

1. Introducción

En esta práctica he utilizado los algoritmos de clasificación *KMeans* y *DBSCAN* sobre un sistema X de 1000 elementos con dos estados cada uno. Para ello he utilizado las implementaciones de estos algoritmos de la librería `scikit-learn`.

2. Método

El programa está dividido en funciones que aislan las distintas funcionalidades del mismo y permiten la reutilización y variación de parámetros del código:

- `kmeans_silhouettes` y `dbscan_silhouettes`: Dado un sistema y un conjunto de parámetros (n para *KMeans* y ϵ para *DBSCAN*) calcula las vecindades correspondientes, con sus valores medios de los coeficientes de *Silhouette*.
- `kmeans_elegir_n_clusters` y `dbscan_elegir_epsilon`: Devuelve el índice del valor máximo de los coeficientes de *Silhouette* calculados anteriormente. Además también dibuja la gráfica de los distintos valores de los coeficientes de *Silhouette* al variar el parámetro del algoritmo, destacando el rojo el valor máximo.
- `dbscan_cluster_centroids`: Calcula los centroides de las vecindades calculadas con *DBSCAN*, útiles para pintar las etiquetas en las gráficas.
- `plot_clusters` y `plot_voronoi`: Gráfica de las vecindades y diagrama de *Voronoi*.
- `apartado1` y `apartado2`: Gestión de los plots y llamadas a las funciones anteriores.

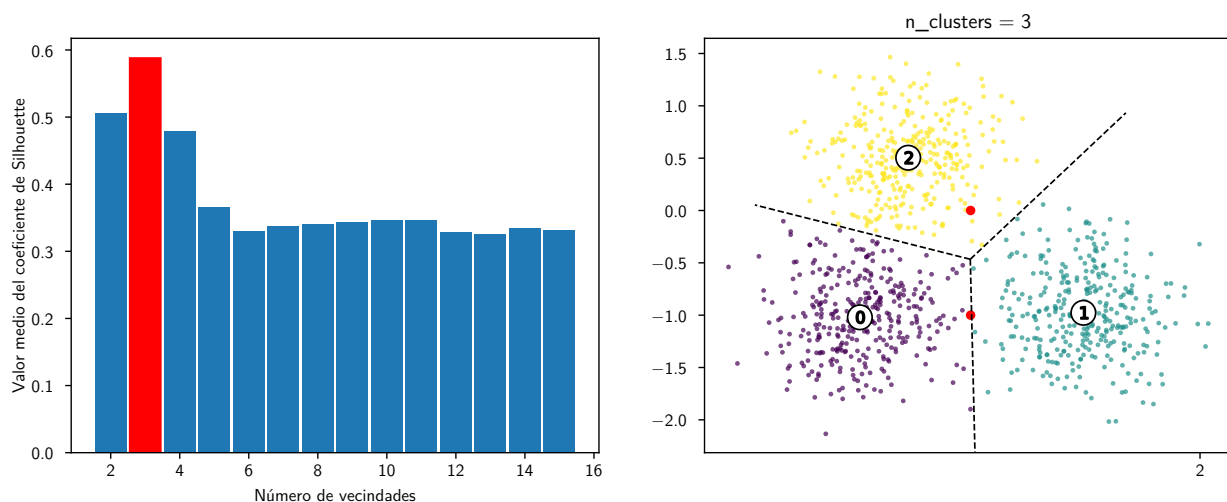
3. Resultados

3.1. Clasificación con algoritmo KMeans y predicción de nuevos estados

El número de vecindades con mayor coeficiente de *Silhouette* para el algoritmo *KMeans* de entre los valores $[2, 15]$ es el 3, como se puede ver en la figura de la izquierda. Además es interesante observar que no hay ningún otro valor que proporcione un coeficiente de *Silhouette* cercano al correspondiente a 3 vecindades. Por otro lado este es precisamente el valor esperado, ya que los estados del sistema han sido generados entorno a tres centros.

En la gráfica de la derecha se pueden observar las vecindades y diagrama correspondientes a este número óptimo de vecindades. Aquí también he dibujado en rojo los elementos $a = (0, 0)$ y $b = (0, -1)$ que queremos clasificar. Con esto he podido comparar el resultado esperado visualmente con el obtenido con el método `kmeans.predict`.

- El punto a está visiblemente en la vecindad 2, lo que coincide con el resultado de `kmeans.predict`
- El punto b está visiblemente en la frontera entre las regiones de voronoi correspondientes a las vecindades 0 y 1. En este caso el método `kmeans.predict` nos dice que el elemento pertenece a la vecindad 0, lo que se corresponde con lo esperado, pero quizás es una información un tanto imprecisa, ya que solo con este dato no sabemos que el punto está muy cerca de la vecindad 1.



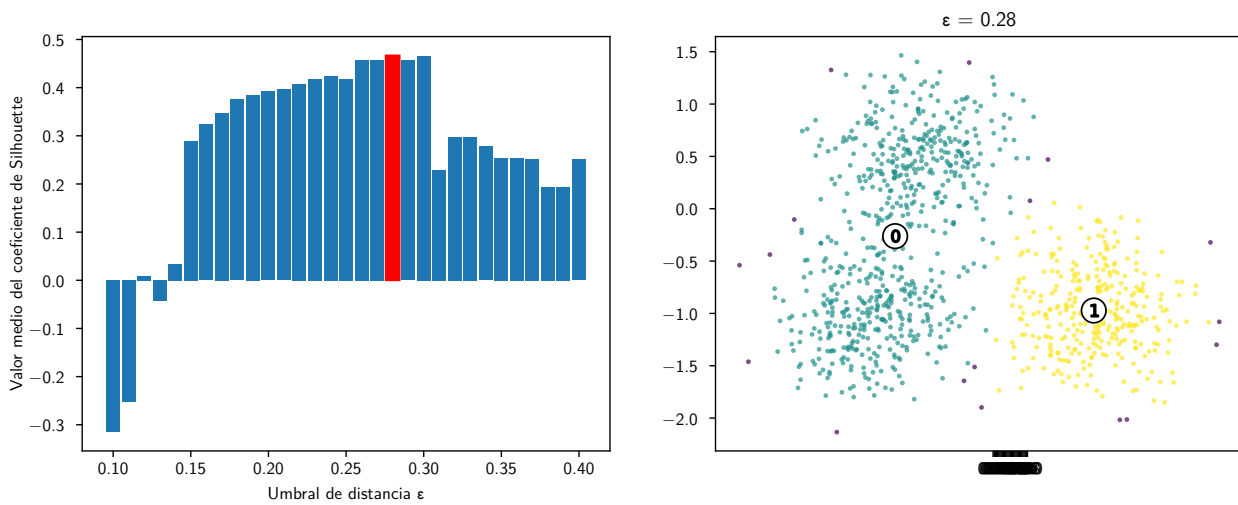
3.2. Clasificación con algoritmo DBSCAN

Al aplicar el algoritmo *DBSCAN* he fijado el número mínimo de elementos en $n_0 = 10$ y he considerado los coeficientes de *Silhouette* correspondientes a los valores del umbral de distancia $\varepsilon \in (0.1, 0.4)$, tanto para la métrica euclídeana como para la de Manhattan.

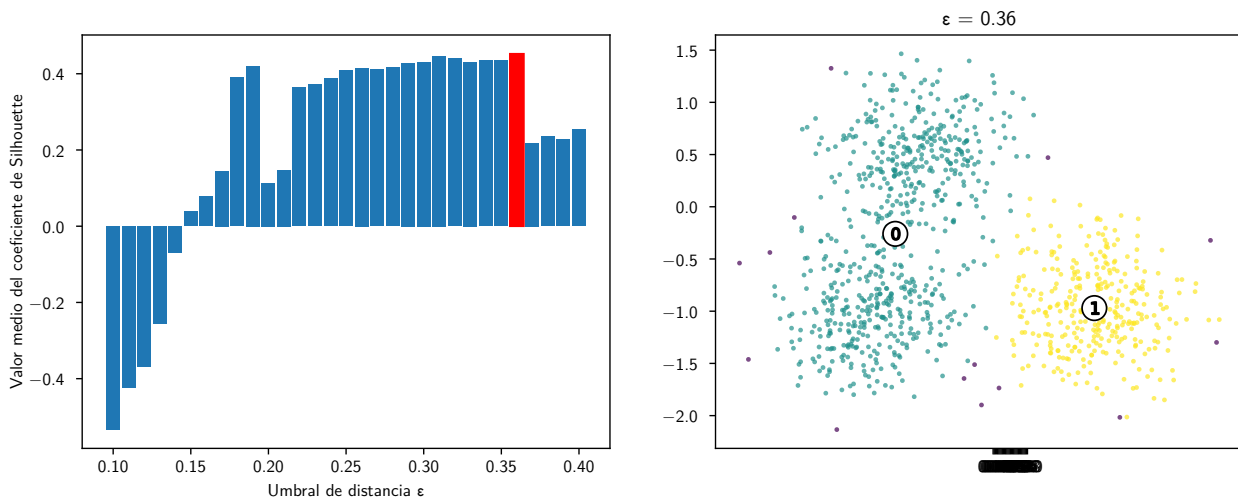
En las gráficas de la izquierda se puede observar que ahora los valores del umbral de distancia con coeficiente de *Silhouette* máximo no están tan diferenciados como en el caso del algoritmo *KMeans*. Es decir, se intuye que existe un entorno del valor óptimo ε en el que el coeficiente de *Silhouette* tiene una variación acotada.

Además, para ambas métricas, los coeficientes de *Silhouette* máximos son considerablemente menores que en el caso anterior. Esto se refleja también en las gráficas de la derecha, en las que están dibujadas las vecindades. En este caso se han distinguido solo dos vecindades, estando agrupadas las vecindades 0 y 2 del apartado anterior en una sola. Con esto se entiende que el coeficiente medio de *Silhouette* sea menor, ya que la vecindad 0 ahora es más grande y sus puntos están "más separados entre sí".

3.2.1. DBSCAN con métrica euclídeana



3.2.2. DBSCAN con métrica de Manhattan



4. Conclusión

Mi conclusión es que el algoritmo *KMeans* tiene una mayor capacidad para distinguir vecindades de estados que están muy cercanas, pero que se acumulan entorno a puntos concretos, mientras que *DBSCAN* no consigue hacer estas distinciones debido a que está considerando los valores de la bola de radio igual al umbral de distancia.

Además no he notado grandes diferencias entre usar una métrica u otra en el algoritmo *DBSCAN*.

5. Código

El siguiente código con la implementación también está adjunto en la entrega y disponible, junto con esta memoria, en un repositorio git en el siguiente enlace: github.com/haztecaso/gcomp22.

```
8 import matplotlib.pyplot as plt
9 import numpy as np
10 from scipy.spatial import Voronoi, voronoi_plot_2d, qhull
11 from sklearn.cluster import KMeans, DBSCAN
12 from sklearn.datasets import make_blobs
13 from sklearn.metrics import silhouette_score
14 from timeit import default_timer as timer
15
16
17 def kmeans_silhouettes(X, ns):
18     silhouettes = []
19     clusters = []
20     for n_clusters in ns:
21         kmeans = KMeans(n_clusters=n_clusters, random_state=0).fit(X)
22         labels = kmeans.labels_
23         silhouettes.append(silhouette_score(X, labels))
24         clusters.append(kmeans)
25     return silhouettes, clusters
26
27
28 def dbscan_silhouettes(X, εs, metric):
29     silhouettes = []
30     clusters = []
31     for ε in εs:
32         db = DBSCAN(eps=ε, min_samples=10, metric=metric).fit(X)
33         labels = db.labels_
34         if len(set(labels)) > 1:
35             silhouettes.append(silhouette_score(X, labels))
36             clusters.append(db)
37     return silhouettes, clusters
38
39
40 def kmeans_elegir_n_clusters(ns, silhouettes, ax = None):
41     silhouettes_max_index = silhouettes.index(max(silhouettes))
42     if ax:
43         ax.set_xlabel("Número de vecindades")
44         ax.set_ylabel("Valor medio del coeficiente de Silhouette")
45         bars = ax.bar(ns, silhouettes, width=(ns[1]-ns[0])*0.9)
46         bars[silhouettes_max_index].set_color('r')
47         plt.xticks(ns)
48     return silhouettes_max_index
49
50
51 def dbscan_elegir_ε(εs, silhouettes, metric, ax = None):
52     silhouettes_max_index = silhouettes.index(max(silhouettes))
53     if ax:
54         ax.set_xlabel("Umbral de distancia ε")
55         ax.set_ylabel("Valor medio del coeficiente de Silhouette")
56         bars = ax.bar(εs, silhouettes, width=(εs[1]-εs[0])*0.9)
57         bars[silhouettes_max_index].set_color('r')
58         plt.xticks(εs)
59     return silhouettes_max_index
60
61
62 def dbscan_cluster_centroids(X, clusters):
63     labels = set(clusters.labels_)
64     core_samples_mask = np.zeros_like(clusters.labels_, dtype=bool)
65     core_samples_mask[clusters.core_sample_indices_] = True
66     return np.asarray([np.mean(X[(clusters.labels_==l)], axis=0) for l in labels if not l == -1])
67
68
69 def plot_clusters(X, clusters, centers, ax = None):
70     if ax is None:
71         _, ax = plt.subplots()
72     labels = clusters.labels_
73     ax.scatter(
74         X[:, 0],
75         X[:, 1],
76         marker=".",
77         s=30,
78         lw=0,
79         alpha=0.7,
80         c=labels.astype(float),
81         edgecolor="k"
82     )
83
84     # Etiquetas de los clusters
85     ax.scatter(
86         centers[:, 0],
87         centers[:, 1],
88         marker="o",
89         c="white",
90         s=200,
91         edgecolor="k",
```

```

92 )
93 for i, c in enumerate(centers):
94     ax.scatter(c[0], c[1], marker=f"${i}$", alpha=1, s=50, edgecolor="k")
95
96 def plot_voronoi(clusters, centers, ax = None):
97     if ax is None:
98         _, ax = plt.subplots()
99
100     lims = (ax.get_xlim(), ax.get_ylim()) # Hack para que voronoi_plot_2d no cambie la escala de la gráfica.
101     try:
102         # Diagrama de voronoi construido a partir de los centros de las vecindades
103         vor = Voronoi(centers)
104         voronoi_plot_2d(vor, show_vertices = False, point_size=0, ax=ax)
105
106         ax.set_xlim(*lims[0]); ax.set_ylim(*lims[1]) # Hack para que voronoi_plot_2d no cambie la escala de la gráfica.
107     except qhull.QhullError:
108         print("ERROR: No se ha podido crear el diagrama de Voronoi, quizás debido a se hayan proporcionado menos de 3 vecindades.")
109
110
111 def apartado1(X):
112     fig, (ax1, ax2) = plt.subplots(1,2)
113     fig.suptitle(f"KMeans", fontweight="bold")
114     # Posibles valores de vecindades
115     ns = range(2,16)
116
117     t1 = timer()
118     # Cálculo del coeficiente de silhouette de cada n
119     silhouettes, kmeans_clusters_list = kmeans_silhouettes(X, ns)
120     t2 = timer()
121
122     # Selecciono el valor de n con mayor coeficiente de silhouette
123     n_clusters_index = kmeans_elegir_n_clusters(ns, silhouettes, ax1)
124     kmeans_clusters = kmeans_clusters_list[n_clusters_index]
125     t3 = timer()
126
127     # Gráfica de la clasificación, con colores y líneas de separación del diagrama de Voronoi
128     centers = kmeans_clusters.cluster_centers_
129     ax2.set_title(f"n_clusters = {ns[n_clusters_index]}")
130     plot_clusters(X, kmeans_clusters, centers, ax2) # Vecindades
131     t4 = timer()
132     plot_voronoi(kmeans_clusters, centers, ax2) # Diagrama de Voronoi
133     t5 = timer()
134
135     # Apartado 3
136     test_data = [[0,0], [0, -1]]
137     print("Apartado 3:")
138     for x, label in zip(test_data, kmeans_clusters.predict(test_data)):
139         ax2.scatter(x[0], x[1], alpha=1, s=20, color="r")
140         print(f"-- El punto {x} pertenece a la vecindad {label}")
141     print()
142     t6 = timer()
143
144     # Tiempos de ejecución
145     print(f"-----Tiempos KMeans")
146
147     Cálculo de los coeficientes de silhouette: {t2-t1}
148     Seleccionando el valor de n: {t3-t2}
149     Gráfica de la clasificación y diagrama de Voronoi: {t4-t3}
150     Gráfica del diagrama de Voronoi: {t5-t4}
151     Predicción de 3 estados nuevos: {t6-t5}
152     TOTAL: {t6-t1}
153     """)
154
155
156 def apartado2(X, metric):
157     fig, (ax1, ax2) = plt.subplots(1,2)
158     fig.suptitle(f"DBSCAN (métrica {metric})", fontweight="bold")
159     # Posibles umbrales de distancia ε
160     es = np.linspace(0.1, 0.4, 31)
161
162     t1 = timer()
163     # Cálculo del coeficiente de silhouette de cada ε
164     silhouettes, dbscan_clusters_list = dbscan_silhouettes(X, es, metric)
165     t2 = timer()
166
167     # Selecciono el valor de ε con mayor coeficiente de silhouette
168     ε_index = dbscan_elegir_ε(es, silhouettes, metric, ax1)
169     dbscan_clusters = dbscan_clusters_list[ε_index]
170     t3 = timer()
171
172     centers = dbscan_cluster_centroids(X, dbscan_clusters)
173
174     # Gráfica de la clasificación, con colores y líneas de separación del diagrama de Voronoi
175     ax2.set_title(f"ε = {es[ε_index]:.2f}")
176     centers = dbscan_cluster_centroids(X, dbscan_clusters)
177     plot_clusters(X, dbscan_clusters, centers, ax2)
178     t4 = timer()
179
180     # Tiempos de ejecución
181     print(f"-----Tiempos DBSCAN (métrica {metric})")
182
183

```

```

184 Cálculo de los coeficientes de silhouette: {t2-t1}
185 Seleccionando el valor de  $\epsilon$ : {t3-t2}
186 Gráfica de la clasificación: {t4-t3}
187 TOTAL: {t4-t1}
188 """
189
190
191
192 def main():
193     # Datos
194     centers = [[-0.5, 0.5], [-1, -1], [1, -1]]
195     X, _ = make_blobs(n_samples=1000, centers=centers, cluster_std=0.4, random_state=0)
196
197     apartado1(X)
198     apartado2(X, 'euclidean')
199     apartado2(X, 'manhattan')
200
201     plt.show()
202
203
204 if __name__ == "__main__":
205     main()

```