Práctica 1 GCOMP: DIAGRAMA DE VORONÓI Y CLUSTERING

Enrique Carro Garrido

I. Introducción (motivación/objetivo de la práctica)

En esta práctica, nos centraremos en la clasificación como un tipo particular de aprendizaje no supervisado, en particular en la estratificación por clases (clustering), utilizando distintos algoritmos.

Cada una de estos algoritmos tiene una ventaja con respecto al resto dependiendo de la estructura que tenga el sistema con el que se trabaja. Los algoritmos utilizados son:

- Familia de algoritmos de clustering o clasificación por k-medias (KMeans), que depende del número de clusters, "k", que se quiera obtener.
- Familia de algoritmos "Density-Based Spatial Clustering of Applications with Noise" (DBSCAN) que depende de una distancia ϵ.

El otro objetivo de la práctica es saber comparar estos algoritmos a partir de una medida de ayuda a la decisión: El coeficiente de Silhouette. De esta manera se puede determinar que algoritmo es el que más ayuda a la clasificación.

II. Material usado (método y datos)

a. El **método** utilizado es la estratificación por clases o *Clustering*. Consiste en el agrupamiento de un conjunto de nuevos estados (representados mediante datos empíricos) en un determinado número de clases o categorías, que van entre 1 y N, donde N es el número de estados de referencia (usados en el entrenamiento). Para su implementación se han utilizado los siguientes algoritmos:

   - Algoritmo KMeans: Se basa en la distribución de los elementos del sistema en un conjunto de k celdas de Voronói para la distancia euclidiana.
   - Algoritmo DBSCAN: El concepto principal del algoritmo es localizar regiones de alta densidad que están separadas entre sí por regiones de baja densidad. Para ello se define la vecindad o ϵ-bola a partir de una distancia. En esta práctica se comparan 2 distancias: La métrica euclidiana (L^2) y la métrica de Manhattan (L^1).

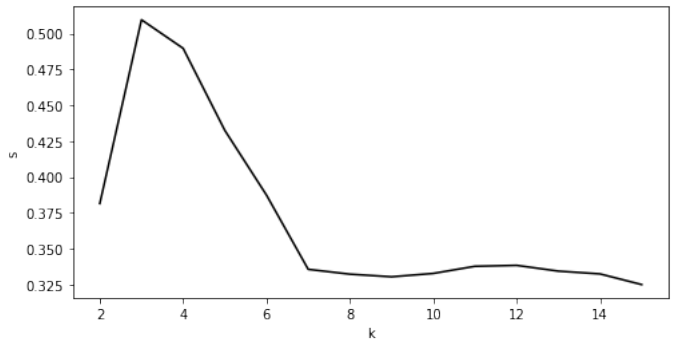
b. Los **datos** utilizados para su desarrollo son:

   - *"Personas_en_la_facultad_matematicas.txt"*, donde se presentan dos variables de estado, X_1 = "nivel de estrés" y X_2 = "afición al rock", para un conjunto A de 1500 personas de la Facultad de Matemáticas. Estos datos son utilizados como conjunto de entrenamiento para los algoritmos mencionados.
   - *"Grados_en_la_facultad_matematicas.txt"*, donde se presenta el listado de pertenencia de cada persona a uno de los 4 grados o doble-grados principales de la Facultad de Matemáticas etiquetados como $0, 1, 2, 3$ para preservar el anonimato de cada caso. Estos datos se utilizan para predecir el grado al que deberían pertenecer una persona dada por su "nivel de estrés" y "afición al rock".
   - *"GCOM2023_practica2_plantilla1.py"*, que aplica un ejemplo de KMeans de la librería sklearn, con métrica euclidiana.
   - *"GCOM2023_practica2_plantilla2.py"*, que aplica un ejemplo de DBSCAN de la librería sklearn, con métrica de Manhattan.

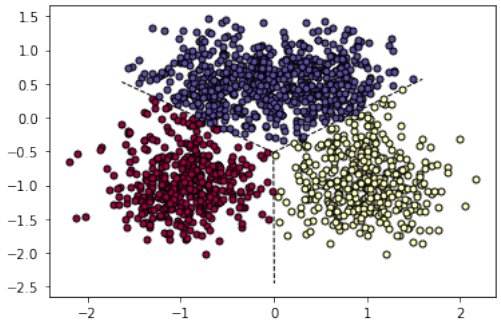
III. Resultados

a. *Obtén el coeficiente $\bar{s}$ de A para diferente número de vecindades $k \in 2, 3, ..., 15$ usando el algoritmo KMeans. Muestra en una gráfica el valor de $\bar{s}$ en función de k y decide con ello cuál es el número óptimo de vecindades. En una segunda gráfica, muestra la clasificación (clusters) resultante con diferentes colores y representa el diagrama de Voronói en esa misma gráfica.*

   Como se puede apreciar claramente en la Figura 1a el valor óptimo del coeficiente de Silhouette ($\bar{s}$) se alcanza con 3 vecindades. Por lo tanto, cogiendo ese número óptimo de vecindades, mostramos en la Figura 1b la clasificación con diferentes colores y el diagrama de Voronói obtenido trazado con líneas discontinuas.

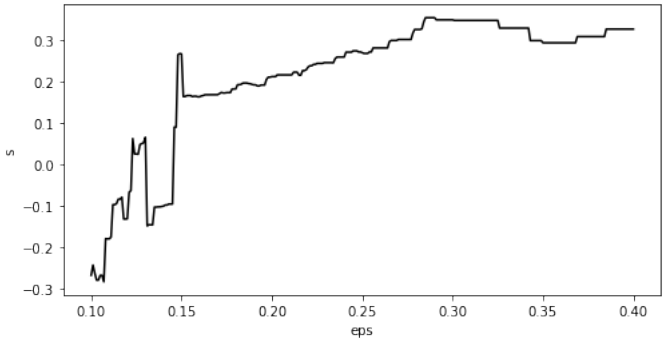


(a) Valor de Silhouette en función del número de clusters en el algoritmo de KMeans

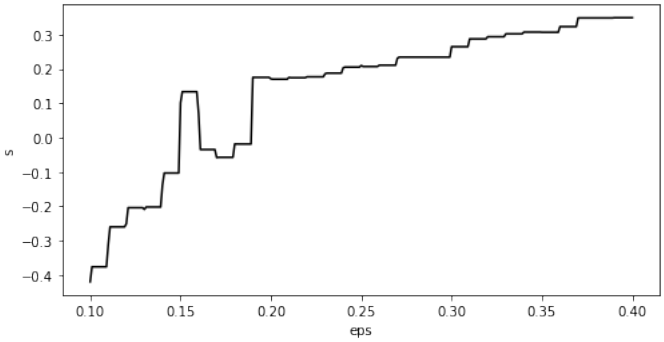


(b) Clasificación y diagrama de Voronói

Figura 1



(a) Silhouette en función de épsilon - DBSCAN (euclidiana)



(b) Silhouette en función de épsilon - DBSCAN (Manhattan)

Figura 2

b. *Obtén el coeficiente $\bar{s}$ para el mismo sistema A usando ahora el algoritmo DBSCAN con la métrica 'euclidean' y luego con 'manhattan'. En este caso, el parámetro que debemos explorar es el umbral de distancia $\epsilon \in (0{,}1, 0{,}4)$, fijando el número de elementos mínimo en $n_0 = 10$. Comparad gráficamente con el resultado del apartado anterior.*

   En la Figura 2 se ven los valores de $\bar{s}$ obtenidos en función del umbral de distancia ϵ.

   Para comparar gráficamente los resultados se tendría que mostrar los valores de $\bar{s}$ en función a una misma variable, que lógicamente tendría que ser el número de clusters. Sin embargo, en el algoritmo de DBSCAN no se puede introducir la k como valor de entrada, sino el umbral de distancia, que a partir del cual se obtiene el número de vecindades. Por lo tanto, se ha decidido coger el máximo valor de Silhouette para cada número de clusters obtenido y así tener una función bien definida para cada algoritmo que vaya de k a $\bar{s}$.

   Las 3 gráficas las hemos incluido en una en la Figura 3 y se puede apreciar que el valor de Silhouette es siempre mayor con el algoritmo de KMeans y por lo tanto el valor óptimo se alcanza cuando $k = 3$ en el algoritmo de KMeans. También se puede apreciar que el valor óptimo de Silhouette tanto para DBSCAN con métrica euclidiana como con métrica Manhattan se alcanza con un solo cluster y es aproximadamente el mismo y se comportan prácticamente de la misma manera; el valor de s decae a medida que aumenta el número de clusters.

c. *¿De qué Grado diríamos que son las personas con coordenadas $a := (0, 0)$ y $b := (0, -1)$? Comprueba tu respuesta con la función "kmeans.predict".*

   a) El grado al que debería pertenecer la persona $a := (0, 0)$ es: 3.

   b) El grado al que debería pertenecer la persona $b := (0, -1)$ es: 2.

IV. Conclusión

A partir de los resultados obtenidos se puede concluir que en este sistema estático el algoritmo KMeans funciona mejor. Esto es debido a que las nubes de puntos se aproximan a una distribución *gaussiana* (Como se puede apreciar en la Figura 1b) y localizar regiones de alta densidad que están separadas por regiones de baja densidad es imposible a menos que se considere un número alto de estados como ruido, empeorando así la clasificación.

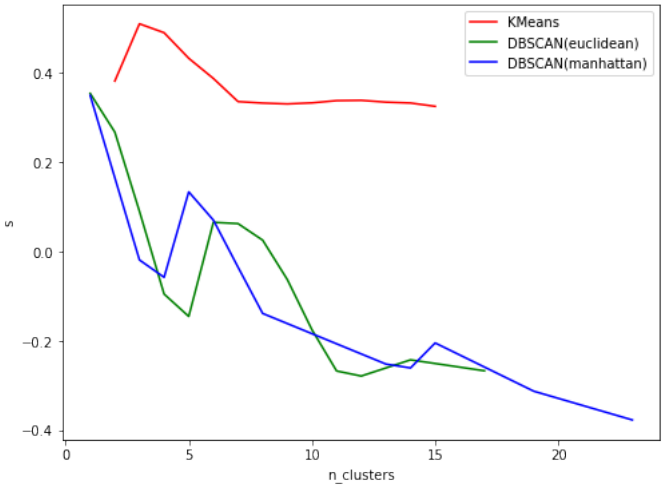


Figura 3: Comparación gráfica de los algoritmos con respecto al valor de Silhouette

V. Anexo con el código utilizado

```
import numpy as np

from sklearn.cluster import KMeans
from sklearn import metrics
from scipy.spatial import ConvexHull, convex_hull_plot_2d
from scipy.spatial import Voronoi, voronoi_plot_2d
import matplotlib.pyplot as plt
from sklearn.cluster import DBSCAN
import sys

# Aquí tenemos definido el sistema A de 1500 elementos (personas) con dos estados
archivo1 = "Personas_en_la_facultad_matematicas.txt"
archivo2 = "Grados_en_la_facultad_matematicas.txt"
X = np.loadtxt(archivo1)
Y = np.loadtxt(archivo2)

# Definimos la siguiente función para no tener que repetir el código cada vez
# que queremos dibujar una gráfica.
def plot_X_to_Y(X, Y, xlabel, ylabel, title):
    """
    Utilizamos esta función para dibujar la gráfica de la función Y = f(X).
    En esta práctica la utilizaremos para representar el valor de Silhouette
    en función del número de clusters o del umbral de distancia.

    Args:
        X (array): Lista con los valores en el eje de la X.
        Y (array): Lista con los valores en el eje de la Y.
        xlabel (string): Etiqueta para el eje de la X.
        ylabel (string): Etiqueta para el eje de la Y.
        title (string): Título de la gráfica.
    """
    plt.figure(figsize=(8,4))
    plt.plot(X, Y, color='black')
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.title(title)

##############################################################################
#                            RESPUESTAS A LOS APARTADOS                      #
##############################################################################

# i) Obtén el coeficiente s de A para diferente número de vecindades
```

```
# k  {2, 3, ..., 15} usando el algoritmo KMeans. Muestra en una
# gráfica el valor de s en función de k y decide con ello cuál
# es el número óptimo de vecindades. En una segunda gráfica, muestra
# la clasificación (clusters) resultante con diferentes colores y
# representa el diagrama de Voronói en esa misma gráfica.

n_clusters_kmeans=range(2,16,1)

# Obtenemos el coeficiente de Silhouette de X para las diferentes
# vecindades definidas en n_clusters
silhouette_kmeans = list()
for k in n_clusters_kmeans:
    kmeans = KMeans(n_clusters=k, random_state=0).fit(X)
    labels = kmeans.labels_
    silhouette_kmeans.append(metrics.silhouette_score(X, labels))

# Mostramos en una gráfica el valor de Silhouette en función de k.
plot_X_to_Y(X=n_clusters_kmeans,
            Y=silhouette_kmeans,
            xlabel='n_cluster',
            ylabel='Silhouette',
            title='Valor de silhouette en función del número de clusters')

# Obtenemos el número óptimo de vecindades
opt_vecindades = n_clusters_kmeans[np.argmax(silhouette_kmeans)]

# Obtenemos la clasificación para ese número de vecindades
kmeans = KMeans(n_clusters=opt_vecindades, random_state=0).fit(X)
labels = kmeans.labels_
centroids_kmeans = kmeans.cluster_centers_

# Representamos el resultado con un plot
unique_labels = set(labels)
colors = [plt.cm.Spectral(each)
for each in np.linspace(0, 1, len(unique_labels))]

voronoi_plot_2d(Voronoi(centroids_kmeans), show_vertices = False)
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.autoscale()
plt.title('Fixed number of KMeans clusters: %d' % opt_vecindades)
plt.show()


# ii) Obtén el coeficiente s para el mismo sistema A usando ahora el algoritmo DBSCAN con la
# métrica 'euclidean' y luego con 'manhattan'. En este caso, el parámetro que debemos explorar
# es el umbral de distancia eps \in (0.1, 0.4), fijando el número de elementos mínimo en n0 = 10.
# Comparad gráficamente con el resultado del apartado anterior.

# Definimos la siguiente función para abstraer el procedimiento realizado y para
# no tener copiar código con distintas distancias.
def clustering_DBSCAN(n0, eps_values, distance):
    """
    Función que dado una lista de valores posibles de épsilon realiza los
    siguientes pasos:
```

```
        - Para cada épsilon calcula su coeficiente de silhouette y el
        número de clusters correspondiente y muestra en una gráfica el
        valor de Silhouette en función de la épsilon.
        - Para que se pueda apreciar mejor como se comporta el coeficiente de
        silhouette con respecto al número de clusters, podemos coger
        para cada cluster el mejor coeficiente de silhouette obtenido.

    Args:
        n0 (integer): Número de elementos mínimo.
        eps_values (list): Valores posibles para el umbral de distancia.
        distance(string): Métrica utilizada en el algoritmo DBSCAN.
    Return:
        silhouette (list): Valores de Silhouette en función al número de cluster,
            se coge el máximo por cada número de vecindades posibles.
        n_clusters (list): Número de vecindades obtenidos con los valores
            de épsilon estudiados.
    """
    silhouette, n_clusters = list(), list()

    # Para cada epsilon calculamos su coeficiente de silhoutte y
    # el número de clusters correspondiente.
    for eps in eps_values:
        db = DBSCAN(eps=eps, min_samples=n0, metric=distance).fit(X)
        labels = db.labels_
        n_clusters.append(len(set(labels)) - (1 if -1 in labels else 0))
        silhouette.append(metrics.silhouette_score(X, labels))

    # Mostramos en una gráfica el valor de Silhouette en función de
    # la épsilon
    plot_X_to_Y(X=eps_values,
                Y=silhouette,
                xlabel='eps',
                ylabel='s',
                title='Valor de silhouette en función del valor de '
                'epsilon con el algoritmo DBSCAN y la métrica ' + distance)

    # Para que se pueda apreciar mejor como se comporta el coeficiente de
    # silhouette con respecto al número de clusters, podemos coger
    # para cada cluster el mejor coeficiente de silhouette obtenido.

    dict = {k : -sys.maxsize for k in sorted(n_clusters)}

    for i in range(len(silhouette)):
        dict[n_clusters[i]] = max(dict[n_clusters[i]], silhouette[i])

    return list(dict.keys()), list(dict.values())

# Fijamos el número de elementos mínimo en n0 = 10 y los valores de eps \in (0.1,0.4)
n0, eps_values = 10, np.arange(0.1, 0.4, 0.001)

# Llamamos a la función de arriba con la distancia euclidiana
n_clusters_db_euclidean, silhouette_db_euclidean = \
    clustering_DBSCAN(n0, eps_values, 'euclidean')

# Llamamos a la función de arriba con la distancia Manhattan
n_clusters_db_manhattan, silhouette_db_manhattan = \
    clustering_DBSCAN(n0, eps_values, 'manhattan')

# Comparamos los 3 resultados obtenidos: KMeans, DBSCAN (euclidean), DBSCAN (manhattan)
plt.figure(figsize=(8,6))
plt.plot(n_clusters_kmeans, silhouette_kmeans, color='r', label='KMeans')
plt.plot(n_clusters_db_euclidean, silhouette_db_euclidean, color='g', label='DBSCAN(euclidean)')
plt.plot(n_clusters_db_manhattan, silhouette_db_manhattan, color='b', label='DBSCAN(manhattan)')
```

```
plt.xlabel('n_clusters')
plt.ylabel('s')
plt.legend()
plt.title('Comparación gráfica de los algoritmos con los coeficientes de silhouette')

# iii) ¿De qué grado diríamos que son las personas con coordenadas a:=(0,0) y b:=(0,-1)?

# Vemos cuantos grados hay en el archivo "Grados_en_la_facultad_de_matematicas.txt"
n_grades = max(Y[:,0])

# Predecimos los clusters al que pertenecen
prediction = kmeans.predict([[0,0], [0,-1]])
cluster_a, cluster_b = prediction[0], prediction[1]

# Para cada persona ver cuál es el grado predominante en el cluster al que pertenecen
# que determinará el grado al que pertenece esa persona.
clusters_Y = kmeans.predict(Y[:,1:3])
counters_of_students_per_grade_in_cluster_a = [
    list([(Y[i][0], clusters_Y[i]) for i in range(len(Y))]).count((grade, cluster_a))
    for grade in range(int(n_grades) + 1)
    ]
print("El grado al que debería pertenecer la persona a:=(0,0) es:",
      np.argmax(counters_of_students_per_grade_in_cluster_a))
counters_of_students_per_grade_in_cluster_b = [
    list([(Y[i][0], clusters_Y[i]) for i in range(len(Y))]).count((grade, cluster_b))
    for grade in range(int(n_grades) + 1)
    ]
print("El grado al que debería pertenecer la persona b:=(0,-1) es:",
      np.argmax(counters_of_students_per_grade_in_cluster_b))
```