Práctica 2: Diagrama de Voronói y Clustering

Geometría Computacional
Subgrupo (U2)

Óscar Mesa Martín

1. Introducción

En esta segunda práctica implementaremos en *python* los algoritmos de **clasificación por k-medias** (o *clustering*) y **DBSCAN** en base a los datos creados por *Robert Monjo Agut* sobre la Facultad de Ciencias Matemáticas (UCM) en dos archivos de texto. Ambos algoritmos serán importaciones de la librería *sklearn*, pues en ella nos apoyaremos para poder realizar un *clustering* (o clasificación de vecindades), así como para poder realizar los respectivos *diagramas de Voronói* o, simplemente, calcular los *Coeficientes de Silhouette*.

2. Método

Empezaremos el experimento partiendo del sistema $S = \{(a_i, \{X_j\}_{j=1}^{2})\}_{i=1}^{1500}$ de datos facilitados y los plasmaremos en $\mathbb{R}^2$ (*Figura 2.1*) apoyándonos en la función *matplotlib.pyplot* (de la librería *matplotlib*) para graficar con comodidad. Luego haremos la **envoltura convexa** (*Figura 2.2*) mediante la función *ConvexHull* (de la librería *scipy*) para poder delimitar la región sobre la que trabajaremos.

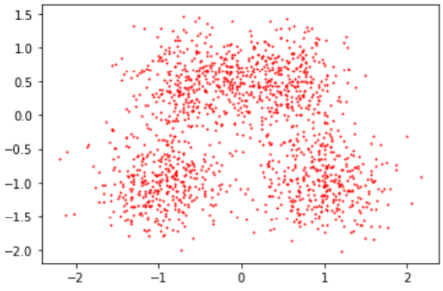

Figura 2.1: Distribución de los Puntos en $\mathbb{R}^2$

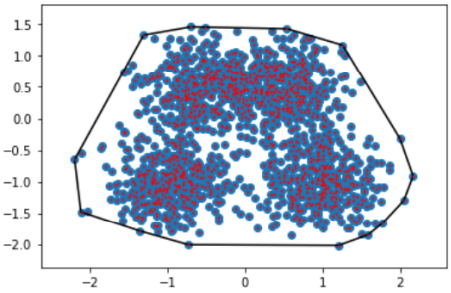

Figura 2.2: Envolvente Convexa del sistema S

I) **Algoritmo de Clasificación por k-medias**:

Ahora procederemos a calcular el *coeficiente de Silhouette* para los clusters (*o vecindades de Voronói*) $k \in \{2, 3, ..., 15\}$, para ello, usaremos la función *metrics.silhouette_score* de la librería *sklearn*. El *coeficiente de Silhouette* es una medida que nos indica la semejanza que tiene un elemento en relación al cluster al que pertenece, yendo desde -1 (si ese elemento no está bien emparejado con su propio cúmulo) hasta 1 (el caso diametralmente opuesto).

Lo haremos apoyándonos en el algoritmo de *k-medias* (que no es más que la clase *KMeans* de la librería *sklearn*), que nos permite conocer la región a la que pertenece cada punto con el método *labels_*. Con esto, implementaremos un *bucle for* sobre el número de vecindades para ver cuál de ellas tiene el mejor (mayor) *coeficiente de Silhouette*.

II) **Algoritmo DBSCAN**:

La idea que recoge este algoritmo es la de clasificar las regiones distinguiendo las vecindades de **alta densidad** que están separadas por regiones de baja densidad. Para ello se establece un **número mínimo de elementos** que han de pertenecer a una región para que sea considerada de alta densidad (*en nuestro caso ese número es 10*).

Para el algoritmo *DBSCAN*, podremos elegir la métrica que usaremos (*Euclídea* ó *Manhattan*). A la hora de implementarlo, nos ayudaremos de la clase *DBSCAN* de la librería *sklean*.

De manera análoga a como hicimos con KMeans, también calcularemos el *coeficiente de Silhouette* pero, esta vez, teniendo en cuenta el ϵ **(épsilon)** que tomaremos, oscilando en el intervalo (0.1, 0.4) y no el número de clusters.

3. Resultados

I) **Algoritmo de Clasificación por k-medias**:

Con nuestros datos, el **número óptimo de clusters** (acorde al mayor *Coeficiente de Silhouette*) es 3, mientras que el **Coeficiente de Silhouette** es 0,509 (*Figura 3.1*).

Una vez descubierto el número de vecindades que va a tener nuestro sistema S, podemos proceder mediante KMeans a identificar cada punto con su cluster (*Figura 3.2*) mediante la función labels_. De igual manera, usando cluster_centers_, designaremos cada uno de los tres centroides.

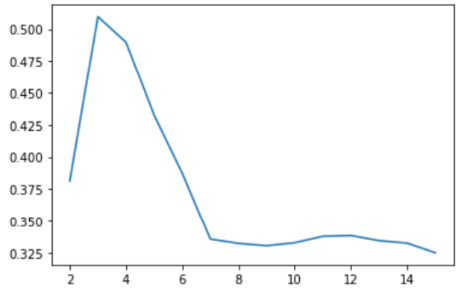

Figura 3.1: Coef. de Silhouette VS Número de Clusters

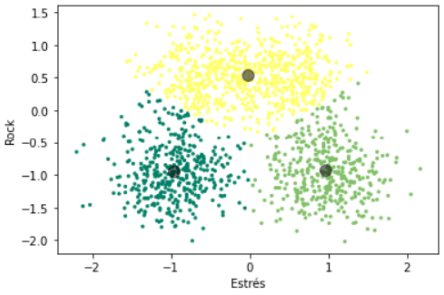

Figura 3.2: Clasificación de los Clusters (KMeans)

Para construir el *Diagrama de Voronói* deseado (*Figura 3.3*) usaremos las funciones *Voronoi* y *voronoi_plot_2d* y así representar las 3 regiones ya delimitadas.

Por otra parte, lejos de predecir a ojo los puntos $\mathbf{a} = (0, 0)$ y $\mathbf{b} = (0, -1)$ viendo el diagrama de Voronói, lo haremos usando la función *kmeans.predict* (*Figura 3.4*). Obviaremos los puntos del sistema, sustituyéndolos por el nombre del Cluster, para mejorar la visualización.

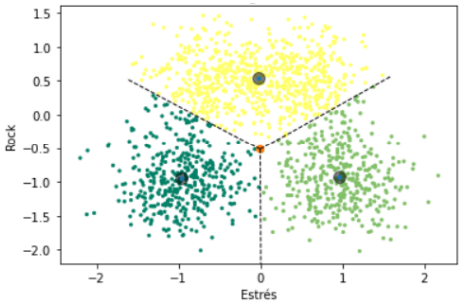

Figura 3.3: Diagrama de Voronói

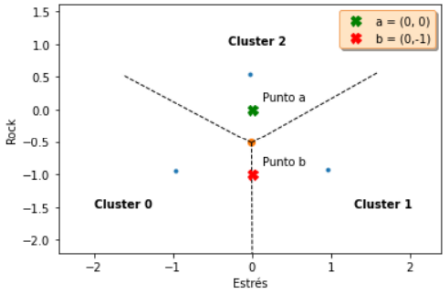

Figura 3.4: Solución del apartado **iii)**

II) **Algoritmo DBSCAN**:

Antes de nada, ejemplificaremos con la métrica *Euclídea*, siendo totalmente simétrico para la *Manhattan*. Haciendo un procedimiento análogo al que hicimos con el Coeficiente de Silhouette para ver el número de Clusters en KMeans (ahora sobre ϵ (*épsilon*)) obtendremos un **Coeficiente de Silhouette** de 0,354 y un **número de Clusters** de 1 (*Figura 3.5*). El método para llegar a la gráfica de los Clusters con *DBSCAN* (*Figura 3.6*) es totalmente análogo al que se hizo con *KMeans*.

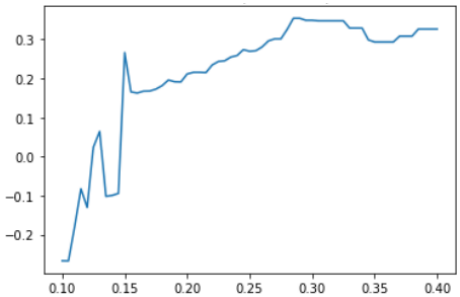

Figura 3.5: Coef. de Silhouette VS ϵ (épsilon)

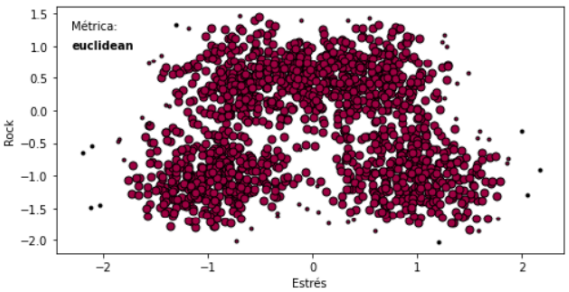

Figura 3.6: Clasificación de los Clusters (DBSCAN)

4. Conclusión

Tan sólo viendo las gráfica nos podemos dar cuenta de la gran diferencia entre los dos algoritmos. Por una parte, *KMeans*, al no regirse por un número mínimo de elementos por cluster (n_0) ni de un umbral de distancia (ϵ), posee los 3 clusters que se apreciaban intuitivamente, esféricos y convexos. Por otra parte, *DBSCAN*, al poner más requisitos sobre las vecindades densas, puede engañarnos gráficamente entendiendo todo el conjunto como un único cluster. Después de haber trabajado en esta práctica, ¿se podría decir que no existen personas raras, sino puntos ruidosos mal clasificados?

5. Anexo (código *.py*)

Código del archivo ***Práctica2_MesaMartínÓscar.py***:

```
##########################################################################
##################                                      ##################
################## PRÁCTICA 2 (GEOMETRÍA COMPUTACIONAL) ##################
####################                                  ####################
##################  (Diagrama de Voronói y Clustering)  ##################
##################                                      ##################
##########################################################################



##########################################################################
#########################                        #########################
######################### IMPORTACIÓN DE LIBRERÍAS #########################
#########################                        #########################
##########################################################################

import numpy as np

# Para el Algoritmo KMeans (K-Medias):
from sklearn.cluster import KMeans

# Para el Algoritmo KMeans (DBSCAN):
from sklearn.cluster import DBSCAN

# Para Envoltura Convexa, Diagrama de Voronói y Cfte de Silhouette:
from scipy.spatial import ConvexHull, convex_hull_plot_2d
from scipy.spatial import Voronoi, voronoi_plot_2d
from sklearn import metrics

# Para las gráficas:
import matplotlib.pyplot as plt

# Para obtener el directorio actual de trabajo:
import os
os.getcwd()

# Para obviar las posibles incompatibilidades del módulo sklearn:
import warnings
warnings.filterwarnings("ignore")



##########################################################################
#########################                        #########################
######################### PREPARACIÓN DE LOS DATOS #########################
#########################                        #########################
##########################################################################


# Tomamos los archivos donde se encuentra nuestro sistema S de:
#   1500 personas
#   2 estados

archivo1 = "Personas_en_la_facultad_matematicas.txt"
archivo2 = "Grados_en_la_facultad_matematicas.txt"
```

```
X = np.loadtxt(archivo1)
Y = np.loadtxt(archivo2)

# Del archivo 2, tomamos el grado al que pertenecen (array):

labels_true = Y[:,0]

# Mostramos los datos facilitados por pantalla:

header = open(archivo1).readline()
print("Archivo de Personas \n")
print(header)
print(X, '\n\n')

# Los situamos sobre una gráfica:

# X[:,0] := Del archivo 1, coge los valores de la primera columna (array)
#                                la variable de estado  X1 (estrés)
# X[:,1] := Del archivo 1, coge los valores de la segunda columna (array)
#                                la variable de estado  X2 (afición)

print("Graficamos los datos facilitados: \n")

plt.plot(X[:,0], X[:,1], 'ro', markersize = 1)
plt.xlabel('Estrés')
plt.xlim(-2.45,2.4)
plt.ylabel('Rock')
plt.ylim(-2.2,1.6)
plt.title("Gráfica de la nube de puntos")
plt.show()

# Realizamos su Envolvente Convexa y la graficamos:

print("\n\nRealizamos su Envolvente Convexa: \n")

envolvente = ConvexHull(X)
convex_hull_plot_2d(envolvente)
plt.plot(X[:,0], X[:,1], 'ro', markersize = 1)
plt.xlabel('Estrés')
plt.xlim(-2.45,2.4)
plt.ylabel('Rock')
plt.ylim(-2.2,1.6)
plt.title("Gráfica de la Envolvente Convexa")
plt.show()



##################################################################################
############################# ALGORITMO K-MEANS ##################################
########################## ------------------------ ##############################
####################### COEFICIENTE DE SILHOUETTE ######################
##################################################################################

# Número de clusters (vecindades) de 2 a 15:
num_clusters = range(2,16)

# Lista donde agregaremos los Coeficientes de Silhouette que generamos:
cfts_silhouette = []

# Bucle para los diferentes números de vecindades (2,...,15):
```

```
for k in num_clusters:
    kmeans = KMeans(n_clusters = k, random_state = 0).fit(X)
    labels = kmeans.labels_
    silhouette = metrics.silhouette_score(X, labels)
    cfts_silhouette.append(silhouette)


# Gráfica de los Coef. de Silhouette en función de k:

print("\n\nSeleccionamos el Número de Clusters en función del",
      "\nCoeficiente de Silhouette que, en este caso, es:\n",
      "    3 = Número de Clusters\n")

plt.plot(num_clusters, cfts_silhouette)
plt.xlabel('Número de Clusters')
plt.ylabel('Coeficiente de Silhouette')
plt.title("Selección del Número de Clusters")
plt.show()



################################################################################
############################## ALGORITMO K-MEANS ###############################
########################## ------------------------- ###########################
###################### CLASIFICACIÓN DE CLUSTERS #######################
################################################################################


# Número de clusters óptimos:
clusters_optimos = num_clusters[cfts_silhouette.index(max(cfts_silhouette))]

# Coeficiente de Silhouette para el número de clusters óptimos:
cft_optimo = max(cfts_silhouette)

# Generamos los datos mediante KMeans:
kmeans = KMeans(n_clusters = clusters_optimos, random_state = 0).fit(X)
centers = kmeans.cluster_centers_
y_kmeans = kmeans.predict(X)
labels = kmeans.labels_
silhouette = metrics.silhouette_score(X, labels)

# Gráfica de la Clasificación de Clusters:

#   - Eje X: Nivel de Estrés
#   - Eje Y: Nivel de Rock
#   - Alpha: Opacidad del centroide (de 0 a 1)
#   - s    : Tamaño de los puntos

print("\n\nClasificamos los tres Clusters gráficamente con",
      "\nsus centros y por colores:\n")

plt.scatter(X[:, 0], X[:, 1], c = y_kmeans, s = 5, cmap = 'summer')
plt.scatter(centers[:, 0], centers[:, 1], c = 'black', s = 100, alpha = 0.5)
plt.xlabel('Estrés')
plt.xlim(-2.45,2.4)
plt.ylabel('Rock')
plt.ylim(-2.2,1.6)
plt.title("Gráfica de la Clasificación de Clusters (KMeans)")
plt.show()
```

```
################################################################################
############################# ALGORITMO K–MEANS #############################
######################### ——————————————————————— #########################
###################### DIAGRAMA DE VORONÓI ######################
################################################################################


# Gráfica del Diagrama de Voronói:

print("\n\nEn la gráfica anterior, realizamos el Diagrama de Voronói:\n")

voro_centers = Voronoi(centers)
voronoi_plot_2d(voro_centers)

plt.scatter(X[:, 0], X[:, 1], c = y_kmeans, s = 5, cmap = 'summer')
plt.scatter(centers[:, 0], centers[:, 1], c = 'black', s = 100, alpha = 0.5)
plt.xlabel('Estrés')
plt.xlim(-2.45,2.4)
plt.ylabel('Rock')
plt.ylim(-2.2,1.6)
plt.title("Gráfica del Diagrama de Voronói")
plt.show()



################################################################################
############################# ALGORITMO K–MEANS #############################
######################### ——————————————————————— #########################
###################### PREDICCIÓN DE PUNTOS ######################
################################################################################

# Implementación de los puntos a y b:
puntos_predecir = [[0, 0], [0, -1]]
problem = np.array(puntos_predecir)

# Gráfica de la Predicción de los puntos:

print("\n\nHacemos desaparecer todos los puntos para facilitar",
      "\nla visualización de los puntos predichos (a y b)\n")

voronoi_plot_2d(voro_centers)
plt.plot(0,  0, 'gX', markersize = 9, label = "a = (0, 0)")
plt.plot(0, -1, 'rX', markersize = 9, label = "b = (0,-1)")
plt.annotate("Punto a", (0 + 0.14,  0.14))
plt.annotate("Punto b", (0 + 0.14, -0.86))
plt.annotate("Cluster 0", (  -2, -1.5), fontweight = "bold")
plt.annotate("Cluster 1", ( 1.3, -1.5), fontweight = "bold")
plt.annotate("Cluster 2", (-0.3,  1  ), fontweight = "bold")
plt.legend(loc       = "best",
           shadow    = True,
           facecolor = "Bisque",
           edgecolor = "SandyBrown")
plt.xlabel('Estrés')
plt.xlim(-2.45,2.4)
plt.ylabel('Rock')
plt.ylim(-2.2,1.6)
plt.title("Gráfica de la Predicción de Puntos")
plt.show()
```

```
# Predicción de los puntos con kmeans.predict:

print("\n\n La predicción (kmeans.predict) para los puntos",
      puntos_predecir,
      ":\n\n    ",
      kmeans.predict(problem), "\n\n")



##################################################################################
###########################  ALGORITMO DBSCAN  ###########################
########################  ————————————————  ########################
######################  COEFICIENTE DE SILHOUETTE  ######################
##################################################################################


elegirMetric = input("¿Qué métrica desea utilizar?:\n"
                     "\n    a) Euclídea    b) Manhattan    \n\n ")

if   elegirMetric == ('a' or 'A'):
    metrica = 'euclidean'
elif elegirMetric == ('b' or 'B'):
    metrica = 'manhattan'

# Lista de los valores (discretizados) de épsilons en (0.1, 0.4):
epsilons = np.arange(0.1,0.4, 0.005)

# Lista donde agregaremos los Coeficientes de Silhouette que generamos:
cfts_silhouette_eps = []

for e in epsilons:
    db              = DBSCAN(eps = e, min_samples = 10,
                             metric = metrica).fit(X)
    labels          = db.labels_
    silhouette_eps = metrics.silhouette_score(X, labels)
    cfts_silhouette_eps.append(silhouette_eps)

# Épsilon óptimo:
eps_optimo = epsilons[cfts_silhouette_eps.index(max(cfts_silhouette_eps))]

# Coeficiente de Silhouette para el épsilon óptimo:
cft_optimo_eps = max(cfts_silhouette_eps)

# Gráfica del número de Cluster en función del épsilon tomado:

print("\n\nSeleccionamos el épsilon que tomaremos en función",
      "\ndel Coeficiente de Silhouette que, en este caso, es:\n",
      "    %0.3f" % eps_optimo)

plt.plot(epsilons, cfts_silhouette_eps)
plt.xlabel('épsilon ($\epsilon$)')
plt.ylabel('Coef. de Silhouette')
plt.title("Selección del Épsilon $(\epsilon)$ óptimo")
plt.show()



# Number of clusters in labels, ignoring noise if present (24-1=23)
n_clusters_ = len(set(labels)) - (1 if -1 in labels else 0)
```

```
# Número de veces que aparece el -1 en labels (número de puntos ruidosos):
n_noise_ = list(labels).count(-1)

print('\n\nNúmero de Clusters esperados:',
      '\n     %d' % n_clusters_)
print('\nNúmero de Puntos Ruidosos:',
      '\n     %d' % n_noise_)
print('\nAdjusted Rand Index:'
      '\n     %0.3f'
      % metrics.adjusted_rand_score(labels_true, labels))



##############################################################################
############################## ALGORITMO DBSCAN ##############################
########################### ---------------------- ###########################
######################### CLASIFICACIÓN DE CLUSTERS ##########################
##############################################################################


db = DBSCAN(eps = eps_optimo, 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_

# Conjunto de etiquetas
unique_labels = set(labels)
# Lista con tantos colores distintos como etiquetas:
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=7)

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

plt.annotate("Métrica:",      ( -2.3, 1.25))
plt.annotate("%s" % metrica, ( -2.3, 0.95), fontweight = "bold")
plt.xlabel('Estrés')
plt.xlim(-2.45,2.4)
plt.ylabel('Rock')
plt.ylim(-2.2,1.6)
plt.title("Gráfica de la Clasificación de Clusters (DBSCAN)")
plt.show()
```