Apéndice A

Código de Python

En este apéndice se detalla la implementación en Python del algoritmo de refinamiento de conjuntos de datos propuesto en este trabajo. El objetivo es implementar un algoritmo que refine conjuntos de datos eliminando ruido y puntos atípicos, basándose en principios de Análisis Topológico de Datos (TDA) para preservar la estructura intrínseca de los datos. Vamos a hacer un resumen del código que hemos usado en Python para filtrar nuestros conjuntos de datos.

A.1. Librerías Importantes

Definimos las librerías más relevantes para ejecutar el algoritmo y visualizar sus resultados

```
import numpy as np # Computacion numerica
import matplotlib.pyplot as plt # Crear graficos y visualizaciones
from ripser import ripser # Calcular la homologia persistente de una nube de
    puntos
from persim import plot_diagrams # Creacion diagramas de persistencia
from persim import bottleneck # Calculo de distancias bottleneck
```

A.2. Generación del Conjunto de Datos de Prueba

A continuación, crearemos una nube de puntos siguiendo la forma de un toroide, al que posteriormente añadiremos los outliers. La construcción se basa en las ecuaciones paramétricas canónicas del toro, que describen la superficie como el resultado de hacer girar un círculo de radio menor r alrededor de un eje que se encuentra a una distancia R (radio mayor) de su centro.

```
def toro(n_puntos=500, ruido=0.2, n_outliers=10):

    # Definimos radios
    R, r = 2, 1

    # Creamos array de distribucion uniforme en [0,2*pi) para el angulo en el
    plano xy y el angulo de la seccion circular
    theta = 2 * np.pi * np.random.rand(n_puntos)
    phi = 2 * np.pi * np.random.rand(n_puntos)

    # Calculamos las coordenadas a traves de las ecuaciones parametricas y las
    unimos
```

```
    x = (R + r * np.cos(theta)) * np.cos(phi)
    y = (R + r * np.cos(theta)) * np.sin(phi)
    z = r * np.sin(theta)
    toro = np.vstack([x, y, z]).T

    # Generamos una matriz de ruido del mismo tamano que la matriz del toro
    ruido_gaussiano = np.random.normal(scale=ruido, size=toro.shape)

    # Sumamos el ruido a las coordenadas de cada punto, desplazandolo
    ligeramente de su posicion original en la superficie ideal.
    toro_ruidoso = toro + ruido_gaussiano

    # Creamos puntos con coordenadas distribuidas uniformemente dentro de un
    cubo definido por los limites [-6,6)
    outliers = np.random.uniform(low=-6, high=6, size=(n_outliers, 3))

# Obtenemos el toro con el ruido en distribucion normal mas los outliers
return np.vstack([toro_ruidoso, outliers])
```

A.3. Filtrado Topológico

Esta función implementa el algoritmo de filtrado topológica. Su propósito es refinar un conjunto de datos eliminando iterativamente puntos que tienen un impacto insignificante en su estructura homológica fundamental.

```
def filtrar_por_impacto_topologico(puntos, epsilon, iteraciones):

    # Creamos una copia de nuestro conjunto a filtrar
    S = puntos.copy()

    for iter in range(iteraciones):

        # Calculamos el diagrama de persistencia de dimension k=0,1,2
        D = ripser(S)['dgms'][k]

        # Creamos la lista donde iremos introduciendo la posicion de los puntos
        a eliminar
        eliminar = []

        for i in range(len(S)):

            # Creamos un array donde eliminamos el punto en la posicion i de S y
            calculamos su diagrama de persistencia de dimension k
            S_i = np.delete(S, i, axis=0)
            D_i = ripser(S_i)['dgms'][]

            # Calculamos la distancia bottleneck entre el diagrama de
            persistencia nuestro conjunto original y el del conjunto sin el
            punto en la posicion i
            d = bottleneck(D, D_i)

```

```
            # Si dicha distancia es menor que nuestro umbral, agregamos la
            posicion i a eliminar
            if d < epsilon:
                eliminar.append(i)
        # Si eliminar sigue vacia rompemos el bucle
        if not eliminar:
            break

        # Eliminamos de S los puntos en las posiciones guardadas en eliminar
        S = np.delete(S, eliminar, axis=0)
    return S
```

Para el caso del último tipo de filtrado presentado en nuestro trabajo, dado que hay que comparar respecto tanto de H_1 como de H_2, tenemos un algoritmo ligeramente distinto a los anteriores.

```
def filtrar_por_impacto_topologico(puntos, epsilon, epsilon2, iteraciones):

    # Creamos una copia de nuestro conjunto a filtrar
    S = puntos.copy()
    for _ in range(iteraciones):

        # Calculamos el diagrama de persistencia de dimension 1 y 2
        resultados = ripser(S, maxdim=2)
        D1 = resultados['dgms'][1]
        D2 = resultados['dgms'][2]

        # Creamos la lista donde iremos introduciendo la posicion de los puntos
        a eliminar
        eliminar = []

        for i in range(len(S)):

            # Creamos un array donde eliminamos el punto en la posicion i de S y
            calculamos sus diagramas de persistencia de dimension 1 y 2
            S_i = np.delete(S, i, axis=0)
            resultados_i = ripser(S_i, maxdim=2)
            D1_i = resultados_i['dgms'][1]
            D2_i = resultados_i['dgms'][2]

            # Calculamos las distancias bottleneck entre los diagramas de
            persistencia nuestro conjunto original y el del conjunto sin el
            punto en la posicion i
            d1 = bottleneck(D1, D1_i)
            d2 = bottleneck(D2, D2_i)

            # Si dichas distancias son menores que nuestros umbrales, agregamos
            la posicion i a eliminar
            if d1 < epsilon and d2 < epsilon2:
                eliminar.append(i)

        # Si eliminar sigue vacia rompemos el bucle
        if not eliminar:
            break
```

```
34
35          # Eliminamos de S los puntos en las posiciones guardadas en eliminar
36          S = np.delete(S, eliminar, axis=0)
37
38      return S
```

A.4. Análisis y Comparación de Resultados

La función *comparar persistencia* es una herramienta de validación y análisis diseñada para evaluar la efectividad del algoritmo de filtrado. Su objetivo principal es comparar la estructura topológica de dos nubes de puntos (típicamente, la original y la filtrada) y proporcionar una medida tanto cuantitativa como cualitativa de su similitud.

```
1  def comparar_persistencia(puntos1, puntos2, dim):
2
3      # Creamos los diagramas
4      dgms1 = ripser(puntos1, maxdim=2)['dgms']
5      dgms2 = ripser(puntos2, maxdim=2)['dgms']
6      dist = bottleneck(dgms1[dim], dgms2[dim])
7
8      # Representacion visual
9      plt.figure(figsize=(10,4))
10     plt.subplot(1,2,1)
11     plt.title('Diagrama persistencia original')
12     plt.scatter(dgms1[dim][:,0], dgms1[dim][:,1])
13     plt.plot([0, max(dgms1[dim][:,1])], [0, max(dgms1[dim][:,1])], 'k--')
14     plt.subplot(1,2,2)
15     plt.title('Diagrama persistencia filtrado')
16     plt.scatter(dgms2[dim][:,0], dgms2[dim][:,1])
17     plt.plot([0, max(dgms2[dim][:,1])], [0, max(dgms2[dim][:,1])], 'k--')
18     plt.show()
19
20     print(f"Distancia bottleneck dimension {dim}: {dist:.4f}")
```

A.5. Visualización de Resultados

La función main sirve como el punto de entrada principal para ejecutar el flujo de trabajo completo del experimento. Su rol es orquestar la secuencia de operaciones: desde la generación de los datos hasta la aplicación del algoritmo de filtrado y la posterior visualización y análisis de los resultados.

```
1  def main():
2      puntos = toro()
3      print(f"Puntos originales: {len(puntos)}")
4
5      # Filtramos el toro
6      puntos_filtrados = filtrar_por_impacto_topologico(puntos,0.02,100)
7      print(f"Puntos tras filtrado: {len(puntos_filtrados)}")
8
9      # Muestra nubes de puntos 3D
```

```
    fig = plt.figure(figsize=(12, 6))

    ax1 = fig.add_subplot(121, projection='3d')
    ax1.scatter(puntos[:,0], puntos[:,1], puntos[:,2], s=10)
    ax1.set_title("Conjunto original")

    ax2 = fig.add_subplot(122, projection='3d')
    ax2.scatter(puntos_filtrados[:,0], puntos_filtrados[:,1], puntos_filtrados
    [:,2], s=10, c="#ff7f0e")
    ax2.set_title("Conjunto filtrado")

    plt.show()

    # Calculamos diagramas de persistencia
    resultados = ripser(puntos, maxdim=2)
    diagramas = resultados['dgms']

    # Muestra diagramas de persistencia
    plot_diagrams(diagramas, show=True)

    # Muestra distancias bottleneck de cada dimension
    comparar_persistencia(puntos, puntos_filtrados, dim=0)
    comparar_persistencia(puntos, puntos_filtrados, dim=1)
    comparar_persistencia(puntos, puntos_filtrados, dim=2)

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