ADIOKO Mickaël
EL AHMAD Tamim

Université Paris-Diderot
Master 1 Mathématiques et Applications

Projet statistique
Sujet 5 : Clustering K-Means

Mme GRIBKOVA Svetlana
M. ZILIO Alessandro

Sommaire

I) La méthode des K-Means :

1) Explication de l'algorithme :

L'algorithme des K-Means est un algorithme de *clustering*, c'est-à-dire qu'il consiste à classifier un jeu de données en plusieurs groupes (K groupes en l'occurrence) dont les éléments, représentés par des vecteurs, présentent des similarités. Ces groupes tendent à être les plus hétérogènes entre eux, et leurs éléments les plus homogènes.

Le fondement du K-Means est d'attribuer des centres aux *clusters* (groupes) et d'attribuer à chacun des éléments le *cluster* dont le centre est le plus proche, au sens de la distance euclidienne. Plus spécifiquement, le principe est le suivant :

1. On choisit le nombre de *clusters* K ;
2. On initialise les K centres de chacun des *clusters* ;
3. On associe à chaque élément du jeu de données le *cluster* dont le centre est le plus proche
4. On met à jour chaque centre selon la formule suivante :

$\mu_k = \frac{1}{|C_k|} \sum_{X_i \in C_k} X_i$ où : - les X_i sont des éléments du jeu de données, donc des vecteurs

- C_k désigne le *cluster* k, donc l'ensemble des éléments du jeu de données appartenant au *cluster* k
- μ_k désigne le centre du *cluster* k, c'est un vecteur de même taille que X_i

En réalité, on remplace les centres de chaque *cluster* par le barycentre de ses points.

5. On répète cette opération jusqu'à ce que les centres ne soient plus modifiés.

2) Comment a-t-on construit cette algorithme ?

Une approche populaire de construction d'algorithmes de *clustering* est de définir une fonction de coût paramétrée sur l'ensemble des partitionnement possibles et le but de l'algorithme va être de trouver le partitionnement minimisant cette fonction de coût. L'algorithme des K-Means ne déroge pas à la règle et son but est de trouver le *clustering* minimisant la fonction de perte quadratique :

$$J = \sum_{i=1}^{K} \sum_{X_j \in C_i} \|X_j - \mu_i\|^2$$

Cette valeur caractérise ce qu'on appelle l'inertie *intra-cluster*, autrement dit, plus est elle faible, et plus les différents *clusters* sont homogènes, c'est-à-dire que leurs éléments sont similaires. Ce qui est normal puisque l'on somme pour chaque *cluster* les distances au carré de chaque élément à son centre.

On va maintenant montrer que cette fonction de perte quadratique décroît à chaque itération de l'algorithme des K-Means.

Lemme L1 : (inertie minimum)

Soit $(X_1, \dots, X_P) \in (\mathbb{R}^N)^P$, P points de $\mathbb{R}^N$, le minimum de la quantité $Q(Y \in \mathbb{R}^N)$:

$$Q(Y) = \sum_{i=1}^{P} \|X_i - Y\|^2$$

est atteint pour $Y=G=\frac{1}{P}\sum_{i=1}^{P} X_i$ le barycentre des points $(X_1,\ldots,X_P)$.

Preuve :

$$\sum_{i=1}^{P}\overrightarrow{GX}_i=\sum_{i=1}^{P}(X_i-G)=\sum_{i=1}^{P}X_i-(P\times(\frac{1}{P})\sum_{i=1}^{P}X_i)=\vec{0}$$

Donc $(\sum_{i=1}^{P}\overrightarrow{GX}_i)\cdot\overrightarrow{YG}=\vec{0}\Rightarrow(\sum_{i=1}^{P}\overrightarrow{GX}_i\cdot\overrightarrow{YG})=\vec{0}$ pour tout $Y\in\mathbb{R}^N$

Soit $Y\in\mathbb{R}^N$:

$$Q(Y)=\sum_{i=1}^{P}\|X_i-Y\|^2=\sum_{i=1}^{P}\|X_i-G+G-Y\|^2=\sum_{i=1}^{P}\|X_i-G\|^2+P\times\|G-Y\|^2+2\sum_{i=1}^{P}(\overrightarrow{GX}_i\cdot\overrightarrow{YG})$$

Or $\sum_{i=1}^{P}\overrightarrow{GX}_i\cdot\overrightarrow{YG}=\vec{0}$

Donc $Q(Y)=\sum_{i=1}^{P}\|X_i-G\|^2+P\times\|G-Y\|^2$

Donc le minimum de Q(Y) est atteint en Y=G.

Preuve de la décroissance de la perte quadratique :

On considère une itération *t* de l'algorithme K-Means. Soit $C_1^{(t-1)},\ldots,C_K^{(t-1)}$ la partition précédente, $\mu_i^{(t-1)}$ le centre du cluster $C_i^{(t-1)}$, et $C_1^{(t)},\ldots,C_K^{(t)}$ la nouvelle partition assignée à l'itération *t*.

D'après le lemme d'inertie minimum, on a : $\sum_{X\in C_i^{(t)}}\|X-\mu_i^{(t)}\|^2\le\sum_{X\in C_i^{(t)}}\|X-\mu_i^{(t-1)}\|^2$

Donc $\sum_{i=1}^{K}\sum_{X\in C_i^{(t)}}\|X-\mu_i^{(t)}\|^2\le\sum_{i=1}^{K}\sum_{X\in C_i^{(t)}}\|X-\mu_i^{(t-1)}\|^2$

De plus, la définition de la nouvelle partition $C_1^{(t)},\ldots,C_K^{(t)}$ minimise l'expression $\sum_{i=1}^{K}\sum_{X\in C_i}\|X-\mu_i^{(t-1)}\|^2$ sur l'ensemble de toutes les partitions $C_1,\ldots,C_K$.

Donc $\sum_{i=1}^{K}\sum_{X\in C_i^{(t)}}\|X-\mu_i^{(t-1)}\|^2\le\sum_{i=1}^{K}\sum_{X\in C_i^{(t-1)}}\|X-\mu_i^{(t-1)}\|^2$

Finalement, $\sum_{i=1}^{K}\sum_{X\in C_i^{(t)}}\|X-\mu_i^{(t)}\|^2\le\sum_{i=1}^{K}\sum_{X\in C_i^{(t-1)}}\|X-\mu_i^{(t-1)}\|^2$.

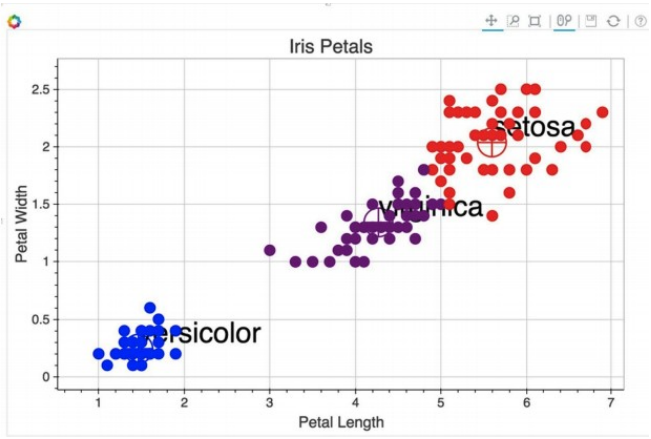
Ce qui conclut la preuve.

Tandis que l'on sait que la perte quadratique décroît à chaque itération de l'algorithme K-Means, il n'y a aucune garanti sur le nombre d'itérations nécessaires pour atteindre la convergence. De plus, il n'y a pas de borne inférieure triviale entre la valeur finale de la perte quadratique lors de l'exécution de l'algorithme et le minimum théorique de cette valeur. En effet, il se peut que K-Means converge vers une valeur qui n'est pas un minimum local. C'est pourquoi pour améliorer les résultats de K-Means il est recommandé de le répéter plusieurs fois avec des initialisations différentes.

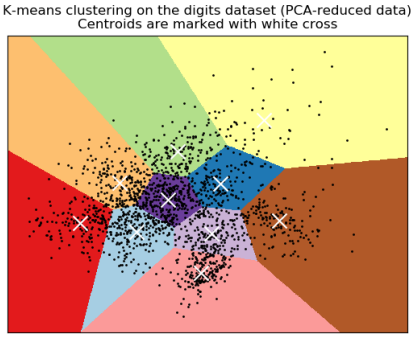
3) Avantages et inconvénients de la méthode des centres mobiles :

Tout d'abord, cette méthode est très simple à comprendre et à mettre en œuvre. De plus, son exécution est généralement rapide, avec l'assurance que la méthode converge (en d'autres termes, l'algorithme prend fin en un temps fini avec probabilité 1) et il est possible de l'appliquer à des données de grandes tailles, et des données de tous types, à condition de définir une bonne notion de distance.

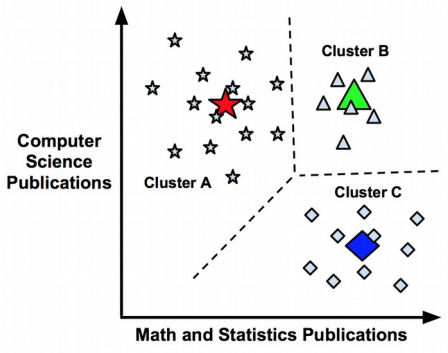
On peut, par exemple, classer des types de fleurs en fonction de la taille et l'épaisseur de leurs pétales ou de leurs sépales:



On peut également classifier des images représentant des chiffres de 0 à 9 écrits de façon manuscrite, en traitant les images qui sont représentées généralement par des matrices n*m*3 de la manière suivante par exemple : on les convertit en niveaux de gris pour avoir des matrices n*m*1, puis on aplatit les matrices pour avoir des vecteurs de taille n*m, on peut donc traiter les images comme des vecteurs de $\mathbb{R}^{n\times m}$ et utiliser la distance euclidienne :



Pour prévoir la disposition des scientifiques assistant à un conférence sur la science des données par exemple, et les regrouper par domaine de prédilection, on peut regarder leurs nombres de publications dans des journaux plus portés sur l'informatique et celles dans des journaux plus portés sur les mathématiques et les statistiques :



Enfin, on peut même classer des textes ou des mails en utilisant par exemple la distance de Levenshtein, qui donne une distance entre 2 chaînes de caractères.

Pour ce qui est des inconvénients, ils résident dans 2 aspects primordiaux de l'algorithme, et sont donc problématiques.

D'abord, le nombre de classes doit être fixé au départ or on ne sait pas *a priori* combien de classes contient notre jeu de données au départ, puisque nous sommes dans un problème de classification non-supervisée.

De plus, le résultat de l'algorithme dépend entièrement de l'initialisation des centres, on ne peut donc pas se contenter d'initialiser aléatoirement les centres pour mettre en œuvre cet algorithme.

Heureusement, des méthodes existent pour pallier à ces deux problèmes.

4) Améliorations de l'initialisation et du nombre de *clusters* :

Soit N le nombre d'éléments dans le jeu de données et K le nombre de classes choisies.

Pour évaluer le résultat d'un *clustering* par la méthode des points mobiles, on peut se baser notamment sur l'évaluation de 2 indices de qualité : l'erreur quadratique moyenne et l'indice de Davies-Bouldin. L'indice de Davies-Bouldin permet de mesurer la compacité et la séparabilité des groupes, de petites valeurs du DB sont indicatives de la présence de groupes compacts et bien séparés :

$$DB=\frac{1}{K}\sum_{i=1}^{K} \max_{j\neq i}\left(\frac{S(C_i)+S(C_j)}{d(\mu_i,\mu_j)}\right) \quad \text{où} \quad S(C_i)=\frac{1}{|C_i|}\sum_{X\in C_i} d(X,C_i)$$

4.1) Amélioration de l'initialisation

a) *Global K-means*

Parlons d'abord de la méthode *Global K-means* visant à atteindre une solution globalement optimale, et à minimiser l'erreur quadratique moyenne. Le but est de trouver les centres un par un en minimisant l'erreur quadratique $J=\sum_{i=1}^{K}\sum_{X_j\in C_i}\|X_j-\mu_i\|^2$.

1. On fixe le premier centre μ_1 comme étant le barycentre des données ;
2. On parcourt le jeu de données en considérant chaque donnée X comme un 2e centre μ_2 , on partitionne les données en fonction du centre le plus proche et on calcule l'erreur quadratique ;
3. On garde le centre μ_2 = X qui minimise l'erreur quadratique ;
4. On applique la méthode des K-means jusqu'à convergence.

Ensuite, on réalise la même opération pour trouver le 3e centre, en initialisant les 2 premiers suivant le résultat de l'étape précédente, et on répète cette opération jusqu'au K^e centre.

b) Initialisation par le mal classé

Le G*lobal K-means* est amorcé par un seul groupe ayant pour représentant le centre de gravité de l'ensemble des données, dans certains cas, cette partie de l'espace est vide (voir figure suivante) ce qui permet de dégrader la classification, nous proposons d'amorcer l'initialisation du K-means avec deux groupes, les centres de ces groupes doivent assurer la séparabilité des données au cours de classification, il est évidant de choisir les deux données les plus éloignées.

L'initialisation, par le mal classé suit ce principe :

1. On crée une matrice de distance
2. On choisit les 2 éléments les plus éloignés (ils représentent les 2 premiers centres)

TANT QUE : le nombre de classes souhaité n'est pas atteint
FAIRE

3. Affecter les individus aux centres disponibles ;
4. Sélectionner un élément mal classé (celui qui possède la plus grande distance de son centre le plus proche) ;
5. Ajouter cet individu à l'ensemble des centres ;
6. Augmenter le nombre des centres ;

Fin TANT QUE

On applique ensuite l'algorithme des K-Means classique avec l'initialisation comme décrite ci-dessus.

c) Approche incrémentale

L'approche incrémental de classification est similaire à celle du *global K-Means*, la différence entre elles réside dans les points suivant :

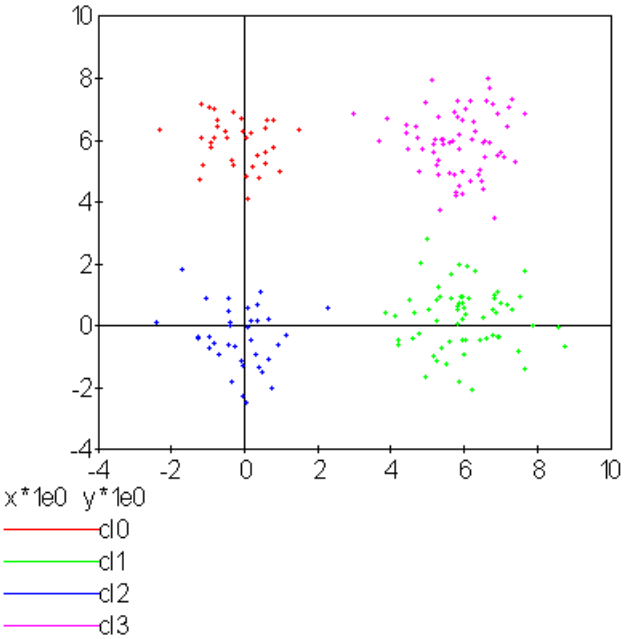
- Le nombre de points initiaux, dans notre cas deux au lieu de un seul dans le global k-means.

- La recherche du nouveau centre ce limite à la recherche de l'élément le mal classé au lieu de testé toutes les données.

1. On fixe 2 premiers centres μ_1 et μ_2 tels que : $\mu_1 = X_1$ et $\mu_2 = X_2$ avec $d(X_1, X_2) = \max_{i,j \in [1,\dots,N]} d(X_i, X_j)$ et N le nombre de données X_i ;
2. On sélectionne un élément mal classé (celui qui possède la plus grande distance de son centre le plus proche) ;
3. On ajoute cet individu à l'ensemble des centres ;
4. On applique la méthode des K-means jusqu'à convergence ;
5. On répète les étapes 2 à 4 jusqu'à obtenir le nombre de *clusters* souhaité.

4.2) Amélioration du choix du nombre de *clusters*

L'algorithme des centres mobiles effectue une classification non supervisée à condition de connaître au préalable le nombre de classes et cette information est rarement disponible. Une alternative consiste à estimer la pertinence des classifications obtenues pour différents nombres de classes, le nombre de classes optimal est celui qui correspond à la classification la plus pertinente. Cette pertinence ne peut être estimée de manière unique, elle dépend des hypothèses faites sur les éléments à classer, notamment sur la forme des classes qui peuvent être convexes ou pas, être modélisées par des lois normales multidimensionnelles, à matrice de covariances diagonales, … On va se pencher sur le critère de Davies-Bouldin, qui est adapté à l'algorithme des centres mobiles. Le critère de Davies-Bouldin est minimum lorsque le nombre de classes est optimal.
La méthode consiste donc à calculer l'indice de Davies-Bouldin pour différents choix de K et de garder celui pour lequel il est minimum :

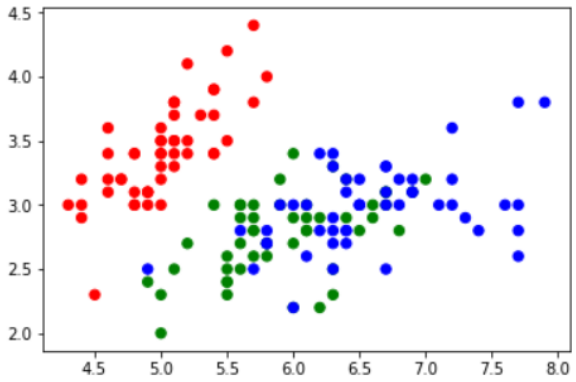


Classification en quatre classes : nombre de classes sélectionnées par le critère de Davies-Bouldin dont les valeurs sont illustrées par le graphe apposé à droite.

II) Mise en œuvre de l'algorithme :

1) Choix du set de données :

On a choisi de tester notre algorithme sur le set de données Iris, initialement publié à l' *UCI Machine Learning Repository: Iris Data Set*, en 1936. Ce set est souvent utilisé pour tester des algorithmes d'apprentissage automatique et des visualisations, et il fait partie des jeux de données disponibles dans la bibliothèque *Scikit-learn* de *Python*. Il contient 3 types de la fleur Iris (*Iris Setosa, Iris Versicolour, Iris Virginica*), 50 instances de chaque type (soit 150 en tout) qui sont des vecteurs de 4 éléments, caractéristiques de chaque fleur et de chaque type de fleur : longueur des sépales en centimètres (entre 4,3 et 7,9 cm), épaisseur des sépales en centimètres (entre 2 et 4,4 cm), longueur des pétales en centimètres (entre 1,0 et 6,9 cm), et épaisseur des pétales en centimètres (entre 0,1 et 2,5 cm). Le jeu de données ne contient pas de valeurs manquantes, ce qui nous dispense de les traiter.

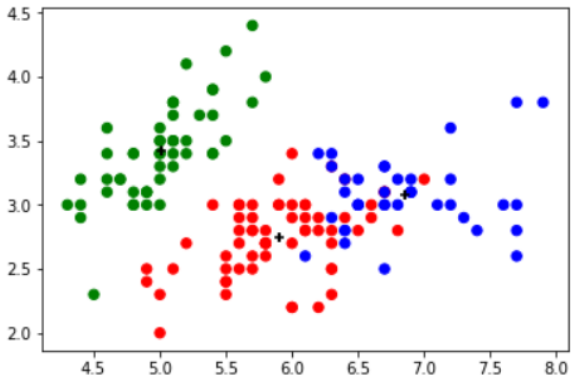


Épaisseur des sépales en fonction de la longueur des sépales,
chaque type de fleur est représenté par une couleur,
ici R : *Setosa*, V : *Versicolour*, B : *Virginica*

Dans la suite, nous représenterons toujours l'épaisseur des sépales en fonction de la longueur des sépales pour représenter les *clusters* avec les couleurs rouge, vert et bleu. Nous représenterons le centre de chaque cluster par une croix noire.

2) Quelques exemples de *clustering* et mise en évidence de la sensibilité à l'initialisation :

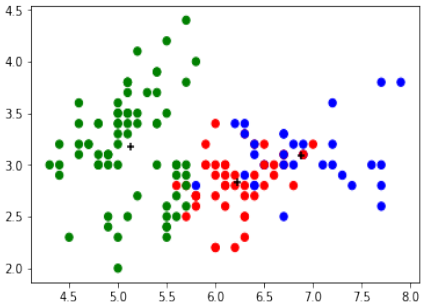
Voici un premier résultat en utilisant la fonction *Kmeans* déjà implémentée dans la bibliothèque *Scikit-learn*, en utilisant la méthode d'initialisation *K-Means++* :



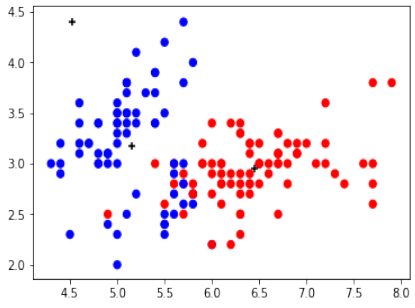
R : 62, V : 50, B : 38

On voit ici que le résultat est assez proche de la réalité, on trouve 62 fleurs de type « rouge », 50 de type « vert » et 38 de type « bleu ». Les Iris *Setosa* (ici en vert) sont bien classées, ce sont en réalité les *Versicolour* et les *Virginica* qui posent problème.

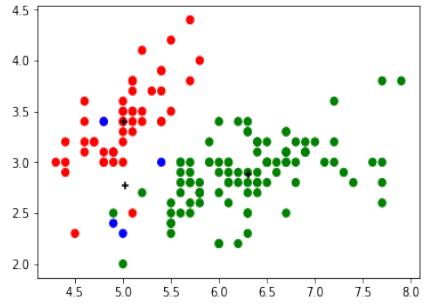
Voici ensuite 5 résultats obtenus avec les algorithmes que nous avons mis en œuvre. Puisque nous connaissions le nombre de classes présentes dans le jeu de données, nous avons à chaque fois exécuté notre algorithme avec le paramètre K=3. Pour chaque figure, les centres ont été initialisés aléatoirement :



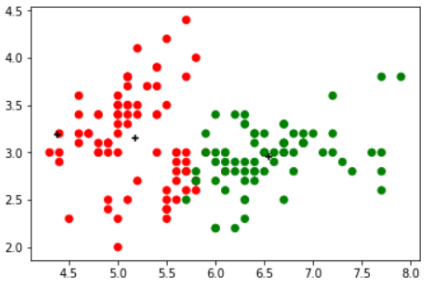
0 : R : 47, V: 71, B : 32



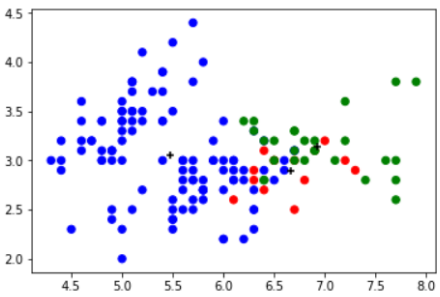
1 :R : 80, V: 0, B : 70



2 : R : 50, V: 96, B : 4



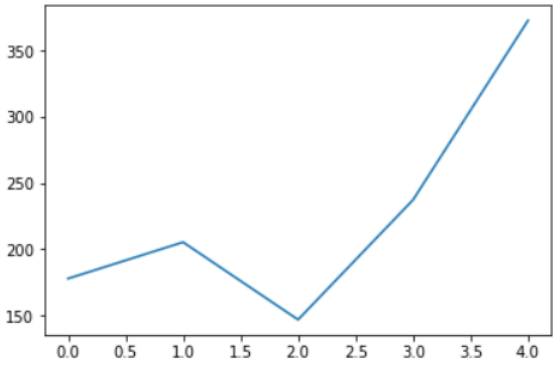
3 : R : 77, V: 73, B : 0



4 :R : 13, V: 28, B : 109

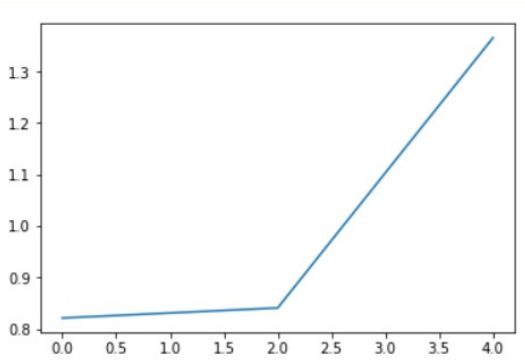
Le partitionnement le plus proche de la réalité est le partitionnement 0. Mais comment savoir quel partitionnement est le meilleur sans connaître les vraies classes des données, autrement dit la réalité ?

On peut d'abord représenter l'erreur quadratique finale de ces 5 partitionnements :



On observe que l'erreur quadratique du partitionnement 0 est la 2^{e} plus basse, ce qui est cohérent mais elle devrait être théoriquement la plus basse. Celle qui la « bat » est l'erreur quadratique du partitionnement 2, ce qui paraît logique au vue de la représentation graphique, l'erreur quadratique étant la somme des distances au carré de chaque élément avec son centre. On voit enfin que la valeur du partitionnement 4 explose mais au vue du *cluster* bleu, rien de surprenant. Cet indicateur est donc bon mais ne garantit pas un résultat infaillible.

On peut maintenant s'intéresser à l'indice de Davies-Bouldin de chaque partitionnement ne contenant pas de classe nulle, ceci entraînant une division par 0 lors du calcul de l'indice. D'ailleurs, on peut dire qu'un partitionnement où une classe est nulle n'est pas correcte car il faudrait dans ce cas réduire le nombre de classes. Voici donc les valeurs de Davies-Bouldin pour les *clusterings* 0, 2 et 4 :



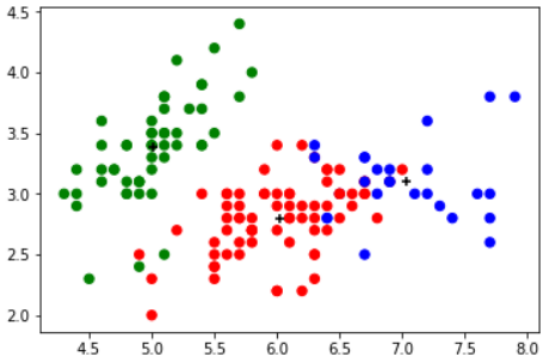
On trouve maintenant des valeurs très similaires entre les partitionnements 0 et 2, et une valeur qui explose encore une fois pour le 4. Cette fois-ci, si on suivait aveuglément ce graphique, on aurait effectivement le bon classement du partitionnement le plus fidèle à la réalité au moins fidèle. Toutefois l'écart très petit entre le *clustering* 0 et 2 en suivant ce critère montre qu'il n'est pas d'une robustesse à toute épreuve.

On peut observer que ces résultats sont très hétérogènes, ce qui illustre parfaitement le fait que la performance de l'algorithme des K-Means est dépendante de son initialisation et que son amélioration est nécessaire.

Maintenant observons les résultats en utilisant des méthodes d'initialisation différentes et non aléatoires, comme *Global K-Means*, l'initialisation par le mal classé et l'approche incrémentale.

a) *Global K-means*

Voici tout d'abord les résultats obtenus avec la méthode *Global K-Means* :

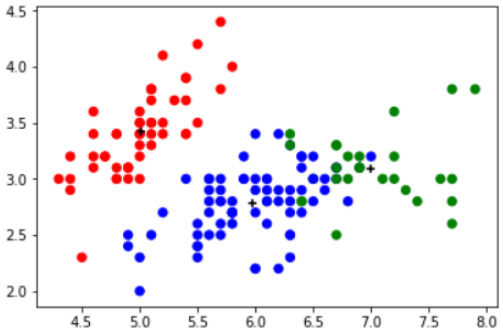


R : 72, V : 52, B : 26

On voit que les résultats sont déjà assez proches de la réalité, mais la classe rouge est un peu trop représentée en dépit de la classe bleue.

b) Initialisation par le mal classé

Regardons maintenant le *clustering* réalisé en initialisant les centres par la méthode du mal classé :

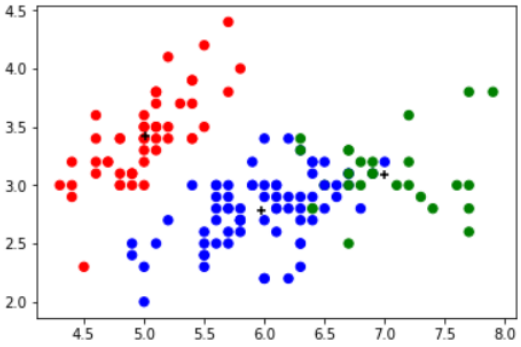


R : 50, V : 28, B : 72

On obtient des résultats extrêmement similaires à la méthode du *Global K-Means*, elle est légèrement meilleure car tous les Iris *Setosa* (en rouge ici) sont bien classés, et 2 *Virginica* précédemment prédits comme *Versicolour* sont maintenant correctement classés. Néanmoins, cette différence est infime.

c) Approche incrémentale

Enfin, voici le partitionnement obtenu grâce à une approche incrémentale (une sorte de mélange entre le *Global K-Means* et l'initialisation par le mal classé) :

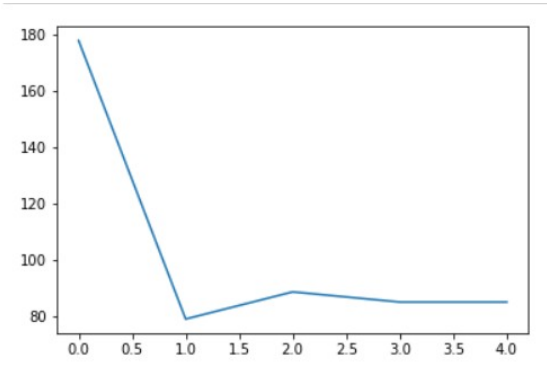


R : 50, V : 28, B : 72

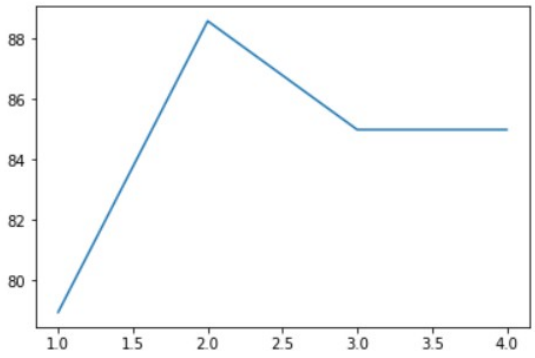
On obtient le même partitionnement qu'avec l'initialisation par le mal classé, car on obtient les mêmes centres.

Comparons maintenant les erreurs quadratiques et les indices de Davies-Bouldin de chaque méthode, avec les abscisses :
- 0 correspondant au partitionnement aléatoire 0, le meilleur qu'on ait obtenu avec une initialisation totalement aléatoirement ;
- 1 correspondant à la méthode *K-Means++* ;
- 2 correspondant à la méthode *Global K-Mean* ;
- 3 correspondant à l'initialisation par le mal classé ;
- 4 correspondant à l'approche incrémentale.

D'abord les erreurs quadratiques :

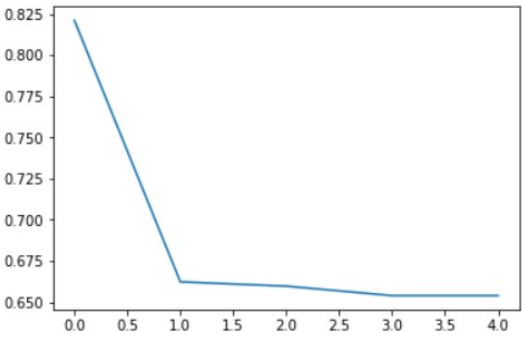


avec le partitionnement 0

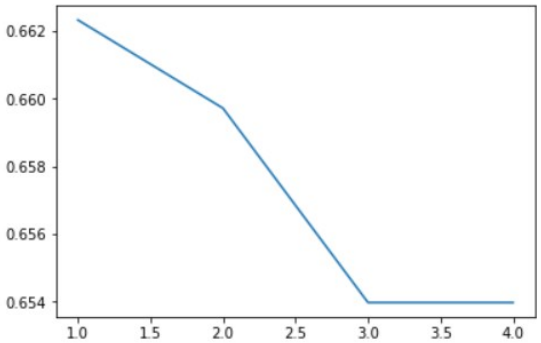


sans le partitionnement 0

Maintenant les indices de Davies-Bouldin :



avec le partitionnement 0



sans le partitionnement 0

On observe d'abord, sans surprise, que ce soit à travers le critère de l'erreur quadratique ou de Davies-Bouldin, le partitionnement 0 est le plus mauvais et de loin. Au regard de l'erreur quadratique, *K-Means++* est assez nettement la meilleure méthode, suivie de l'initialisation par le mal classé et l'approche incrémentale qui ont le même résultat, et enfin *Global K-Means*. Ce résultat est plutôt logique puisqu'il suit effectivement la qualité des partitionnements par rapport à la réalité. Concernant le critère de Davies-Bouldin, on obtient maintenant 4 valeurs très proches, mais cette fois-ci l'initialisation par le mal classé et l'approche incrémentale ont le meilleur indice, suivi de *Global K-Means* et *K-Means+*. Ce résultat est également logique puisque dans le calcul de l'indice de Davies-Bouldin, chaque terme de la somme est divisé par une distance entre centres de *clusters*, et l'initialisation par le mal classé et l'approche incrémentale place les centres en tentant de maximiser les distances entre eux.

On voit donc que les différentes variantes de la méthode palliant à l'initialisation aléatoire améliorent effectivement les partitionnements obtenus, seulement ils ne sont toujours pas fidèles à la réalité, à part pour les iris *Setosa*. Ceci est dû au fait que les 4 caractéristiques que nous disposons permettent de bien différencier les iris *Setosa* des 2 autres espèces au sens de la distance euclidienne, mais pas les iris *Versicolour* et *Virginica* de manière efficace. Ces 2 groupes ne représentant pas 2 sphères distinctes dans l'espace $\mathbb{R}^4$.

3) Choix du nombre K de *clusters* :

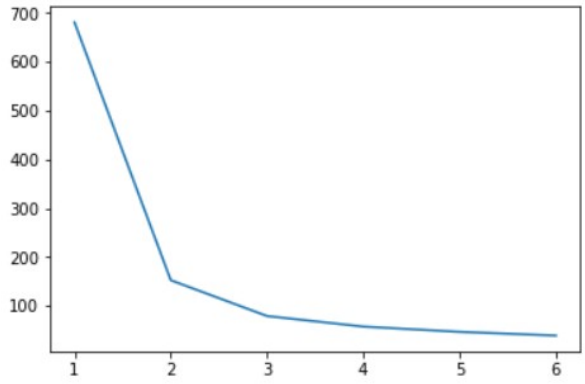
Maintenant que l'on a mis en évidence la sensibilité de K-Means à l'initialisation et étudier quelques alternatives qui s'offrent à nous, penchons-nous sur le choix du nombre de *clusters* K. En effet, ce choix est primordial car il est le seul hyper-paramètre du problème à régler au préalable, et en réalité on ne connaît pas le nombre de classes que contient notre jeu de données. Différentes méthodes s'offrent à nous, comme l'étude des indices d'erreur quadratique et de Davies-Bouldin pour sélectionner le K optimal. Nous allons nous intéresser à ces 2 dernières méthodes.

a) Erreur quadratique

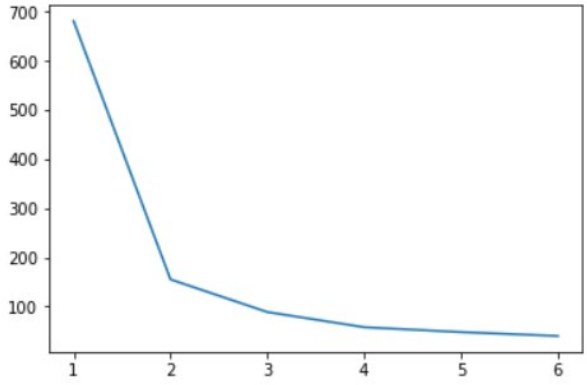
La méthode est la suivante :

- On applique une méthode de K-Means dont l'initialisation n'est pas totalement aléatoire (la méthode basique) au jeu de données en faisant varier K (on choisit de ne pas utiliser le K-Means basique car l'aléatoire total biaise les résultats) ;
- On trace le graphe des erreurs quadratiques des partitionnements obtenus en fonction de K ;
- On regarde la valeur de K à partir de laquelle la diminution de l'erreur quadratique n'est plus significative : cette valeur est optimale.

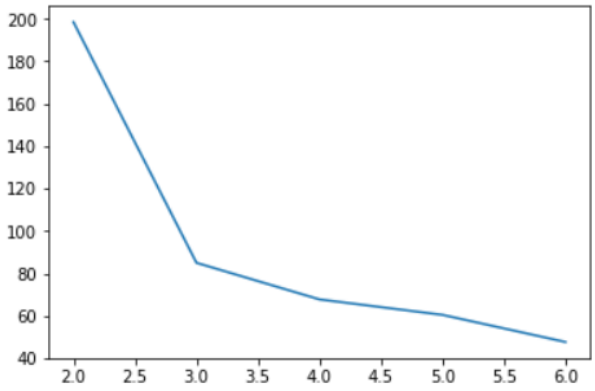
Pourquoi ne choisirait-on pas tout simplement le K pour lequel l'erreur quadratique est minimale ?
En réalité, l'erreur quadratique est décroissante en fonction de K, car plus on augmente le nombre de *clusters*, plus on augmente le nombre de centres, et donc plus on diminue la distance de chaque point avec son centre le plus proche. Donc prendre le plus petit reviendrait à toujours prendre K le plus grand. C'est pourquoi on s'intéresse plutôt à la variation de l'erreur quadratique et le moment où celle-ci commence à décroître lentement.
Voici différents graphes de l'erreur quadratique en fonction de K :



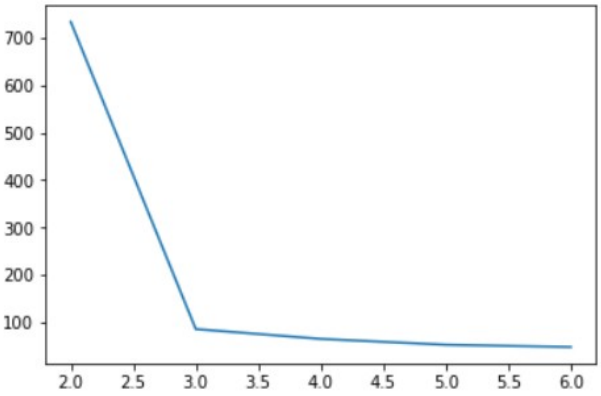
K-Means++



Global K-Means



initialisation par le mal classé



approche incrémentale

On a commencé à K=1 pour le *K-Means++* et le *Global K-Means* qui correspond uniquement à positionner un centre comme barycentre de l'ensemble des données et associé toutes les données à ce seul et unique *cluster*. Quant à l'initialisation par le mal classé et l'approche incrémentale, l'étude commence à K=2 car au vu du fonctionnement de ces 2 algorithmes, on commence nécessairement avec 2 centres à l'initialisation. On voit donc très clairement dans le cas de *K-Means++* et l'approche incrémentale que c'est à partir de K=3 que la diminution n'est plus significative. Dans le cas de *Global K-Means* et l'initialisation par le mal classé, la conclusion est la même, seulement la différence est légèrement moins évidente.

Cette méthode montre donc satisfaction, le seul problème étant de réellement quantifier ce caractère « significatif » et de trouver exactement le K correct, ce qui peut être délicat dans certains cas.

b) Indice de Davies-Bouldin

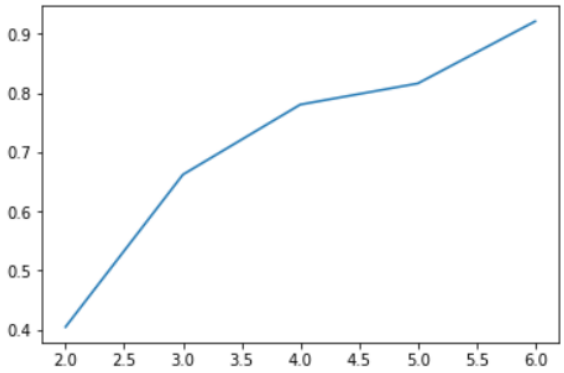
En utilisant le critère de Davies-Bouldin, la méthode est similaire :

- On applique une méthode de K-Means dont l'initialisation n'est pas totalement aléatoire (la méthode basique) au jeu de données en faisant varier K (on choisit de ne pas utiliser le K-Means basique car l'aléatoire total biaise les résultats) ;

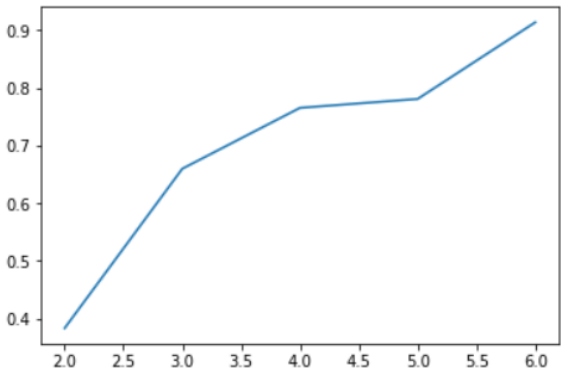
- On trace le graphe des indices de Davies-Bouldin des partitionnements obtenus en fonction de K ;

- On regarde la valeur de K pour laquelle l'indice de Davies-Bouldin est minimal : cette valeur est optimale.

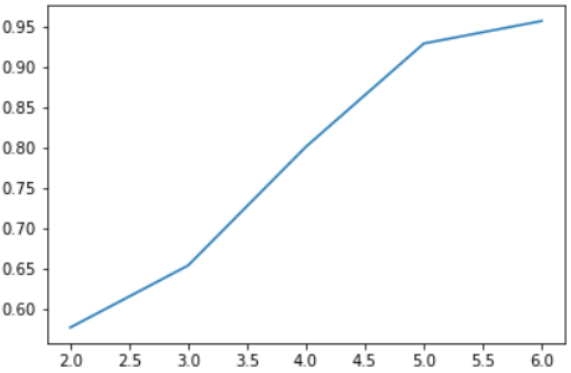
Voici donc différents graphes de l'indice de Davies-Bouldin en fonction de K (on commence à K=2, autrement l'indice est nul) :



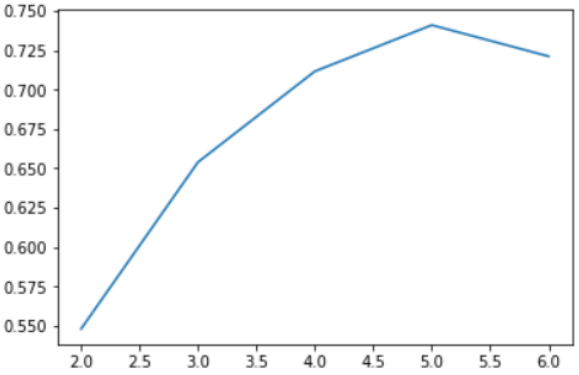
K-Means++



Global K-Means



initialisation par le mal classé



approche incrémentale

On observe que dans chaque cas, l'indice de Davies-Bouldin est croissant en fonction de K, à l'exception de l'approche incrémentale où la valeur à K=6 et plus faible que celle à K=5. Ce résultat peut être expliquer par le fait que plus le nombre de *clusters* est petit, plus ils sont séparables. Toutefois, cela nous induit donc en erreur dans la détermination du nombre de *clusters* K, même si les résultats ne sont pas incohérents et que K=3 est toujours la 2^{e} valeur la plus faible. D'ailleurs, dans le cas de l'initialisation par le mal classé, la différence entre K=2 et K=3 est faible.

Dans notre cas, cette méthode n'est par conséquent pas très concluante mais apporte de plutôt bons résultats généralement.

Conclusion

Durant ce projet, nous avons pu mettre en œuvre l'algorithme des K-Means et ses variantes afin de le rendre plus efficace. Il est simple à comprendre et à appliquer, et fournit des résultats globalement satisfaisants. Cependant, nous avons pu observer que cet algorithme est limité. En effet, nous nous sommes interrogés sur les raisons de ses résultats approximatifs. Au cours de nos recherches, nous avons constaté que l'algorithme des K-Means présentait des inconvénients autres que son initialisation et le problème de détermination du nombre de clusters. Outre le fait qu'une expérience scientifique ne peut jamais réellement être menée à la perfection, il subsiste un facteur qui nuit à son efficacité : la forme des clusters car l'algorithme suppose que nous traitons avec des clusters sphériques et de taille ou densité égale.

Ainsi, l'algorithme des K-Means présente des performances moins compétitives que d'autres algorithmes plus sophistiquées du fait d'une variance élevée de ses résultats.

Bibliographie

- Shai Shalev-Shwartz, Shai Ben-David, 2014. *Understanding Machine Learning:From Theory to Algorithms*, publié par *Cambridge University Press*. Chapitre 22, *Clustering*, p. 307-322.

- Z.Guellil, L.Zaoui. *Proposition d'une solution au problème d'initialisation cas du K-means*, Université des sciences et de la technologie d'Oran MB, Université Mohamed Boudiaf USTO – ORAN – Algérie.

- Site de Xavier Dupré, section *Clustering*, Chapitre k-means : http://www.xavierdupre.fr/app/mlstatpy/helpsphinx/c_clus/kmeans.html#.

- Site de scikit-learn : https://scikit-learn.org/stable/index.html

Annexe

```
%matplotlib inline

from sklearn.datasets import load_iris
import numpy as np
import matplotlib as mlp
import matplotlib.pyplot as plt
from math import *
from sklearn.cluster import KMeans
from sklearn.decomposition import PCA

# On charge le set de donnees iris de sklearn

iris = load_iris()

# On regarde le set de donnees et ses differents attributs

print(iris)
print(iris.data)
print(iris.feature_names)
print(iris.target)
print(iris.target_names)

# On regarde la forme des donnees et les valeurs min et max de chaque colonne pour l'initialisation des centres

print(len(iris.data))
print(iris.data.shape)
print(iris.data[:,0].min())
print(iris.data[:,0].max())
print(iris.data[:,1].min())
print(iris.data[:,1].max())
print(iris.data[:,2].min())
print(iris.data[:,2].max())
print(iris.data[:,3].min())
print(iris.data[:,3].max())

# Fonction pour initialiser les centres dans l'algorithme k-means, qui renvoie un numpy arrray de dimension (K,4)

def init(K):
    column0 = 4.0*np.random.rand(K,1)+4.0
    column1 = 2.5*np.random.rand(K,1)+2.0
    column2 = 6.0*np.random.rand(K,1)+1.0
    column3 = 2.5*np.random.rand(K,1)
    return np.concatenate((column0, column1, column2, column3), axis = 1)

centers = init(3)

centers.shape
```

```
# Fonction calculant la distance entre 2 points de meme dimension

def distance(x1, x2):
    dist = 0
    for i in range(len(x1)):
        dist += (x1[i]-x2[i])**2
    return sqrt(dist)

print(distance(centers[0], iris.data[0]))

x1 = np.zeros(3)
x2 = np.ones(3)
print(distance(x1,x2))
print(sqrt(3))

# Fonction permettant de trouver le centre le plus proche d'un point x quelconque, elle renvoie l'indice du centre dans le numpy array centers

def min_dist(x, centers):
    mini = distance(x, centers[0])
    mini_index = 0
    for i in range(1,len(centers)):
        if mini>distance(x, centers[i]):
            mini = distance(x, centers[i])
            mini_index = i
    return mini_index

for i in range(20):
    print(min_dist(iris.data[i], centers))

x3 = 2*np.ones(3)
list_test = [x2,x3]
print(min_dist(x1, list_test))

# Fonction qui fait le partitionnement des donnees a chaque centre le plus proche
# elle renvoie la liste clusters de la taille du nombre de donnes contenant les indices des centres les plus proches, correspondant aux clusters
# et la taille de chaque clusters dans une liste clusters_size

def partitionning(data, centers):
    clusters = []
    clusters_size = np.zeros(len(centers))
    for i in range(len(data)):
        j=min_dist(data[i], centers)
        clusters.append(j)
        clusters_size[j]+=1
    return clusters, clusters_size

clusters, clusters_size = partitionning(iris.data, centers)
print(len(clusters))
print(clusters)
print(clusters_size)
```

```
# Fonction qui met a jour les centres en les remplacant par les barycentres de chaque cluster

def update_centers(data, centers, clusters, clusters_size):
    for i in range(len(centers)):
        if clusters_size[i]!=0:
            new_center = np.zeros(len(data[0]))
            for j in range(len(clusters)):
                if clusters[j]==i:
                    new_center += data[j]
            new_center *= 1.0/clusters_size[i]
            centers[i] = new_center

update_centers(iris.data, centers, clusters, clusters_size)

# Fonction qui retourne True si chaque element x1[i][j] du numpy array x1 est egal a chaque element x2[i][j] du numpy array x2
# Utile pour savoir quand s'arreter dans l'algorithme k-means

def equal(x1,x2):
    for i in range(len(x1)):
        for j in range(len(x1[i])):
            if x1[i][j]!=x2[i][j]:
                return False
    return True

# Fonction k-means reutilisant les precedentes fonctions, renvoyant les centres, les clusters et les tailles des clusters

def K_means(data, K):
    centers = init(K)
    clusters, clusters_size = partitionning(data, centers)
    centers_update = centers
    update_centers(data, centers_update, clusters, clusters_size)
    while equal(centers_update,centers)==False:
        centers = centers_update
        clusters, clusters_size = partitionning(data, centers)
        update_centers(data, centers_update, clusters, clusters_size)
    return centers, clusters, clusters_size

# On teste avec 5 initialisations differentes

centers0, clusters0, clusters_size0 = K_means(iris.data, 3)
centers1, clusters1, clusters_size1 = K_means(iris.data, 3)
centers2, clusters2, clusters_size2 = K_means(iris.data, 3)
centers3, clusters3, clusters_size3 = K_means(iris.data, 3)
centers4, clusters4, clusters_size4 = K_means(iris.data, 3)

# On regarde les tailles des clusters dans chaque essai et les vraie classes des donnees

print(clusters_size0)
print(clusters_size1)
print(clusters_size2)
print(clusters_size3)
```

```
print(clusters_size4)

print(iris.target)

# On applique la methode k-means++ de sklearn a nos donnees

kmeans = KMeans(n_clusters=3, random_state=0).fit(iris.data)

real_clusters_size = np.zeros(3)
for i in kmeans.labels_:
    real_clusters_size[int(i)] += 1

print(real_clusters_size)

# On affiche l'epaisseur des sepales en fonction de leurs longueur pour observer la vraie repartition des classes
# et celles dans nos differents essais de k-means

colormap=np.array(['Red','green','blue'])
plt.scatter(iris.data[:,0], iris.data[:,1],c=colormap[iris.target],s=40)
plt.show()

plt.scatter(iris.data[:,0], iris.data[:,1],c=colormap[kmeans.labels_],s=40)
plt.scatter(kmeans.cluster_centers_[:,0], kmeans.cluster_centers_[:,1], c="k", marker="+")
plt.show()

plt.scatter(iris.data[:,0], iris.data[:,1],c=colormap[clusters0],s=40)
plt.scatter(centers0[:,0], centers0[:,1], c="k", marker="+")
plt.show()

plt.scatter(iris.data[:,0], iris.data[:,1],c=colormap[clusters1],s=40)
plt.scatter(centers1[:,0], centers1[:,1], c="k", marker="+")
plt.show()

plt.scatter(iris.data[:,0], iris.data[:,1],c=colormap[clusters2],s=40)
plt.scatter(centers2[:,0], centers2[:,1], c="k", marker="+")
plt.show()

plt.scatter(iris.data[:,0], iris.data[:,1],c=colormap[clusters3],s=40)
plt.scatter(centers3[:,0], centers3[:,1], c="k", marker="+")
plt.show()

plt.scatter(iris.data[:,0], iris.data[:,1],c=colormap[clusters4],s=40)
plt.scatter(centers4[:,0], centers4[:,1], c="k", marker="+")
plt.show()

# On calcule l'erreur quadratique de nos differents essais

def err_quadr(data,clusters,centers):
    J=0
    for i in range(len(centers)):
        for j in range(len(clusters)):
            if clusters[j]==i:
```

```
            J += distance(data[j],centers[i])**2
    return J

print(clusters_size0)
print(clusters_size1)
print(clusters_size2)
print(clusters_size3)
print(clusters_size4)

print(err_quadr(iris.data,clusters0,centers0))
print(err_quadr(iris.data,clusters1,centers1))
print(err_quadr(iris.data,clusters2,centers2))
print(err_quadr(iris.data,clusters3,centers3))
print(err_quadr(iris.data,clusters4,centers4))

# On affiche les valeurs d'erreur quadratique pour nos 5 essais aleatoires

J0 = err_quadr(iris.data,clusters0,centers0)
J1 = err_quadr(iris.data,clusters1,centers1)
J2 = err_quadr(iris.data,clusters2,centers2)
J3 = err_quadr(iris.data,clusters3,centers3)
J4 = err_quadr(iris.data,clusters4,centers4)

plt.plot([0,1,2,3,4], [J0,J1,J2,J3,J4])
plt.show()

# On applique k-means++ pour k allant de 1 a 6

kmeans_k1 = KMeans(n_clusters=1, init="k-means++", random_state=None).fit(iris.data)
kmeans_k2 = KMeans(n_clusters=2, init="k-means++", random_state=None).fit(iris.data)
kmeans_k3 = KMeans(n_clusters=3, init="k-means++", random_state=None).fit(iris.data)
kmeans_k4 = KMeans(n_clusters=4, init="k-means++", random_state=None).fit(iris.data)
kmeans_k5 = KMeans(n_clusters=5, init="k-means++", random_state=None).fit(iris.data)
kmeans_k6 = KMeans(n_clusters=6, init="k-means++", random_state=None).fit(iris.data)

real_clusters_size_k1 = 150*np.ones(1)

real_clusters_size_k2 = np.zeros(2)
for i in kmeans_k2.labels_:
    real_clusters_size_k2[int(i)] += 1

real_clusters_size_k3 = np.zeros(3)
for i in kmeans_k3.labels_:
    real_clusters_size_k3[int(i)] += 1

real_clusters_size_k4 = np.zeros(4)
for i in kmeans_k4.labels_:
    real_clusters_size_k4[int(i)] += 1

real_clusters_size_k5 = np.zeros(5)
for i in kmeans_k5.labels_:
    real_clusters_size_k5[int(i)] += 1
```

```
real_clusters_size_k6 = np.zeros(6)
for i in kmeans_k6.labels_:
    real_clusters_size_k6[int(i)] += 1

print(real_clusters_size_k1)
print(real_clusters_size_k2)
print(real_clusters_size_k3)
print(real_clusters_size_k4)
print(real_clusters_size_k5)
print(real_clusters_size_k6)

# On observe l'evolution de l'erreur quadratique en fonction de k

J_k1 = err_quadr(iris.data, kmeans_k1.labels_, kmeans_k1.cluster_centers_)
J_k2 = err_quadr(iris.data, kmeans_k2.labels_, kmeans_k2.cluster_centers_)
J_k3 = err_quadr(iris.data, kmeans_k3.labels_, kmeans_k3.cluster_centers_)
J_k4 = err_quadr(iris.data, kmeans_k4.labels_, kmeans_k4.cluster_centers_)
J_k5 = err_quadr(iris.data, kmeans_k5.labels_, kmeans_k5.cluster_centers_)
J_k6 = err_quadr(iris.data, kmeans_k6.labels_, kmeans_k6.cluster_centers_)

print(J_k1)
print(J_k2)
print(J_k3)
print(J_k4)
print(J_k5)
print(J_k6)

K = [1,2,3,4,5,6]
J = [J_k1,J_k2,J_k3,J_k4,J_k5, J_k6]

plt.plot(K, J)
plt.show()

# On calcule l'indice de Davies-Bouldin en le decomposant en 3 fonctions

# D'abord les termes S

def S(data,clusters,clusters_size,i,centers):
    S=0
    for n in range(len(clusters)):
        if clusters[n]==i:
            S=S+distance(data[n],centers[i])
    S=S/(clusters_size[i])
    return S

# Ensuite chaque terme de la somme

def DBterm(data,clusters,clusters_size,i,centers):
    DBterm=0
    for n in range (len(centers)):
        if n!=i:
            DBterm=max(DBterm,(S(data,clusters,clusters_size,i,centers)
+S(data,clusters,clusters_size,n,centers))/(distance(centers[i],centers[n])))
```

```
    return DBterm

# L'indice final

def DB(data,clusters,clusters_size,centers):
    DB=0
    for i in range (len(centers)):
        DB=DB+DBterm(data,clusters,clusters_size,i,centers)
    DB=DB/len(centers)
    return DB

DB0 = DB(iris.data,clusters0,clusters_size0,centers0)

print(DB0)

# On recalcule l'indice DB en changeant le calcul des termes S, maintenant on prend p=2 dans la formule generale
# ce qui correspond plus a un ecart-type. Mais on observe des resultats tres similaires.

def S_2(data,clusters,clusters_size,i,centers):
    S=0
    for n in range(len(clusters)):
        if clusters[n]==i:
            S=S+distance(data[n],centers[i])**2
    S=sqrt(S/(clusters_size[i]))
    return S

def DBterm_2(data,clusters,clusters_size,i,centers):
    DBterm=0
    for n in range (len(centers)):
        if n!=i:
            DBterm=max(DBterm,(S_2(data,clusters,clusters_size,i,centers)
+S_2(data,clusters,clusters_size,n,centers))/(distance(centers[i],centers[n])))
    return DBterm


def DB_2(data,clusters,clusters_size,centers):
    DB=0
    for i in range (len(centers)):
        DB=DB+DBterm_2(data,clusters,clusters_size,i,centers)
    DB=DB/len(centers)
    return DB

DB0_2 = DB_2(iris.data,clusters0,clusters_size0,centers0)

print(DB0₂)

# On regarde l'evolution de l'indice DB en fonction de k pour la methode k-means++

DBB_k2 = DB(iris.data,kmeans_k2.labels_,real_clusters_size_k2,kmeans_k2.cluster_centers_)
DBB_k3 = DB(iris.data,kmeans_k3.labels_,real_clusters_size_k3,kmeans_k3.cluster_centers_)
DBB_k4 = DB(iris.data,kmeans_k4.labels_,real_clusters_size_k4,kmeans_k4.cluster_centers_)
DBB_k5 = DB(iris.data,kmeans_k5.labels_,real_clusters_size_k5,kmeans_k5.cluster_centers_)
```

```
DBB_k6 = DB(iris.data,kmeans_k6.labels_,real_clusters_size_k6,kmeans_k6.cluster_centers_)


K = [2,3,4,5,6]
DBBB = [DBB_k2,DBB_k3,DBB_k4,DBB_k5,DBB_k6]

plt.plot(K, DBBB)
plt.show()

# On regarde de meme l'evolution de l'indice DB en fonction de k pour la methode k-means++ avec la formule 2

DBB_2_k2 = DB_2(iris.data,kmeans_k2.labels_,real_clusters_size_k2,kmeans_k2.cluster_centers_)
DBB_2_k3 = DB_2(iris.data,kmeans_k3.labels_,real_clusters_size_k3,kmeans_k3.cluster_centers_)
DBB_2_k4 = DB_2(iris.data,kmeans_k4.labels_,real_clusters_size_k4,kmeans_k4.cluster_centers_)
DBB_2_k5 = DB_2(iris.data,kmeans_k5.labels_,real_clusters_size_k5,kmeans_k5.cluster_centers_)
DBB_2_k6 = DB_2(iris.data,kmeans_k6.labels_,real_clusters_size_k6,kmeans_k6.cluster_centers_)


K = [2,3,4,5,6]
DBBB_2 = [DBB_2_k2,DBB_2_k3,DBB_2_k4,DBB_2_k5,DBB_2_k6]

plt.plot(K, DBBB_2)
plt.show()

# Fonction qui renvoie un numpy array contenant les coordonnees du barycentre d'un ensemble de donnees

def barycentre(data):
    G=np.zeros(len(data[0]))
    for i in range(len(data)):
        G += data[i]
    G = G/len(data)
    return G

print(barycentre(iris.data))

# Fonction appliquant la methode global k-means

def global_K_means(data, K):
    # initialisation avec le premier centre barycentre des donnees
    C1 = np.reshape(barycentre(data), (1,len(data[0])))
    centers = C1
    new_centers = []
    for n in range(K-1):
        # a chaque iteration, on teste chaque donnee en la rajoutant aux centres, partitionnant les donnees,
        # calculant l'erreur quadratique correspondant, et on garde le point dont l'erreur quadratique est
minimale
        new_centers.append(data[0])
        centers2 = np.append(centers, np.reshape(new_centers[n], (1,len(data[0]))), axis=0)
        clusters, clusters_size = partitionning(data, centers2)
        J = err_quadr(data,clusters,centers2)
        for i in data:
            centers2 = np.append(centers, np.reshape(i, (1,len(data[0]))), axis=0)
```

```
            clusters, clusters_size = partitionning(data, centers2)
            if J > err_quadr(data,clusters,centers2):
                J = err_quadr(data,clusters,centers2)
                new_centers[n] = i
        # et on applique k-means classique quand on trouve le centre supplementaire a chaque etape jusqu'a la convergence
        # jusqu'a ce qu'on ait le nombre de centres souhaites
        centers = np.append(centers,  np.reshape(new_centers[n], (1,len(data[0]))), axis=0)
        clusters, clusters_size = partitionning(data, centers)
        centers_update = centers
        update_centers(data, centers_update, clusters, clusters_size)
        while equal(centers_update,centers)==False:
                centers = centers_update
                clusters, clusters_size = partitionning(data, centers)
                update_centers(data, centers_update, clusters, clusters_size)
    return centers, clusters, clusters_size

centers0_glob, clusters0_glob, clusters_size0_glob = global_K_means(iris.data, 3)

# on analyse le resultat obtenu par cette methode

print(clusters_size0_glob)
print(err_quadr(iris.data, clusters0_glob, centers0_glob))
print(DB(iris.data,clusters0_glob,clusters_size0_glob,centers0_glob))

plt.scatter(iris.data[:,0], iris.data[:,1],c=colormap[clusters0_glob],s=40)
plt.scatter(centers0_glob[:,0], centers0_glob[:,1], c="k", marker="+")
plt.show()

# On fait les etudes pour le choix de k avec cette methode

centers_glob_k2, clusters_glob_k2, clusters_size_glob_k2 = global_K_means(iris.data, 2)
centers_glob_k3, clusters_glob_k3, clusters_size_glob_k3 = global_K_means(iris.data, 3)
centers_glob_k4, clusters_glob_k4, clusters_size_glob_k4 = global_K_means(iris.data, 4)
centers_glob_k5, clusters_glob_k5, clusters_size_glob_k5 = global_K_means(iris.data, 5)
centers_glob_k6, clusters_glob_k6, clusters_size_glob_k6 = global_K_means(iris.data, 6)

print(clusters_size_glob_k2)
print(clusters_size_glob_k3)
print(clusters_size_glob_k4)
print(clusters_size_glob_k5)
print(clusters_size_glob_k6)

# Methode de l'erreur quadratique
# on se permet pour k=1 d'utiliser k-means++ car cela revient au meme, le seul centre est le barycentre

J_k1 = err_quadr(iris.data, kmeans_k1.labels_, kmeans_k1.cluster_centers_)
J_glob_k2 = err_quadr(iris.data, clusters_glob_k2, centers_glob_k2)
J_glob_k3 = err_quadr(iris.data, clusters_glob_k3, centers_glob_k3)
J_glob_k4 = err_quadr(iris.data, clusters_glob_k4, centers_glob_k4)
J_glob_k5 = err_quadr(iris.data, clusters_glob_k5, centers_glob_k5)
J_glob_k6 = err_quadr(iris.data, clusters_glob_k6, centers_glob_k6)
```

```
print(J_k1)
print(J_glob_k2)
print(J_glob_k3)
print(J_glob_k4)
print(J_glob_k5)
print(J_glob_k6)

K = [1,2,3,4,5,6]
J_glob = [J_k1,J_glob_k2,J_glob_k3,J_glob_k4,J_glob_k5, J_glob_k6]

plt.plot(K, J_glob)
plt.show()

# Methode de DB

DB_glob_k2 = DB(iris.data,clusters_glob_k2,clusters_size_glob_k2,centers_glob_k2)
DB_glob_k3 = DB(iris.data,clusters_glob_k3,clusters_size_glob_k3,centers_glob_k3)
DB_glob_k4 = DB(iris.data,clusters_glob_k4,clusters_size_glob_k4,centers_glob_k4)
DB_glob_k5 = DB(iris.data,clusters_glob_k5,clusters_size_glob_k5,centers_glob_k5)
DB_glob_k6 = DB(iris.data,clusters_glob_k6,clusters_size_glob_k6,centers_glob_k6)

K_glob = [2,3,4,5,6]
DB_glob = [DB_glob_k2,DB_glob_k3,DB_glob_k4,DB_glob_k5,DB_glob_k6]

plt.plot(K_glob, DB_glob)
plt.show()

# Fonction renvoyant les indices des 2 elements les plus eloignes dans un ensemble de donnees

def max_dist(data):
    i,j = 0,1
    maxi = distance(data[0],data[1])
    for k in range(len(data)-1):
        for l in range(k+1,len(data)):
            if distance(data[k],data[l]) > maxi:
                i,j = k,l
                maxi = distance(data[k],data[l])
    return i,j

# Fonction renvoyant l'indice de l'element mal classe dans un ensemble de donnees partitionne
# celui qui est le plus eloigne de son centre

def mal_classe(data, centers, clusters):
    i = 0
    maxi = distance(data[0],centers[clusters[0]])
    for k in range(1,len(data)):
        if distance(data[k],centers[clusters[k]]) > maxi:
            i = k
            maxi = distance(data[k],centers[clusters[k]])
    return i

# Fonction renvoyant le numpy array des centres par l'initialisation par le mal classe
```

```
def init_mal_classe(data, K):
    i,j = max_dist(data)
    C1, C2 = np.reshape(data[i], (1,len(data[0]))), np.reshape(data[j], (1,len(data[0])))
    centers = np.append(C1, C2, axis=0)
    for n in range(K-2):
        clusters, clusters_size = partitionning(data, centers)
        k = mal_classe(data, centers, clusters)
        centers = np.append(centers, np.reshape(data[k], (1,len(data[0]))), axis=0)
    return centers

# Fonction appliquant k-means apres l'initialisation par le mal classe

def K_means_mal_classe(data, K):
    centers = init_mal_classe(data, K)
    clusters, clusters_size = partitionning(data, centers)
    centers_update = centers
    update_centers(data, centers_update, clusters, clusters_size)
    while equal(centers_update,centers)==False:
        centers = centers_update
        clusters, clusters_size = partitionning(data, centers)
        update_centers(data, centers_update, clusters, clusters_size)
    return centers, clusters, clusters_size

centers0_mal_cl, clusters0_mal_cl, clusters_size0_mal_cl = K_means_mal_classe(iris.data, 3)

# On analyse les resultats par cette methode

print(clusters_size0_mal_cl)
print(err_quadr(iris.data, clusters0_mal_cl, centers0_mal_cl))
print(DB(iris.data,clusters0_mal_cl,clusters_size0_mal_cl,centers0_mal_cl))

print(clusters0_mal_cl)
plt.scatter(iris.data[:,0], iris.data[:,1],c=colormap[clusters0_mal_cl],s=40)
plt.scatter(centers0_mal_cl[:,0], centers0_mal_cl[:,1], c="k", marker="+")
plt.show()

# On fait les etudes pour le choix de k avec cette methode

centers_mal_cl_k2, clusters_mal_cl_k2, clusters_size_mal_cl_k2 = K_means_mal_classe(iris.data, 2)
centers_mal_cl_k3, clusters_mal_cl_k3, clusters_size_mal_cl_k3 = K_means_mal_classe(iris.data, 3)
centers_mal_cl_k4, clusters_mal_cl_k4, clusters_size_mal_cl_k4 = K_means_mal_classe(iris.data, 4)
centers_mal_cl_k5, clusters_mal_cl_k5, clusters_size_mal_cl_k5 = K_means_mal_classe(iris.data, 5)
centers_mal_cl_k6, clusters_mal_cl_k6, clusters_size_mal_cl_k6 = K_means_mal_classe(iris.data, 6)

print(clusters_size_mal_cl_k2)
print(clusters_size_mal_cl_k3)
print(clusters_size_mal_cl_k4)
print(clusters_size_mal_cl_k5)
print(clusters_size_mal_cl_k6)
```

```
# Methode de l'erreur quadratique

J_mal_cl_k2 = err_quadr(iris.data, clusters_mal_cl_k2, centers_mal_cl_k2)
J_mal_cl_k3 = err_quadr(iris.data, clusters_mal_cl_k3, centers_mal_cl_k3)
J_mal_cl_k4 = err_quadr(iris.data, clusters_mal_cl_k4, centers_mal_cl_k4)
J_mal_cl_k5 = err_quadr(iris.data, clusters_mal_cl_k5, centers_mal_cl_k5)
J_mal_cl_k6 = err_quadr(iris.data, clusters_mal_cl_k6, centers_mal_cl_k6)

print(J_mal_cl_k2)
print(J_mal_cl_k3)
print(J_mal_cl_k4)
print(J_mal_cl_k5)
print(J_mal_cl_k6)

K = [2,3,4,5,6]
J_mal_cl = [J_mal_cl_k2,J_mal_cl_k3,J_mal_cl_k4,J_mal_cl_k5, J_mal_cl_k6]

plt.plot(K, J_mal_cl)
plt.show()

# Methode de DB

DB_mal_cl_k2 = DB(iris.data,clusters_mal_cl_k2,clusters_size_mal_cl_k2,centers_mal_cl_k2)
DB_mal_cl_k3 = DB(iris.data,clusters_mal_cl_k3,clusters_size_mal_cl_k3,centers_mal_cl_k3)
DB_mal_cl_k4 = DB(iris.data,clusters_mal_cl_k4,clusters_size_mal_cl_k4,centers_mal_cl_k4)
DB_mal_cl_k5 = DB(iris.data,clusters_mal_cl_k5,clusters_size_mal_cl_k5,centers_mal_cl_k5)
DB_mal_cl_k6 = DB(iris.data,clusters_mal_cl_k6,clusters_size_mal_cl_k6,centers_mal_cl_k6)

K_mal_cl = [2,3,4,5,6]
DB_mal_cl = [DB_mal_cl_k2,DB_mal_cl_k3,DB_mal_cl_k4,DB_mal_cl_k5,DB_mal_cl_k6]

plt.plot(K_mal_cl, DB_mal_cl)
plt.show()

# Fonction appliquant l'approche incrementale de k-means

def incr_K_means(data, K):
    # initialisation des 2 premiers centres avec les 2 elements les plus eloignes du set
    i,j = max_dist(data)
    C1, C2 = np.reshape(data[i], (1,len(data[0]))), np.reshape(data[j], (1,len(data[0])))
    centers = np.append(C1, C2, axis=0)
    clusters, clusters_size = partitionning(data, centers)
    for n in range(K-2):
        # a chaque iteration on prend le mal classe et on applique k-means
        # jusqu'a ce qu'on ait le nombre de centres souhaites
        k = mal_classe(data, centers, clusters)
        centers = np.append(centers, np.reshape(data[k], (1,len(data[0]))), axis=0)
        clusters, clusters_size = partitionning(data, centers)
        centers_update = centers
        update_centers(data, centers_update, clusters, clusters_size)
        while equal(centers_update,centers)==False:
                centers = centers_update
                clusters, clusters_size = partitionning(data, centers)
```

```
            update_centers(data, centers_update, clusters, clusters_size)
    return centers, clusters, clusters_size

centers0_incr, clusters0_incr, clusters_size0_incr = incr_K_means(iris.data, 3)

# On analyse les resultats par cette methode

print(clusters_size0_incr)
print(err_quadr(iris.data, clusters0_incr, centers0_incr))
print(DB(iris.data,clusters0_incr,clusters_size0_incr,centers0_incr))

print(clusters0_incr)
plt.scatter(iris.data[:,0], iris.data[:,1],c=colormap[clusters0_incr],s=40)
plt.scatter(centers0_incr[:,0], centers0_incr[:,1], c="k", marker="+")
plt.show()

# On fait les etudes pour le choix de k avec cette methode

centers_incr_k2, clusters_incr_k2, clusters_size_incr_k2 = incr_K_means(iris.data, 2)
centers_incr_k3, clusters_incr_k3, clusters_size_incr_k3 = incr_K_means(iris.data, 3)
centers_incr_k4, clusters_incr_k4, clusters_size_incr_k4 = incr_K_means(iris.data, 4)
centers_incr_k5, clusters_incr_k5, clusters_size_incr_k5 = incr_K_means(iris.data, 5)
centers_incr_k6, clusters_incr_k6, clusters_size_incr_k6 = incr_K_means(iris.data, 6)

print(clusters_size_incr_k2)
print(clusters_size_incr_k3)
print(clusters_size_incr_k4)
print(clusters_size_incr_k5)
print(clusters_size_incr_k6)

# Methode de l'erreur quadratique

J_incr_k2 = err_quadr(iris.data, clusters_incr_k2, centers_incr_k2)
J_incr_k3 = err_quadr(iris.data, clusters_incr_k3, centers_incr_k3)
J_incr_k4 = err_quadr(iris.data, clusters_incr_k4, centers_incr_k4)
J_incr_k5 = err_quadr(iris.data, clusters_incr_k5, centers_incr_k5)
J_incr_k6 = err_quadr(iris.data, clusters_incr_k6, centers_incr_k6)

print(J_incr_k2)
print(J_incr_k3)
print(J_incr_k4)
print(J_incr_k5)
print(J_incr_k6)

K = [2,3,4,5,6]
J_incr = [J_incr_k2,J_incr_k3,J_incr_k4,J_incr_k5, J_incr_k6]

plt.plot(K, J_incr)
plt.show()
```

```
# Methode de DB

DB_incr_k2 = DB(iris.data,clusters_incr_k2,clusters_size_incr_k2,centers_incr_k2)
DB_incr_k3 = DB(iris.data,clusters_incr_k3,clusters_size_incr_k3,centers_incr_k3)
DB_incr_k4 = DB(iris.data,clusters_incr_k4,clusters_size_incr_k4,centers_incr_k4)
DB_incr_k5 = DB(iris.data,clusters_incr_k5,clusters_size_incr_k5,centers_incr_k5)
DB_incr_k6 = DB(iris.data,clusters_incr_k6,clusters_size_incr_k6,centers_incr_k6)

K_incr = [2,3,4,5,6]
DB_incr = [DB_incr_k2,DB_incr_k3,DB_incr_k4,DB_incr_k5,DB_incr_k6]

plt.plot(K_incr, DB_incr)
plt.show()

# On compare les differentes methodes en evoluant k

# Par l'erreur quadratique

K = [2,3,4,5,6]

J = [J_k2,J_k3,J_k4,J_k5, J_k6]

J_glob = [J_glob_k2,J_glob_k3,J_glob_k4,J_glob_k5, J_glob_k6]

J_mal_cl = [J_mal_cl_k2,J_mal_cl_k3,J_mal_cl_k4,J_mal_cl_k5, J_mal_cl_k6]

J_incr = [J_incr_k2,J_incr_k3,J_incr_k4,J_incr_k5, J_incr_k6]

plt.plot(K, J, "b")
plt.plot(K, J_glob, "g")
plt.plot(K, J_mal_cl, "r")
plt.plot(K, J_incr, "y")

plt.show()

plt.plot(K, J, "b")
plt.plot(K, J_glob, "g")
plt.plot(K, J_mal_cl, "r")

plt.show()

# Par le critère de DB

K = [2,3,4,5,6]

DBBB = [DBB_k2,DBB_k3,DBB_k4,DBB_k5,DBB_k6]

DB_glob = [DB_glob_k2,DB_glob_k3,DB_glob_k4,DB_glob_k5,DB_glob_k6]

DB_mal_cl = [DB_mal_cl_k2,DB_mal_cl_k3,DB_mal_cl_k4,DB_mal_cl_k5,DB_mal_cl_k6]

DB_incr = [DB_incr_k2,DB_incr_k3,DB_incr_k4,DB_incr_k5,DB_incr_k6]
```

```
plt.plot(K, DBBB, "b")
plt.plot(K_glob, DB_glob, "g")
plt.plot(K_mal_cl, DB_mal_cl, "r")
plt.plot(K_incr, DB_incr, "y")
plt.show()

# On compare les differentes  methodes pour k=3, avec et sans un des meilleurs resultats aleatoires qu'on ait obtenu

# Par l'erreur quadratique

J_rand = 177.9083199730297

plt.plot([0,1,2,3,4],[J_rand, J_k3, J_glob_k3, J_mal_cl_k3, J_incr_k3])
plt.show()

plt.plot([1,2,3,4],[J_k3, J_glob_k3, J_mal_cl_k3, J_incr_k3])
plt.show()

# Par le critère de DB

DB_rand = 0.821020412792

plt.plot([0,1,2,3,4],[DB_rand, DBB_k3, DB_glob_k3, DB_mal_cl_k3, DB_incr_k3])
plt.show()

plt.plot([1,2,3,4],[DBB_k3, DB_glob_k3, DB_mal_cl_k3, DB_incr_k3])
plt.show()
```