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 y datos

Como datos he utilizado el sistema X proporcionado en la plantilla:

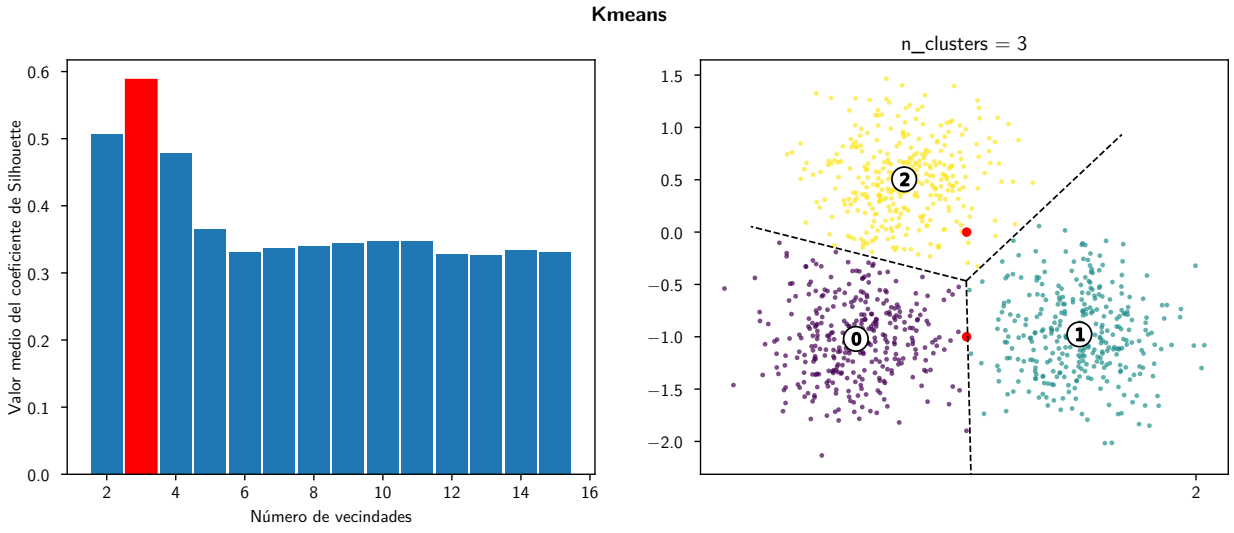
```
193     # Datos
194     centers = [[-0.5, 0.5], [-1, -1], [1, -1]]
```

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

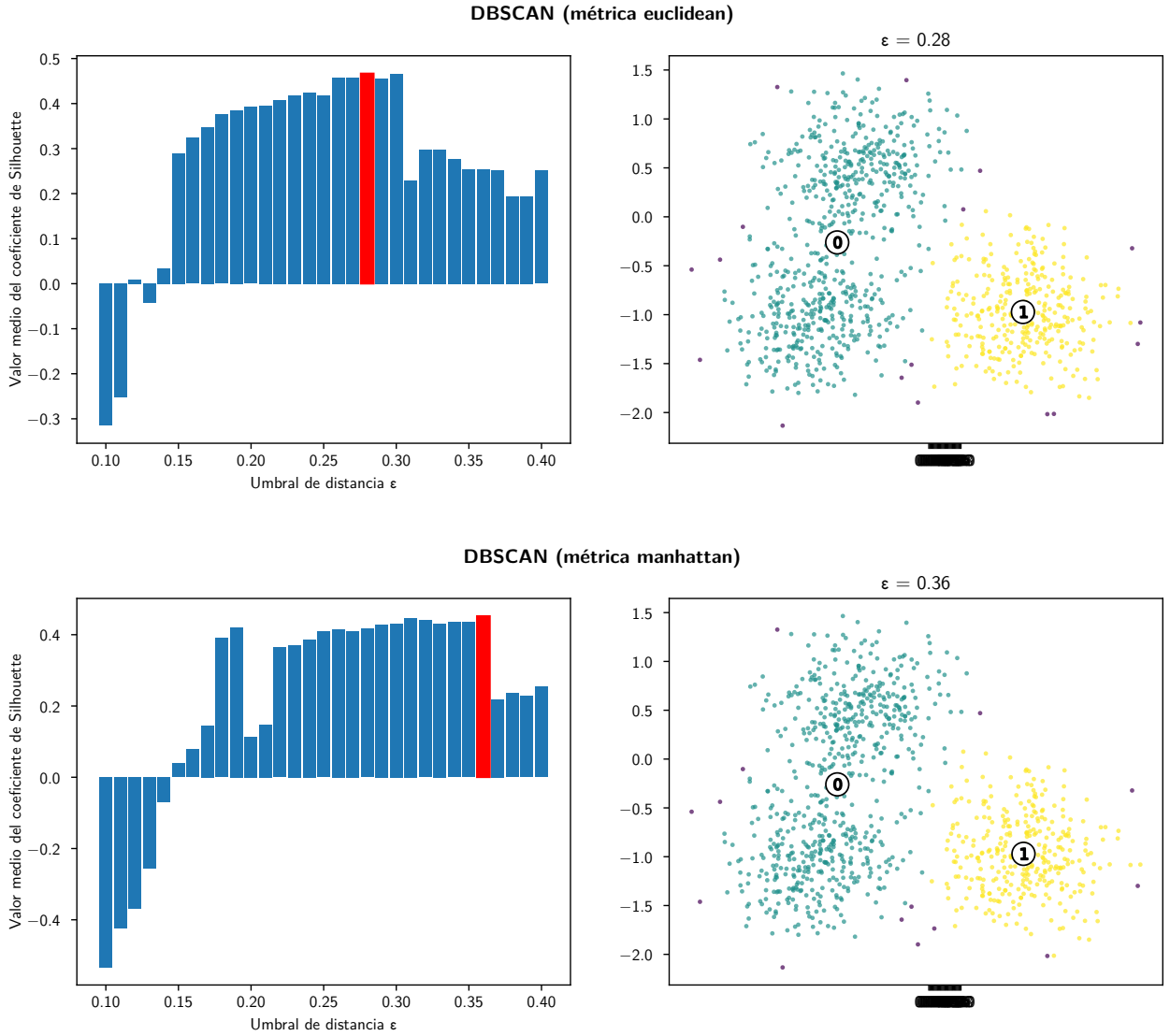
- `kmeans_silhouettes` y `dbscan_silhouettes`: Dado un sistema y un conjunto de parámetros (n para *KMeans* y ϵ para *DBSCAN*) calcula las vecindades correspondientes, con sus valores medios de los coeficientes de Silhouette.
- `kmeans_elegir_n_clusters` y `dbscan_elegir_ε`: Devuelve el índice del valor 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`: Gráfica de las vecindades.
- `plot_voronoi`: Gráfica del 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



3.2. Clasificación con algoritmo DBSCAN



4. 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.

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.spatial import Voronoi, voronoi_plot_2d, qhull
from sklearn.cluster import KMeans, DBSCAN
from sklearn.datasets import make_blobs
from sklearn.metrics import silhouette_score
from timeit import default_timer as timer


def kmeans_silhouettes(X, ns):
    silhouettes = []
    clusters = []
    for n_clusters in ns:
        kmeans = KMeans(n_clusters=n_clusters, random_state=0).fit(X)
        labels = kmeans.labels_
        silhouettes.append(silhouette_score(X, labels))
        clusters.append(kmeans)
    return silhouettes, clusters


def dbscan_silhouettes(X, εs, metric):
    silhouettes = []
    clusters = []
    for ε in εs:
```

```
        db = DBSCAN(eps=ε, min_samples=10, metric=metric).fit(X)
        labels = db.labels_
        if len(set(labels)) > 1:
            silhouettes.append(silhouette_score(X, labels))
            clusters.append(db)
    return silhouettes, clusters


def kmeans_elegir_n_clusters(ns, silhouettes, ax = None):
    silhouettes_max_index = silhouettes.index(max(silhouettes))
    if ax:
        ax.set_xlabel("Número de vecindades")
        ax.set_ylabel("Valor medio del coeficiente de Silhouette")
        bars = ax.bar(ns, silhouettes, width=(ns[1]-ns[0])*.9)
        bars[silhouettes_max_index].set_color('r')
        plt.xticks(ns)
    return silhouettes_max_index


def dbscan_elegir_ε(εs, silhouettes, metric, ax = None):
    silhouettes_max_index = silhouettes.index(max(silhouettes))
    if ax:
        ax.set_xlabel("Umbral de distancia ε")
        ax.set_ylabel("Valor medio del coeficiente de Silhouette")
        bars = ax.bar(εs, silhouettes, width=(εs[1]-εs[0])*.9)
        bars[silhouettes_max_index].set_color('r')
        plt.xticks(εs)
    return silhouettes_max_index


def dbscan_cluster_centroids(X, clusters):
    labels = set(clusters.labels_)
    core_samples_mask = np.zeros_like(clusters.labels_, dtype=bool)
    core_samples_mask[clusters.core_sample_indices_] = True
    return np.asarray([np.mean(X[(clusters.labels_==l)], axis=0) for l in labels if not l == -1])


def plot_clusters(X, clusters, centers, ax = None):
    if ax is None:
        _, ax = plt.subplots()
    labels = clusters.labels_
    ax.scatter(
            X[:, 0],
            X[:, 1],
            marker=".",
            s=30,
            lw=0,
            alpha=0.7,
            c=labels.astype(float),
            edgecolor="k"
    )

    # Etiquetas de los clusters
    ax.scatter(
        centers[:, 0],
        centers[:, 1],
        marker="o",
        c="white",
        s=200,
        edgecolor="k",
    )
    for i, c in enumerate(centers):
        ax.scatter(c[0], c[1], marker=f"${i}$", alpha=1, s=50, edgecolor="k")

def plot_voronoi(clusters, centers, ax = None):
    if ax is None:
        _, ax = plt.subplots()

    lims = (ax.get_xlim(), ax.get_ylim()) # Hack para que voronoi_plot_2d no cambie la escala de la gráfica.
    try:
        # Diagrama de voronoi construido a partir de los centros de las vecindades
        vor = Voronoi(centers)
        voronoi_plot_2d(vor, show_vertices = False, point_size=0, ax=ax)

        ax.set_xlim(*lims[0]); ax.set_ylim(*lims[1]) # Hack para que voronoi_plot_2d no cambie la escala de la gráfica.
    except qhull.QhullError:
        print("ERROR: No se ha podido crear el diagrama de Voronoi, quizás debido a se hayan proporcionado menos de 3 vecindades.


def apartado1(X):
    fig, (ax1, ax2) = plt.subplots(1,2)
    fig.suptitle(f"Kmeans", fontweight="bold")
    # Posibles valores de vecindades
    ns = range(2,16)

    t1 = timer()
    # Cálculo del coeficiente de silhouette de cada n
    silhouettes, kmeans_clusters_list = kmeans_silhouettes(X, ns)
    t2 = timer()

    # Selecciono el valor de n con mayor coeficiente de silhouette
```

```
    n_clusters_index = kmeans_elegir_n_clusters(ns, silhouettes, ax1)
    kmeans_clusters = kmeans_clusters_list[n_clusters_index]
    t3 = timer()

    # Gráfica de la clasificación, con colores y líneas de separación del diagrama de Voronoi
    centers = kmeans_clusters.cluster_centers_
    ax2.set_title(f"n_clusters = {ns[n_clusters_index]}")
    plot_clusters(X, kmeans_clusters, centers, ax2) # Vecindades
    t4 = timer()
    plot_voronoi(kmeans_clusters, centers, ax2)     # Diagrama de Voronoi
    t5 = timer()

    # Apartado 3
    test_data = [[0,0], [0, -1]]
    print("Apartado 3:")
    for x, label in zip(test_data, kmeans_clusters.predict(test_data)):
        ax2.scatter(x[0], x[1], alpha=1, s=20, color="r")
        print(f"- El punto {x} pertenece a la vecindad {label}")
    print()
    t6 = timer()

    # Tiempos de ejecución
    print(f"""Tiempos KMeans
--------------
Cálculo de los coeficientes de silhouette:          {t2-t1}
Seleccionando el valor de n:                        {t3-t2}
Gráfica de la clasificación y diagrama de Voronoi:  {t4-t3}
Gráfica del diagrama de  Voronoi:                   {t5-t4}
Predicción de 3 estados nuevos:                     {t6-t5}
TOTAL:                                              {t6-t1}
""")


def apartado2(X, metric):
    fig, (ax1, ax2) = plt.subplots(1,2)
    fig.suptitle(f"DBSCAN (métrica {metric})", fontweight="bold")
    # Posibles umbrales de distancia ε
    εs = np.linspace(0.1, 0.4, 31)

    t1 = timer()
    # Cálculo del coeficiente de silhouette de cada ε
    silhouettes, dbscan_clusters_list = dbscan_silhouettes(X, εs, metric)
    t2 = timer()


    # Selecciono el valor de ε con mayor coeficiente de silhouette
    ε_index = dbscan_elegir_ε(εs, silhouettes, metric, ax1)
    dbscan_clusters = dbscan_clusters_list[ε_index]
    t3 = timer()

    centers = dbscan_cluster_centroids(X, dbscan_clusters)

    # Gráfica de la clasificación, con colores y líneas de separación del diagrama de Voronoi
    ax2.set_title(f"ε = {εs[ε_index]:.2f}")
    centers = dbscan_cluster_centroids(X, dbscan_clusters)
    plot_clusters(X, dbscan_clusters, centers, ax2)
    t4 = timer()

    # Tiempos de ejecución
    print(f"""Tiempos DBSCAN (métrica {metric})
---------------------------------
Cálculo de los coeficientes de silhouette:  {t2-t1}
Seleccionando el valor de ε:                {t3-t2}
Gráfica de la clasificación:                {t4-t3}
TOTAL:                                      {t4-t1}
""")



def main():
    # Datos
    centers = [[-0.5, 0.5], [-1, -1], [1, -1]]
    X, _ = make_blobs(n_samples=1000, centers=centers, cluster_std=0.4, random_state=0)

    apartado1(X)
    apartado2(X, 'euclidean')
    apartado2(X, 'manhattan')

    plt.show()


if __name__ == "__main__":
    main()
```