In [1]:
```
import numpy as np
import matplotlib.pyplot as plot
```

K-Means

Mise en situation

Je cherche à faire de la compression d'image, en essayant de regrouper des couleurs similaires en une seule.

Je voudrais un algorithme qui me regroupe les couleurs similaires tout seul.

Les K-moyennes (ou K-Means)

Le monde de l'apprentissage non supervisé

Depuis deux semaines, on n'a vu que des algorithmes de type "apprentissage supervisé" :

- apprendre à sortir la classe d'un objet, à partir d'un ensemble connu d'objets avec leurs classes
- apprendre à modéliser une fonction numérique à partir d'exemples d'entrées-sorties

Il existe aut*re* domaine majeur dans l'apprentissage automatique : l'apprentissage non-supervisé. En un mot, j'ignore ce qu'il faut faire exactement mais je laisse l'algorithme choisir au mieux. Ou à peu près :)

Le principe

Le principe de cet algorithme est le suivant:

- je dispose d'un ensemble de *m* données (vecteurs de réels, ça ça ne change pas)
- je les classe en k "clusters"
- à chaque itération, je voudrais trouver une meilleure classification que la précédente

Bon, il reste à définir "meilleure". Et accessoirement, l'algorithme s'arrête : il n'y a que k^m classifications différentes, donc un nombre fini, et forcément on finira une meilleure classification je ne risque pas de boucler sur des partitionnements déjà vu (puisqu'ils sont moins bons).

Terminologie

- Une donnée X est définie par son vecteur de réels
- Une classe z est définie par son centre C_z, un autre vecteur de même dimension
- La donnée appartient à la classe z si et seulement si la distance avec le centre de de la classe, C_z, est inférieure à la distance avec n'importe quel autre centre de classe.

Par distance, on entend norme euclidienne.

Autrement dit si $||X - C_z|| = min_{c_i}(||X - C_i||)$, alors j est la classe à laquelle X appartient.

L'algorithme

- On commence par initialiser les k centres un peu au hasard
- On boucle:
  - On classe les données par rapport aux centres
  - On calcule le barycentre des données de chaque classe, pour définir de nouveaux centres
- On s'arrête quand les centres ne bougent plus.

La fonction de coût

Comme on l'a déjà vu, et comme pour beaucoup d'algorithmes du monde du machine learning, on cherche à minimiser une fonction de coût, ici, elle vaut :

$J(C_1, C_2, \ldots, C_k) = \frac{1}{m}\sum_{z}\sum_{j|X_j \in C_z} ||X - C_z||^2$, avec f la fonction de classification. On additionne toutes les distances des points avec leur plus proche centre.

Autre formulation : on mesure la distance globale des points avec leurs classes. Et on essaye de minimiser ça. Par contre ça va se faire sans descente de gradient.

Lorsqu'on commence une itération, on a une certaine valeur de coût global. On commence par faire une classification : le coût ne change pas, puisque les centres ne changent pas.

Une fois qu'on a identifié les éléments d'une classe, on aimerait remplacer le centre initial par le meilleur point possible en terme de coût. Ce point, c'est le barycentre. A ce moment là on bascule sur de nouvelles valeurs pour les C_i, qui va faire diminuer la fonction de coût.

Le choix de départ

Plutôt que de prendre des points au hasard dans l'espace vectoriel considéré, il est déjà plus utile de piocher des points au hasard dans le jeu de données : on évite de se retrouver trop loin.

Cependant, il y a un faible dans cette méthode. Supposons que je veuille identifier 3 clusters, et que mes points de départs soient dans le cluster "vert" pour deux d'entre eux, et dans un des deux autres pour le 3ème : je risque de découper mon cluster numéro 1 en deux parties et de regrouper mes deux autres clusters en un seul...

Il y a plusieurs méthodes pour contourner ce problème, aucune n'est parfaite mais la plus efficace à ce jour reste celle des *k-means++*. En gros:

- Le premier centre au hasard (en piochant dans les données disponibles bien entendu)
- Pour chaque centre suivant
  - On calcule la distance minimale de chaque point avec les centres de départ déjà identifiés
  - On en fait une distribution de probabilité
  - On tire un point au hasard suivant cette distribution, et on en fait un centre

Autrement dit, quitte à ajouter une nouvelle classe, on favorise sa création près des points les plus éloignés. Cette méthode d'initialisation fournit des résultats plutôt bons dans la plupart des cas.

Simulation rapide

Validons rapidement le concept avec quelques exemples simples.

In [2]:
```
# Calculer toutes les distances
def distances(k, data, centers):
    return np.concatenate(
        np.sum((data-centers[i,:])**2, axis = 1).reshape(-1,1) for i in range(k)
    ), axis = 1)

# Trouver les classes par rapport aux centres
def find_class(k, data, centers):
    return np.argmin(distances(k, data, centers), axis = 1)

# K-Means, avec les points initiaux en paramètre
def k_means(k, data, centers):
    m = data.shape[0]
    while(True):
        classes = find_class(k, data, centers)
        new_centers = np.concatenate([np.mean(data[classes == i], axis = 0).reshape(1,-1) for i in range(k)])
        if np.sum(new_centers != centers) == 0: break;
        centers = new_centers
    return centers
```

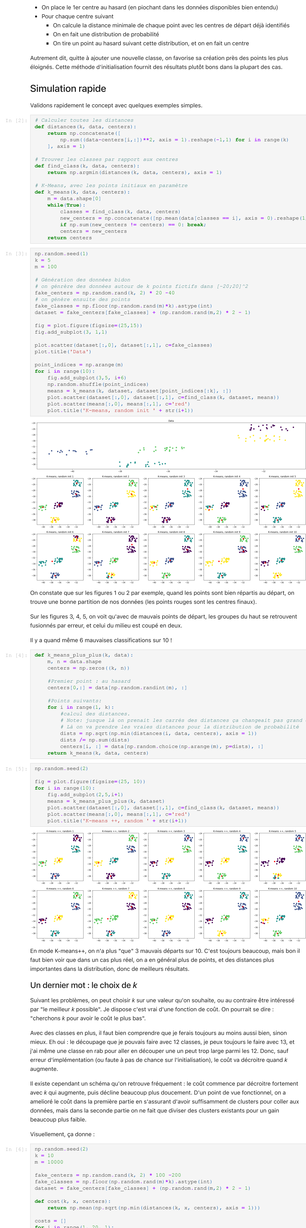
In [3]:
```
np.random.seed(1)
k = 5
m = 100

# Générations des données bidon
# On génère des données autour de k points fictifs dans [-20;20]^2
fake_centers = np.random.rand(k, 2) * 20 -40
# On génère ensuite des points
fake_classes = np.floor(np.random.rand(m)*k).astype(int)
dataset = fake_centers[fake_classes] + (np.random.rand(m,2) * 2 - 1)

fig = plot.figure(figsize=(25,15))
fig.add_subplot(3, 1,1)

plot.scatter(dataset[:,0], dataset[:,1], c=fake_classes)
plot.title("Data")

point_indices = np.arange(m)
for i in range(10):
    fig.add_subplot(3,5, i+6)
    np.random.shuffle(point_indices)
    means = k_means(k, dataset, dataset[point_indices[:k], :])
    plot.scatter(dataset[:,0], dataset[:,1], c=find_class(k, dataset, means))
    plot.scatter(means[:,0], means[:,1], c='red')
    plot.title("K-means, random init " + str(i+1))
```



On constate que sur les figures 1 ou 2 par exemple, quand les points sont bien répartis au départ, on trouve une bonne partition de nos données (les points rouges sont les centres finaux).

Sur les figures 3, 4, 5, on voit qu'avec de mauvais points de départ, les groupes du haut se retrouvent fusionnés par erreur, et celui du milieu est coupé en deux.

Il y a quand même 6 mauvaises classifications sur 10 !

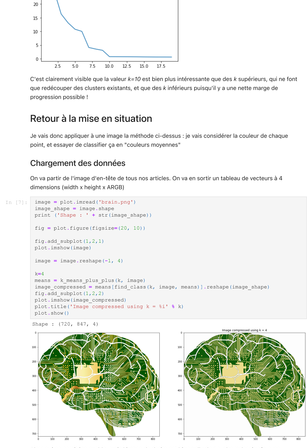
In [4]:
```
def k_means_plus_plus(k, data):
    m, n = data.shape
    centers = np.zeros((k, n))

    #Premier point : au hasard
    centers[0,:] = data[np.random.randint(m), :]

    #Puits suivants:
    for i in range(1, k):
        #calcul des distances.
        # Notez: jusque là on prenait les carrés des distances ce changeait pas grand
        # Là on va prendre les vraies distances pour la distribution de probabilité
        dists = np.sqrt(np.min(distances(i, data, centers), axis = 1))
        dists /= np.sum(dists)
        centers[i, :] = data[np.random.choice(range(m), p=dists), :]
    return k_means(k, data, centers)
```

In [5]:
```
np.random.seed(2)

fig = plot.figure(figsize=(25, 10))
for i in range(10):
    fig.add_subplot(2,5,i+1)
    means = k_means_plus_plus(k, dataset)
    plot.scatter(dataset[:,0], dataset[:,1], c=find_class(k, dataset, means))
    plot.scatter(means[:,0], means[:,1], c='red')
    plot.title("K-means ++, random " + str(i+1))
```



En mode K-means++, on n'a plus "que" 3 mauvais départs sur 10. C'est toujours beaucoup, mais bon il faut bien voir que dans un cas plus réel, on a en général plus de points, et des distances plus importantes dans la distribution, donc de meilleurs résultats.

Un dernier mot : le choix de *k*

Suivant les problèmes, on peut choisir *k* sur une valeur qu'on souhaite, ou au contraire être intéressé par "le meilleur *k* possible". Je dispose c'est vrai d'une fonction de coût. On pourrait se dire : "cherchons *k* pour avoir le coût le plus bas".

Avec des classes en plus, il faut bien comprendre que je ferais toujours au moins aussi bien, sinon mieux. En out : le découpage que je pouvais faire avec 12 classes, je peux toujours le faire avec 13, et j'ai même une classe en rab pour aller en découper une un peut trop large parmi les 12. Donc, sauf erreur d'implémentation (ou faute à pas de chance sur l'initialisation), le coût va décroître quand *k* augmente.

Il existe cependant un schéma qu'on retrouve fréquemment : le coût commence par décroître fortement avec *k* qui augmente, puis décline beaucoup plus doucement. D'un point de vue fonctionnel, on a amélioré le coût dans la première partie en s'assurant d'avoir suffisamment de clusters pour coller aux données, mais dans la seconde partie on ne fait que diviser des clusters existants pour un gain beaucoup plus faible.

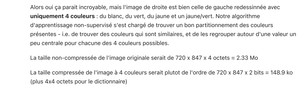
Visuellement, ça donne :

In [6]:
```
np.random.seed(2)
k = 15
m = 10000

fake_centers = np.random.rand(k, 2) * 100 -200
fake_classes = np.floor(np.random.rand(m)*k).astype(int)
dataset = fake_centers[fake_classes] + (np.random.rand(m,2) * 2 - 1)

def cost(k, x, centers):
    return np.mean(np.sqrt(np.min(distances(k, x, centers), axis = 1)))

costs = []
for i in range(1, 20, 1):
    costs.append(cost(i, dataset, k_means_plus_plus(i, dataset)))
plot.plot(range(1, 20, 1), costs)
plot.title("Cost for k classes")
```



C'est clairement visible que la valeur *k=10* est bien plus intéressante que des *k* supérieurs, qui ne font que redécouper des clusters existants, et que des *k* inférieurs puisqu'il y a une nette marge de progression possible !

Retour à la mise en situation

Je vais donc appliquer à une image la méthode ci-dessus : je vais considérer la couleur de chaque point, et essayer de classifier ça en "couleurs moyennes"

Chargement des données

On va partir de l'image d'en-tête de tous nos articles. On va en sortir un tableau de vecteurs à 4 dimensions (width x height x ARGB)

In [7]:
```
image = plot.imread("brain.png")
image_shape = image.shape
print ('Shape : ' + str(image_shape))

fig = plot.figure(figsize=(20, 10))

fig.add_subplot(1,2,1)
plot.imshow(image)

image = image.reshape(-1, 4)

k=4
means = k_means_plus_plus(k, image)
image_compressed = means[find_class(k, image, means)].reshape(image_shape)
fig.add_subplot(1,2,2)
plot.imshow(image_compressed)
plot.title('Image compressed using k = %i' % k)
plot.show()
```

```
Shape : (720, 847, 4)
```


Alors oui ça paraît incroyable, mais l'image de droite est bien celle de gauche redessinée avec **uniquement 4 couleurs** : du blanc, du vert, du jaune et un jaune/vert. Notre algorithme d'apprentissage non-supervisé s'est chargé de trouver un bon partitionnement des couleurs présentes - i.e. de trouver des couleurs qui sont similaires, et de les regrouper autour d'une valeur un peu centrale pour chacune des 4 couleurs possibles.

La taille non-compressée de l'image originale serait de 720 x 847 x 4 octets = 2.33 Mo

La taille compressée de l'image à 4 couleurs serait plutot de l'ordre de 720 x 847 x 2 bits = 148.9 ko (plus 4x4 octets pour le dictionnaire)