El algoritmo *k-means* aplicado a clasificación y procesamiento de imágenes

Clasificación

El aprendizaje de máquina (*Machine Learning*) estudia el aprendizaje automático a partir de datos (*data-driven*, gobernado por los datos) para conseguir hacer prediciones precisas a partir de observaciones con datos previos.

La clasificación automática de objetos o datos es uno de los objetivos del aprendizaje de máquina. Podemos considerar tres tipos de algoritmos:

- **Clasificación supervisada:** disponemos de un conjunto de datos (por ejemplo, imágenes de letras escritas a mano) que vamos a llamar datos de entrenamiento y cada dato está asociado a una etiqueta (a qué letra corresponde cada imagen). Construímos un modelo en la fase de entrenamiento (training) utilizando dichas etiquetas, que nos dicen si una imagen está clasificada correcta o incorrectamente por el modelo. Una vez construído el modelo podemos utilizarlo para clasificar nuevos datos que, en esta fase, ya no necesitan etiqueta para su clasificación, aunque sí la necesitan para evaluar el porcentaje de objetos bien clasificados.
- **Clasificación no supervisada:** los datos no tienen etiquetas (o no queremos utilizarlas) y estos se clasifican a partir de su estructura interna (propiedades, características).
- **Clasificación semisupervisada:** algunos datos de entrenamiento tienen etiquetas, pero no todos. Este último caso es muy típico en clasificación de imágenes, donde es habitual disponer de muchas imágenes mayormente no etiquetadas. Estos se pueden considerar algoritmos supervisados que no necesitan todas las etiquetas de los datos de entrenamiento.

En esta práctica vamos a ver varios ejemplos de utilización del algoritmo de clasificación no supervisada *k-means* para la clasificación y procesamiento de imágenes.

El algoritmo k-means

K-means es un algoritmo de clasificación no supervisada (clusterización) que agrupa objetos en *k* grupos basándose en sus características. El agrupamiento se realiza minimizando la suma de distancias entre cada objeto y el centroide de su grupo o cluster. Se suele usar la distancia cuadrática.

El algoritmo consta de tres pasos:

1. **Inicialización:** una vez escogido el número de grupos, *k*, se establecen *k* centroides en el espacio de los datos, por ejemplo, escogiéndolos aleatoriamente.
2. **Asignación objetos a los centroides:** cada objeto de los datos es asignado a su centroide más cercano.
3. **Actualización centroides:** se actualiza la posición del centroide de cada grupo tomando como nuevo centroide la posición del promedio de los objetos pertenecientes a dicho grupo.

Se repiten los pasos 2 y 3 hasta que los centroides no se mueven, o se mueven por debajo de una distancia umbral en cada paso.

El algoritmo *k-means* resuelve un **problema de optimización**, siendo la función a optimizar (minimizar) la suma de las distancias cuadráticas de cada objeto al centroide de su cluster.

Los objetos se representan con vectores reales de d dimensiones $(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n)$ y el algoritmo *k-means* construye k grupos donde se minimiza la suma de distancias de los objetos, dentro de cada grupo $\mathbf{S} = \{S_1, S_2, \ldots, S_k\}$, a su centroide. El problema se puede formular de la siguiente forma:

$$\min_{\mathbf{S}} E(\boldsymbol{\mu_i}) = \min_{\mathbf{S}} \sum_{i=1}^{k} \sum_{\mathbf{x}_j \in S_i} \|\mathbf{x}_j - \boldsymbol{\mu}_i\|^2 \quad (1)$$

donde $\mathbf{S}$ es el conjunto de datos cuyos elementos son los objetos $\mathbf{x}_j$ representados por vectores, donde cada uno de sus elementos representa una característica o atributo. Tendremos k grupos o clusters con su correspondiente centroide $\boldsymbol{\mu_i}$.

En cada actualización de los centroides, desde el punto de vista matemático, imponemos la condición necesaria de extremo a la función $E(\boldsymbol{\mu_i})$ que, para la función cuadrática (1) es

$$\frac{\partial E}{\partial \boldsymbol{\mu}_i} = 0 \implies \boldsymbol{\mu}_i^{(t+1)} = \frac{1}{\left|S_i^{(t)}\right|} \sum_{\mathbf{x}_j \in S_i^{(t)}} \mathbf{x}_j$$

y se toma el promedio de los elementos de cada grupo como nuevo centroide.

Las principales ventajas del método *k-means* son que es un método sencillo y rápido. Pero es necesario decidir el valor de k y el resultado final depende de la inicialización de los centroides. En principio no converge al mínimo global sino a un mínimo local.

Ejercicio 1

Implementar el método *k-means* de forma que clasifique datos con dos dimensiones. Utilizarlo para clasificar en tres grupos los datos generados a continuación.

Para que las figuras se inserten en este notebook, en lugar de abrir una ventana nueva, ejecutamos

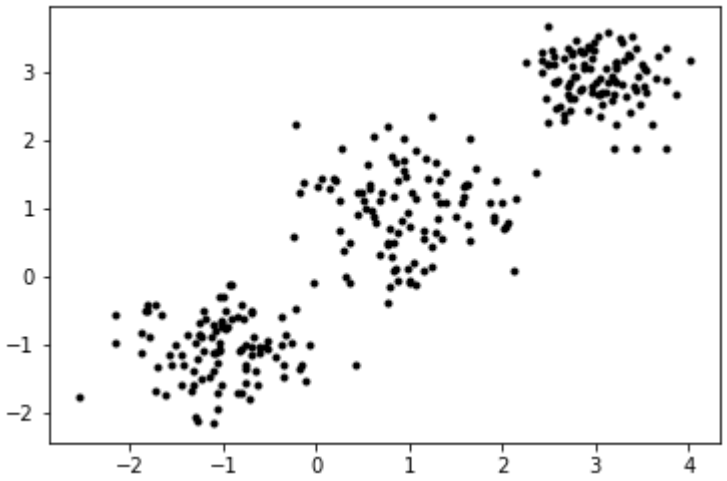
In [1]:
```
%matplotlib inline
```

Importamos las librerías y funciones necesarias

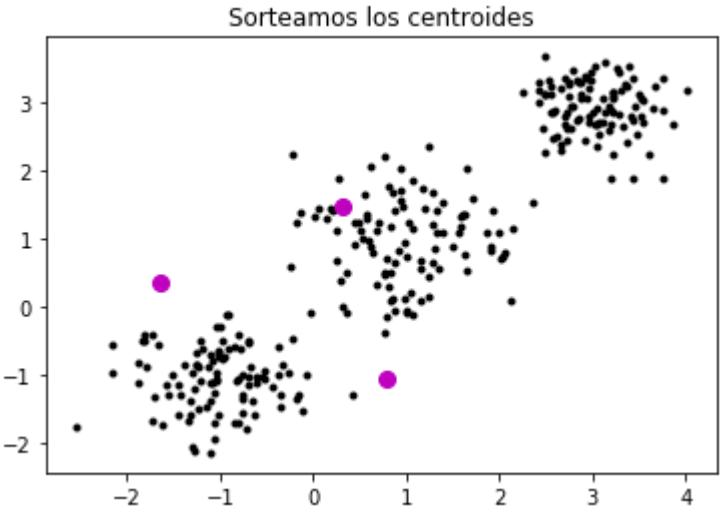
In [2]:
```
import numpy as np
import matplotlib.pyplot as plt
```

Generamos datos aleatorios 2D para tres clusters y los representamos

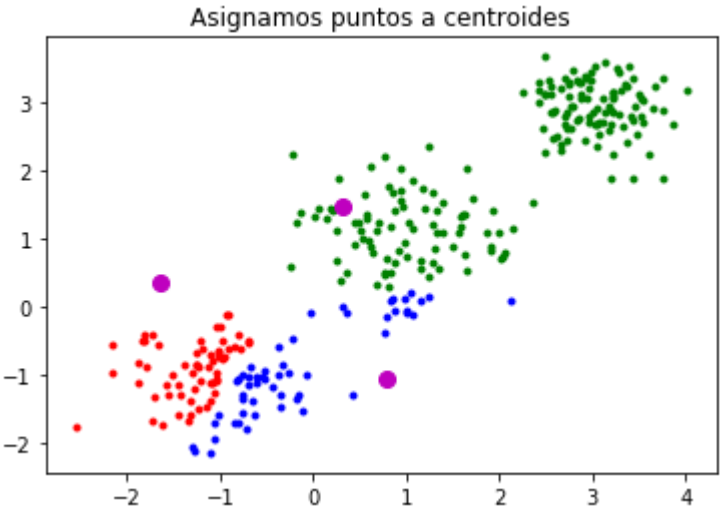
In [3]:
```
np.random.seed(7)

x1 = np.random.standard_normal((100,2))*0.6+np.ones((100,2))
x2 = np.random.standard_normal((100,2))*0.5-np.ones((100,2))
x3 = np.random.standard_normal((100,2))*0.4-2*np.ones((100,2))+5
X = np.concatenate((x1,x2,x3),axis=0)

plt.plot(X[:,0],X[:,1],'k.')
plt.show()
```

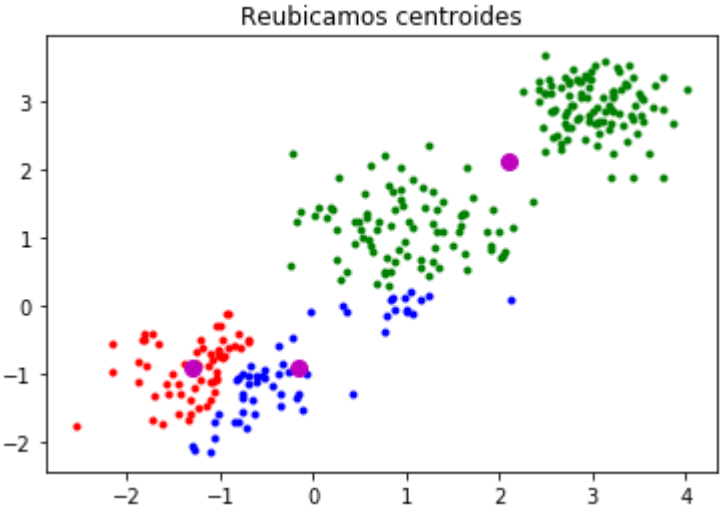


In [4]:
```
%run Ejercicio1.py
```

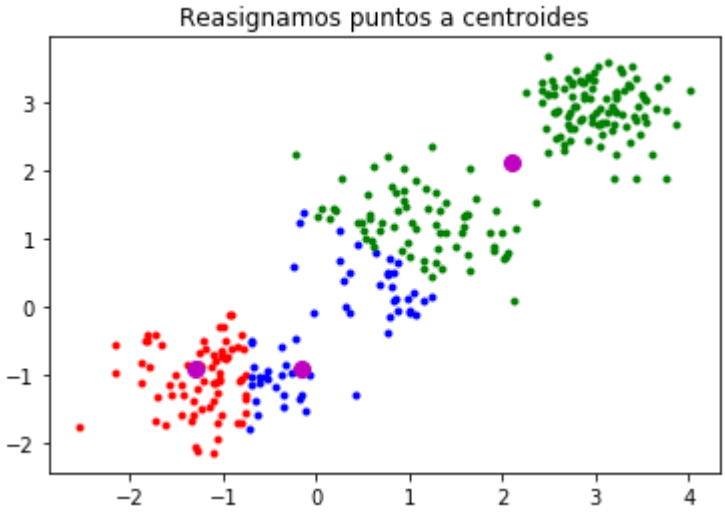


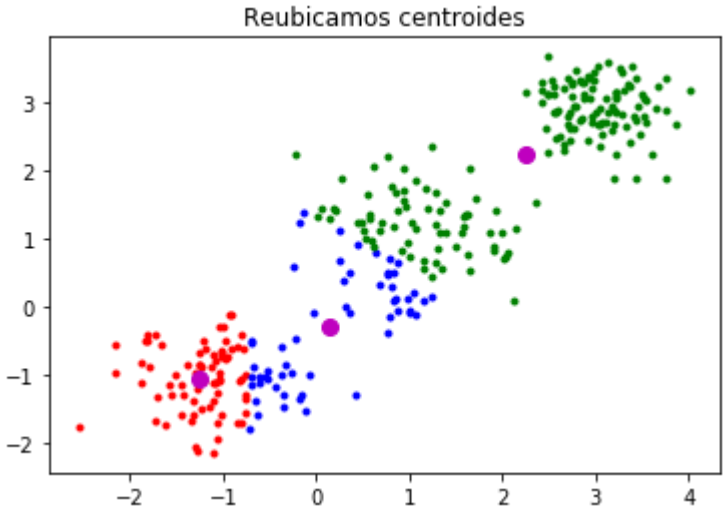
********** iteración 0



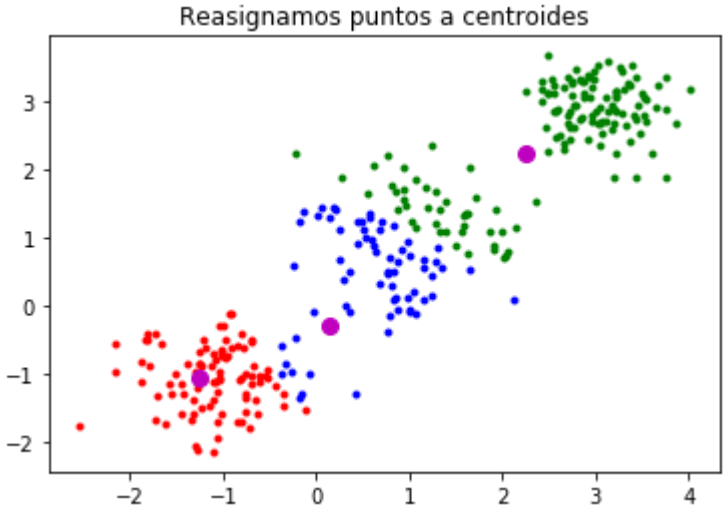


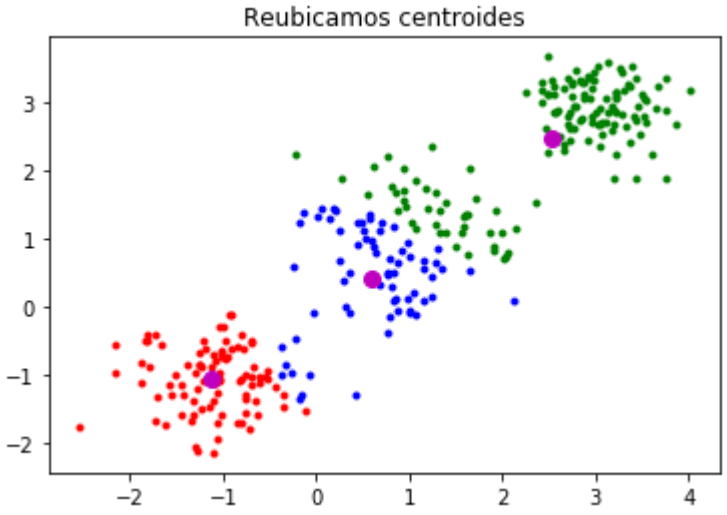
********** iteración 1


Reasignamos puntos a centroides

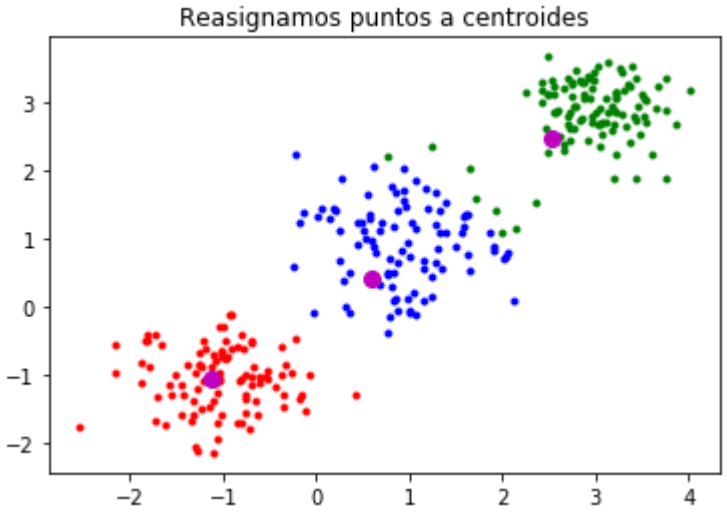

Reubicamos centroides

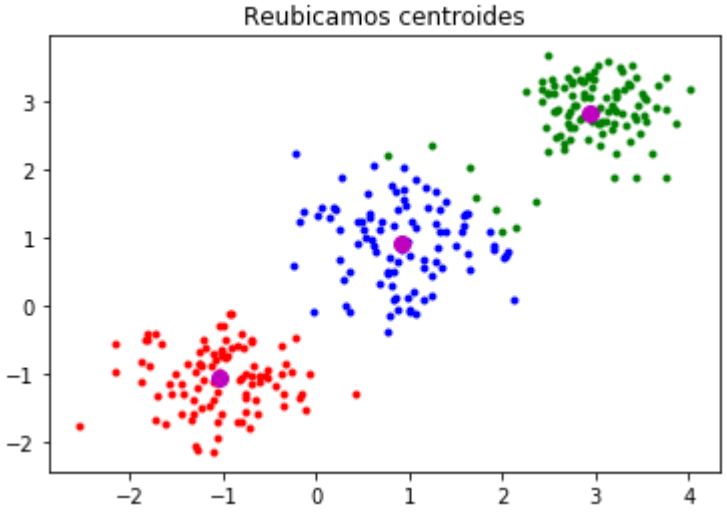
********** iteración 2


Reasignamos puntos a centroides

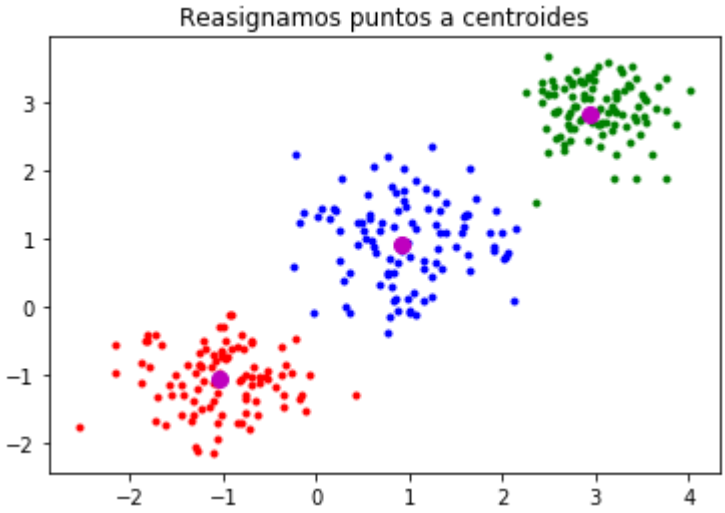


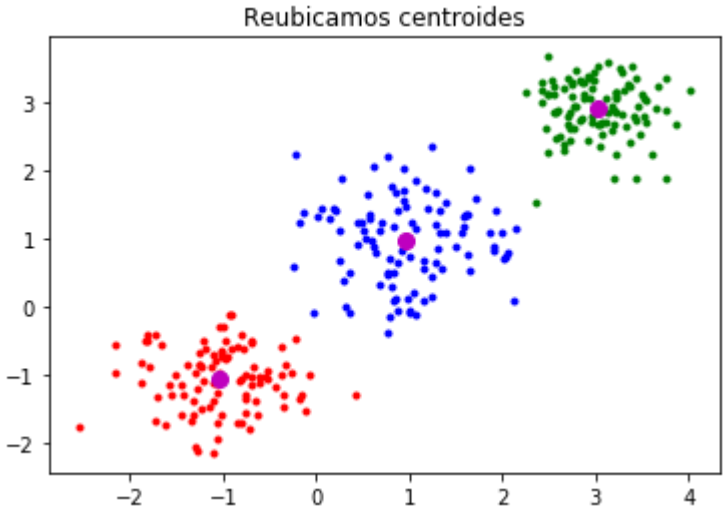
********** iteración 3



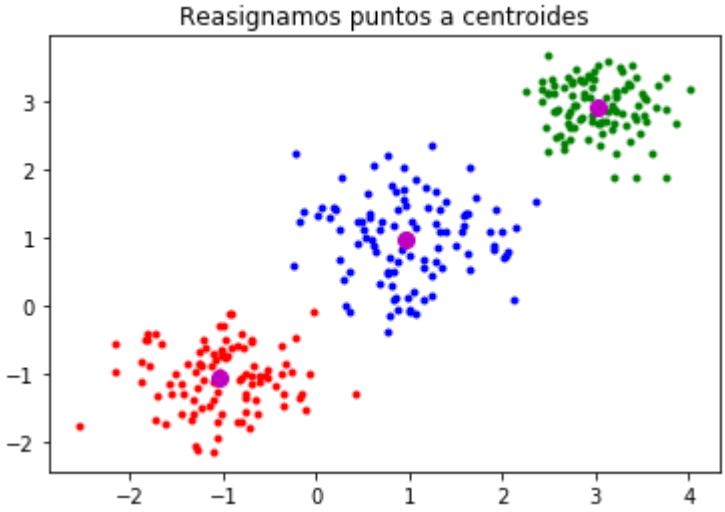


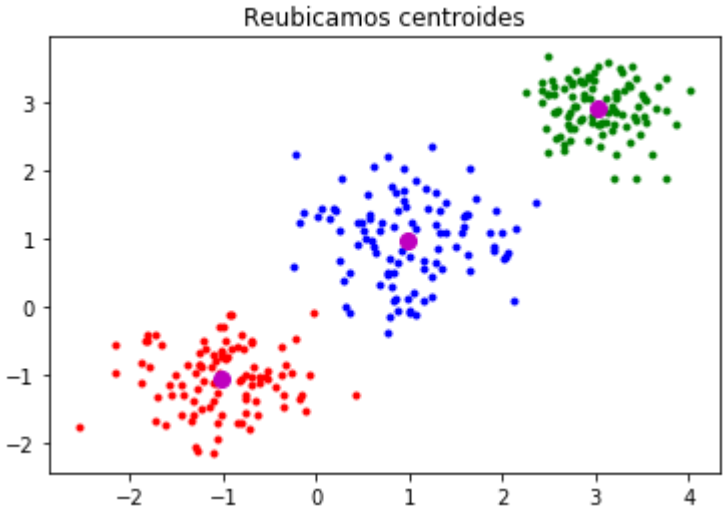
********** iteración 4


Reasignamos puntos a centroides

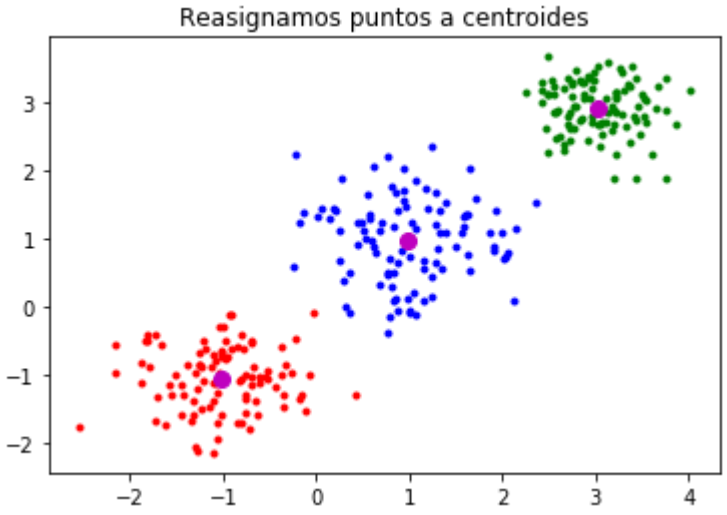

Reubicamos centroides

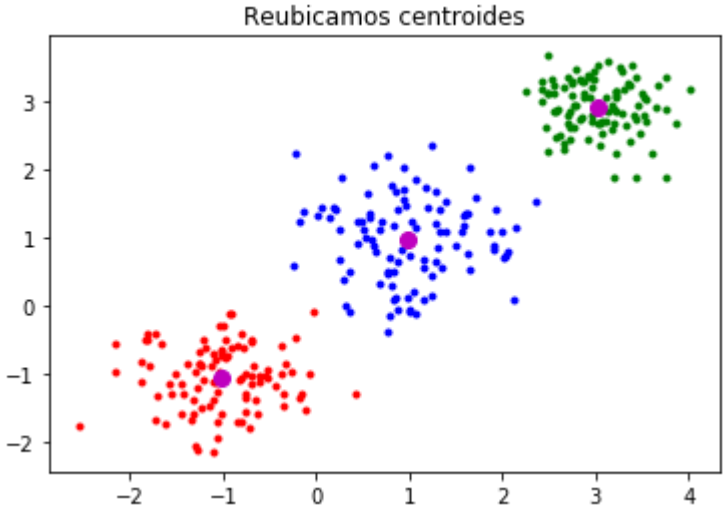
********** iteración 5


Reasignamos puntos a centroides

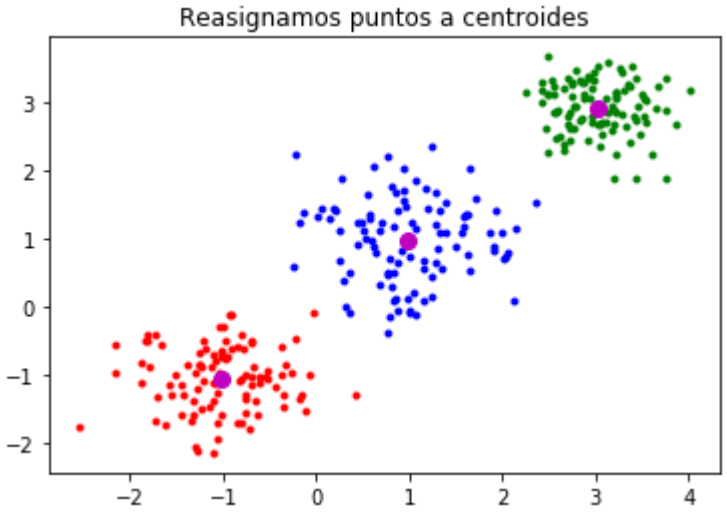


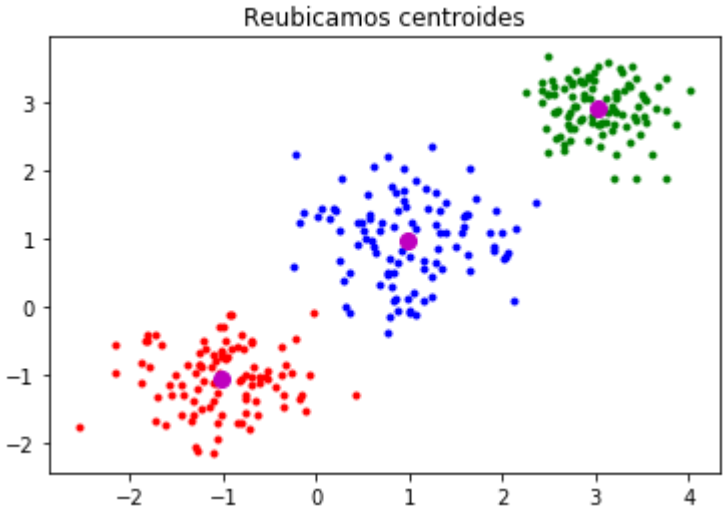
********** iteración 6





********** iteración 7






```
<matplotlib.figure.Figure at 0x10b149400>
```

En los demás ejercicios de esta práctica vamos a usar la función `KMeans` de la librería `sklearn`.

In [5]:
```
from sklearn.cluster import KMeans
```

Agrupamos los puntos con *k-means* usando $k = 3$

In [6]:
```
n = 3
k_means = KMeans(n_clusters=n)
k_means.fit(X)
```

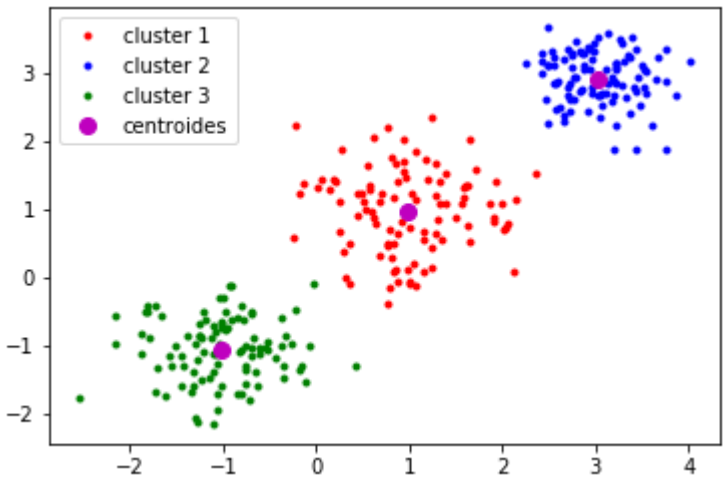
Out[6]:
```
KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
    n_clusters=3, n_init=10, n_jobs=1, precompute_distances='auto',
    random_state=None, tol=0.0001, verbose=0)
```

El resultado son tres centroides en torno a los cuales se agrupan los puntos y las etiquetas para cada punto que indican a qué cluster pertenece dicho punto

In [7]:
```
centroides = k_means.cluster_centers_
etiquetas = k_means.labels_
```

Dibujamos ahora los puntos y los centroides, utilizando un color distinto para los puntos de cada cluster

In [8]:
```
plt.plot(X[etiquetas==0,0],X[etiquetas==0,1],'r.', label='cluster 1')
plt.plot(X[etiquetas==1,0],X[etiquetas==1,1],'b.', label='cluster 2')
plt.plot(X[etiquetas==2,0],X[etiquetas==2,1],'g.', label='cluster 3')

plt.plot(centroides[:,0],centroides[:,1],'mo',markersize=8, label='cen
troides')

plt.legend(loc='best')
plt.show()
```



Clasificación de dígitos con k-means

Vamos a clasificar dígitos de la base de datos contenida en la librería `sklearn` de python utilizando el algoritmo *k-means*.

Comenzamos cargando las librerías que vamos a usar:

las habituales

In [9]:
```
import numpy as np
import matplotlib.pyplot as plt
from PIL import Image
```

las que contienen las funciones del método *k-means*

In [10]:
```
from sklearn.cluster import KMeans
```

y para cargar las imágenes de los dígitos

In [11]:
```
from sklearn.datasets import load_digits
```

Cargamos los datos, que están incluídos en la librería `sklearn`

In [12]:
```
digits = load_digits()
data = digits.data
```

In [13]:
```
print(data.shape)
```

```
(1797, 64)
```

En la matriz `data`, cada fila se corresponde con la imagen de un dígito. Los píxeles de la imagen rectangular de 8×8 píxeles se han recolocado en una fila de 64 elementos. Por lo tanto, cada fila es un objeto o dato. Las características o propiedades de cada objeto son las intensidades de gris de cada pixel. Es decir, tenemos, para cada imagen, 64 propiedades.

Para visualizar mejor los dígitos, vamos a invertir los colores

In [14]:
```
data = 255-data
```

Fijamos la semilla para sortear los centroides iniciales, para que los resultados obtenidos aquí sean repetibles

In [15]:
```
np.random.seed(1)
```

Como tenemos 10 dígitos diferentes (del 0 al 9) escogemos agrupar las imágenes en 10 clusters

In [16]:
```
n = 10
```

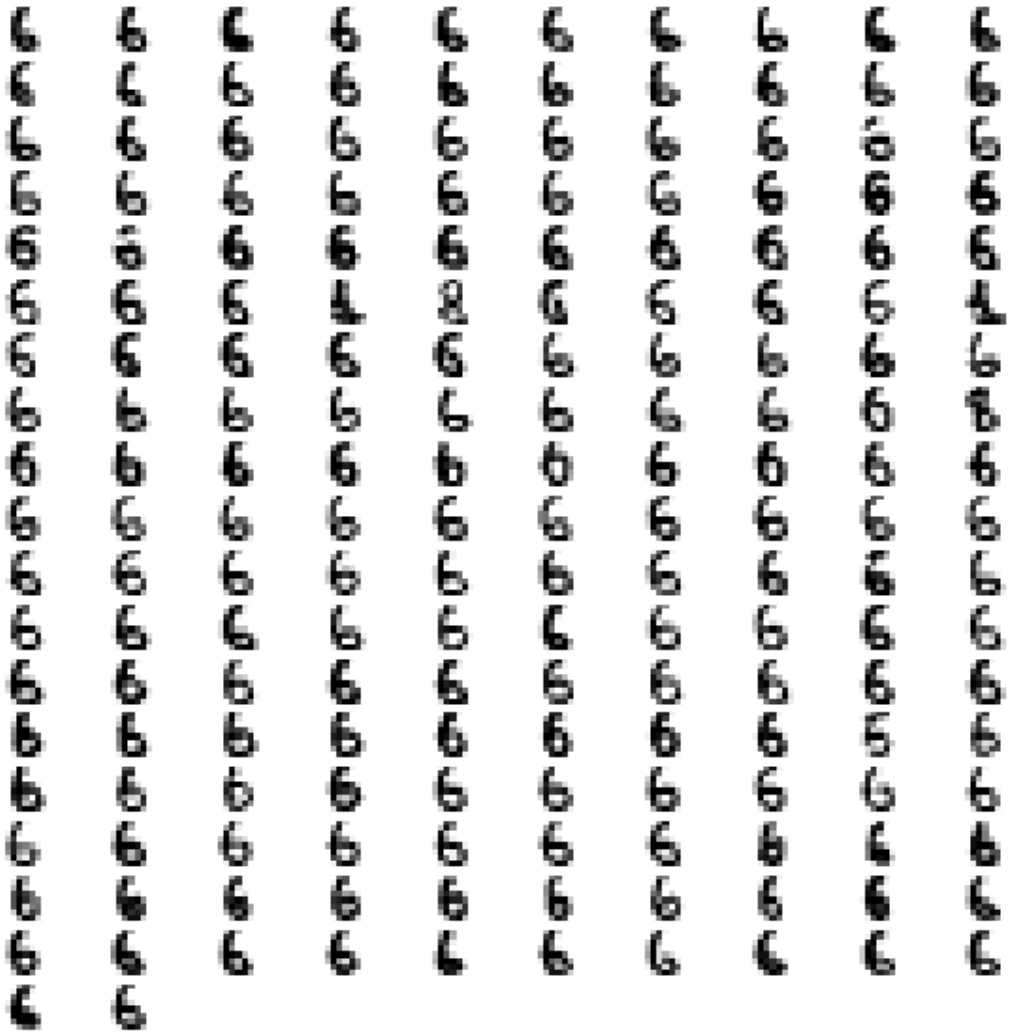
Clasificamos los datos utilizando *k-means*

In [17]:
```
kmeans = KMeans(n_clusters=n,init='random')
kmeans.fit(data)
Z = kmeans.predict(data)
```

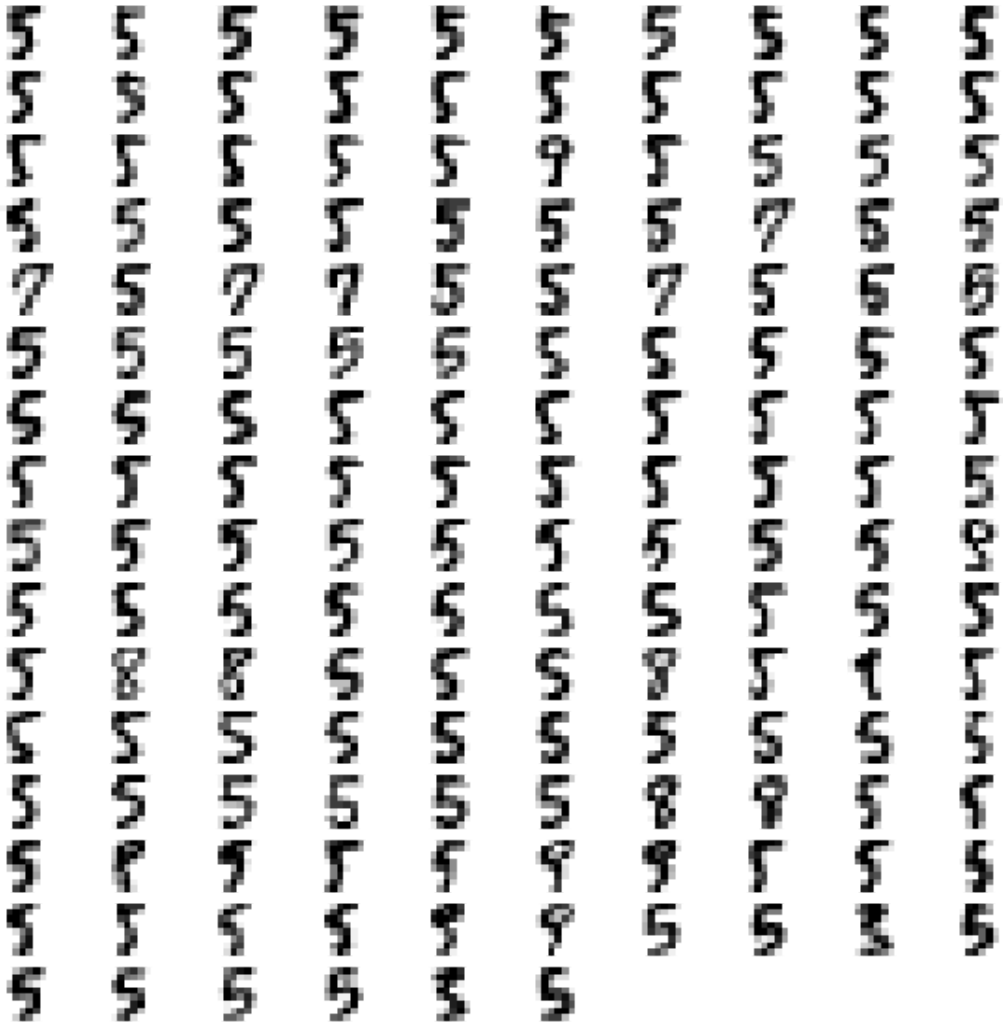
Dibujamos los clusters resultantes

In [18]:
```
for i in range(0,n):

    fila = np.where(Z==i)[0] # filas en Z donde están las imagenes de
cada cluster
    num = fila.shape[0]      # numero imagenes de cada cluster
    r = np.floor(num/10.)    # numero de filas menos 1 en figura de sa
lida

    print("cluster "+str(i))
    print(str(num)+" elementos")

    plt.figure(figsize=(10,10))
    for k in range(0, num):
        plt.subplot(r+1, 10, k+1)
        imagen = data[fila[k], ]
        imagen = imagen.reshape(8, 8)
        plt.imshow(imagen, cmap=plt.cm.gray)
        plt.axis('off')
    plt.show()
```

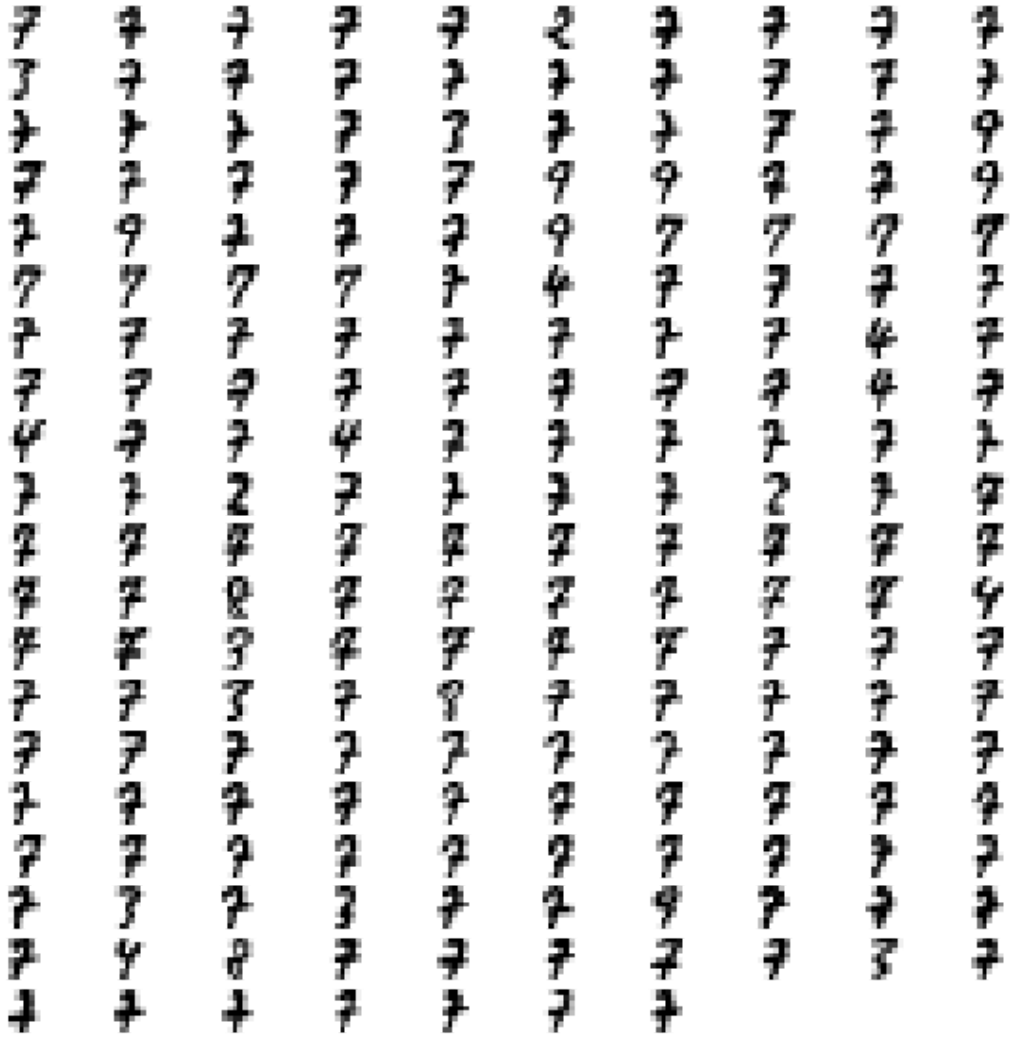
cluster 0
182 elementos



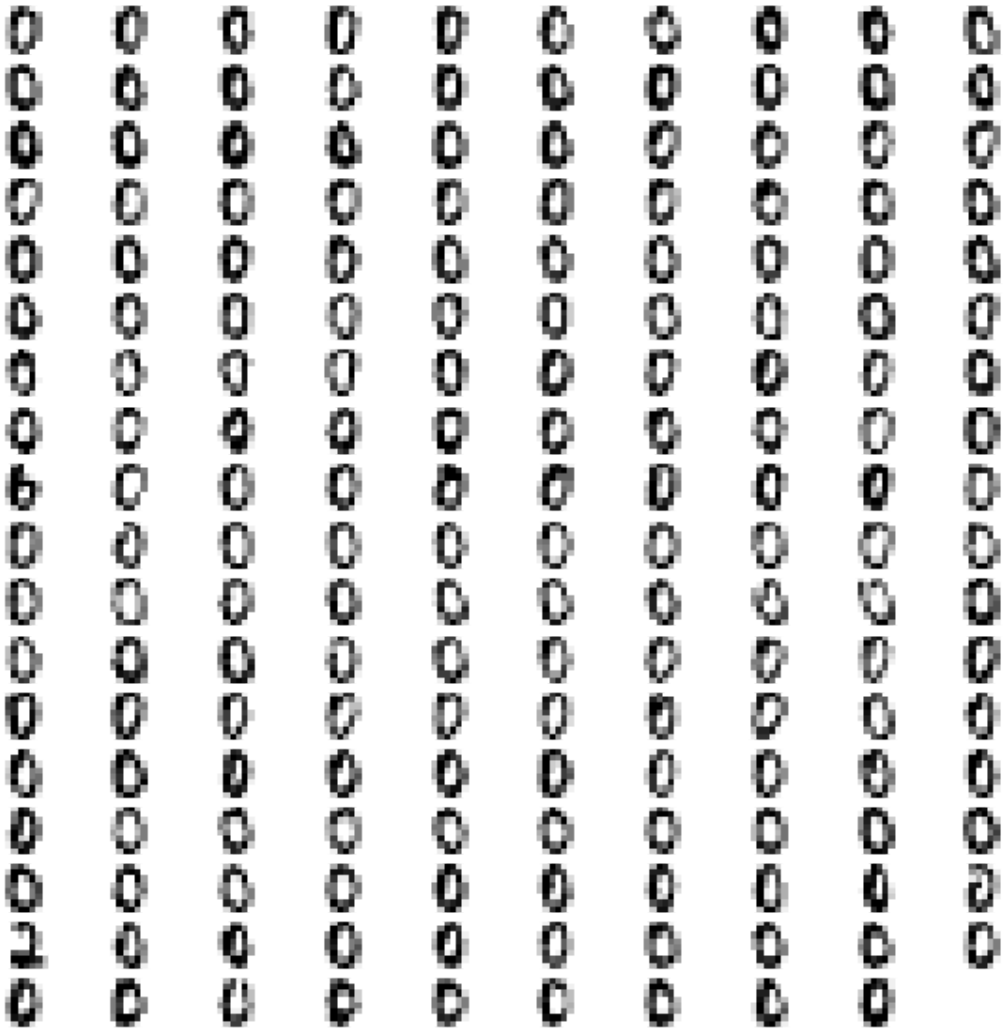
cluster 1
156 elementos



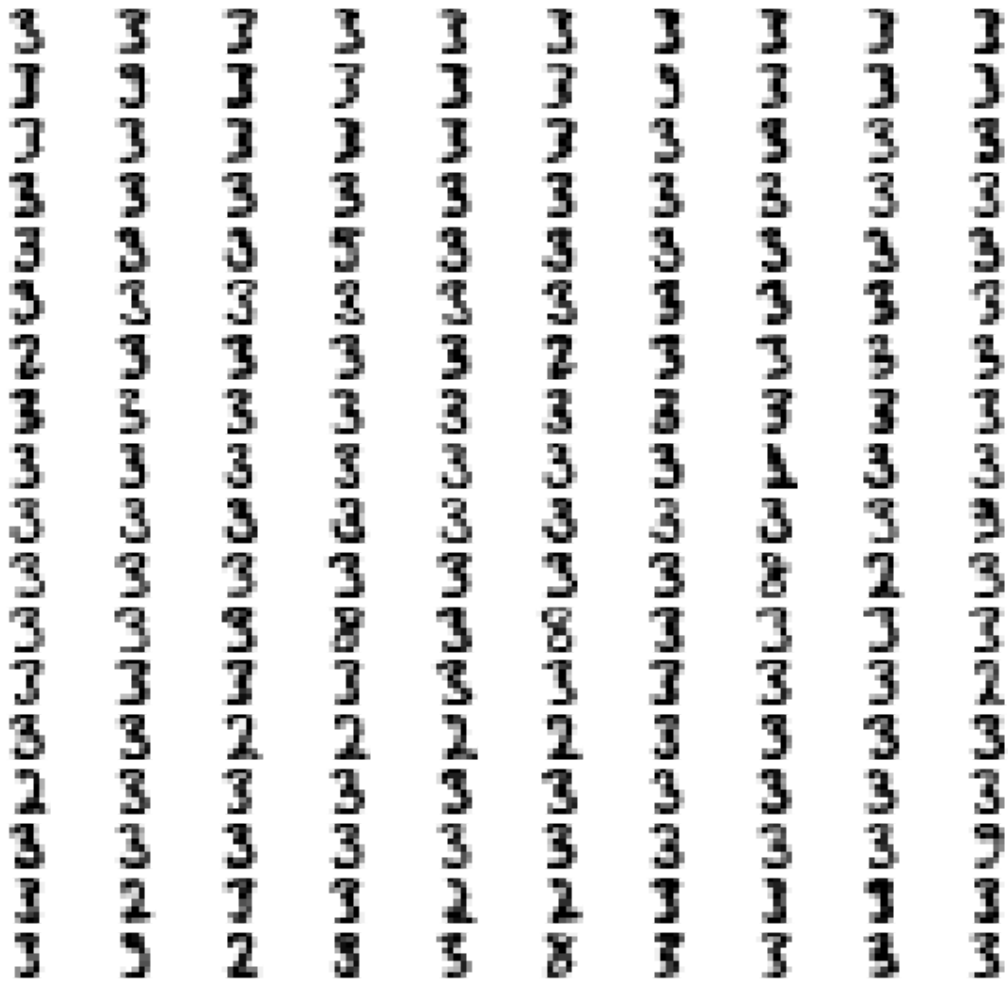
cluster 2
197 elementos



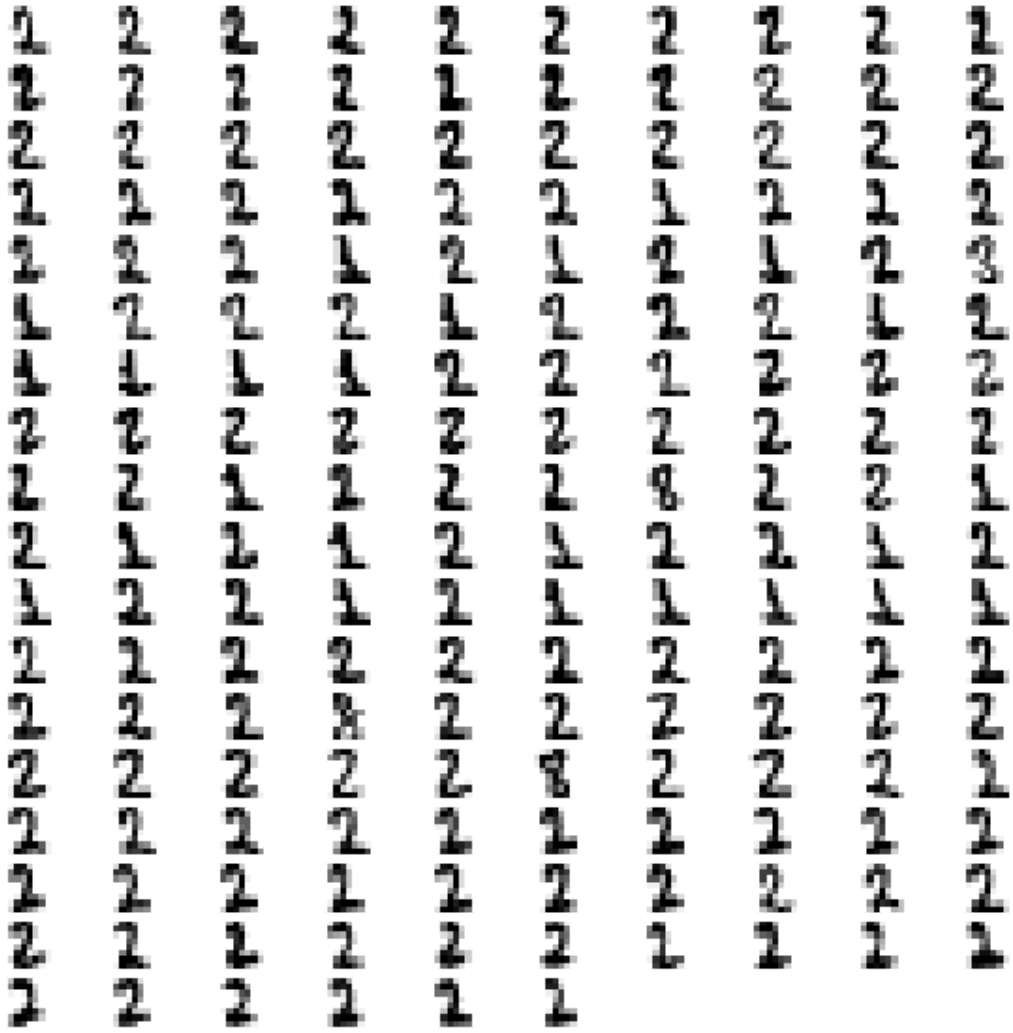
cluster 3
179 elementos



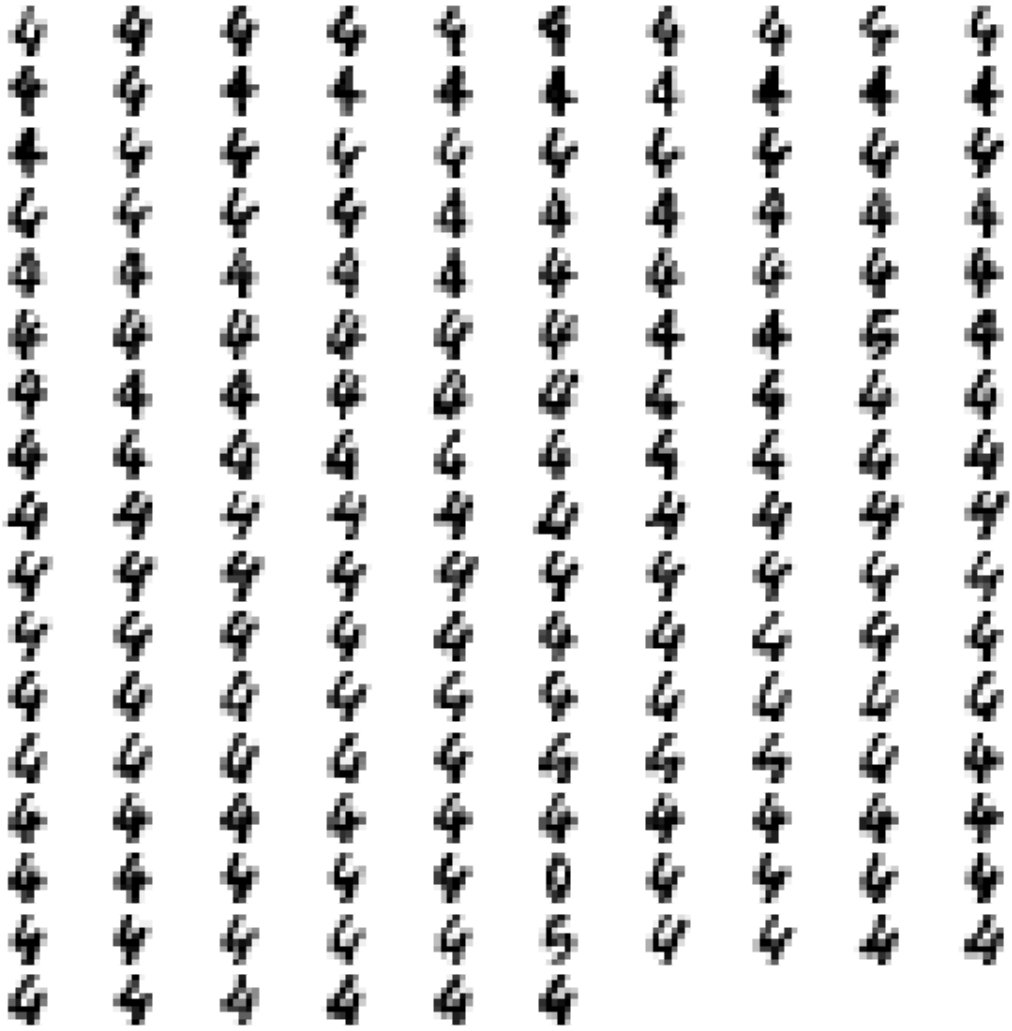
cluster 4
180 elementos



cluster 5
176 elementos

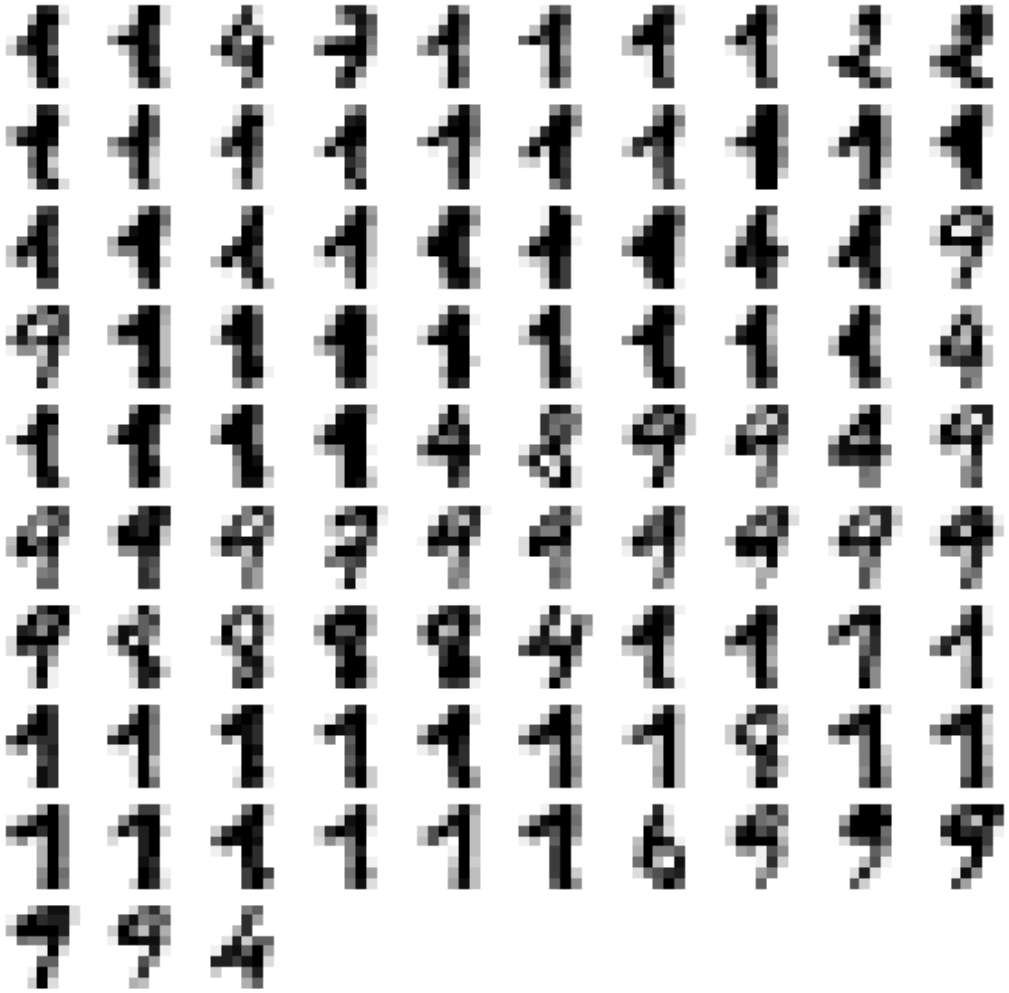


cluster 6
166 elementos



cluster 7
242 elementos

cluster 8
93 elementos



cluster 9
226 elementos

Cuantificación de imágenes con k-means

Cuantificación es una técnica de compresión con pérdida que consiste en agrupar todo un rango de valores en uno solo. Si cuantificamos el color de una imagen, reducimos el número de colores necesarios para representarla y el tamaño del fichero de la misma disminuye. Esto es importante, por ejemplo, para representar una imagen en dispositivos que sólo dan soporte a un número limitado de colores.

Vamos a cuantificar el color de la imagen siguiente utilizando *k-means*.

Cargamos la imagen

In [23]:
```
I = Image.open("tienda.jpg")
```

La transformamos en una matriz `numpy` y la visualizamos

In [24]:
```
I1 = np.asarray(I,dtype=np.float32)/255
plt.figure(figsize=(12,12))
plt.imshow(I1)
plt.axis('off')
plt.show()
```

En este caso nuestros objetos son los pixeles y sus características son las intensidades de rojo, verde y azul asociada a cada uno. Por lo tanto tenemos tantos datos u objetos como píxeles y tres características o propiedades para cada píxel. Tendremos tantos colores distintos como ternas RGB diferentes. Contamos el número de colores distintos

In [26]:
```
w, h = I.size
colors = I.getcolors(w * h)
num_colores = len(colors)
num_pixels = w*h

print (u'Número de pixels  = ', num_pixels)
print (u'Número de colores = ', num_colores)
```

```
Número de pixels  =  272640
Número de colores =  172388
```

Para poder aplicar *k-means* necesitamos una matriz con tantas filas como píxeles y para cada fila/pixel 3 columnas, una para cada intensidad de color (rojo, verde y azul).

Extraemos cada canal por separado

In [27]:
```
R = I1[:,:,0]
G = I1[:,:,1]
B = I1[:,:,2]
```

Convertimos las matrices en matrices de una dimensión y construímos la matriz de tres columnas descrita arriba.

In [28]:
```
XR = R.reshape((-1, 1))
XG = G.reshape((-1, 1))
XB = B.reshape((-1, 1))

X = np.concatenate((XR,XG,XB),axis=1)
```

Agruparemos los 172388 colores en 60 grupos o nuevos colores, que se corresponderán con los centroides obtenidos con el *k-means*

In [29]:
```
n = 60
k_means = KMeans(n_clusters=n)
k_means.fit(X)
```

Out[29]:
```
KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
    n_clusters=60, n_init=10, n_jobs=1, precompute_distances='auto',
    random_state=None, tol=0.0001, verbose=0)
```

Los `centroides` finales son los nuevos colores y cada pixel tiene ahora una `etiqueta` que dice a qué grupo o cluster pertenece

In [30]:
```
centroides = k_means.cluster_centers_
etiquetas = k_means.labels_
```

A partir de las `etiquetas` y los colores (intensidades de rojo, verde y azul) de los `centroides` reconstruimos la matriz de la imagen utilizando únicamente los colores de los centroides.

In [31]:
```
m = XR.shape
for i in range(m[0]):
    XR[i] = centroides[etiquetas[i]][0]
    XG[i] = centroides[etiquetas[i]][1]
    XB[i] = centroides[etiquetas[i]][2]
XR.shape = R.shape
XG.shape = G.shape
XB.shape = B.shape
XR = XR[:, :, np.newaxis]
XG = XG[:, :, np.newaxis]
XB = XB[:, :, np.newaxis]

Y = np.concatenate((XR,XG,XB),axis=2)
```

Representamos la imagen con 60 colores

In [32]:
```
plt.figure(figsize=(12,12))
plt.imshow(Y)
plt.axis('off')
plt.show()
```

El número de píxeles es el de la foto inicial, y el número de colores de esta imagen es igual al número de centroides.

In [33]:
```
print (u'Número de pixels  = ', num_pixels)
print (u'Número de colores = ', n)
```

```
Número de pixels  =  272640
Número de colores =  60
```

Guardamos ahora la imagen en formato jpg

In [34]:
```
Y1 = np.floor(Y*255)
Image.fromarray(Y1.astype(np.uint8)).save("tienda_comprimida.jpg")
```

Compara el tamaño del fichero inicial y el fichero final: 176 KB el fichero inicial y 107 KB el fichero final.

Ejercicio 2

Contar el número de colores en la imagen siguiente, `holi.jpg`. A partir de ella crear una imagen donde el número de colores se ha reducido a 10 utilizando el método *k-means*.

In [36]:
```
%run Ejercicio2.py
```

```
Num. pixels  =  229760
Num. colores =  118708
```

```
Num. pixels  =  229760
Num. colores =  10
```

Ejercicio 3

Obtener la tercera imagen a partir de la primera imagen (`che-guevara.jpg`) que se muestra a continuación.

In [37]:
```
%run Ejercicio3.py
```

Segmentación de imágenes con k-means

La segmentación divide una imagen en regiones con propiedades internas coherentes. Se puede segmentar una imagen utilizando el color.

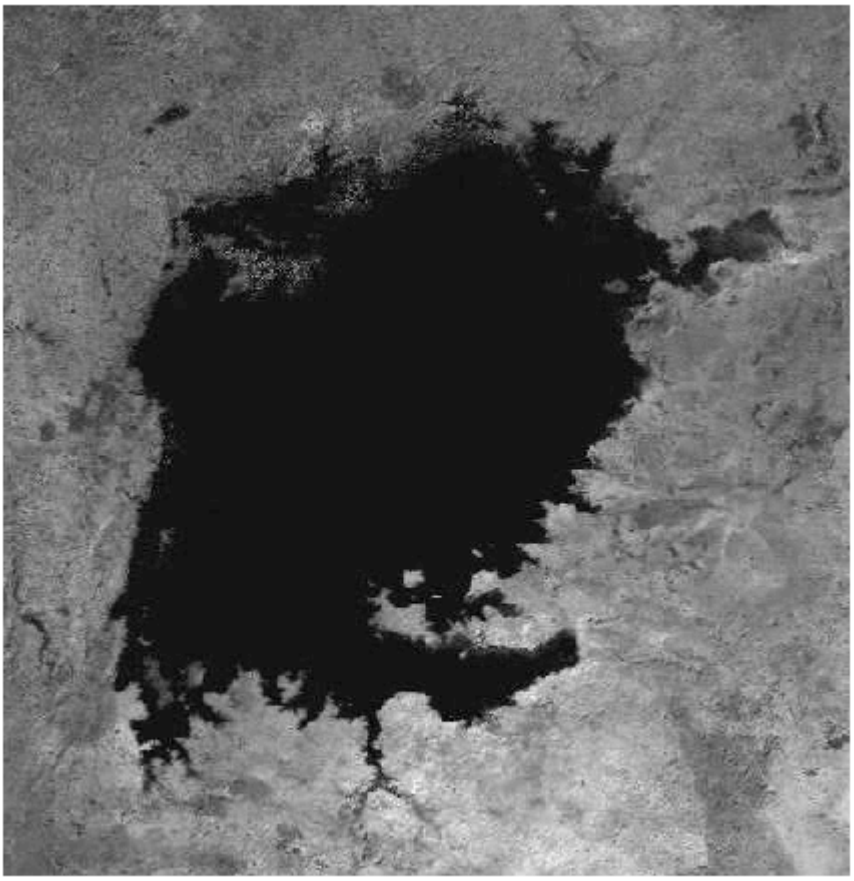
El proceso es similar al de cuantización de imágenes. La diferencia es el objetivo con el que se agrupan los píxeles: agrupamos los píxeles para separar los elementos significativos de una imagen y así poder extraer cierta información de alguno de ellos. Por ejemplo, calcular el tamaño de un tumor a partir de imágenes médicas, el porcentaje de mica en una roca granítica, el área de un lago a partir de una foto aérea.

Comencemos con este último ejemplo. Si tenemos la siguiente imagen tomada desde un satélite del lago Victoria y el área que cubre es aproximadamente 210000 km^2, podemos calcular, a partir del porcentaje del área de la imagen, el área del lago.

In [38]:
```
I = Image.open("Lago_Victoria.jpg")

plt.figure(figsize=(8,8))
plt.imshow(I)
plt.axis('off')
plt.show()
```

Para simplificar el problema, convertimos la imagen de color a blanco y negro

In [39]:
```
I1 = I.convert('L')
I2 = np.asarray(I1,dtype=np.float)

plt.figure(figsize=(8,8))
plt.imshow(I2,cmap='gray')
plt.axis('off')
plt.show()
```



Preparamos la matriz para aplicar *k-means*. Ahora tendrá tantas filas como píxeles pero sólo una columna, la intensidad de gris.

In [40]:
```
X = I2.reshape((-1, 1))
```

Agrupamos los píxeles en tres clusteres con *k-means*

In [41]:
```
k_means = KMeans(n_clusters=3)
k_means.fit(X)
```

Out[41]:
```
KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300,
    n_clusters=3, n_init=10, n_jobs=1, precompute_distances='auto',
    random_state=None, tol=0.0001, verbose=0)
```

Extraemos el valor de los `centroides` y las `etiquetas` de cada pixel

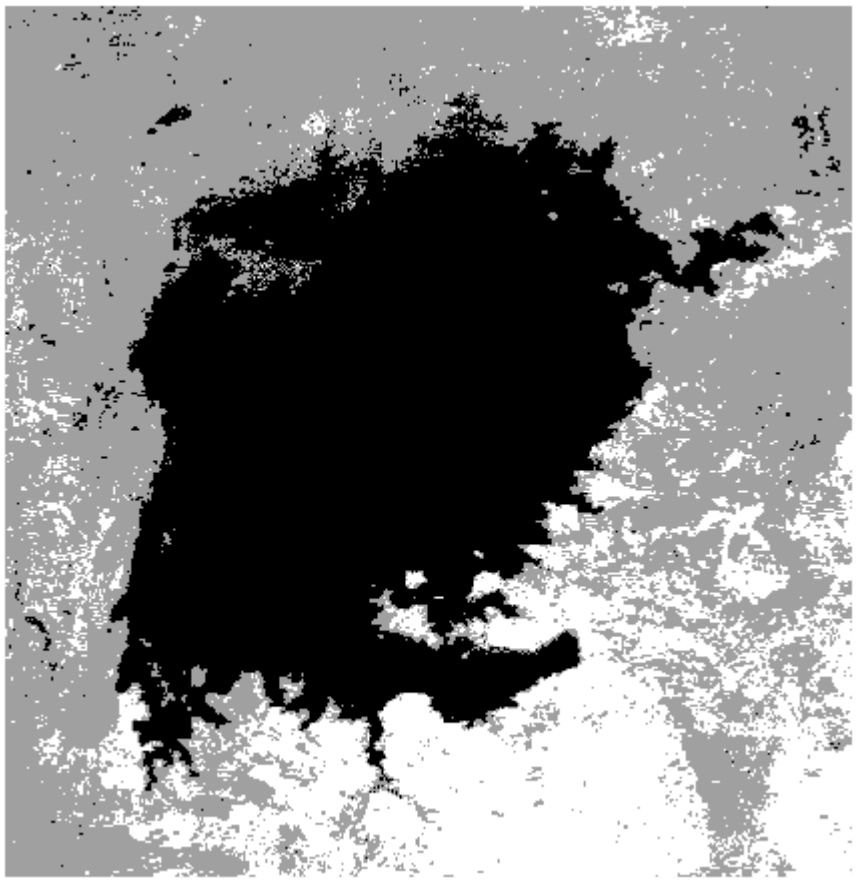
In [42]:
```
centroides = k_means.cluster_centers_
etiquetas = k_means.labels_
```

Reconstruimos la imagen utilizando las tres intensidades de los centroides

In [43]:
```
I2_compressed = np.choose(etiquetas, centroides)
I2_compressed.shape = I2.shape
```

Visualizamos la foto reconstruida

In [44]:
```
plt.figure(figsize=(8,8))
plt.imshow(I2_compressed,cmap='gray')
plt.axis('off')
plt.show()
```



Contamos el número de píxeles de color negro (los de intensidad de gris más baja)

In [45]:
```
I2 = (I2_compressed-np.min(I2_compressed))/(np.max(I2_compressed)-np.m
in(I2_compressed))*255
I2 = Image.fromarray(I2.astype(np.uint8))
w, h =I2.size
colors = I2.getcolors(w * h)
print (colors)
```

```
[(79002, 0), (110746, 161), (47372, 255)]
```

Hay 79002 píxeles con intensidad 18. Calculamos el porcentaje respecto al número total de píxeles de la foto y tenemos el porcentaje del área del lago respecto al área total representada por la foto

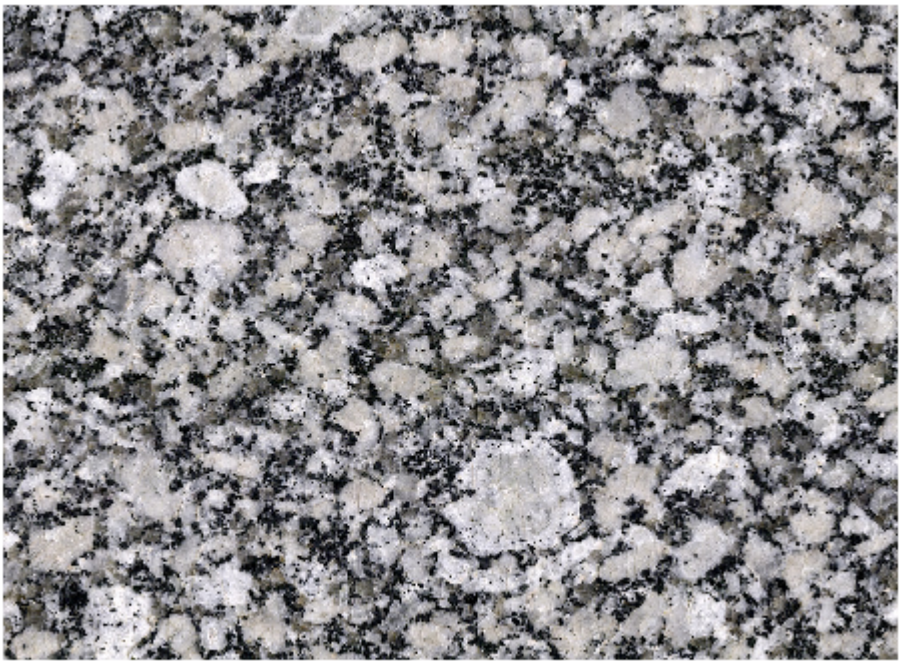
In [46]:
```
print (u'Área = ',  float(210000)*float(colors[0][0])/float(w*h), 'km
2')
```

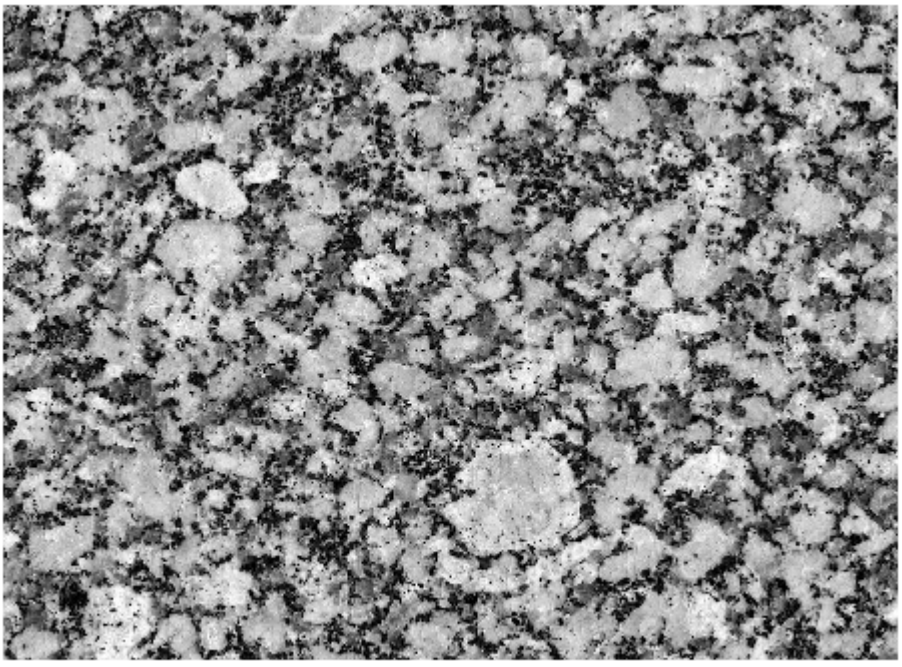
```
Área =  69966.34615384616 km2
```

Ejercicio 4

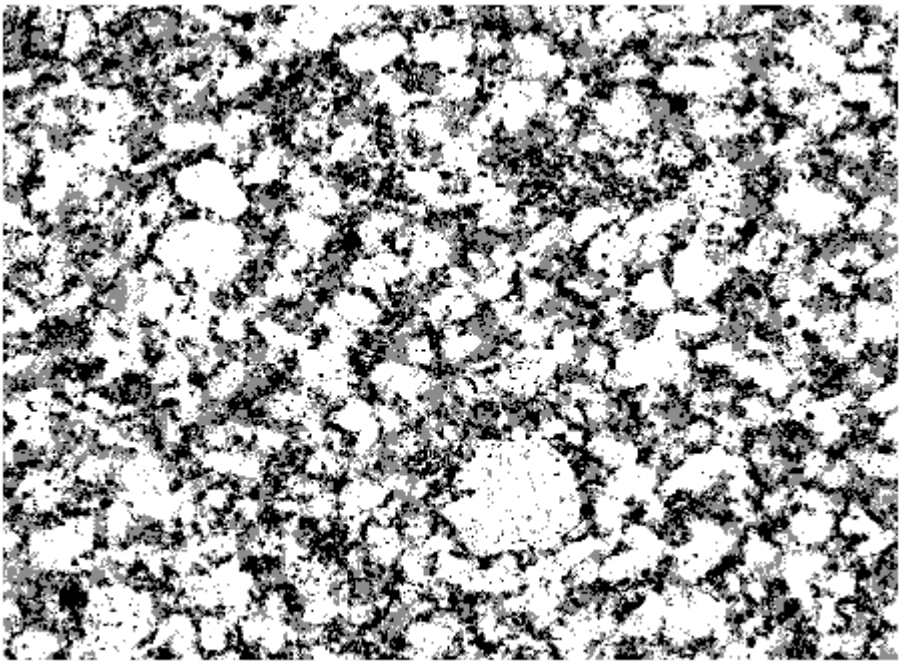
Calcular el porcentaje de mica (es el mineral de color más oscuro) en la roca de granito cuya sección muestra la foto. Ejecutar 10 veces la función *k-means* y quedarse con la iteración que dé menor error siendo `error = k_means.inertia_`.

In [48]:
```
%run Ejercicio4.py
```





Error	Porcentaje de mica en el granito
374889848	22.18
374889848	22.18
374794828	21.89
374889848	22.18
374889848	22.18
374844787	21.62
374889848	22.18
374889848	22.18
374889848	22.18
374844787	21.62



```
[(165788, 0), (230659, 140), (370529, 255)]
Error min.    Porcentaje de mica en el granito
 374844787      21.62
```

Ficheros

- Tienda (http://www.unioviedo.es/compnum/laboratorios_py/ficheros/tienda.jpg)
- Holi (http://www.unioviedo.es/compnum/laboratorios_py/ficheros/holi.jpg)
- Che Guevara (http://www.unioviedo.es/compnum/laboratorios_py/ficheros/che-guevara.jpg)
- Lago Victoria (http://www.unioviedo.es/compnum/laboratorios_py/ficheros/Lago_Victoria.jpg)
- Granito (http://www.unioviedo.es/compnum/laboratorios_py/ficheros/granito.jpg)

Referencias fotos

- Imagen tienda.jpg (https://pixabay.com/es/estambul-pavo-gran-bazar-de-viaje-1643752/)
- Imagen holi.jpg (https://pixabay.com/es/holi-india-vivid-vacaciones-671899/)
- Imagen che-guevara.jpg (https://es.wikipedia.org/wiki/Ernesto_Guevara)
- Imagen Lago_Victoria.jpg (https://es.wikipedia.org/wiki/Lago)
- Imagen granito.jpg (https://www.flickr.com/photos/jsjgeology/25904916323)