TME 8 : Algorithme k-means

Le principe de l'algorithme k-means est de trouver une partition de l'espace d'entrée en considérant la densité d'exemples pour caractériser ces partitions. Un cluster C_i correspond à la donnée d'un prototype $\mu_i \in \mathbb{R}^d$ dans l'espace d'entrée. Chaque exemple x est affecté au cluster le plus proche selon une distance (euclidienne ou autre) entre l'exemple et le prototype du cluster. Soit $s_C : \mathbb{R} \to \mathbb{N}$ la fonction d'affectation associé au clustering $C = \{C_1, C_2, \ldots, C_k\}$: $s_C(x) = argmin_i \|\mu_i - x\|^2$. La fonction de coût sur un ensemble de données $\{x_1, \ldots, x_n\}$ généralement considérée dans ce cadre est la moyenne des distances intra-clusters : $\frac{1}{n}\sum_{i=1}^{n}\sum_{j|s_C(x_j)=i}\|\mu_i - x_j\|^2 = \frac{1}{n}\sum_{i=1}^{n}\|\mu_{s_C(x_i)} - x_i\|^2$. C'est également ce qu'on appelle le coût de reconstruction : effectivement, dans le cadre de cette approche, chaque donnée d'entrée peut être "représentée" par le prototype associé : on réalise ainsi une compression de l'information (n.b. : beaucoup de liens existent entre l'apprentissage et la théorie de l'information, la compression et le traitement de signal). L'algorithme fonctionne en deux étapes, (la généralisation de cet algorithme est appelé algorithme E-M, Expectation-Maximization) :

- à partir d'un clustering C^t, les prototypes $\mu_i^t = \frac{1}{|C_i|}\sum_{x_j \in C_i^t} x_j$, barycentres des exemples affectés à ce cluster ;
- à partir de ces nouveaux barycentres, calculer la nouvelle affectation (le prototype le plus proche).

Ces deux étapes sont alternées jusqu'à stabilisation.

Cet algorithme peut être utilisé pour faire de la compression d'image (connu également sous le nom de quantification vectorielle). Une couleur est codée par un triplet (r, g, b) qui dénote le mélange de composantes rouge, vert et bleu. En limitant le nombre de couleurs possibles dans l'image, on réalise une compression en limitant la longueur de codage de chaque couleur. L'objectif est de trouver quelles couleurs doivent être présentent dans notre *dictionnaire* afin de minimiser l'erreur entre l'image compressée et l'image originale (remarque : c'est exactement l'erreur de reconstruction ci-dessus). En considérant l'ensemble des pixels de l'image comme la base d'exemples non supervisée, le nouveau codage de chaque pixel peut être obtenu par le résultat de l'algorithme k-means sur cette base d'exemple.

Le bout de code suivant permet de lire, afficher, modifier, sauver une image au format `png` et de la stocker dans un tableau de taille $l \times h \times c$, l la largeur de l'image, h la hauteur, et c 3 généralement pour les 3 couleurs (parfois 4, la 4eme dimension étant pour la transparence, non utilisée ici).

```
1 import matplotlib.pyplot as plt
2
3 im=plt.imread("fichier.png")[:,:,:3] #on garde que les 3 premieres composantes, la
      transparence est inutile
4 im_h,im_l,_=im.shape
5 pixels=im.reshape((im_h*im_l,3)) #transformation en matrice n*3, n nombre de pixels
6 imnew=pixels.reshape((im_h,im_l,3)) #transformation inverse
7 plt.imshow(im) #afficher l'image
```

Implémentez l'algorithme k-means. Expérimentez la compression : choisissez une image, construisez avec k-means l'image compressée et affichez là. Etudier en fonction du nombre de clusters (couleurs) choisis comment évolue l'erreur de reconstruction.

Quel est le gain en compresssion effectué ?

Sachant que souvent une image peut être découpée en région de tonalité homogène, voyez-vous une amélioration possible pour augmenter la compression tout en diminuant l'erreur de compression ?