DUCHANOY Colin
LANCERY Hugo
QUIN Hugo
SOUCOURRE Vincent

IPSA
ÉCOLE D'INGÉNIEURS
DE L'AIR, DE L'ESPACE ET DE LA MOBILITÉ DURABLE

Bureau d'étude Ma325 : Variations sur l'astuce des noyaux

AERO 3

M. COUFFIGNAL – M. EL MAHBOUBY

Mars 2022

Table des matières

.....

Exercice 1 : Reconstruction de surface avec les moindres carrées à noyaux

Dans cet exercice nous allons utiliser l'astuce des noyaux afin de reconstruire une surface (2D) à partir de la donnée des points de $\mathbb{R}^3$.
Supposons donné un ensemble (sous la forme d'une matrice) de points de m points de $\mathbb{R}^3$:

$$P=\begin{pmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ : & : & : \\ x_m & y_m & z_m \end{pmatrix}$$

Commençons par écrire la forme « linéaire » de la recherche d'une surface passant « au plus près de ces points ». On cherche donc un plan :

$$II: \alpha x + \beta y + \gamma z + \delta = 0$$

Où les (x, y, z) parcours $\mathbb{R}^3$.

Pour python nous prendrons la matrice P suivante :

$$P=\begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 1.5 \\ 0 & 1 & 0.5 \\ 1 & 1 & 1 \\ 0.5 & 0.5 & 0 \end{pmatrix}$$

1. Ecriture de la forme « linéaire »
   a) Quels types de plans l'hypothèse : « $\gamma \neq 0$ » supprime ?

Si $\gamma \neq 0$, cela signifie que tous les plans tel que :

$$\alpha x + \beta y + \delta = 0$$

Ainsi tous les plans parallèles à l'axe (Oz) ne sont pas solutions de ce problème sont supprimés

   b) Problème de la recherche des coefficients a, b, d sous la forme d'un problème de moindre carrée

On cherche à écrire le problème sous la forme de moindre carrée :

$$X^* = \arg\ min \|AX - Z\|_2^2$$

On a :

$$X^* = \arg\ min \|ax + by + d - z\|_2^2$$

$$\Leftrightarrow X^* = \arg\ min \left\| \begin{pmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ : & : & : \\ x_m & y_m & 1 \end{pmatrix} \begin{pmatrix} a \\ b \\ d \end{pmatrix} - \begin{pmatrix} z_1 \\ z_2 \\ : \\ z_m \end{pmatrix} \right\|_2^2$$

$$\Leftrightarrow X^* = \arg\ min \|AX - Z\|_2^2$$

On retrouve bien la forme souhaitée avec :

$$A = \begin{pmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ : & : & : \\ x_m & y_m & 1 \end{pmatrix}, X = (a, b, d)^T \; et \; Z = (z_1, z_2, .., z_m)^T$$

c) Rédaction du programme Python

i. Fonction Python : *MCsurface(P)*

Dans cette partie nous allons implémenter une fonction qui nous renvoie une solution au sens des moindres carrés :

$$X^* = (a^*, b^*, d^*)^T$$

Ainsi que l'erreur commise : $\|AX^* - Z\|$

```
def MCsurface(P):

    A=np.ones((np.shape(P))) #Création d'une matrice de 1 de shape (5,3)
    #On construit A en récupérant les 2 premières colonnes
    A[:,:2]=P[:,:2]
    Z=P[:,-1:] #On construit Z en récupérant la dernière colonne

    solMC=np.linalg.pinv(A)@Z #Calcul de la solution classique
    erreur=np.linalg.norm(A@solMC-Z) #Calcul de l'erreur
    print("erreur MCsurface classique: ", erreur)

    return solMC , erreur
```

Tout d'abord on crée une matrice A de même taille que P, puis on lui ajoute les valeurs. Les deux premières colonnes de A seront les mêmes que P.
On construit Z qui sera sous la forme d'un vecteur colonne récupérant la dernière colonne de P.
En utilisant la fonction `np.linalg.pinv(A)@Z` on peut calcule la solution du problème.
Puis on détermine l'erreur commise avec `np.linalg.norm(A@solMC-Z)`

ii. Fonction Python : *Visualisation(P,X,nb)*

```
def Visualisation(P,X,nb):

    n,m=np.shape(P) #Récupération de la shape de P
    #Descrétisation de x
    discreteX=np.linspace(np.min(P[:,0])-0.5,np.max(P[:,0])+0.5,nb)
    #Descrétisation de y
    discreteY=np.linspace(np.min(P[:,1])-0.5,np.max(P[:,1])+0.5,nb)
    #Passage en meshgrid pour le tracé
    discreteX,discreteY=np.meshgrid(discreteX,discreteY)

    #Calcul de la surface solution
    fxy=(X[0]*discreteX+X[1]*discreteY+X[2]).T

    #TRACE DE LA SURFACE ET DES POINTS
    fig = plt.figure()
    ax = fig.add_subplot(projection='3d')
    surf = ax.plot_surface(discreteX,discreteY,fxy,
                label='Solution MC classique', color='blue', alpha=0.2)
    #erreur de la fonction legend en 3d avec une surface
    #cette solution vient de stackflow
    surf._facecolors2d=surf._facecolor3d
    surf._edgecolors2d=surf._edgecolor3d
    ax.legend()
    for i in range(n):
        ax.scatter(P[i][0],P[i][1],P[i][2],s=20)
    plt.show()

    return()
```

2. Généralisation du procédé précédent

On s'autorise à avoir des surfaces non-linéaires grâce à l'astuce des noyaux.

a) Utilisation de l'astuce des noyaux pour transformer notre problème :

Utilisons l'astuce des noyaux pour transformer notre problème : Supposons donné k : $\mathbb{R}^3 \times \mathbb{R}^3 \rightarrow \mathbb{R}$ un noyau.

Alors d'après le théorème de Mercer il existe un espace de Hilbert H muni d'un produit scalaire et une application ϕ: $\mathbb{R}^3 \rightarrow$H tels que :

$$k(x,y) = \langle \phi(x), \phi(y) \rangle \qquad \forall (x,y) \epsilon\, \mathbb{R}^3 \times \mathbb{R}^3$$

Ainsi le problème se récrit sous la forme

$$c^* = \arg min \sum_{i=1}^{m} \left(\langle R, \phi(\omega_i) \rangle - z_i \right)^2$$

Avec

$$R = \sum_{i=1}^{m} c_i \, k_{\omega i}$$

Et $\omega_i = (x_i, y_i, 1)^T$ et $c^* = \left({c_1}^*, {c_2}^*, \dots\dots, {c_m}^*\right)$

b) Ecriture matricielle sous la forme : $\|K\,c - Z\|_2^2$

On sait que

$$c^* = \arg min \sum_{i=1}^{m} \left(\langle R, \phi(\omega_i)\rangle - z_i\right)^2$$

Avec

$$R = \sum_{i=1}^{m} c_i \, k_{\omega i}$$

On a alors :

$$\sum_{i=1}^{m} \left(\langle R, \phi(\omega_i)\rangle - z_i\right)^2$$

$$\Leftrightarrow \sum_{i=1}^{n} \left(\left\langle \sum_{j=1}^{m} c_j \, k_{\omega j} \, , \phi(\omega_i) \right\rangle - z_i \right)^2$$

$$\Leftrightarrow \sum_{i=1}^{n} \left(\sum_{j=1}^{m} c_j \left\langle k_{\omega j} \, , \phi(\omega_i) \right\rangle - z_i \right)^2$$

Or : $\left\langle k_{\omega j} \, , \phi(\omega_i) \right\rangle = K(\omega_i, \omega_j)$

$$\Leftrightarrow \sum_{i=1}^{n} \sum_{j=1}^{m} k_{ij} \, c_j \, - z_i$$

Exemple pour j=1 on aurait :

$$k_{i1} \, c_1 + k_{i2} \, c_2 + \cdots + k_{im} \, c_m \, = \begin{pmatrix} k_{11} & k_{12} & \dots & k_{1m} \\ k_{21} & k_{22} & \dots & k_{2m} \\ : & : & : & : \\ k_{n1} & k_{n2} & \dots & k_{nm} \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ : \\ c_m \end{pmatrix}$$

Ainsi on obtient donc :

$$\Leftrightarrow \|K\,c - Z\|_2^2$$

Avec K la matrice de Gram $\begin{pmatrix} k_{11} & k_{12} & \dots & k_{1m} \\ k_{21} & k_{22} & \dots & k_{2m} \\ : & : & : & : \\ k_{n1} & k_{n2} & \dots & k_{nm} \end{pmatrix}$ et c=$(c_1, c_2, .., c_m)^T$

c) Ecriture de la fonction solution

$$f_k(x,y) = \left\langle R^*, \phi\left(\begin{pmatrix} x \\ y \\ 1 \end{pmatrix}\right)\right\rangle = \sum_{i=n}^{m} c_{i^*} k\left(w_j, \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}\right)$$

On a :

$$\left\langle R^*, \phi\left(\begin{pmatrix} x \\ y \\ 1 \end{pmatrix}\right)\right\rangle$$

Avec :

$$R^* = \sum_{i=n}^{m} c_i^* k_{\omega i}$$

$$\Leftrightarrow \left\langle \sum_{i=n}^{m} c_i^* k_{\omega i}, \phi\left(\begin{pmatrix} x \\ y \\ 1 \end{pmatrix}\right)\right\rangle$$

$$\Leftrightarrow \sum_{i=n}^{m} c_i^* \left\langle k_{\omega i}, \phi\left(\begin{pmatrix} x \\ y \\ 1 \end{pmatrix}\right)\right\rangle$$

$$\Leftrightarrow \sum_{i=n}^{m} c_i^* k(\omega i, \begin{pmatrix} x \\ y \\ 1 \end{pmatrix})$$

3. Résolution avec Python au sens des moindres avec noyau

a) Fonction Python : *MCKsurface(P,kern)*

Dans cette partie nous allons implémenter une fonction qui nous renvoie la solution au sens des moindres carres avec noyaux :

$$c^* = \left({c_1}^*, {c_2}^*, \dots\dots\dots, {c_m}^*\right)$$

Ainsi que l'erreur commise $\|Kc^* - Z\|$

```python
def MCKsurface(P,kern):

    n,m=np.shape(P) #Récupération de la shape de P
    A=np.ones((np.shape(P))) #Création d'une matrice de 1 de shape (5,3)
    #On construit A en récupérant les 2 premières colonnes
    A[:,:2]=P[:,:2]
    Z=P[:,-1:] #On construit Z en récupérant la dernière colonne

    K=np.zeros((n,n)) #initialisation de K
    for i in range(n) :
        for j in range(n) :
            #calcul des Kij avec un kernel
            K[i,j]=kern(A[i,:].reshape((3,1)),A[j,:].reshape((3,1)))

    solMCK=np.linalg.pinv(K)@Z #calcul de la solution avec noyau
    erreur=np.linalg.norm(K@solMCK-Z) #Calcul de l'erreur

    return solMCK , erreur
```

Dans un premier temps on recrée la matrice A et la matrice Z comme nous l'avons vu précédemment.
On crée la matrice K qui sera la matrice contenant les noyaux entre les différentes lignes de A.
On parcourt tout la matrice K puis on lui insère les valeurs.
Kern peut correspondre au noyau gaussien, polynomiale etc..

On peut ensuite calculer la solution au sens des moindre carres en faisant `np.linalg.pinv(K)@Z` ainsi que l'erreur commise avec `np.linalg.norm(K@solMCK-Z)`.

b) Fonction Python : *VisualisationK(P,c*,nb, kern)*

```
#Fonction calcul de la Matrice S (surface solution)
def fK(c_etoile,discreteX,discreteY,A,n,kern):
    s=0
    for i in range(n) :

s+=c_etoile[i]*kern(A[i].reshape((3,1)),np.array([discreteX,discreteY,1]
,dtype=object).reshape((3,1)))
    return s
```

```
def VisualisationK(P,c_etoile,nb, kern):
    n,m=np.shape(P) #Récupération de la shape de P
    A=np.ones((np.shape(P))) #Création d'une matrice de 1 de shape (5,3)
    A[:,:2]=P[:,:2] #On construit A en récupérant les 2 premières
colonnes
    discreteX=np.linspace(np.min(P[:,0])-0.5,np.max(P[:,0])+0.5,nb)
#Descrétisation de x
    discreteY=np.linspace(np.min(P[:,1])-0.5,np.max(P[:,1])+0.5,nb)
#Descrétisation de y
    discreteX,discreteY=np.meshgrid(discreteX,discreteY) #Passage en
meshgrid pour le tracé

    fK(c_etoile,discreteX,discreteY,A,n,kern)

    S=np.zeros(np.shape(discreteX))
    for i in range(np.shape(discreteX)[0]) :
            S[i]=fK(c_etoile, discreteX[i],discreteY[i],A,n,kern)

    fig = plt.figure()
    ax = fig.add_subplot(projection='3d')
    surf = ax.plot_surface(discreteX,discreteY,S,label='Solution avec
noyau', color='blue', alpha=0.2)
    surf._facecolors2d=surf._facecolor3d #erreur de la fonction legend
en 3d avec une surface
    surf._edgecolors2d=surf._edgecolor3d #cette solution vient de
stackflow
    ax.legend()
    for i in range(n):
        ax.scatter(P[i][0],P[i][1],P[i][2],s=20)

    plt.show()

    return
```

Dans cette fonction on récupère la matrice P et c* qui sera le résultat de *Mcksurface* puis on renvoie en sortie le tracé 3D de la surface solution.

c) Question défi

On recherche le noyau minimisant l'erreur commise pour l'ensemble de points contenu dans le fichier : « P_test.npy ».

```
#faisons une liste des kernels importé
Liste_kernels=[kern_gauss,kern_sigmo,kern_ratio_quad,kern_mquad,kern_inv
_mquad,kern_cauchy,kern_log]

for i in range (len(Liste_kernels)):
    b.append(MCKsurface(P,Liste_kernels[i])[1])
j=b.index(min(b)) #comparaison du meilleur kernel

VisualisationK(P,MCKsurface(P,Liste_kernels[j])[0],nb=20,
kern=Liste_kernels[j])
print("erreur optimale MCKsurface avec noyau: ",
MCKsurface(P,Liste_kernels[j])[1])
```

Nous avons rajouté dans le code un « ***.format( )*** » qui nous montre le nom du kernel qui minimise l'erreur. Il s'agit ici du noyau logarithmique.

4. Utilisation d'autres ensemble de points.

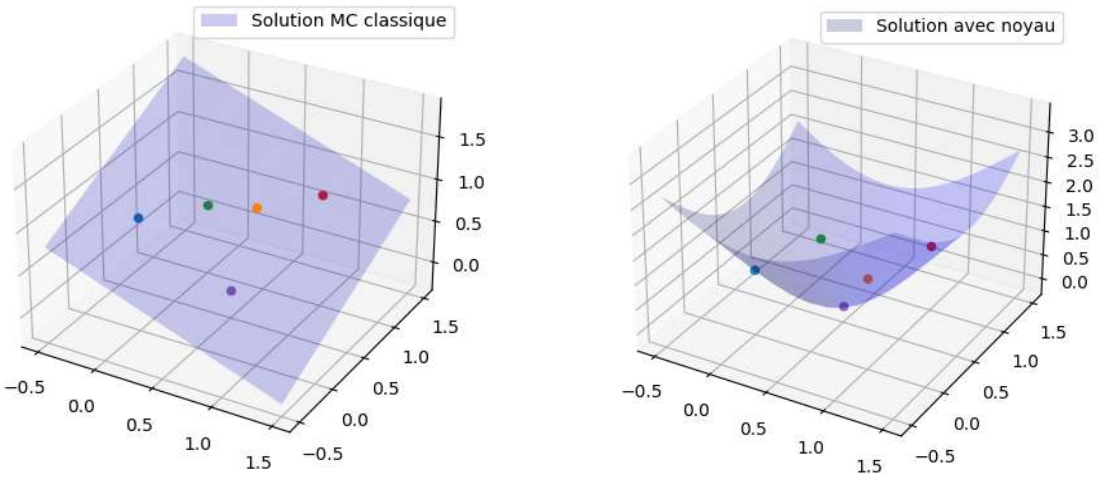


Figure 1- Comparaison entre les deux méthodes

On observe donc que la méthode des noyaux nous permet d'établir une surface approchant au plus près des points contrairement à la méthode des moindres carres.

Exercice 2 : Partitionnement d'image avec ACP et k-moyennes

Partie 1 : L'ACP le retour du retour du retour…

Notre objectif est d'utiliser l'ACP pour les pixels d'une image afin de la partitionner. Le problème de cette méthode est le grand nombre de pixels présents dans une image, ce qui rend la décomposition en éléments simples impossible pour nos ordinateurs. Nous chercherons donc une formulation de l'ACP différente de celle que nous utilisions jusqu'à présent, qui nécessite moins d'espace mémoire.
Pour ce faire, nous repartirons du travail effectué au cours de l'exercice 1 du TD2.
A partir de la matrice $X \in \mathcal{M}_{nd}(\mathbb{R})$ associée aux données du tableau étudié. Notons ind_i la ligne représentant les données d'un individu et X_i la colonne représentant un type de donnée.
Notons

$$ind_g = \frac{1}{n}\sum_{i=1}^{n} ind_i$$

Le vecteur ligne centre de gravité et la nouvelle matrice centrée :

$$X_c = \begin{pmatrix} ind_1 - ind_g \\ ind_2 - ind_g \\ \vdots \\ ind_n - ind_g \end{pmatrix}$$

Nous noterons ensuite X_{rc} la matrice centrée réduite des données tel que l'estimateur de covariance soit défini par :

$$C = {X_{rc}}^T X_{rc} \in \mathcal{M}_{d,d}(\mathbb{R})$$

1. Justifions que les directions principales sont données par les vecteurs propres de U dans la diagonalisation de la matrice de covariance $C = UDU^T$ si l'on range les valeurs propres de C par ordre décroissant.

Nous avons vu en MA314, que la matrice de covariance est symétrique donc C est diagonalisable dans une base orthonormée. Alors il existe une matrice P orthogonale, de passage et inversible telle que $P^{-1} = P^T$ et D une matrice diagonale vérifiant :

$$C = PDP^T$$

On définit la première composante principale comme le vecteur centré où :

$$v = arg\,max(x_c v)$$

On appelle un tel v, la direction principale. Ainsi on rappelle qu'il nous faut résoudre le problème suivant :

$$arg\,max(x_c v) = \arg\max\,(v^T C v)$$

En remplaçant l'expression de C, on obtient :

$$\arg\max(v^T C v)$$
$$\Leftrightarrow \arg\max(v^T P D P^T v)$$

On utilise le fait que P est orthogonale pour obtenir que :

$$v^T v = v^T P P^T v$$

$$\Leftrightarrow (P^T v)^T P^T v$$

On peut donc poser U=$P^T v$ et remplacer dans le problème précédent :

$$\arg\max(v^T P D P^T v)$$
$$\Leftrightarrow \arg\max(U^T D U)$$

On remarque donc ainsi que les directions principales sont données par les vecteurs propres de U dans la diagonalisation de la matrice de covariance.

De plus on la projection des données X sur les q-premiers axes principaux qui est donnée par :

$$X(q) = X_{rc} U_q$$

Où U_q est la matrice de taille (d, q) formée des q-premiers vecteurs colonnes de U

2. Démonstration de la diagonalisation à partir de la SVD de C

Nous savons que la matrice de covariance est une matrice symétrique ainsi en réalisant la décomposition SVD de cette matrice, on obtient $C^T C$ est une nouvelle matrice symétrique.
En recherchant les vecteurs propres de cette matrice on obtient la matrice V^T
On sait que :

$$C\ v_i = \sigma_i u_i$$

Avec u_i les vecteurs colonnes de U.

Comme C est une matrice symétrique alors on a :

$$U^T = V^T$$

Ainsi on obtient :

$$C = U \Sigma V^T = U \Sigma U^T$$

Avec Σ la matrice diagonale contenant les valeurs singulières de $C^T C$

3. Fonction Python : *ACP(X,q)*

Les matrices X proviennent des données de Iris et de Mnist
Pour cette partie nous ferons appel à des fonctions vues en MA 314 :
Esperance, variance et centre_red.

```
#%%Données Iris
Ebrut=np.genfromtxt("iris.csv" , dtype=str , delimiter=',') #données
brutes
labelscolonne=Ebrut[ 0 , : -1]
labelsligne=Ebrut[1 : , -1]
X=Ebrut[1 : , : -1].astype('float')
#%% Données Mnist
train_data=np.loadtxt('mnist_train.csv', delimiter=',')
#%%
X=(train_data[:,1:]).astype('float')
```

On cherche à réaliser l'ACP d'un tableau de données X et renvoyant la matrice $X(q)$
On prendra q donnée par la règle de Kaiser avec q le plus grand des indices tels que :

$$\lambda_q \geq \frac{1}{d} \sum_{\lambda \in Spec\,(C)} \lambda$$

On implémente dans un premier temps la règle de Kaiser afin de déterminer q.

```
def Kaiser(X):
    n,d=np.shape(X)
    C=Xrc.T@Xrc
    S=np.linalg.eigvals(C)
    lambdas=np.sum(S)
    q=0
    for i in range(np.shape(S)[0]):
        if S[i] >= (1/d)*lambdas:
            q=q+1
    return q
q=Kaiser(X)
```

Dans un premier temps on récupère les dimensions de notre matrice X. Ensuite crée une variable `C=Xrc.T@Xrc` (estimateur de covariance) obtenue en faisant `Xrc=centre_red(X)`.
On stocke dans S les valeurs propres de C avec la fonction. `np.linalg.eigvals(C)`
Ensuite on effectue la somme des valeurs propres (`lambdas=np.sum(S)`) puis on applique la règle de Kaiser avec une boucle comme défini au-dessus.
Si la condition est respectée alors l'algorithme nous renvoie la valeur de q.

```
def Acp(X,q):
    """
    Xrc=centre_red(X)
    C=Xrc.T@Xrc
    # On peut la méthode svd sur la matrice de covariance, après que X
    # ait été centrée et réduite.
    U,S,Vt=np.linalg.svd(C)
    Uq=U[:,:q]
    Xq=Xrc@Uq
    return Xq

Xq=Acp(X,q)
```

Puis on réalise l'ACP de notre tableau de données X. Cela nous donnera en sortie la projection des données X sur les q-premiers axes principaux.

4. Test de la robustesse de l'algorithme en utilisant des données déjà rencontrées : mnist_train.csv et iris.csv

En utilisant les bases de données mnist_train.csv et iris.csv, n voit que nos fonctions sont opérationnelles même pour des matrices de grande taille comme mnist_train (60000x784 valeurs). Elles sont donc assez robustes pour être utilisée sur des images.

5. Fonction Python : *ACPpond(X,q,W)*

Nous allons maintenant introduire une matrice diagonale :

$$\sqrt{W} = \begin{pmatrix} \sqrt{w_1} & & & \\ & \sqrt{w_1} & & \\ & & \backslash & \\ & & & \sqrt{w_d} \end{pmatrix}$$

Et modifier la matrice de covariance par une version pondérée :

$$C_w = \frac{1}{\sum_{i=1}^{d} w_i} (X_{rc})^T X_{rc} \sqrt{W}$$

```
    def ACPond(X,q,W)   :

     s=np.sum(W)
     Xrc=centre_red(X)
     Cw=(1/s)*(Xrc@W).T @ Xrc@W
     # Cw devient la nouvelle matrice de covariance de l'ACP, on effectue
ensuite
     # la même démarche que l'ACP classique.
     U,S,Vt=np.linalg.svd(Cw)
     Uq=U[:,:q]
     Xqp=Xrc@Uq
     return Xqp
```

On définit donc la fonction ACPond(X,q,W)
On réalise une décomposition SVD de notre matrice `Cw=(1/s)*(Xrc@W).T @ Xrc@W` avec la fonction `U,S,Vt=np.linalg.svd(Cw)`.
On détermine les `Uq=U[:,:q]` puis on effectue la projection des données de notre matrice X (`.Xqp=Xrc@Uq`) .
A la fin on demande à l'algorithme de comparer les deux méthodes pour voir si elle donne des résultats similaires (ACP et ACPond).
Tests de l'ACP pondérée sur les données de mnist_train.csv et iris.csv
On vérifie bien qu'en prenant W = I, le paramètre n'influe pas et on effectue un ACP classique.

Partie 2 : L'algorithme des k-moyennes.

Dans cette partie nous allons voir (revoir si vous avez déjà réaliser le bonus du BE de Ma314) un des algorithmes
les plus connus « résolvant » le problème du partitionnement de données : l'algorithme des k-moyennes (k-means en anglais). Nous allons commencer par décrire formellement le problème du partitionnement de données.

Le problème du partitionnement de données consiste à trouver une partition S = $\{S_1, S_2, \cdots, S_k\}$ de l'ensemble D en k parties minimisant la quantité suivante :

$$\sum_{j=1}^{k} \sum_{a_i \epsilon S_j} \|a_i - \mu_j\|_2^2$$

Où μ_j est le barycentre de l'ensemble S_j :

$$\mu_j = \frac{1}{Card(S_j)} \sum_{a_i \epsilon S_j} a_i$$

Nous allons voir l'algorithme des k-moyennes qui se veut une solution heuristique au problème de partitionnement des données.
Algorithme des k-moyennes classique : Soit A ∈ M_{mn} (ℝ) une matrice que nous allons considérer par blocs de lignes :

$$A = \begin{pmatrix} a_1 \\ a_2 \\ : \\ a_m \end{pmatrix}$$

Avec $a_i = (a_{i1}, a_{i2}, \dots, a_{ik})$

On cherche à créer un programme Kmoy(A,k,ϵ) qui prend en entrée une matrice A, un entier k ≥ 2 et un réel $\epsilon > 0$ et qui renvoie une liste de longueur k de matrices incarnant une partition de taille k de l'ensemble D=$\{a_1, a_2, \dots, a_m\}$ en suivant la procédure suivante

1. Etape 1

On choisit k vecteurs a_l distincts aux hasards parmi les lignes de A : = $\{a_{l1}, a_{l2}, \dots, a_{lm}\}$ en définissant :$\mu_j{}^{(0)}$=a_{lj}

```
def Kmoy(A,k,epsilon,afficher=False):
    m,n=np.shape(A)
    colour = ['b', 'g', 'k', 'y', 'm', 'pink']
    muo=[]
    for i in range(k):
        r=rnd.randint(0,m)
        muo.append(A[r,:])
```

Tout d'abord on définit les paramètres souhaités (k, ϵ). On définit la liste vide `muo` qu'on remplira de vecteurs aléatoires grâce à une boucle. On définit r une variable qui nous permet de générer un nombre aléatoire (`r=rnd.randint(0,m)`) correspondant à l'indice d'une ligne de A. Ainsi en fonction du nombre k choisi on aura un nombre de vecteurs aléatoires dans `muo` .

2. Etape 2

On définit les partitions $S_j{}^{(1)}$ qui contiennent les a_i les plus proches de $\mu_j{}^{(0)}$ ce qui signifie que :

$$S_j{}^{(1)} = \{a_i |\ \left\| a_i - \mu_j{}^{(0)} \right\|_2 \leq \left\| a_i - \mu_p{}^{(0)} \right\|_2, \forall p \epsilon [1,k]\ \setminus \{j\}\}$$

3. Etape 3

On calcule les barycentres des partitions $S_j{}^{(1)}$:

$$\mu_k{}^{(1)} = \frac{1}{card(S_j{}^{(1)})} \sum_{a_i \epsilon S_j{}^{(1)}} a_i$$

Ainsi que la matrice (considérée par blocs lignes) :

$$\mu^{(1)} = \begin{pmatrix} \mu_1{}^{(1)} \\ \mu_2{}^{(1)} \\ \vdots \\ \mu_k{}^{(1)} \end{pmatrix}$$

Pour chaque ligne de A on va créer une liste D contenant la distance à chaque barycentre. On récupère l'indice de ce minimum, qui correspond aussi au numéro de la partition associé

4. Etape 4

Puis on itère les étapes précédentes jusqu'à ce qu'il existe un entier f :

$$\left\|\mu^{(f-1)} - \mu^{(f)}\right\| \leq \epsilon$$

5. Etape 5

Puis on renvoie la matrice S suivante :

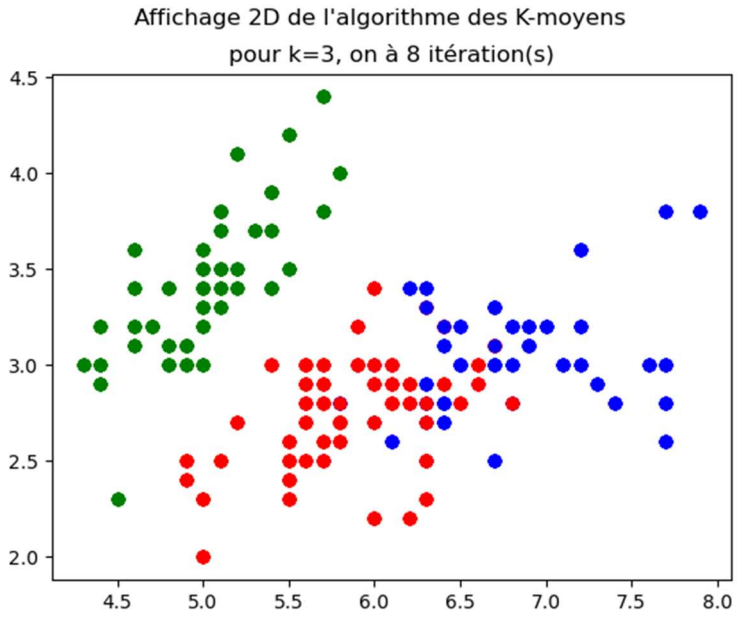
$$S = \begin{pmatrix} & 1 \\ S_1 & : \\ & 1 \\ & 2 \\ S_2 & : \\ & 2 \\ : & : \\ & k \\ S_k & : \\ & k \end{pmatrix}$$

Où les S_j sont les matrices dont les vecteurs lignes sont données par les vecteurs de la partition $S_j^{(f)}$ et la dernière colonne le numéro j de la partition associé.

Les étapes 2 à 5 se font dans la même boucle *while*.
Cette dernière continue tant que l'algorithme n'est pas stable c'est-à-dire que les barycentres se déplacent encore.

On commence par initialiser notre partition des lignes se A en k catégories.
On initialise ainsi S1 qui servira aux calculs et S qui contient une colonne de plus remplie du numéro de la partition associée.
On calcule ensuite pour chacune des lignes de A sa distance aux k-barycentres.
On ajoute ensuite cette ligne à la catégorie dont elle est la plus proche du barycentre.
Lors des premières itérations il est possible qu'une partition de S1 soit vide, pour éviter les problèmes On ajoute un point 0 dans ces partitions vides. Ce point n'aura pas d'impact sur les barycentres muo
Enfin on calcule le déplacement moyen des barycentres pour la condition d'arrêt ainsi que les nouveaux barycentres.
Si la condition est respectée l'algorithme s'arrête et retourne S
Le mode *afficher* permet comme son nom l'indique de visualiser en 2D le partitionnement en k-catégories.

Afin de vérifier le bon fonctionnement de notre programme on le teste avec la base de données Iris.
En se référant au BE de Ma314 on sait que cette base de données peut se regrouper selon 3 partitions.
On exécute donc le programme des k-Moyens sur la base de données iris pour k=3.
On obtient l'affichage suivant :



On distingue bien les 3 partitions des données. Ces dernières correspondent pratiquement aux catégories déjà existantes pour iris.

Partie 3 : L'algorithme de partitionnement.

1. Fonction Python : *choixpts(img, nbpts)*

```
def choixpts(image, nbpts) :
#création d'une matrice qui contiendra tout les pixels choisi
uniquement(480,nbpts,3)
    L=[]
#création d'une matrice de meme taille que l'image ici (480,640,3)
    L_tot=np.zeros(np.shape(image))
    for i in range (1,np.shape(image)[0]-1):
#récupération d'un pixel random en évitant les bords
        Pts_random=np.random.randint(1, np.shape(image)[1]-1, nbpts)
#on insert dans la matrice vierge le pixels choisi
        L.insert(i,np.array(image[i,Pts_random,:]))
#on remplace dans la matrice vierge le pixels choisi
        L_tot[i,Pts_random]=image[i,Pts_random]
#L etait une liste on le repasse en array() pour utiliser cv2
    L=np.array(L)
    cv2.imwrite("L_tot.jpg",L_tot) #création de l'image de L_tot
    cv2.imwrite("L.jpg", L) #création de l'image de L
    return L,L_tot
```

On crée une matrice L qui contiendra tous les pixels choisis aléatoirement en fonction du `nbpts` exigé.
Ensuite on crée une matrice `L_tot` de la même taille que l'image étudié.
On parcourt l'image en effectuant une boucle puis grâce à la fonction `np.random.randint`, on récupère un pixel qu'on ajoute dans notre matrice vierge L.
Puis on remplace ce pixel dans notre matrice `L_tot`.
On redimensionne l'image afin de pouvoir utiliser CV2 et ses différentes fonctions.

2. Fonction Python : *data_pixels(img, L)*

```
def data_pixels(img, L) :
    print("Récupération des datas de l'image en cours ...")
    n,m,p=np.shape(img)
    k=0
#création de la matrice initiale de taille(np.shape(L)[0]*nbpts,8)
    data_img=np.zeros((np.shape(L)[0]*nbpts,8))
#Suivant le pixel parcouru dans la matrice L_tot qui contient uniquement
les pixels randoms de l'image à leur position

    for i in tqdm(range (n)):
        for j in range(m):
#si le pixel n'est pas noir alors on le rentre dans le tableau de datas
                if np.mean(L[i,j])!=0:
                    data_img[k,0]=i
                    data_img[k,1]=j
                    data_img[k,2]=img[i,j,0]
                    data_img[k,3]=img[i,j,1]
                    data_img[k,4]=img[i,j,2]
                    for p in range (3):
                        data_img[k,5]=(int(img[i+1,j+1][p])+
                        int(img[i,j+1][p])+int(img[i-1,j+1][p])+
                        int(img[i+1,j][p])+int(img[i,j][p])+
                        int(img[i-1,j][p])+int(img[i+1,j-1][p])+
                        int(img[i,j-1][p])+int(img[i-1,j-1][p]))//9
                    k=k+1
    return data_img
```

On récupère la taille De l'image puis on crée la matrice vide `data_img` qu'on obtient avec les dimensions la matrice L ainsi que `nbpts` grâce à la fonction `choixpts`.
On parcourt ainsi la matrice `L_tot` qui contient des pixels aléatoires grâce à une boucle sur les lignes puis sur les colonnes.
En mettant une condition `np.mean(L[i,j])!=0` sur les différents coefficients de la matrice, on sait quelle pixel est noir ou pas . Si le pixel parcouru n'est pas noir alors on le rentre dans d `data_img` et on récupère les informations correspondantes.
A la fin on se trouve donc avec la matrice `data_img` de taille nbpts×8 et dont les lignes sont données par :

$$(c_{i1}, c_{i2}, c_{i3}, c_{i4}, c_{i5}, c_{i6}, c_{i7}, c_{i8})$$

Avec :

- c_{i1} l'indice de ligne du pixel i, c_{i2} l'indice de colonne
- c_{i3}, c_{i4}, c_{i5} les trois intensités de couleurs sur les trois couches de l'image.
- c_{i6}, c_{i7}, c_{i8} les trois moyennes des intensités de couleurs sur les 9 pixels autour du pixel i (lui compris)

3. Fonction Python : *ACPimg(img, L)*

```
def ACPimg(tabl,q=2):
    Xrc=centre_red2(tabl) #on centre réduit les données
    C=Xrc.T@Xrc
    n,d=np.shape(tabl) #on récupere la shape du tabl
#3 racines réelles simples, C est donc diagonalisable
    Uq=np.zeros((d,q))
    U=np.linalg.eig(C)[1]
    for i in range(q):
        Uq[:,i]=U[:,i]
#on calcul Xq contenant les catégories du tabl centré réduit
    Xq=Xrc@Uq
    return(Xq)
```

4. Création de deux fonctions

   a) Fonction Python : *Kmoyimg(img, L)*

Nous reprenons la fonction Kmoyen vu dans la partie 2.

   b) Fonction Python : *Masque(img, L)*

```
def Masque(S):
#création d'une matrice de meme taille que l'image
    imagef=np.zeros(np.shape(image))
    for i in S[0]:
#on impose la couleur du pixel de la catégorie trouvé par l'ACP
            imagef[int(c[i][0]),int(c[i][1])]=[50,50,200]

    for j in S[1]:
#on impose la couleur du pixel de la catégorie trouvé par l'ACP
            imagef[int(c[j][0]),int(c[j][1])]=[50,200,50]
    return imagef
```

5. Algorithme d'intpainting et création de la fonction *RemplissageMasque(imgmasqueponctuel)*

```
def RemplissageMasque(imgmasqueponctuel):
    print("Création du masque ...")
    mask=cv2.cvtColor(imgmasqueponctuel,cv2.COLOR_BGR2GRAY)
    mask[mask>0]=255
    mask=255*np.ones(np.shape(mask))-mask

imgmasque=cv2.inpaint(imgmasqueponctuel,np.uint8(mask),3,cv2.INPAINT_NS)
    print("terminé !\n")
    return imgmasque
```

Dans un premier temps nous allons utiliser la fonction `cv2.inpaint` permettant d'obtenir le masque global à l'aide du programme précèdent contenant `imgmasqueponctuel`.

```
def Masquetot(img,imgmasque):
    print("Création du masque global en cours ...")
    n,m,p=np.shape(img)
#création de l'image du masque global1 en fond blanc
    imageselec1=np.ones((n,m,p))*255
#création de l'image du masque global2 en fond blanc
    imageselec2=np.ones((n,m,p))*255
    for i in tqdm(range (n)):
        for j in range(m):
#Si le pixel est inférieur a la moyenne alors on récupère le véritable
pixel de l'image pour le mettre dans l'image masque blanc

            if np.mean(imgmasque[i,j][1]) < np.mean(imgmasque[:,:][1]):
                imageselec1[i,j]=img[i,j]
            else:
                imageselec2[i,j]=img[i,j]
    return imageselec1,imageselec2
```

On définit par la suite la fonction `Masquetot` permettant d'afficher le rendu final avec les couches séparés. On récupère la taille de l'image étudié puis on crée deux matrices (`imageselec1`, `imageselec2`) qui contiendront le rendu final.
A l'aide d''une boucle on parcourt les différents indices de `imgmasque`. Si le pixel à une valeur inferieur à la moyenne alors on récupère les indices où se situe le pixel puis on l'ajoute à notre matrice `imageselec1` sinon on l'ajoute à `imageselec2`.
Cela nous permet ainsi d'obtenir 2 images.

Exemple :

Avec feuille.jpg :

Figure 2- Image originelle

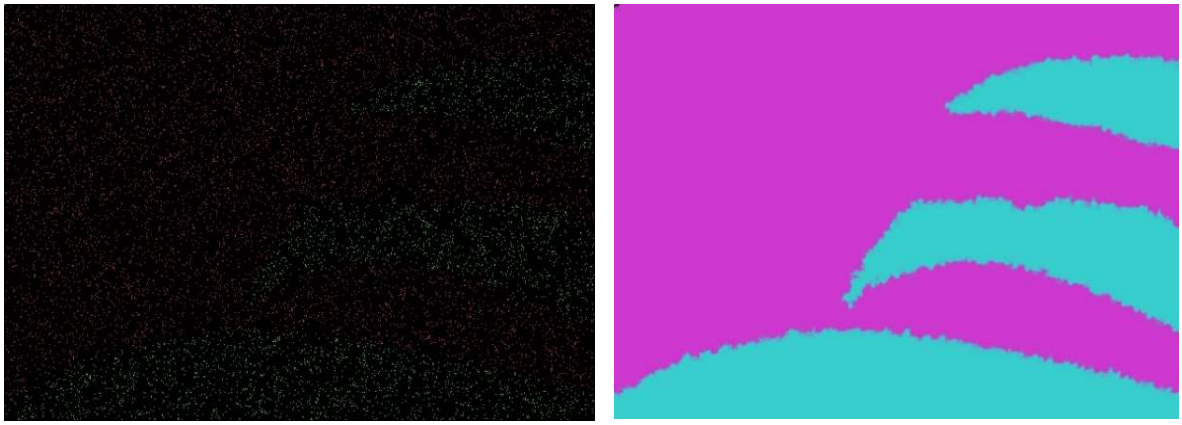


Figure 3- Masque Ponctuel (à gauche), masque après inpainting (à droite)

Figure 4- Rendu final avec les couches séparés

Avec nebuleuse.jpg :

Figure 5- Image originelle

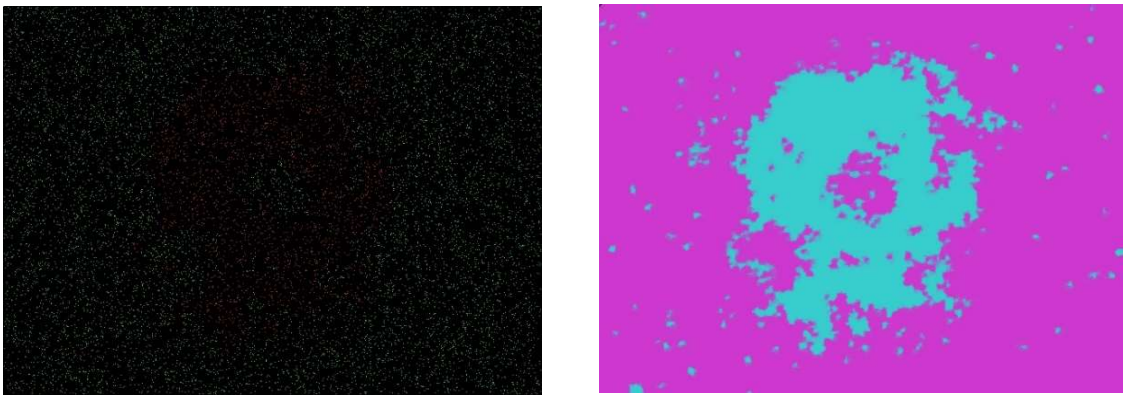


Figure 6- Masque Ponctuel (à gauche), masque après inpainting (à droite)

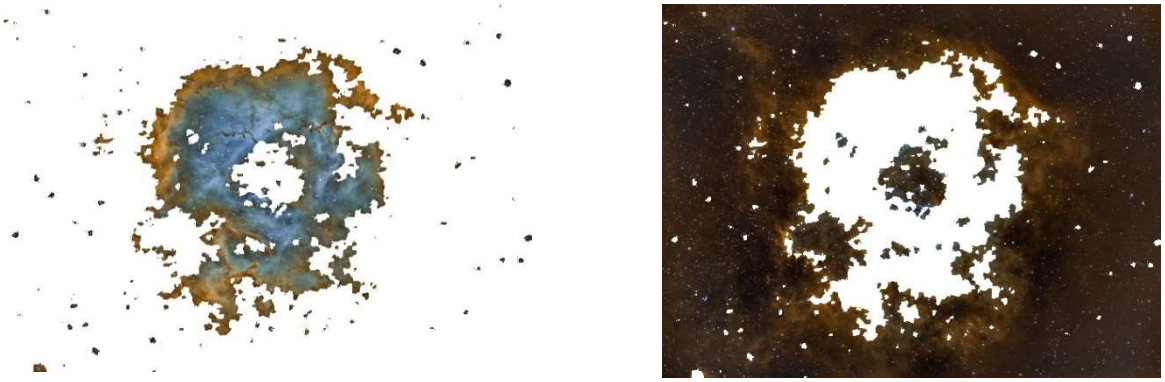


Figure 7- Rendu final avec les couches séparés

6. Bonus

On remarque que l'utilisation du Kmoy sans ACP permet de faire apparaitre une partie des catégories au centre de l'image l'image ou l'influence de la position du pixel est la moins forte et donc celle de la couleur la plus forte.

Figure 8- Image originelle

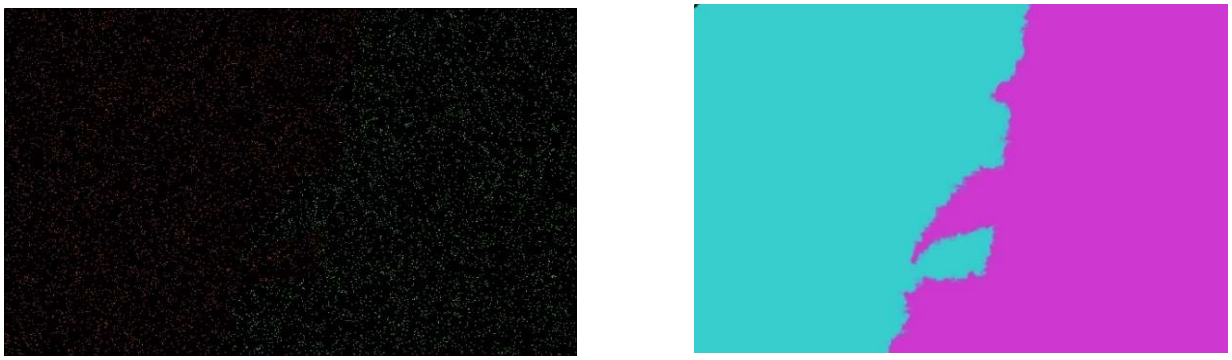


Figure 9- Masque Ponctuel (à gauche), masque après inpainting (à droite)

Figure 10- Rendu final avec les couches séparés