1. Introducción

En esta práctica clasificaremos un sistema X de 1000 elementos a partir del número de vecindades de Voronói o clusters. Para ello usaremos los algoritmos *KMeans* y *DBSCAN* de la librería *scikit-learn* de python.

2. Material empleado

2.1. Clustering con *KMeans*

El algoritmo *KMeans* tiene como parámetro de entrada el número de clusters deseados n y el conjunto X a clasificar. El algoritmo devuelve el conjunto clasificado y los centroides de cada cluster.

El algoritmo toma al principio n centroides y en cada iteración clasifica cada elemento como perteneciente al cluster cuya distancia al centroide del cluster sea mínima. Al final de cada iteración cambia el centroide de cada cluster siendo el nuevo la media de todos los puntos que pertenecen a ese cluster. Si al final de una iteración la distancia entre cada centroide y el anterior es menor que un cierto ϵ el algoritmo termina.

Para ver cuál es el valor óptimo de clusters probamos con $n \in \{1, 2, 3..,15\}$ y de cada clasificación calculamos su coeficiente de *Silhouette* $\bar{s}$ y escogemos el que mayor valor $\bar{s}$ tenga.

El coeficiente de *Silhouette* se define como $\bar{s} := \frac{1}{N}\sum_{q=1}^{n}\sum_{i\in V_q}\frac{b_i-a_i}{max\{a_i,b_i\}}$ siendo V_q cada uno de los clusters, $a_i = \frac{1}{|V_q|-1}\sum_{j\in V_q} d_p(i,j)$ y $b_i = \text{mín}_{k\neq q}\frac{1}{|V_k|}\sum_{j\in V_k} d_p(i,j)$. Este coeficiente es una medida de ayuda a la decisión para saber si la clasificación ha sido buena.

2.2. Clustering con *DBSCAN*

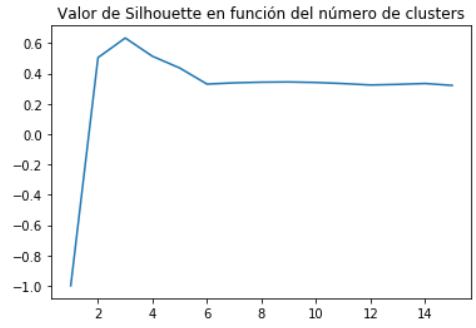
Este algoritmo tiene como parámetros de entrada ϵ que es la distancia mínima considerada según la p-norma, n_0 que es el número mínimos de elementos en un cluster y el conjunto X a clasificar. El algoritmo devuelve los elementos clasificados.

El algoritmo clasifica cada elemento como perteneciente al cluster de los elementos que están en la ϵ-bola centrada en él. Si no tiene elementos suficientes (menos de n_0) cerca de él se le clasifica como ruido aunque después se revisitará para ver si pertenece a algún cluster existente.

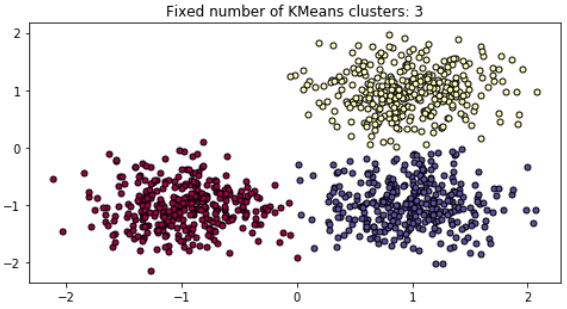
Para ver cuál es valor óptimo de cluster fijamos $n_0 = 10$ y probamos con $\epsilon \in (0{,}1, 1)$ tomando valores con paso 0.05. Al igual que en el apartado utilizamos el coeficiente de *Silhouette* para saber con que ϵ ha sido la mejor clasificación.

3. Resultados

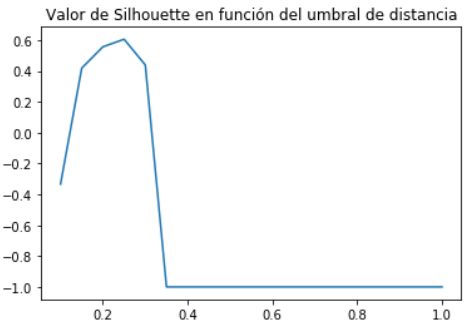
KMeans



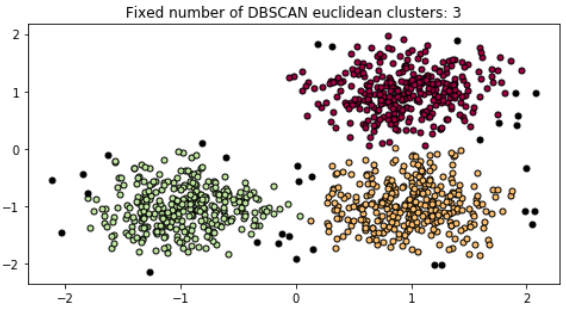
El valor óptimo de Silhouette con el algoritmo KMeans es 0.634 y se obtiene con 3 clusters



DBSCAN euclideo



El valor óptimo de Silhouette es 0.606 y se obtiene con umbral de distancia 0.25 que da lugar a 3 clusters



Los resultados obtenidos con el algoritmo *DBSCAN* y la distancia Manhattan son muy similares a los obtenidos con la distancia euclídea. Lo único reseñable es que se observa más ruido en la muestra.

4. Conclusión

Ambos algoritmos obtienen resultados muy similares siendo un poco mejor el valor de *Silhouette* del algoritmo *KMeans*. Los dos algoritmos obtienen 3 clusters como era de esperar porque el conjunto X estaba generado por puntos en torno a los centros (1,1), (-1,-1) y (1,-1).

Uno de los posibles motivos por los que el algoritmo *DBSCAN* obtiene un peor coeficiente de *Silhouette* es porque considera a los puntos que son ruido como otro cluster.

Cabe también remarcar que cuando hay un único cluster el término b_i no está bien definido. En este caso se considera que $b_i = 0$ y obteniendose un valor de *Silhouette* de -1.

5. Código

```
"""
Plantilla 1 de la práctica 3

Referencia:
    https://scikit-learn.org/stable/modules/clustering.html
    https://scikit-learn.org/stable/modules/generated/
        sklearn.cluster.KMeans.html
    https://docs.scipy.org/doc/scipy/reference/spatial.distance.html
"""

import numpy as np

from sklearn.cluster import DBSCAN
from sklearn.cluster import KMeans
from sklearn import metrics
from sklearn.datasets import make_blobs

import matplotlib.pyplot as plt

# #############################################################################
# Aquí tenemos definido el sistema X de 1000 elementos de dos estados
# construido a partir de una muestra aleatoria entorno a unos centros:
centers = [[1, 1], [-1, -1], [1, -1]]
X, labels_true = make_blobs(n_samples=1000, centers=centers,
    cluster_std=0.4, random_state=0)
#Si quisieramos estandarizar los valores del sistema, haríamos:
#from sklearn.preprocessing import StandardScaler
#X = StandardScaler().fit_transform(X)

print('MUESTRA')
plt.plot(X[:,0],X[:,1],'ro', markersize=1)
plt.show()
print('\n\n\n')

'''
A partir de aquí, código escrito por
-Jorge Sainero Valle
-Lucas de Torre Barrio
utilizando el código proporcionado
'''
```

```
# Representamos el resultado con un plot
def graficaKMeans(n_clusters):
    kmeans = KMeans(n_clusters=n_clusters, random_state=0).fit(X)
    labels = kmeans.labels_
    unique_labels = set(labels)
    colors = [plt.cm.Spectral(each)
              for each in np.linspace(0, 1, len(unique_labels))]

    plt.figure(figsize=(8,4))
    for k, col in zip(unique_labels, colors):
        if k == -1:
            # Black used for noise.
            col = [0, 0, 0, 1]

        class_member_mask = (labels == k)

        xy = X[class_member_mask]
        plt.plot(xy[:, 0], xy[:, 1], 'o', markerfacecolor=tuple(col),
                 markeredgecolor='k', markersize=5)

    plt.title('Fixed number of KMeans clusters: %d' % n_clusters)
    plt.show()

def kMeans(k):
    global X
    kmeans = KMeans(n_clusters=k, random_state=0).fit(X)
    labels = kmeans.labels_
    #para que no falle cuando solo hay un cluster
    if k==1:
       silhouette = -1
    else:
        silhouette = metrics.silhouette_score(X, labels)
    return silhouette

def bestSilhouetteKmeans():
    print('KMeans')
    sils=np.zeros(15)
    maxi=-1
    for i in range(1,16):
        sils[i-1]=kMeans(i)
        if sils[i-1]>maxi:
            maxi=sils[i-1]
            j=i
```

```
    plt.title('Valor de Silhouette en función del número de clusters')
    plt.plot(np.linspace(1,15,15),sils)
    plt.show()
    print("El valor óptimo de Silhouette con el algoritmo KMeans es %0.3f"
        % maxi, "y se obtiene con", j, "clusters")
    graficaKMeans(j);
    print('\n\n\n')

def graficaDBSCAN(epsilon,metrica):
    db = DBSCAN(eps=epsilon, min_samples=10, metric=metrica).fit(X)
    core_samples_mask = np.zeros_like(db.labels_, dtype=bool)
    core_samples_mask[db.core_sample_indices_] = True
    labels = db.labels_
    unique_labels = set(labels)
    colors = [plt.cm.Spectral(each)
              for each in np.linspace(0, 1, len(unique_labels))]

    plt.figure(figsize=(8,4))
    for k, col in zip(unique_labels, colors):
        if k == -1:
            # Black used for noise.
            col = [0, 0, 0, 1]

        class_member_mask = (labels == k)

        xy = X[class_member_mask]
        plt.plot(xy[:, 0], xy[:, 1], 'o', markerfacecolor=tuple(col),
                 markeredgecolor='k', markersize=5)
    #Restamos uno para no contar el ruido
    plt.title('Fixed number of DBSCAN '+metrica+' clusters: %d'
        % (len(unique_labels)-1))
    plt.show()

def dbscan(epsilon,metrica):
    db = DBSCAN(eps=epsilon, min_samples=10, metric=metrica).fit(X)

    core_samples_mask = np.zeros_like(db.labels_, dtype=bool)
    core_samples_mask[db.core_sample_indices_] = True
    labels = db.labels_

    # Number of clusters in labels, ignoring noise if present.
    n_clusters_ = len(set(labels)) - (1 if -1 in labels else 0)
    #para que no falle cuando solo hay un cluster
```

```
    if n_clusters_<=1:
        silhouette=-1
    else:
        silhouette=metrics.silhouette_score(X, labels)
    return [n_clusters_,silhouette]

def bestSilhouetteDBSCAN():
    print('DBSCAN euclideo')
    silsEuc=[[dbscan(i,'euclidean'),i] for i in np.arange(0.1,1.05,0.05)]
    silsMan=[[dbscan(i,'manhattan'),i] for i in np.arange(0.1,1.05,0.05)]
    n_clusters_euc=0
    maxieuc=-1
    epsilon_euc=0
    n_clusters_man=0
    maximan=-1
    epsilon_man=0

    for sil in silsEuc:
        if (sil[0][1]>maxieuc):
            maxieuc = sil[0][1]
            n_clusters_euc = sil[0][0]
            epsilon_euc=sil[1]
    plt.title('Valor de Silhouette en función del umbral de distancia')
    plt.plot([silsEuc[i][1] for i in range(len(silsEuc))],
        [silsEuc[i][0][1] for i in range(len(silsEuc))])
    plt.show()
    print("El valor óptimo de Silhouette es %0.3f" % maxieuc,
        "y se obtiene con umbral de distancia %0.2f" % epsilon_euc,
        "que da lugar a",n_clusters_euc,"clusters")
    graficaDBSCAN(epsilon_euc,'euclidean')
    print('\n\n\n')

    print('DBSCAN Manhattan')
    for sil in silsMan:
        if (sil[0][1]>maximan):
            maximan = sil[0][1]
            n_clusters_man = sil[0][0]
            epsilon_man=sil[1]
    plt.title('Valor de Silhouette en función del umbral de distancia')
    plt.plot([silsMan[i][1] for i in range(len(silsMan))],
        [silsMan[i][0][1] for i in range(len(silsMan))])
    plt.show()
    print("El valor óptimo de Silhouette es %0.3f" % maximan,
```

```
        "y se obtiene con umbral de distancia %0.2f" % epsilon_man,
        "que da lugar a",n_clusters_man,"clusters")
    graficaDBSCAN(epsilon_euc,'manhattan')

bestSilhouetteKmeans()
bestSilhouetteDBSCAN()
```