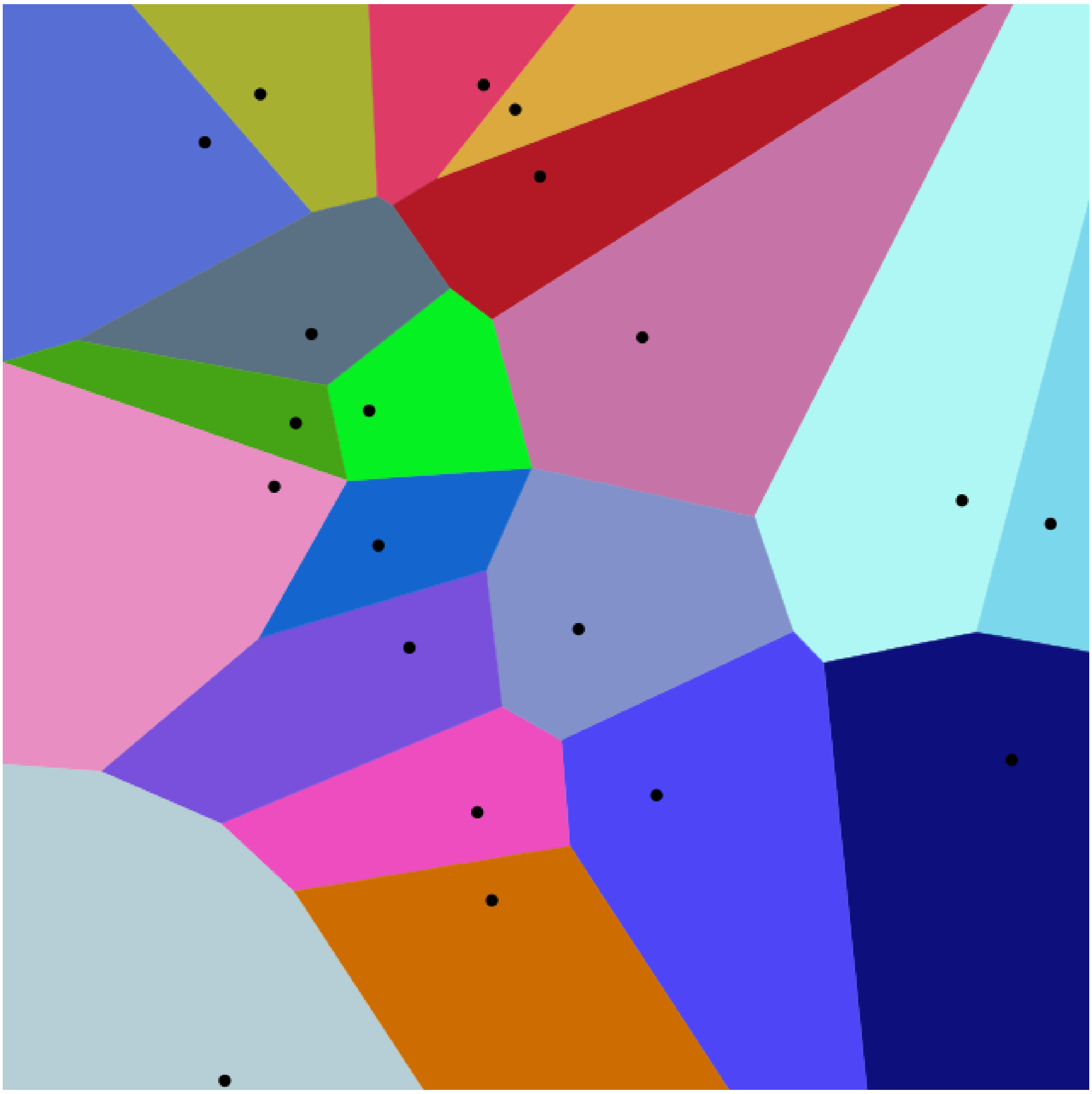
GEOMETRÍA COMPUTACIONAL

PRÁCTICA 3



DAVID SEIJAS PÉREZ

1. Introducción

En esta tercera práctica queremos aprender a clasificar un sistema con un número determinado de elementos, con dos estados cada uno de ellos, a partir de un número determinado de cluster o vencidades de Voronoi. Además, utilizaremos el coeficiente de Silhouette para determinar el número optimo de estas vecindades.

2. Material Usado

Para el desarrollo de esta práctica he utilizado diversas librerías dadas por Python. Primero, como siempre, hemos usado *matplotlib.pyplot* para la representación de gráfica y, esta vez, además hemos utilizado *Voronoi* y *voronoi_plot_2d* para representar el diagrama de Voronoi. Hemos utilizado también las librerías *KMeans* y *DBSCAN* para usar los algoritmos con el mismo nombre. Por último, hemos usado *metrics* y *make_blobs* para, principalmente, calcular el coeficiente *Silhouette* y el sistema X, respectivamente.

Además, he cogido las plantillas *GCOM2022-practica3_plantilla1* y *GCOM2022-practica3 _plantilla2* para reusar diversas partes del código como la representación gráfica de los clusters y los cálculo hechos en ambos algoritmos.

3. Metodología

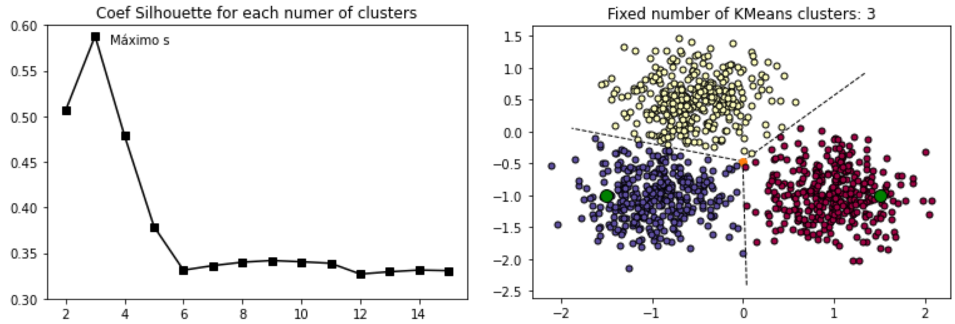
Para el primer apartado, he hecho la la función *mostrarKMEANS*, reutilizando código de la plantilla 1, que muestra los clusters calculados por el algoritmo **KMeans** del sistema para un k dado. Esta función es utilizada solo para representar gráficamente el sistema junto al diagrama de Voronoi al final del apartado. He realizado, también, la función *apartado1* en la que realizamos el algoritmo KMeans para las distintas ks especificadas y calculamos el coeficiente Silhouette asociada a cada una. Posteriormente, muestro en una gráfica la relación de este coeficiente según el k correspondiente y, así, determinamos el mejor número de vecindades, asociado al máximo coeficiente. Para este k vuelvo a realizar el algoritmo para poder mostrar en una gráfica el sistema clusterizado con los datos apropiados.

En el segundo apartado, la función *mostrarDBSCAN* muestra, como en la plantilla 2, los clusters obtenidos según el algoritmo **DBSCAN**. En la función *apartado2* he desarrollado el algoritmo DBSCAN, como se aporta en la plantilla2, para una distancia dada por parámetros y para todos los valores de ϵ desde 0.1 a 0.4 con paso 0.005. Mi idea ha sido recorrer todos los ϵ aplicando el algoritmo para ver con cual obtenemos el mejor coeficiente de Silhouette y, así, tener el mejor número de cluster para cada distancia. Una vez hallado este umbral de distancia, he vuelto a aplicar el algoritmo para poder mostrar por pantalla la gráfica con los datos óptimos. Además, con una pequeña modificación de la función, no incluida en el código, he estudiado como se comportaba este algoritmo para distintos valores de ϵ con cada una de las distancias.

En ambos apartados, en las funciones de mostrar resultados, he representado los puntos a y b del apartado 3 resaltados para ver gráficamente la pertenencia de cada uno a su cluster.

4. Resultados

1. En el primer apartado he obtenido que el número óptimo de vecindades es $k = 3$ para las cuales obtenemos el coeficiente de Silhouette $\bar{s} = 0.588$. A continuación muestro las gráficas obtenidas.



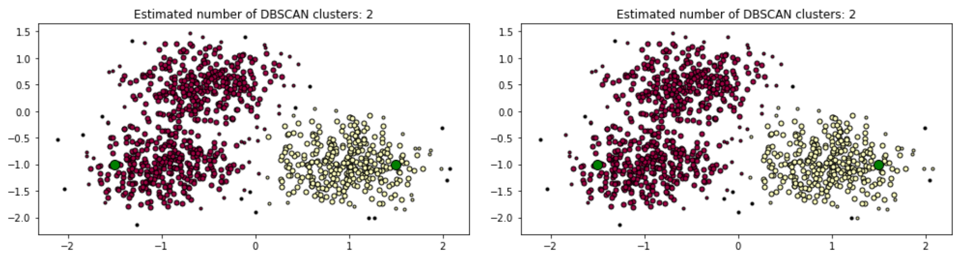
(a) Relación entre $\bar{s}$ y k (b) Diagrama Voronoi con a y b

2. En el segundo apartado he obtenido que el número estimado de clusters es 2 para ambas distancias en los mejores casos. Para la distancia Euclídea (a) he obtenido:

   - Umbral de distancia con el que se alcanza el mejor $\bar{s}$: $\epsilon = 0.28$
   - Número estimado de elementos ruidosos: 17
   - Coeficiente de Silhouette: $\bar{s} = 0.467$

   y para la distancia Manhattan (b):

   - Umbral de distancia con el que se alcanza el mejor $\bar{s}$: $\epsilon = 0.365$
   - Número estimado de elementos ruidosos: 13
   - Coeficiente de Silhouette: $\bar{s} = 0.546$



(a) Métrica Euclídea (b) Métrica Manhattan

Además, he analizado el comportamiento del algoritmo en las dos distancias según el parámetro ϵ. Para la distancia euclídea, con valores de $\epsilon > 3$ he obtenido un solo cluster con más elementos ruidosos y peores valores de $\bar{s}$ cuanto mayor es el valor. Decrementando ϵ se mantienen los dos clusters hasta 0.15, a partir de donde empiezan a aumentar mucho los clusters y llegamos, incluso, a obtener valores $\bar{s} < 0$. El comportamiento para la distancia Manhattan es parecido, aunque se comporta mejor con valores más grande ϵ, mantiene los dos clusters para valores más grandes de este parámetro y con mejores $\bar{s}$. Sin embargo, para valores más pequeños se comporta peor, a partir de $\epsilon < 2$ empiezan a aumentar mucho los clusters obteniendo valores más pequeños del coeficiente de Silhouette, hasta aproximadamente $\bar{s} = -0.43$

3. Para el apartado 3, en la función *apartado1* observamos que el punto a pertenece al cluster 2 y el b al 0, según la notación del algoritmo KMeans. La pertenencia de estos puntos a las vecindades con los dos algoritmos se muestran en los gráficos.

5. Conclusión

Me parecen fascinantes los métodos de clasificación vistos y cómo pueden varias las clasificaciones modificando ligeramente los datos. Por ejemplo, en el apartado 2 para algunos valores de ϵ se pueden obtener resultados realmente malos, incluso con valores del coeficiente de Silhouette negativo y una cantidad de clusters absurda para el sistema que estamos estudiando. Esto nos hace ver la importancia de determinar los valores óptimos para los cuales los algoritmos de clasificación se comportan como esperamos. Además, con el apartado 2 vemos también como se consiguen mejores resultado con la distancia Manhattan, lo cual me ha sorprendido bastante pues pensaba que sería al revés, mejor con la euclídea.

6. Anexo: Código

```
'''
DAVID SEIJAS PEREZ
Practica 3
'''

import numpy as np
from sklearn.cluster import DBSCAN
from sklearn.cluster import KMeans
from sklearn import metrics
from sklearn.datasets import make_blobs
from scipy.spatial import Voronoi, voronoi_plot_2d
import matplotlib.pyplot as plt


'''
Funcion que muestra los clusters calculados con KMEANS
'''
def mostrarKMEANS(X, labels, nClusters):
    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.plot(problem[:,0],problem[:,1],'ko', markersize=10,
        markerfacecolor="green")
    plt.axis('tight')
    plt.title('Fixed number of KMeans clusters: %d' % nClusters)
    plt.show()


def apartado1():
    ks = [i for i in range(2, 16)]
    coefsSil = [0]*14

    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)

    #Algoritmo KMEANS para las distintas vencidades k
    for i in range(len(ks)):
        kmeans = KMeans(n_clusters=ks[i], random_state=0).fit(X)
        labels = kmeans.labels_
        silhouette = metrics.silhouette_score(X, labels)
        #mostrarKMEANS(X, labels, problem, ks[i])

        '''
        # Etiqueta de cada elemento (punto)
        print(kmeans.labels_)
        #  ndice  de los centros de vencindades o regiones de Voronoi
            para cada elemento (punto)
        print(kmeans.cluster_centers_)
        #Coeficiente de Silhouette
        '''
        print("Silhouette Coefficient for k = " + str(ks[i])
              + ": %0.3f" % silhouette)
        coefsSil[i] = silhouette


    plt.ylim(0.3, 0.6)
    plt.plot(ks, coefsSil, 'ks-')
    plt.title('Coef Silhouette for each numer of clusters')
    plt.text(3.5, 0.58, "M ximo s")
    plt.show()
    #Con esto vemos que el mejor n mero de vecindades es k=3
    k = 3

    #Calculamos de nuevo todo para mostrar los clusters
    kmeans = KMeans(n_clusters=k, random_state=0).fit(X)
    labels = kmeans.labels_
```

```

    #Mostramos Diagrama de Vornoi junto con los clusters
    vor = Voronoi(kmeans.cluster_centers_)
    voronoi_plot_2d(vor)
    mostrarKMEANS(X, labels, k)
    #Al quitar el plt.figure de mostrarClusters no nos elimina la figura de voronoi creada antes

    #Calculamos a qu  clusters pertenecen los puntos a y b (Apartado 3)
    clases_pred = kmeans.predict(problem)
    print("Clasificaci n de los puntos a=(0,0) y b=(0,-1) para Kmeans:")
    print(clases_pred)
    print("\n-------------------------------\n")



'''
Apartado 2
'''

'''
Funcion que muestra los cluster calculados con DBSCAN
'''
def mostrarDBSCAN(X, labels, core_samples_mask, nClusters):
    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 & core_samples_mask]
        plt.plot(xy[:, 0], xy[:, 1], 'o', markerfacecolor=tuple(col),
                 markeredgecolor='k', markersize=5)

        xy = X[class_member_mask & ~core_samples_mask]
        plt.plot(xy[:, 0], xy[:, 1], 'o', markerfacecolor=tuple(col),
                 markeredgecolor='k', markersize=3)

    plt.plot(problem[:,0],problem[:,1],'ko', markersize=10, markerfacecolor="green")
    plt.title('Estimated number of DBSCAN clusters: %d' % nClusters)
    plt.show()


def apartado2(distancia):
    epsilons = [0.1 + 0.005*i for i in range(60)]
```

```
    silMax = 0
    epsilonMax = 0.1

    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)

    #Algoritmo DBSCAN
    for i in range(len(epsilons)):
        db = DBSCAN(eps=epsilons[i], min_samples=10, metric=distancia).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)
        n_noise_ = list(labels).count(-1)

        silhouette = metrics.silhouette_score(X, labels)
        if(silhouette > silMax):
            silMax = silhouette
            epsilonMax = epsilons[i]


    db = DBSCAN(eps=epsilonMax, min_samples=10, metric=distancia).fit(X)
    core_samples_mask = np.zeros_like(db.labels_, dtype=bool)
    core_samples_mask[db.core_sample_indices_] = True
    labels = db.labels_
    n_clusters_ = len(set(labels)) - (1 if -1 in labels else 0)
    n_noise_ = list(labels).count(-1)

    print(distancia + " DISTANCE")
    print('Best epsilon: %0.3f' % epsilonMax)
    print('Estimated number of clusters: %d' % n_clusters_)
    print('Estimated number of noise points: %d' % n_noise_)
    print("Adjusted Rand Index: %0.3f"
          % metrics.adjusted_rand_score(labels_true, labels))
    print("Silhouette Coefficient: %0.3f"
          % silMax)
    print("\n------------------------------\n")

    mostrarDBSCAN(X, labels, core_samples_mask, n_clusters_)


problem = np.array([[-1.5, -1], [1.5, -1]])
apartado1()
apartado2("euclidean")
apartado2("manhattan")
```