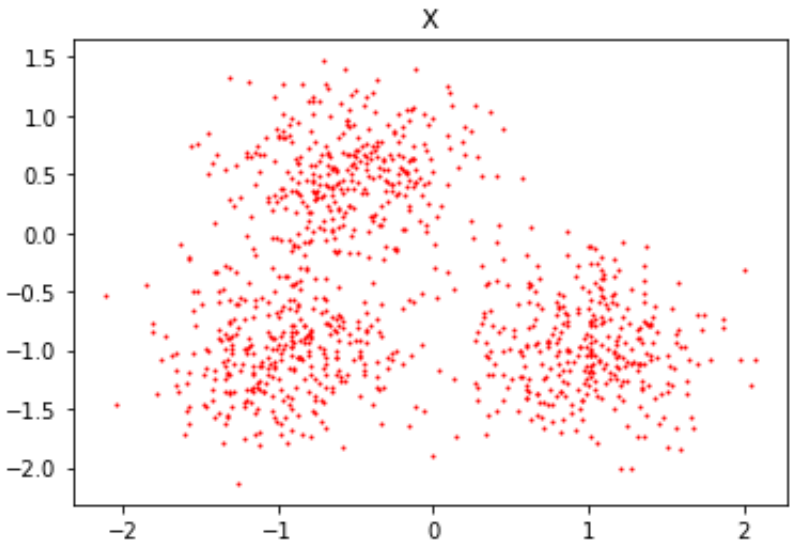
PRÁCTICA 3: Diagrama de Voronói y Clustering

Geometría Computacional

Belén Sánchez Centeno
belsan05@ucm.es

14 de marzo de 2022



Date Performed: 23 de febrero y 2 de marzo de 2022
Code Partner: Martín Fernández de Diego

1. Introducción

En esta práctica se pretende clasificar un sistema X de 1000 elementos (con dos estados cada uno) a partir de un determinado número de vecindades de Voronói. Para determinar el número óptimo de vecindades o clusters, la medida que emplearemos es el *coeficiente de Silhouette* ($\overline{s}$), que puede utilizarse directamente desde la librería `sklearn`.

Primer Objetivo

Obtener el coeficiente $\overline{s}$ de X para diferente número de vecindades $k \in \{2, 3, ..., 15\}$ utilizando el algoritmo KMeans. Mostrar en una gráfica el valor de $\overline{s}$ en función de k y decidir con ello cuál es el número óptimo de vecindades. En una segunda gráfica, mostrar la clasificación (clusters) resultante con diferentes colores y representar el diagrama de Voronói en esa misma gráfica.

Segundo Objetivo

Obtener el coeficiente $\overline{s}$ para el mismo sistema X 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 $n_0 = 10$. Comparar gráficamente con el resultado del apartado anterior.

Tercer Objetivo

¿A qué vecindad pertenecen los elementos con coordenadas $a := (0, 0)$ y $b := (0, -1)$? Comprobar la respuesta con la función `kmeans.predict`.

2. Material utilizado

Se resuelve el problema con un script de Python, en el que se usan funciones de implementación propia, de las plantillas proporcionadas y de las siguientes bibliotecas: `matplotlib.pyplot`, `numpy`, `sklearn` y `scipy`. Quedan detalladas a continuación:

- `plot_silhouette_kmeans`, que, dado un conjunto X de puntos, muestra en una gráfica el *coeficiente de Silhouette* ($\overline{s}$) en función del número vecindades, entre 2 y 15. Para ello se utiliza iterativamente el algoritmo KMeans ya implementado en `sklearn.cluster`, con `random_state=0`, y la función `silhouette_score` de `sklearn.metrics`. Simultáneamente se comparan los coeficientes con el máximo, para decidir así cuál es el número óptimo de vecindades de forma automática, empezando con un máximo igual a -1, que es el mínimo valor posible. La función devuelve el óptimo calculado.

- `plot_clusters_kmeans`, que, dado un conjunto X de puntos y el número de vecindades óptimo calculado por la función anterior, muestra gráficamente los clusters. En este caso se ejecuta el algoritmo KMeans una sola vez y, a partir del resultado obtenido, se colorean los puntos de cada clúster de un mismo color, se marca el centro (indicando además el número de etiqueta del clúster asignado por el algoritmo) y se representa el diagrama de Voronói, con las funciones `Voronoi` y `voronoi_plot_2d` facilitadas por `scipy.spatial`. Todas estas representaciones se superponen, y es necesario ajustar los límites de la gráfica resultante para que se muestren todos los puntos de X.

- `plot_silhouette_dbscan`, análoga a `plot_silhouette_kmeans` con el algoritmo alternativo a KMeans DBSCAN, también implementado en `sklearn.cluster`. Además de X, se recibe como argumento la métrica que se va a utilizar para clasificar los puntos. En este caso la gráfica del *coeficiente de Silhouette* ($\overline{s}$) a mostrar es en función del umbral de distancia (épsilon), entre 0,1 y 0,4 y en nuestro caso con un paso de 0,01 por iteración. También se devuelve el épsilon óptimo calculado.

- `plot_clusters_dbscan`, otra versión de `plot_clusters_kmeans` con el algoritmo DBSCAN pero que únicamente muestra la representación de los clústers coloreados.

- `distancia_euclidea`, que, dados dos puntos (en este caso un punto cualquiera y el centro de uno de los clusters), devuelve la distancia euclídea que los separa, es decir:

$$d_e((x_1, x_2), (y_1, y_2)) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2} \tag{1}$$

- `distancia_manhattan`, análoga a la anterior con la distancia manhattan:

$$d_m((x_1, x_2), (y_1, y_2)) = (x_1 - y_1) + (x_2 - y_2) \tag{2}$$

Los datos iniciales, es decir, la inicialización del conjunto X de 1000 elementos repartidos por regiones de mayor y menor densidad a estudiar, se toma de las plantillas "GCOM2022-practica3_plantilla1" y "GCOM2022-practica3_plantilla2" proporcionadas, en las que se indican los centros $(-0,5, 0,5)$, $(-1, -1)$ y $(1, -1)$. Adicionalmente, se utiliza la función `kmeans.predict` para comprobar qué vecindad habría asignado a un punto concreto el algoritmo KMeans, tras haberlo calculado previamente con `distancia_euclidea` y `distancia_manhattan`.

3. Resultados

3.1. Primer objetivo

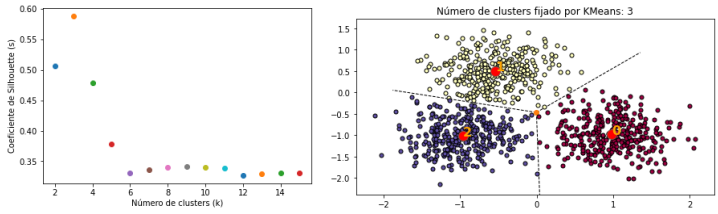


Figura 1: Resultado obtenido con el algoritmo KMeans:
Número óptimo de vecindades: 3

3.2. Segundo objetivo

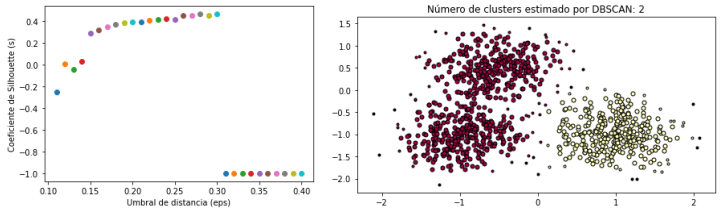


Figura 2: Resultado obtenido con el algoritmo DBSCAN y la distancia euclidiana:
Umbral de distancia euclidiana óptimo: 0.2799999999999999
Número óptimo de vecindades: 2

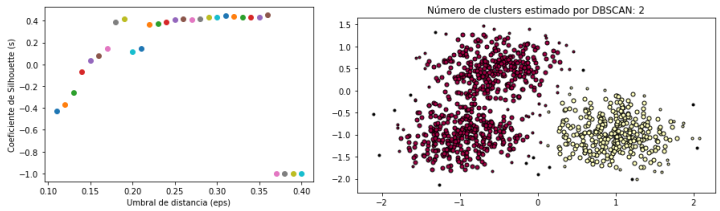


Figura 3: Resultado obtenido con el algoritmo DBSCAN y la distancia manhattan:
Umbral de distancia manhattan óptimo: 0.3599999999999999
Número óptimo de vecindades: 2

3.3. Tercer objetivo

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

4. Conclusión

Comparando los resultados con la clusterización que parece más evidente a simple vista, es más eficaz el algoritmo KMeans, que toma centros arbitrarios (tantos como se le indiquen), clasifica los puntos de X como vecindades de esos centros y los recalcula a otros más óptimos hasta que apenas haya variación, obteniendo así 3 clústers en su configuración óptima. Por otro lado, DBSCAN, que toma vecindades en un umbral de distancia dado por el épsilon y se va expandiendo, obtiene solamente 2 clústers en su configuración más óptima y, disminuyendo épsilon para forzar a que encuentre 3, el resultado es una clusterización mucho menos equilibrada que la de KMeans.

5. Anexo: Código utilizado

```
"""
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]
        class_member_mask = (labels == k)
        xy = X[class_member_mask]
        plt.plot(xy[:, 0], xy[:, 1], 'o', markerfacecolor=tuple(col),
            markeredgecolor='k', markersize=5)
    # Mostramos los centros
    plt.plot(centers[:,0],centers[:,1],'o', markersize=12,
        markerfacecolor="red")
    for i in range(len(centers)):
        plt.text(centers[i,0],centers[i,1],str(i),color='orange',
            fontsize=16,fontweight='black')
    # Configuramos los atributos de la gráfica con sus límites
    plt.title('Número de clusters fijado por KMeans: %d' % n_clusters)
    plt.xlim([np.min(X[:,0])-0.25,np.max(X[:,0])+0.25])
    plt.ylim([np.min(X[:,1])-0.25,np.max(X[:,1])+0.25])
    plt.show()

    return kmeans

"""
Dado un conjunto de puntos X, el tipo de métrica y la sensibilidad de
    búsqueda del umbral de distancia
muestra la gráfica de los coeficientes de Silhouette para cada umbral
    de distancia
devuelve el umbral de distancia óptimo asociado al mayor coeficiente de
     Silhouette
"""
def plot_silhouette_dbscan(X,metric,step=0.01):
    # Mostramos los coeficientes de Silhouette para cada épsilon
    # y obtenemos el épsilon asociado al mayor coeficiente de todos
    max_s = -1
    for epsilon in np.arange(0.11,0.4,step):
        # Utilizamos el algoritmo de DBSCAN para mínimo 10 elementos
        db = DBSCAN(eps=epsilon, min_samples=10, metric=metric).fit(X)
        labels = db.labels_
        # Aseguramos el valor de Silhouette si el número de clusters es
            1
        n_clusters_ = len(set(labels)) - (1 if -1 in labels else 0)
        silhouette = metrics.silhouette_score(X, labels) if n_clusters_
            != 1 else -1
        # Decidimos computacinalmente el número óptimo de clusters
        if max_s < silhouette:
```

```
            max_s = silhouette
            max_eps = epsilon
        plt.plot(epsilon, silhouette, 'o')
    plt.xlabel("Umbral de distancia (eps)")
    plt.ylabel("Coeficiente de Silhouette (s)")
    plt.show()

    return max_eps

"""
Dado un conjunto de puntos X, el tipo de métrica y el umbral de
    distancia
muestra los clusters estimados por el algoritmo DBSCAN con la métrica
    dada
"""
def plot_clusters_dbscan(X, metric, epsilon):
    # Tomamos el épsilon óptimo devuelto por el coeficiente de
        Silhouette
    # y volvemos a ejecutar DBSCAN para mostrar los clusters por
        colores
    db = DBSCAN(eps=epsilon, min_samples=10, metric=metric).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)

    print("Número óptimo de vecindades: ", n_clusters_)

    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):
        # Black used for noise.
        if k == -1:
            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.title('Número de clusters estimado por DBSCAN: %d' %
        n_clusters_)
    plt.show()

def distancia_euclidea(punto,centro):
    return np.sqrt((punto[0]-centro[0])**2 + (punto[1]-centro[1])**2)

def distancia_manhattan(punto,centro):
    return abs(punto[0]-centro[0]) + abs(punto[1]-centro[1])

# FORMATO
class Formato:
    BOLD = "\033[1m"
```

```
    RESET = "\033[0m"

# Aquí tenemos definido el sistema X de 1000 elementos de dos estados
# construido a partir de una muestra aleatoria entorno a unos centros:
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)
#Si quisieramos estandarizar los valores del sistema, haríamos:
#from sklearn.preprocessing import StandardScaler
#X = StandardScaler().fit_transform(X)

#Envolvente convexa, envoltura convexa o cápsula convexa
hull = ConvexHull(X)
convex_hull_plot_2d(hull)

plt.plot(X[:,0],X[:,1], 'ro', markersize=1)
plt.show()

# APARTADO i)
print("\n" + Formato.BOLD + "Apartado i)" + Formato.RESET)

max_k = plot_silhouette_kmeans(X)
print("Número óptimo de vecindades: ", max_k)
kmeans = plot_clusters_kmeans(X,max_k)

# APARTADO ii)
print("\n" + Formato.BOLD + "Apartado ii)" + Formato.RESET)

# Euclidean
euclidean_metric = 'euclidean'
euclidean_max_eps = plot_silhouette_dbscan(X,euclidean_metric)
print("Umbral de distancia euclidiana óptimo: ",euclidean_max_eps)
plot_clusters_dbscan(X,euclidean_metric,euclidean_max_eps)

# Manhattan
manhattan_metric = 'manhattan'
manhattan_max_eps = plot_silhouette_dbscan(X,manhattan_metric)
print("Umbral de distancia manhattan óptimo: ",manhattan_max_eps)
plot_clusters_dbscan(X,manhattan_metric,manhattan_max_eps)

# APARTADO iii)
print("\n" + Formato.BOLD + "Apartado iii)" + Formato.RESET)

centers = kmeans.cluster_centers_

punto1 = [0,0]
distancias_euclideas = [distancia_euclidea(punto1,centers[i]) for i in
    range(len(centers))]
cluster1 = distancias_euclideas.index(min(distancias_euclideas))
print("El punto ",punto1," se encuentra en el cluster ",cluster1," según
    la distancia euclídea")
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])
```