Práctica 3 - Geometría Computacional

Marcos Herrero Agustín

1. Introducción

Esta práctica consiste en clasificar un sistema X formado por 1000 elementos, con 2 estados cada uno. Para clasificarlo, se utilizarán los algoritmos KMeans y DBSCAN implementados en la biblioteca sklearn de Python. A continuación, se compararán los resultados obtenidos por ambos algoritmos.

2. Datos y condiciones iniciales

Para la realización de la práctica se han utilizado los siguientes datos y condiciones iniciales:

- La información con la que generar el sistema X a clasificar: centros, número de puntos y desviación.
- La semilla $random\_seed = 0$ para asegurar que el comportamiento estocástico de la generación del sistema y de la clasificación por KMeans sea el mismo en todas las ejecuciones.
- El rango discreto $\{2, \dots, 15\}$ en el que buscar el número óptimo de clusters en el apartado i).
- El rango continuo $(0.1, 0.4)$ en el que buscar el umbral de distancia óptimo en el apartado ii).
- El número de mínimo de puntos $n_0 = 10$ que ha de tener un entorno para que DBSCAN lo considere denso.
- Los puntos $a = (0, 0)$ y $b = (0, -1)$ a clasificar en el apartado iii).

3. Metodología

En el apartado i), para cada $k \in \{2, \dots, 15\}$ se ha ejecutado el algoritmo KMeans con k clusters sobre el sistema X. Tras cada ejecución, se ha calculado el coeficiente de Silhouette para medir cómo de buena es la clasificación. Al final, nos quedamos con el número de clusters que maximiza este coeficiente.

En el apartado ii), para distintos valores de ϵ en $(0.1, 0.4)$ se ha ejecutado el algoritmo DBSCAN con umbral de distancia ϵ sobre el sistema X y se ha calculado el coeficiente de Silhouette. Nos quedamos con el umbral de distancia que maximiza este coeficiente. Esto se ha realizado tanto con la distancia euclídea como con la distancia Manhattan. El error en el cálculo del ϵ óptimo se deduce de la granularidad utilizada al recorrer el intervalo, mientras que el error en el cálculo del coeficiente de Silhouette se estima como la máxima diferencia en valor absoluto en un entorno próximo al óptimo.

En el apartado iii), se ha utilizado la función *predict* para determinar los clusters de los elementos a y b en la clasificación óptima del apartado i).

4. Resultados y discusión

4.1. Apartado i)

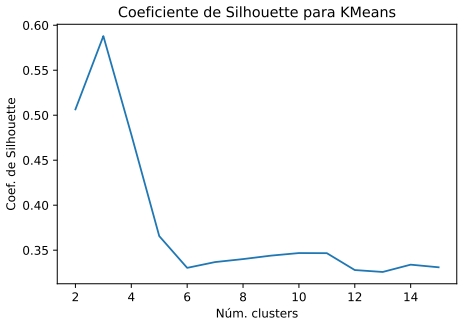

Figura 1

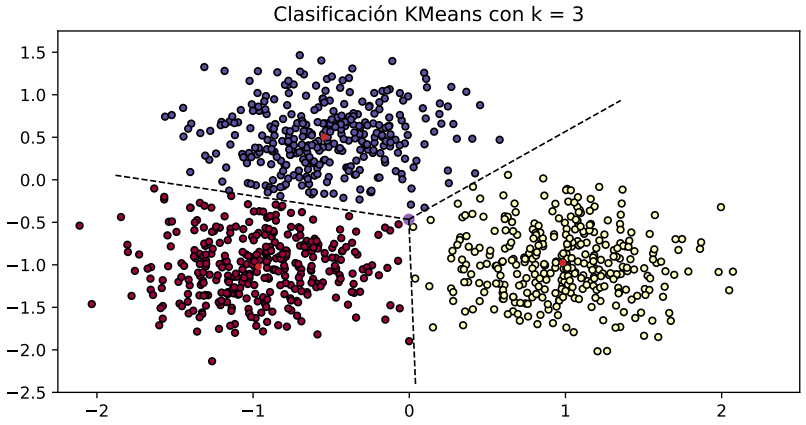

Figura 2

En la figura 1 se muestra el coeficiente de Silhouette obtenido para la ejecución de KMeans con cada número de clusters en $\{2, \dots, 15\}$. Se observa que el máximo coeficiente de Silhouette, que es de 0.588, se alcanza para 3 clusters. Por tanto, la distribución en clusters óptima utilizando KMeans es la que utiliza 3 clusters. Esta clasificación, junto con su diagrama de Voronoi asociado, se muestra en la figura 2.

4.2. Apartado ii)

En la figura 3 se muestra el coeficiente de Silhouette obtenido para la ejecución de DBSCAN con métrica euclídea para los umbrales de distancia en el intervalo $(0.1, 0.4)$. Se observa que el máximo coeficiente de Silhouette, que es de 0.467 ± 0.007, se alcanza para el umbral de distancia 0.280 ± 0.001. Por tanto, la distribución en clusters óptima utilizando DBSCAN con métrica

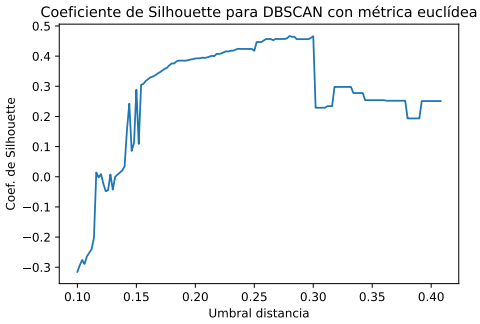

Figura 3

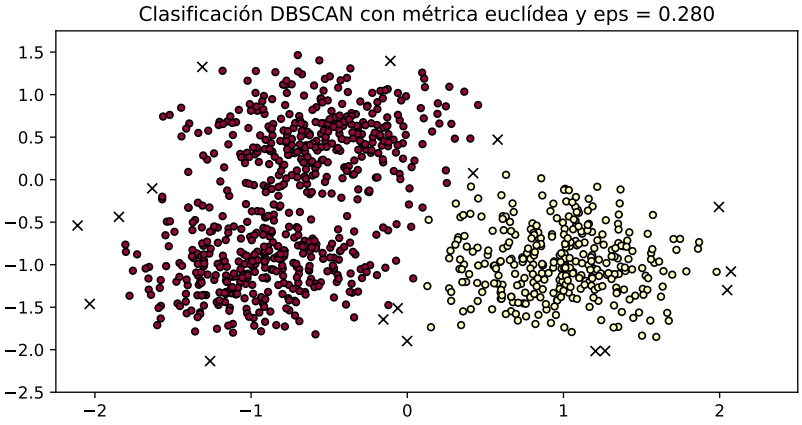

Figura 4

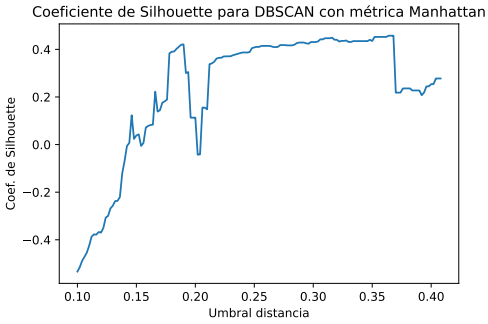

Figura 5

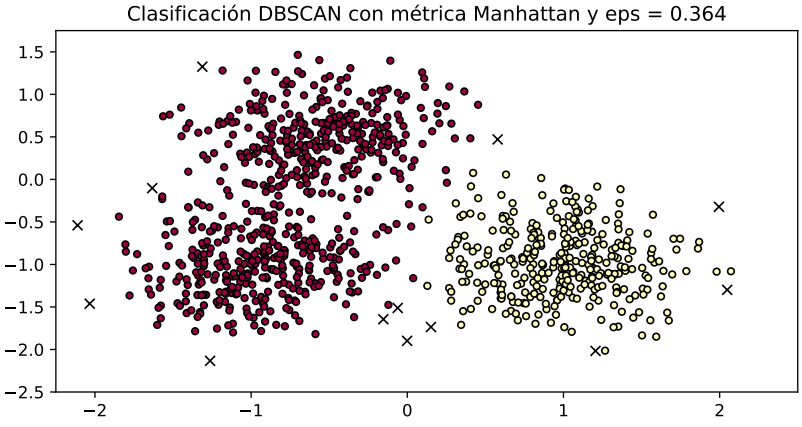

Figura 6

euclídea es la que utiliza 0.280 como umbral de distancia. Esta clasificación se muestra en la figura 4, en la que los elementos marcados con una x son los considerados ruido.

En la figura 5 se muestra el coeficiente de Silhouette obtenido para la ejecución de DBSCAN con métrica Manhattan para los umbrales de distancia en el intervalo $(0.1, 0.4)$. Se observa que el máximo coeficiente de Silhouette, que es de 0.457 ± 0.005, se alcanza para el umbral de distancia 0.364 ± 0.001. Por tanto, la distribución en clusters óptima utilizando DBSCAN con métrica Manhattan es la que utiliza 0.364 como umbral de distancia. Esta clasificación se muestra en la figura 6 (nuevamente, los elementos marcados con una x son los considerados ruido).

Al comparar las figuras 4 y 6 observamos que las diferencias entre el uso de DBSCAN con una u otra distancia son mínimas. Únicamente varían unos pocos elementos, que pasan de considerarse ruido a incluirse en un cluster o viceversa.

En cambio, comparando estas con la figura 2, se ve que sí hay una diferencia notable en la clasificación entre utilizar el algoritmo KMeans y DBSCAN. KMeans identifica correctamente el número de clusters que hemos generado y los determina de forma bastante ajustada. En cambio, DBSCAN solo detecta 2 clusters. Otra diferencia entre ambos es que el algoritmo de KMeans tiene que clasificar todos los elementos en alguno de los clusters, mientras que DBSCAN marca como ruido algunos de los elementos más aislados.

4.3. Apartado *iii)*

Vamos a clasificar los elementos $a := (0, 0)$ y $b := (0, -1)$ según la clasificación óptima del sistema con KMeans, que utiliza 3 clusters.

Observando la figura 2 podemos ver que el elemento $a = (0, 0)$ pertenece al cluster azul, por situarse en la región de Voronoi que contiene a este. Podemos llegar a esta misma conclusión utilizando la función *predict* de kmeans.

Análogamente, se observa en la figura que el elemento $b = (0, -1)$ pertenece al cluster rojo. De nuevo, podemos comprobarlo con la función *predict*.

5. Conclusiones

Hemos comprobado que para este caso concreto, en el que algunos de los clusters están muy próximos, el algoritmo DBSCAN no funciona adecuadamente. Al no haber una zona de baja densidad entre ambos, el algoritmo los identifica como uno solo. Por el contrario, el algoritmo KMeans sí los determina casi perfectamente.

Por tanto, se deduce que KMeans es el algoritmo apropiado para determinar clusters generados a partir de un centro y una desviación estándar respecto de él, como el usado en esta práctica. En cambio, DBSCAN debería usarse en otros contextos, por ejemplo, aquellos con abundancia de ruido.

Apéndice A Código utilizado

```
"""
Práctica 3 de Geometría Computacional

Autor: Marcos Herrero
"""

import numpy as np
import matplotlib.pyplot as plt

from sklearn.cluster import KMeans,DBSCAN
from sklearn import metrics
from sklearn.datasets import make_blobs
from scipy.spatial import Voronoi, voronoi_plot_2d


"""
Funciones auxiliares
"""

#Representa los elementos distribuidos en clusters y, opcionalmente,
#el diagrama de Voronoi asociado
def representarClusters(title, filename, labels, centers = None):
    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):

        size = 4
        marker = 'o'
        if k == -1:
            # Black used for noise.
            col = [0, 0, 0, 1]
            size = 6
            marker='x'

        class_member_mask = (labels == k)

        members = X[class_member_mask]
        plt.plot(members[:, 0], members[:, 1], marker, markerfacecolor=tuple(col),
                 markeredgecolor='k', markersize=size)

    axes = plt.gca()

    if not(centers is None):
        vor = Voronoi(centers)
        fig = voronoi_plot_2d(vor,ax=axes)

    plt.title(title)
    axes.set_xlim(-2.25,2.5)
    axes.set_ylim(-2.5,1.75)
    fig = plt.gcf()
    fig.savefig(filename,format='pdf')
    plt.show()

#Representa los clusters obtenidos por KMeans
def representarClustersKMeans(kmeans,filename):
    title = "Clasificación KMeans con k = {}".format(kmeans.n_clusters)
    labels = kmeans.labels_
    centers = kmeans.cluster_centers_
    representarClusters(title, filename, labels, centers)

#Representa los clusters obtenidos por DBSCAN
```

```
def representarClustersDBSCAN(db,filename):
    title =''

    if db.metric == 'euclidean':
        title = "Clasificación DBSCAN con métrica euclídea y eps = {:.3f}".format(
            db.eps)
    elif db.metric == 'manhattan':
        title = "Clasificación DBSCAN con métrica Manhattan y eps = {:.3f}".format(
            db.eps)

    labels = db.labels_
    representarClusters(title, filename, labels)

#Función que clasifica el sistema X usando KMeans para diferentes valores de k
#y devuelve la clasificación que da un mayor coeficiente de silhouette.
#También muestra gráficamente el coeficiente de silhouette obtenido para
# cada valor de k
def resolverKMeans(X):
    optimkmeans = None
    optimsilhouette = -2
    silhouette = np.empty(len(range(2,16)))
    ind = 0

    for k in range(2,16):

        kmeans = KMeans(n_clusters= k,random_state=0).fit(X)
        labels = kmeans.labels_
        silhouette[ind] = metrics.silhouette_score(X, labels)

        if silhouette[ind] > optimsilhouette:
            optimsilhouette = silhouette[ind]
            optimkmeans = kmeans


        ind += 1

    print("Número de clusters óptimo: {}".format(optimkmeans.n_clusters))
    print("Coeficiente de Silhouette: {}".format(optimsilhouette))

    plt.title("Coeficiente de Silhouette para KMeans")
    plt.plot(range(2,16),silhouette)
    plt.xlabel("Núm. clusters")
    plt.ylabel("Coef. de Silhouette")
    fig = plt.gcf()
    fig.savefig("silhouettekmeans.pdf",format='pdf')
    plt.show()



    return optimkmeans

#Función que resuelve el apartado ii) para el sistema X
def resolverDBSCAN(X,metrica):
    errorEps = 0.001 #error en la estimación del umbral optimo: lo fijamos nosotros


    optimdb = None
    optimsilhouette = -2
    epsilons = np.arange(0.1,0.41,2*errorEps)
    silhouette = np.empty(epsilons.size)
    ind = 0

    for epsilon in epsilons:

        db = DBSCAN(eps=epsilon, min_samples=10, metric=metrica).fit(X)
        labels = db.labels_
        silhouette[ind] = metrics.silhouette_score(X, labels)
```

```
        if silhouette[ind] > optimsilhouette:
            optimsilhouette = silhouette[ind]
            optimdb = db


        ind += 1

    #Estimación del error del coeficiente
    errorCoef = 0 #error en la estimación del coef. de Silhouette óptimo: hay que calcular
    delta = errorEps/20

    for e in np.arange(optimdb.eps - 2*errorEps,optimdb.eps + 2*errorEps,delta):
        db = DBSCAN(eps=e, min_samples=10, metric=metrica).fit(X)
        labels = db.labels_
        errorCoef = max(errorCoef,abs(optimsilhouette - metrics.silhouette_score(X,labels))


    print("Epsilon óptimo: {} | Error : {}".format(optimdb.eps,errorEps))
    print("Coeficiente de Silhouette: {} | Error : {} ".format(optimsilhouette,errorCoef))

    title =''
    filename=''
    if metrica == 'euclidean':
        title = "Coeficiente de Silhouette para DBSCAN con métrica euclídea"
        filename = "silhouettedbscaneucl.pdf"

    elif metrica == 'manhattan':
        title = "Coeficiente de Silhouette para DBSCAN con métrica Manhattan"
        filename = "silhouettedbscanmanh.pdf"

    plt.title(title)
    plt.plot(epsilons,silhouette)
    plt.xlabel("Umbral distancia")
    plt.ylabel("Coef. de Silhouette")
    fig = plt.gcf()
    fig.savefig(filename,format='pdf')
    plt.show()

    return optimdb

"""
Generar el sistema
"""
centers = [[-0.5, 0.5], [-1, -1], [1, -1]]
X, labels_true = make_blobs(n_samples=1000, centers=centers, cluster_std=0.4,
                            random_state=0)

"""
Apartado i): Obtener el número de clusters óptimo utilizando KMeans y el
coeficiente de Silhouette . Mostrar en un gráfica el coeficiente de Silhouette
para k =2,3,...,15. Representar la clasificación para el número óptimo de
clusters y el  diagrama de Voronoi en la misma gráfica.
"""

print("Apartado i):")
kmeans = resolverKMeans(X)
representarClustersKMeans(kmeans, "kmeans.pdf")
print()

"""
Apartado ii): Obtener el umbral de distancia epsilon óptimo utilizando el
DBSCAN y el coeficiente de Silhouette. Hacerlo con métrica euclidean y luego
con manhattan y comparar con la gráfica del apartado anterior
"""

print("Apartado ii):")

#Con métrica euclídea
```

```
print("Con métrica euclídea:")
dbeuclid = resolverDBSCAN(X,'euclidean')
representarClustersDBSCAN(dbeuclid,"dbscaneuclid.pdf")
print()

#Con métrica Manhattan
print("Con métrica Manhattan:")
dbmanh = resolverDBSCAN(X,'manhattan')
representarClustersDBSCAN(dbmanh,"dbscanmanh.pdf")
print()

"""
Apartado iii): Decidir a qué vecindad pertenecen los puntos a = (0,0) y
b = (0,-1)
"""

print("Apartado iii):")

a= [0,0]
b = [0,-1]

#Con kmeans
print("Según KMeans:")

clustera = kmeans.predict([a])
centroclusta = kmeans.cluster_centers_[clustera]
print("* El elemento a = {} pertenece al cluster {}, de centro {}".format(a,
        clustera[0],centroclusta))

clusterb = kmeans.predict([b])
centroclustb = kmeans.cluster_centers_[clusterb]
print("* El elemento b = {} pertenece al cluster {}, de centro {}".format(b,
        clusterb[0],centroclustb))
```

Apéndice B Resultado de la ejecución

```
Apartado i):
Número de clusters óptimo: 3
Coeficiente de Silhouette: 0.5879553162684444

Apartado ii):
Con métrica euclídea:
Epsilon óptimo: 0.28000000000000014 | Error : 0.001
Coeficiente de Silhouette: 0.46695850538182737 | Error : 0.0072153035681513655

Con métrica Manhattan:
Epsilon óptimo: 0.3640000000000002 | Error : 0.001
Coeficiente de Silhouette: 0.4574317766022809 | Error : 0.005040725836159721

Apartado iii):
Según KMeans:
* El elemento a = [0, 0] pertenece al cluster 2, de centro [[-0.5443386   0.50413529]]
* El elemento b = [0, -1] pertenece al cluster 0, de centro [[-0.96555004 -1.02042969]
```