Université de Montpellier **HMMA308**
2018/2019 **Alexandre Gramfort, Thierry Guillemot, Joseph Salmon**

TP n° 5 : Clustering

Pour ce travail vous devez déposer un **unique** fichier au format `nom_prenom.ipynb` sur le site moodle du cours (si moodle ne l'accepte pas, zipper cet unique fichier en fichier .zip)

Vous devez charger votre fichier sur Moodle, avant le vendredi 19/09/2018, 23h55.

La note totale est sur **20** points répartis comme suit :

— qualité des réponses aux questions : **15** pts,

— qualité de rédaction, de présentation et d'orthographe : **2** pts,

— indentation, style PEP8, commentaires adaptés, etc. : **2** pts,

— absence de bug : **1** pt.

Les personnes qui n'auront pas rendu leur devoir avant la limite obtiendront **zéro**.

Rappel : aucun travail par mail ne sera accepté !

- 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 $\mathbb{R}^d$, le but est de regrouper les points par groupes (si possible homogènes), ou *clusters*. Notez qu'à la différence d'un problème supervisé il n'y a pas d'étiquettes associées dans ce contexte.

- 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

Dans cette section on s'intéressera aux algorithmes où le nombre de *clusters* (noté K dans la suite) est fixé au préalable par l'utilisateur.

L'algorithme de 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 G est appelé l'*inertie* :

$$G_X(\mathcal{C}_1, \dots, \mathcal{C}_K) = \sum_{k=1}^{K} \sum_{i \in \mathcal{C}_k} \|\mathbf{x}_i - \mu_k\|_2^2$$

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

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

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

$$j = \underset{k \in [K]}{\arg\min} \|\mathbf{x}_i - \mu_k\|_2 = \underset{k \in [K]}{\arg\min} \|\mathbf{x}_i - \mu_k\|_2^2 \quad .$$

Rem : on départage les ex-æquo aléatoirement si besoin.

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].

1) En vous basant sur le squelette de code dans le fichier `kmeans.py`, implémentez l'algorithme K-Means. Le fichier termine avec un exemple de code qui vous servira de validation. Il utilise la fonction `make_blobs` qui permet de simuler un dataset.
2) L'inertie décroît-elle bien au cours des itérations ? Vous afficherez la décroissance de l'inertie en fonction des itérations.
3) En faisant varier l'initialisation du générateur de nombres aléatoires, observez que la solution trouvée n'est pas toujours la même. Vous pourrez augmenter le nombre de clusters simulés pour mettre d'avantage en évidence le phénomène.
4) Comparez votre implémentation avec celle de `scikit-learn`, en terme de résultat et temps de calcul. Vous pourrez utiliser la commande `%%timeit` d'IPython pour évaluer le temps de calcul d'une cellule du notebook. Voici un exemple d'utilisation de `scikit-learn` :

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

Calcul optimal du nombre de cluster

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 le nombre optimal de clusters. Nous proposons de définir une méthode pour le calculer.

On s'intéresse toujours à un jeu de données X avec n points. Soit $X_t, t \in [T]$, T tirages indépendants de n points aléatoires tirés selon une distribution uniforme dans la boite englobante de X. À partir des inerties G_X et G_{X_t} des K-means appliqués à X et à X_t, nous définissons la différence :

$$\delta(k) = \mathrm{Gap}(k) - (\mathrm{Gap}(k+1) - \sigma(k+1))$$

où $\mathrm{Gap}(k) \in \mathbb{R}$ correspond à la différence entre l'espérence du log des inerties de G_{X_t} et le log de l'inertie G_X :

$$\mathrm{Gap}(k) = \mathbb{E}[\log(G_{X_t})] - \log(G_X)$$

et où $\sigma(k)$ est défini par :

$$\sigma(k) = \sqrt{\frac{T+1}{T}}\mathrm{std}(\log(X_t))$$

avec std qui reprensente l'ecart-type.

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

5) Complétez la fonction `compute_log_inertia_rand` de sorte qu'elle estime la moyenne et l'écart type des $\log(G_{X_t})$.
6) Complétez la fonction `compute_gap` qui calcule, pour un jeu de donnée X fixé, les valeurs de $\mathrm{Gap}(k)$ et de $\delta(k)$ pour différentes valeurs de k.
7) Analyser les valeurs de $\mathrm{Gap}(k)$ et de $\delta(k)$ pour différentes valeurs de k et proposez une méthode pour permettre de définir automatiquement le nombre K des K-means. Vous utiliserez la fonction `sklearn.datasets.make_blobs` qui permet de générer un ensemble de K clusters et la fonction `plot_gap` pour visualiser correctement vos différentes données.
8) Complétez la fonction `optimal_n_clusters_search` avec la méthode que vous avez proposée et vérifiez son fonctionnement sur différents jeux de données.

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 seulement K couleurs, avec K bien plus faible que les 256^3 valeurs possible. L'image peut ainsi être codée de façon plus compacte.

Nous proposons de comparer les résultats obtenus avec différentes techniques de compression :

9) Appliquer l' algorithme K-Means pour réduire le nombre de niveaux de couleurs de l'image à K=8 valeurs possibles. Visualiser le résultat sous forme d'une image. A partir de quelle valeur de K ne voit-on plus de différence avec l'image d'origine ?

10) Appliquer l' algorithme Spectral Clustering pour réduire le nombre de niveaux de couleurs de l'image à 8 valeurs possibles. Attention, pour des raisons de performances et d'utilisation mémoire, il faut absolument utiliser la version du Spectral Clustering utilisant les matrices creuses. Visualisez et comparez les résultats obtenus avec K-Means.

Indication : Pour lire l'image utiliser la commande `img = scipy.ndimage.imread('china.jpg')`. Pour utiliser vos estimateurs 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.

Références

[BMDG05] A. Banerjee, S. Merugu, I. S. Dhillon, and J. Ghosh. Clustering with Bregman divergences. *J. Mach. Learn. Res.*, 6 :1705–1749, 2005. 2