble $[\min(X_{:,1}), \max(X_{:,1})] \times \cdots \times [\min(X_{:,p}), \max(X_{:,p})]$ (*i.e.,* on tire uniformément dans le plus petit hypercube qui contient toutes les $n$ observations initiales). À partir des inerties $I_X(K)$ et $I_{X^{(b)}}(K)$ des $K$-Means appliqués à $X$ et à $X^{(b)}$,on choisit comme nombre de clusters, le plus petit entier $k$ tel que :

$$\mathrm{Gap}(k) \geq \mathrm{Gap}(k+1) - \sigma(k+1) \ ,$$

où $\mathrm{Gap}(k) \in \mathbb{R}$ correspond à la différence entre la moyenne du log des inerties $I_{X^{(b)}}(k)$ et le log de l'inertie $I_X(k)$ :

$$\mathrm{Gap}(k) = \frac{1}{B}\sum_{b=1}^{B}[\log(I_{X^{(b)}}(k))] - \log(I_X(k)) \ ,$$

et où $\sigma(k+1)$ est défini par :

$$\sigma(k+1) = \sqrt{\frac{B+1}{B}}\sqrt{\frac{1}{B}\sum_{b=1}^{B}\left[\log(I_{X^{(b)}}(k+1)) - \frac{1}{B}\sum_{b'=1}^{B}\log(I_{X^{(b')}})(k+1)\right]^2} \ .$$

En vous basant sur le squelette de code fourni dans le fichier `gap.py` :

5) Complétez la fonction `compute_log_inertia_rand` dans de sorte qu'elle estime la moyenne et l'écart type des $\log(I_{X^{(b)}})$.

6) Complétez la fonction `compute_gap` qui calcule, pour un jeu de donnée $X$ fixé, les valeurs de $\text{Gap}(k)$ et de $\delta(k)$ pour différentes valeurs de $k$. On prendra $B = 10$ pour le nombre de répétition.

7) Analyser les valeurs de $\text{Gap}(k)$ pour différentes valeurs de $k$. On sélectionne alors le $k$ qui maximise la statistique $\text{Gap}(k)$. Appliquer cette procédure avec un jeu de données `sklearn.datasets.make_blobs` qui permet de générer un ensemble de $K$ (*e.g.*, $K = 5$) clusters. On pourra aussi utiliser la fonction `plot_gap` pour visualiser correctement l'impact de $\text{Gap}(k)$.

8) Complétez la fonction `optimal_n_clusters_search` avec la méthode que vous avez proposée et vérifiez son fonctionnement sur différents jeux de données.

## Gaussian Mixture Models - GMM

Un des défauts de l'algorithme $K$-Means est qu'il produit des *clusters* taille et forme similaire : l'hypothèse implicite est que chaque *cluster* a la même variance. Les modèles de mélanges Gaussiens (GMM) permettent de relaxer cette hypothèse. En effet, on peut voire l'algorithme $K$-Means comme une version similaire de l'algorithme EM appliquée à un mélange de gaussiennes, dont les matrices de variance/covariance sont identiques (et scalaires). Un algorithme de type EM comporte principalement deux étapes alternées :

1) une étape d'estimation de moyenne (en anglais "E" renvoie à *expectation*)
2) une étape d'optimisation (en anglais "M" renvoie à *Maximization*) qui pour le $K$-Means est une étape de minimisation.

Pour des précisions sur l'interprétation des méthodes de type EM et des différences entre EM et $K$-Means on pourra consulter :

http://ttic.uchicago.edu/~dmcallester/ttic101-07/lectures/em/em.pdf.

Pour la version générale, chaque *cluster* est modélisé comme une distribution gaussienne dont la (matrice de) covariance est estimée au cours des itérations. Afin de mettre en évidence la capacité des GMM à estimer la covariance des classes, exécutez l'exemple suivant :

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn import mixture

n_samples = 300
np.random.seed(0)
C = np.array([[0., -0.7], [3.5, .7]])
X = np.r_[np.dot(np.random.randn(n_samples, 2), C),
          np.random.randn(n_samples, 2) + np.array([20, 20])]

clf = mixture.GaussianMixture(n_components=2, covariance_type='full')
clf.fit(X)
labels = clf.predict(X)
x = np.linspace(-20.0, 30.0)
y = np.linspace(-20.0, 40.0)
XX, YY = np.meshgrid(x, y)
Z = np.log(-clf.score_samples(np.c_[XX.ravel(), YY.ravel()])).reshape(XX.shape)
plt.close('all')
CS = plt.contour(XX, YY, Z)
CB = plt.colorbar(CS, shrink=0.8, extend='both')
plt.plot(X[labels == 0, 0], X[labels == 0, 1], 'or')
plt.plot(X[labels == 1, 0], X[labels == 1, 1], 'ob')
plt.axis('tight')
plt.show()
```

3) Les GMMs sont des modèles probabilistes d'estimation de densité. Il est possible de calculer la vraisemblance d'un ensemble de points. En exploitant la méthode *score* de l'objet *GMM* qui calcule la log-vraisemblance des observations, estimer le nombre de classes par validation croisée (on pourra retenir une moitié des observations pour l'entraînement, et l'autre moitié pour le test).

4) Par ailleurs, les critères AIC et BIC permettent d'évaluer la pertinence d'un modèle, sans étape de validation croisée. Les deux critères comportent un terme mesurant la qualité de l'ajustement (le maximum de vraisemblance au sein du modèle), pénalisé par un terme mesurant la complexité du modèle (la dimension de l'espace des paramètres aussi appelé nombre de degrés de liberté du modèle). Ce terme de pénalité permet de prendre en compte un éventuel sur-apprentissage du modèle, de sorte que l'AIC et le BIC peuvent être calculés sur le jeu de données d'entraînement. Ces critères sont implémentés dans les méthodes `bic` et `aic` de l'objet `GMM`. En vous servant du code de `scikit-learn` donnez les formules des deux critères. Pour plus de détails sur l'interprétation théorique de ces deux critères, le lecteur intéressé pourra consulter http://www.cbcb.umd.edu/~hcorrada/PracticalML/pdf/lectures/selection.pdf.

5) Afficher les critères AIC d'une part, et BIC d'autre part en fonction du nombre de classes. Que peut-on en conclure sur le nombre de classes ? Comparez les résultats obtenus respectivement avec le BIC, l'AIC et la validation croisée.

### Application à la compression d'images

Les algorithmes de *clustering* peuvent aussi servir à la quantification vectorielle. Dans une image couleur, chaque pixel est codé sur 3 valeurs (rouge, vert et bleu, RGB en anglais), chacune codée sur un octet (entier entre 0 et 255). L'idée de la quantification vectorielle est de coder chaque pixel sur un nombre possible de couleurs $K$, bien plus faible que les $256^3$ valeurs possible. L'image peut ainsi être codée de façon plus compacte.

6) Appliquer les algorithmes $K$-Means et GMM pour réduire le nombre de niveaux de couleurs de l'image `china.jpg` à 3 valeurs possibles.

7) Réutiliser les trois méthodes de sélection de modèle présentées précédemment pour les GMM pour estimer le nombre de classes. Les trois méthodes sont-elles en accord ?

8) Expliquer le paramètre `covariance_type` de l'estimateur `GMM`. Les conclusions de la question précédente sont-elles impactées par ce paramètre ? Une représentation graphique claire est idéale pour répondre à cette question.

**Indication :** Pour lire l'image utiliser la commande `img = scipy.ndimage.imread('china.jpg')`. Pour utiliser un estimateur de scikit-learn il faudra effectuer un `reshape` de l'image afin de travailler avec un `array` à 2 dimensions de taille "nombre de pixels fois trois", car il y a trois couleurs pour un codage RGB (Rouge/Vert/Bleu).

## Références


[AV07] D. Arthur and S. Vassilvitskii. k-means++ : The advantages of careful seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1027–1035. Society for Industrial and Applied Mathematics, 2007. 1

[BMDG05] A. Banerjee, S. Merugu, I. S. Dhillon, and J. Ghosh. Clustering with Bregman divergences. *J. Mach. Learn. Res.*, 6 :1705–1749, 2005. http://machinelearning.wustl.edu/mlpapers/paper_files/BanerjeeMDG05.pdf. 1

[TWH01] R. Tibshirani, G. Walther, and T. Hastie. Estimating the number of clusters in a data set via the gap statistic. 63(2) :411–423, 2001. 4