PRÁCTICA 3: DIAGRAMA DE VORONÓI Y CLUSTERING

Autor *Martín Fernández de Diego*
Compañera de código *Belén Sánchez Centeno*

1. INTRODUCCIÓN

A partir de un sistema X de 1000 elementos construido como muestra aleatoria entorno a unos centros definidos, se pide determinar una clasificación por vecindades —o clusters— a través de tres algoritmos distintos: KMeans, DBSCAN con métrica euclidiana y DBSCAN con métrica manhattan. También se pide representar el diagrama de Voronói correspondiente, entre otros.

2. MATERIAL USADO

2.1. Apartado i)

El **clustering por KMeans** toma k centroides arbitrarios y clasifica cada punto de cierto sistema X de forma que pertenezca al cluster asociado al centroide más cercano. Después, toma el punto medio de cada cluster como centroide de la siguiente iteración. El algoritmo cicla hasta que la distancia entre los nuevos puntos medios y los centroides de la anterior iteración sea inferior a cierto épsilon.

Se han utilizado las funciones:

- `plot_silhouette_kmeans(sistema)`
- `plot_clusters_kmeans(sistema)`

Para encontrar el **coeficiente de Silhouette**[1] con el algoritmo KMeans, en la primera función, el sistema X es entrenado para $k \in \{2, 3, ..., 15\}$ vecindades con la instrucción `KMeans(num_vecindades,estado_aleatorio)`. A continuación, se calcula el valor con la librería `sklearn` y la función `silhouette_score(sistema,vecindades)` que recibe X y las vecindades devueltas por KMeans. Por último, se muestra una **primera gráfica** con los coeficientes —eje de ordenadas— correspondientes a cada número de vecindades —eje de abscisas— y se decide computacionalmente cuál es el número de vecindades que corresponde al mayor coeficiente de Silhouette.

La segunda función es la encargada de, dado el mejor k estimado por Silhouette, recalcular KMeans y, con ayuda de la plantilla ofrecida por el profesor *Robert Monjo*, **representar los clusters** por colores sobre una **segunda gráfica**.

[1]El coeficiente de Silhouette es un indicador de ayuda a la decisión para saber cuán de buena ha sido una clasificación.

Adicionalmente, se añade la posición y nombre de los centros de los clusters para calcular y superponer el **teselado de Voronói**[2] con `voronoi = Voronoi(centros)` y `voronoi_plot_2d(voronoi)`.

2.2. Apartado ii)

El **clustering por DBSCAN** toma un punto arbitrario del sistema X y lo clasifica en función del número de puntos que haya en la ϵ-bola centrada en él respecto a n_0. Si es mayor, es un centro, se crea una clase y se marcan sus vecinos —como miembros— para ser visitados. Si es menor, se clasifica como ruido a la espera de, quizás, revisitarse. Antes de volver a tomar un punto arbitrario, se realizan las visitas pendientes y se marcan como puntos fronterizos si el número de puntos en su ϵ-bola es inferior a n_0 o, de nuevo centros, si es superior. Termina cuando no se pueden crear nuevas clases.

Se han utilizado las funciones:

- `plot_silhouette_dbscan(sistema,métrica)`
- `plot_clusters_dbscan(sistema,métrica)`

De manera semianáloga al apartado anterior, se halla el **coeficiente de Silhouette**, esta vez con el algoritmo DBSCAN, en la primera función. El sistema X es entrenado para un umbral $\epsilon \in (0{,}1, 0{,}4)$ con la instrucción `DBSCAN(epsilon,mínimo,métrica)`. A continuación, se obtiene el coeficiente de la misma forma que antes. Por último, se muestra una **primera gráfica** con los coeficientes —eje de ordenadas— correspondientes a cada valor de umbral —eje de abscisas— y se decide computacionalmente cuál es el umbral correspondiente al mayor coeficiente de Silhouette.

La segunda función, volviendo a hacer uso del material cedido en las plantillas, muestra una **segunda gráfica** con la **separación por clusters** en colores que arroja DBSCAN. Los puntos negros son los puntos de ruido del algoritmo.

2.3. Apartado iii)

A partir del cálculo de los centros de las vecindades y para cierta definición de distancia, se puede estimar a qué cluster pertenece un punto dado.

Se han utilizado las funciones:

[2]El diagrama de Voronói de un conjunto de marcas es la división del espacio en regiones de forma que a cada punto se le asigna la región formada por los puntos que son más cercanos a ella que a ninguna otra marca.

- `distancia_euclidea(punto,centro)`
- `distancia_manhattan(punto,centro)`

Se calcula la distancia —euclidiana o manhattan— del punto a cada centroide del sistema. El centroide más cercano identificará el cluster al que pertenece el nuevo punto.

Dados dos puntos del plano $P = (p1, p2)$ y $Q = (q1, q2)$, la fórmula de la **distancia euclidiana** implementada es

$$d_e(P, Q) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2}$$

y la fórmula de la **distancia manhattan** implementada es

$$d_m(P, Q) = |q_1 - p_1| + |q_2 - p_2|.$$

3. RESULTADOS

3.1. Apartado i)

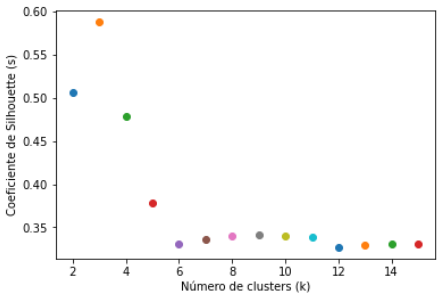

Fig. 1. Coeficientes de Silhouette por KMeans

El **clustering por KMeans** decide una clasificación en 3 clusters, en base al coeficiente de Silhouette más alto.

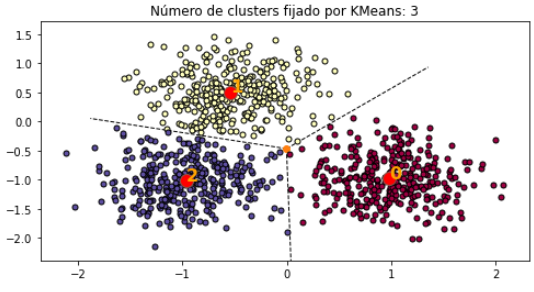

Fig. 2. Clusters por KMeans con teselado de Voronoi

3.2. Apartado ii)

3.2.1. *Métrica euclidiana*

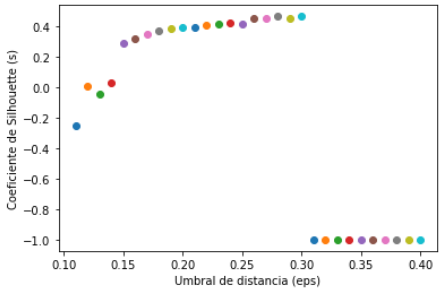

Fig. 3. Coeficientes de Silhouette por DBSCAN

El **clustering por DBSCAN** decide un umbral de 0.28 que se traduce en 2 clusters.

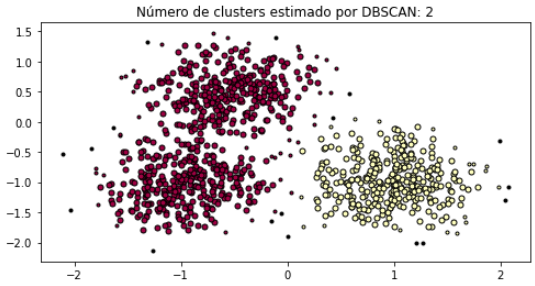

Fig. 4. Clusters por DBSCAN

3.2.2. *Métrica manhattan*

Por restricciones de formato y dada su similaridad con la euclidiana, no se mostrará el caso de la métrica manhattan.

3.3. Apartado iii)

El punto $(0, 0)$ se encuentra en el cluster 1 y el punto $(0, -1)$ en el cluster 2 según las distancias euclídea y manhattan. La comprobación corrobora este resultado.

4. CONCLUSIÓN

Los algoritmos de clusterización han arrojado resultados diferentes para un mismo sistema X. El funcionamiento de cada algoritmo está directamente relacionado. Mientras que KMeans categoriza en base al teselado de Voronói una vez encontrados unos centroides suficientemente buenos, DBSCAN expande la región de una categoría sin importar su centroide mientras haya suficiente densidad de puntos.

Dado que el coeficiente de Silhouette debe interpretarse como ayuda a la decisión, se podría tomar un umbral menor a 0,15 para diferenciar más de 2 clusters con DBSCAN.

5. ANEXO CON EL SCRIPT Y CÓDIGO UTILIZADO

5.1. Código

```
"""
PRÁCTICA 3: DIAGRAMA DE VORONOI Y CLUSTERING
Belén Sánchez Centeno
Martín Fernández de Diego
"""

import numpy as np
import matplotlib.pyplot as plt

from sklearn.cluster import KMeans
from sklearn.cluster import DBSCAN
from sklearn import metrics
from sklearn.datasets import make_blobs
from scipy.spatial import ConvexHull, convex_hull_plot_2d
from scipy.spatial import Voronoi, voronoi_plot_2d

"""
Dado un conjunto de puntos X
muestra la gráfica de los coeficientes de Silhouette para cada número de clusters
devuelve el número óptimo de clusters asociado al mayor coeficiente de Silhouette
"""
def plot_silhouette_kmeans(X):
    # Mostramos los coeficientes de Silhouette para cada k
    # y obtenemos el k asociado al mayor coeficiente de todos
    max_s = -1
    for k in range(2,16):
        kmeans = KMeans(n_clusters=k, random_state=0).fit(X)
        labels = kmeans.labels_
        silhouette = metrics.silhouette_score(X, labels)
        # Decidimos computacinalmente el número óptimo de clusters
        if max_s < silhouette:
            max_s = silhouette
            max_k = k
        plt.plot(k, silhouette, 'o')
    plt.xlabel("Número de clusters (k)")
    plt.ylabel("Coeficiente de Silhouette (s)")
    plt.show()

    return max_k

"""
Dado un conjunto de puntos X y el número de vecindades óptimo n_clusters
muestra los clusters estimados por el algoritmo KMeans
devuelve la instancia de KMeans para el apartado iii)
"""
def plot_clusters_kmeans(X,n_clusters):
    # Tomamos el número de vecindades óptimo devuelto por el coeficiente de Silhouette
    # y volvemos a ejecutar KMeans para mostrar los clusters por colores
    kmeans = KMeans(n_clusters=n_clusters, random_state=0).fit(X)
    labels = kmeans.labels_
    centers = kmeans.cluster_centers_
    # Mantenemos la misma proporción al resto en el tamaño de la gráfica
    fig = plt.figure(figsize=(8,4))
    ax = fig.add_subplot(111)

    # Mostramos el teselado de Voronoi
    vor = Voronoi(centers)
    voronoi_plot_2d(vor,ax=ax)

    # 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))]
    for k, col in zip(unique_labels, colors):
        if k == -1:
            # Black used for noise.
            col = [0, 0, 0, 1]
```

```
67          class_member_mask = (labels == k)
68          xy = X[class_member_mask]
69          plt.plot(xy[:, 0], xy[:, 1], 'o', markerfacecolor=tuple(col), markeredgecolor='k
    ', markersize=5)
70      # Mostramos los centros
71      plt.plot(centers[:,0],centers[:,1],'o', markersize=12, markerfacecolor="red")
72      for i in range(len(centers)):
73          plt.text(centers[i,0],centers[i,1],str(i),color='orange',fontsize=16,fontweight=
    'black')
74      # Configuramos los atributos de la gráfica con sus límites
75      plt.title('Número de clusters fijado por KMeans: %d' % n_clusters)
76      plt.xlim([np.min(X[:,0])-0.25,np.max(X[:,0])+0.25])
77      plt.ylim([np.min(X[:,1])-0.25,np.max(X[:,1])+0.25])
78      plt.show()
79
80      return kmeans
81
82  """
83  Dado un conjunto de puntos X, el tipo de métrica y la sensibilidad de búsqueda del
        umbral de distancia
84  muestra la gráfica de los coeficientes de Silhouette para cada umbral de distancia
85  devuelve el umbral de distancia óptimo asociado al mayor coeficiente de Silhouette
86  """
87  def plot_silhouette_dbscan(X,metric,step=0.01):
88      # Mostramos los coeficientes de Silhouette para cada épsilon
89      # y obtenemos el épsilon asociado al mayor coeficiente de todos
90      max_s = -1
91      for epsilon in np.arange(0.11,0.4,step):
92          # Utilizamos el algoritmo de DBSCAN para mínimo 10 elementos
93          db = DBSCAN(eps=epsilon, min_samples=10, metric=metric).fit(X)
94          labels = db.labels_
95          # Aseguramos el valor de Silhouette si el número de clusters es 1
96          n_clusters_ = len(set(labels)) - (1 if -1 in labels else 0)
97          silhouette = metrics.silhouette_score(X, labels) if n_clusters_ != 1 else -1
98          # Decidimos computacinalmente el número óptimo de clusters
99          if max_s < silhouette:
100             max_s = silhouette
101             max_eps = epsilon
102         plt.plot(epsilon, silhouette, 'o')
103     plt.xlabel("Umbral de distancia (eps)")
104     plt.ylabel("Coeficiente de Silhouette (s)")
105     plt.show()
106
107     return max_eps
108
109 """
110 Dado un conjunto de puntos X, el tipo de métrica y el umbral de distancia
111 muestra los clusters estimados por el algoritmo DBSCAN con la métrica dada
112 """
113 def plot_clusters_dbscan(X,metric,epsilon):
114     # Tomamos el épsilon óptimo devuelto por el coeficiente de Silhouette
115     # y volvemos a ejecutar DBSCAN para mostrar los clusters por colores
116     db = DBSCAN(eps=epsilon, min_samples=10, metric=metric).fit(X)
117     core_samples_mask = np.zeros_like(db.labels_, dtype=bool)
118     core_samples_mask[db.core_sample_indices_] = True
119     labels = db.labels_
120     # Number of clusters in labels, ignoring noise if present.
121     n_clusters_ = len(set(labels)) - (1 if -1 in labels else 0)
122     n_noise_ = list(labels).count(-1)
123
124     print("Número óptimo de vecindades: ", n_clusters_)
125
126     unique_labels = set(labels)
127     colors = [plt.cm.Spectral(each) for each in np.linspace(0, 1, len(unique_labels))]
128     plt.figure(figsize=(8,4))
129     for k, col in zip(unique_labels, colors):
130         # Black used for noise.
131         if k == -1:
132             col = [0, 0, 0, 1]
133         class_member_mask = (labels == k)
```

```
134             xy = X[class_member_mask & core_samples_mask]
135             plt.plot(xy[:, 0], xy[:, 1], 'o', markerfacecolor=tuple(col), markeredgecolor='k
        ', markersize=5)
136             xy = X[class_member_mask & ~core_samples_mask]
137             plt.plot(xy[:, 0], xy[:, 1], 'o', markerfacecolor=tuple(col), markeredgecolor='k
        ', markersize=3)
138         plt.title('Número de clusters estimado por DBSCAN: %d' % n_clusters_)
139         plt.show()
140
141     def distancia_euclidea(punto,centro):
142         return np.sqrt((punto[0]-centro[0])**2 + (punto[1]-centro[1])**2)
143
144     def distancia_manhattan(punto,centro):
145         return abs(punto[0]-centro[0]) + abs(punto[1]-centro[1])
146
147     # FORMATO
148     class Formato:
149         BOLD = "\033[1m"
150         RESET = "\033[0m"
151
152     # Aquí tenemos definido el sistema X de 1000 elementos de dos estados
153     # construido a partir de una muestra aleatoria entorno a unos centros:
154     centers = [[-0.5, 0.5], [-1, -1], [1, -1]]
155     X, labels_true = make_blobs(n_samples=1000, centers=centers, cluster_std=0.4,
            random_state=0)
156     #Si quisieramos estandarizar los valores del sistema, haríamos:
157     #from sklearn.preprocessing import StandardScaler
158     #X = StandardScaler().fit_transform(X)
159
160     #Envolvente convexa, envoltura convexa o cápsula convexa
161     hull = ConvexHull(X)
162     convex_hull_plot_2d(hull)
163
164     plt.plot(X[:,0],X[:,1],'ro', markersize=1)
165     plt.show()
166
167     # APARTADO i)
168     print("\n" + Formato.BOLD + "Apartado i)" + Formato.RESET)
169
170     max_k = plot_silhouette_kmeans(X)
171     print("Número óptimo de vecindades: ", max_k)
172     kmeans = plot_clusters_kmeans(X,max_k)
173
174     # APARTADO ii)
175     print("\n" + Formato.BOLD + "Apartado ii)" + Formato.RESET)
176
177     # Euclidean
178     euclidean_metric = 'euclidean'
179     euclidean_max_eps = plot_silhouette_dbscan(X,euclidean_metric)
180     print("Umbral de distancia euclidiana óptimo: ",euclidean_max_eps)
181     plot_clusters_dbscan(X,euclidean_metric,euclidean_max_eps)
182
183     # Manhattan
184     manhattan_metric = 'manhattan'
185     manhattan_max_eps = plot_silhouette_dbscan(X,manhattan_metric)
186     print("Umbral de distancia manhattan óptimo: ",manhattan_max_eps)
187     plot_clusters_dbscan(X,manhattan_metric,manhattan_max_eps)
188
189     # APARTADO iii)
190     print("\n" + Formato.BOLD + "Apartado iii)" + Formato.RESET)
191
192     centers = kmeans.cluster_centers_
193
194     punto1 = [0,0]
195     distancias_euclideas = [distancia_euclidea(punto1,centers[i]) for i in range(len(centers
            ))]
196     cluster1 = distancias_euclideas.index(min(distancias_euclideas))
197     print("El punto ",punto1,"se encuentra en el cluster ",cluster1,"según la distancia
            euclídea")
198     distancias_manhattan = [distancia_manhattan(punto1,centers[i]) for i in range(len(
```

```
    centers))]
cluster1 = distancias_manhattan.index(min(distancias_manhattan))
print("El punto ",punto1,"se encuentra en el cluster ",cluster1,"según la distancia
    manhattan")
print("Comprobación: ",kmeans.predict([punto1])[0])

punto2 = [0,-1]
distancias_euclideas = [distancia_euclidea(punto2,centers[i]) for i in range(len(centers
    ))]
cluster2 = distancias_euclideas.index(min(distancias_euclideas))
print("El punto ",punto2,"se encuentra en el cluster ",cluster2,"según la distancia
    euclídea")
distancias_manhattan = [distancia_manhattan(punto2,centers[i]) for i in range(len(
    centers))]
cluster2 = distancias_manhattan.index(min(distancias_manhattan))
print("El punto ",punto2,"se encuentra en el cluster ",cluster2,"según la distancia
    manhattan")
print("Comprobación: ",kmeans.predict([punto2])[0])
```

5.2. Ejecución

```
Apartado i)
Número óptimo de vecindades:  3

Apartado ii)
Umbral de distancia euclidiana óptimo:  0.2799999999999999
Número óptimo de vecindades:  2
Umbral de distancia manhattan óptimo:  0.3599999999999999
Número óptimo de vecindades:  2

Apartado iii)
El punto  [0, 0] se encuentra en el cluster  1 según la distancia euclídea
El punto  [0, 0] se encuentra en el cluster  1 según la distancia manhattan
Comprobación:  1
El punto  [0, -1] se encuentra en el cluster  2 según la distancia euclídea
El punto  [0, -1] se encuentra en el cluster  2 según la distancia manhattan
Comprobación:  2
```