Informe Final - Procesamiento De Imágenes Médicas

Edinson Orlando Dorado Dorado

14 de junio de 2024

Resumen

Este es un trabajo que compendia lo visto en el curso Introducción al Procesamiento de Imágenes Médicas dictado por PhD. Jose Bernal en el primer semestre del 2024 en Universidad del Valle Colombia.

1. Introducción

En este documento se encontrará la explicación de lo implementado en mi repositorio de github creado para segmentar imágenes médicas. Se explicará desde la subida del archivo de imágenes que soporta, algoritmos de segmentación como thresholding, isodata, region growing, k-means; se explicará el filtro de la media y la mediana, 2 algoritmos de estandarización, y por último algoritmos de detección de bordes.

2. Consideraciones iniciales - archivos nifti

2.1. Cargar y entender archivos nifti

Las imágenes médicas en este caso son cargadas por medio de archivos nifti. En python se usa la libreria nibabel.

```
import nibabel as nib
file_path = ""
current_data = None
img_aux = nib.load(self.file_path)
data = img_aux.get_fdata()
```

```
[[[0. 0. 0. ... 0. 0. 0.]
  [0. 0. 0. ... 0. 0. 0.]
  [0. 0. 0. ... 0. 0. 0.]
  ...
  [0. 0. 0. ... 0. 0. 0.]
  [0. 0. 0. ... 0. 0. 0.]
  [0. 0. 0. ... 0. 0. 0.]]

 [[0. 0. 0. ... 0. 0. 0.]
  [0. 0. 0. ... 0. 0. 0.]
  [0. 0. 0. ... 0. 0. 0.]
  ...
  [0. 0. 0. ... 0. 0. 0.]
  [0. 0. 0. ... 0. 0. 0.]
  [0. 0. 0. ... 0. 0. 0.]]
```

Figura 1: Estructura de data

2.2. Visualizacion de las tres axis

Para visualizar data, se hace uso de la libreria de python matplotlib.

```
import matplotlib.pyplot as plt

# Create figure and axes
fig, axes = plt.subplots(1, 3, figsize=(15, 5))
```

```
# Initial slice index
initial_slice = 100

# Remove innnecesario info
axes[0].axis('off')
axes[1].axis('off')
axes[2].axis('off')

# Display initial slices
ax0_img = axes[0].imshow(data[:, :, initial_slice])
ax1_img = axes[1].imshow(data[:, initial_slice, :])
ax2_img = axes[2].imshow(data[initial_slice, :, :].T, origin='lower')
```

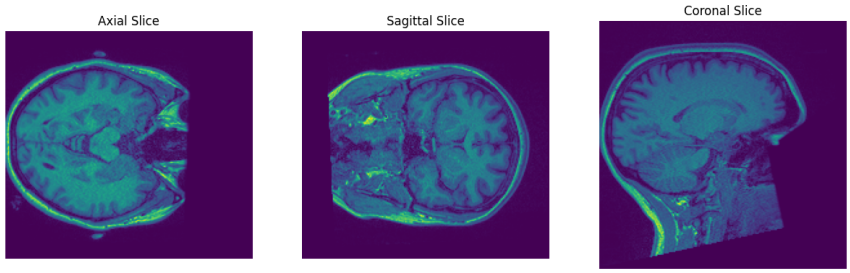

Figura 2: Visualización de un archivo nifti

3. Algoritmos de segmentación

Para segmentar los siguientes algoritmos, se hace uso de la intensidad que tiene cada voxel.

```
histogram = plt.hist(loadMedicalImage('file.nii')[loadMedicalImage('file.nii')>10],bins=750,color='blue',alpha=0.7)
```

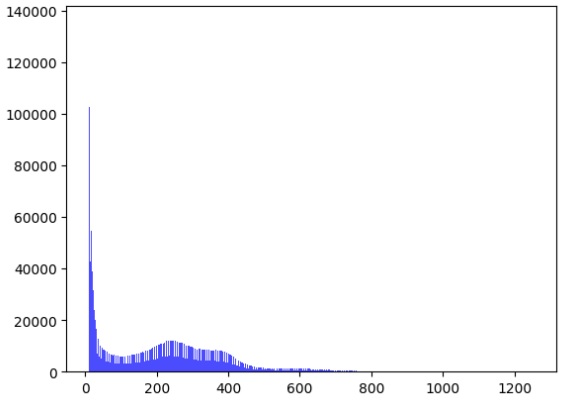

Figura 3: Histograma de las intensidades de data

3.1. Thresholding

Este algoritmo usa un valor entero para separar las intensidades en background y foreground.

```
def algoThresholding(inputData: np.ndarray, tau: int ) -> np.ndarray:
    data = inputData > tau
    return data
```

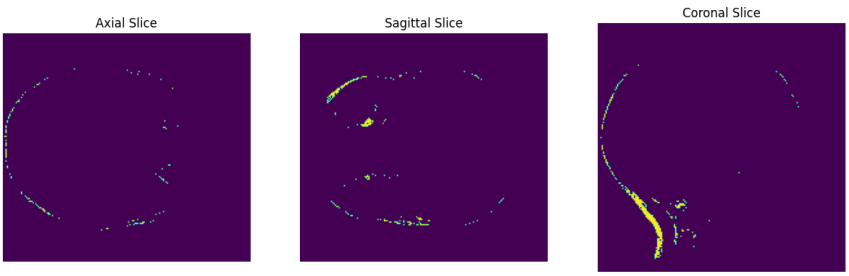

Figura 4: Resultado del algoritmo thresholding con un threshold de 650

3.1.1. Ventajas

Simplicidad: El algoritmo es simple y fácil de implementar. Solo requiere comparar los valores de los píxeles con un umbral y generar una imagen binaria.
Eficiencia: La operación es computacionalmente eficiente, ya que solo implica comparaciones y puede ser ejecutada rápidamente incluso en imágenes grandes.
Claridad en la Segmentación: Proporciona una segmentación clara y directa, diferenciando fácilmente entre regiones de interés y el fondo.

3.1.2. Desventajas

Sensibilidad al Umbral: La calidad de la segmentación depende críticamente del valor del umbral tau. Un valor mal elegido puede llevar a una segmentación incorrecta.
Dependencia de la Iluminación: Es sensible a variaciones en la iluminación y contraste de la imagen, lo que puede afectar negativamente el resultado.
Limitado a Imágenes con Contraste Claro: Funciona bien solo en imágenes donde hay un contraste claro entre las regiones de interés y el fondo. No es adecuado para imágenes complejas o con gradientes suaves.

3.2. Isodata

Este algoritmo encuentra un numero óptimo para hacer tresholding, hallando de forma iterativa el valor que hace que el promedio de las intensidades del background y foreground den una diferencia menor a una tolerancia dada.

```
def isodataAlgo(inputData: np.ndarray) -> np.ndarray:
    tau_init = 300
    t = 0
    tau_t = tau_init
    tolerancia = 0.001
    data = None
    while True:
        data = inputData > tau_t

        m_foreground = inputData[data == 1].mean()
        m_background = inputData[data == 0].mean()

        tau_new = 0.5 * (m_foreground + m_background)

        if abs(tau_new - tau_t) < tolerancia:
            break
        t = t + 1

        tau_t = tau_new

    return data
```

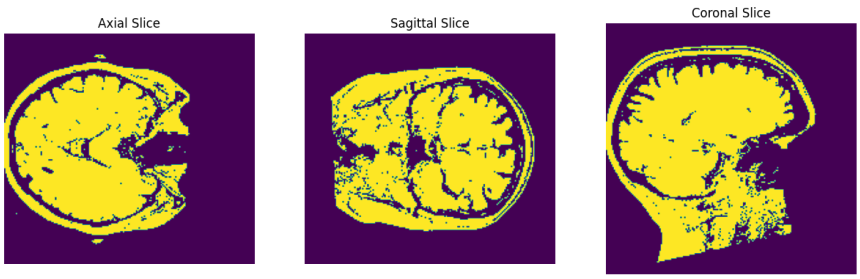

Figura 5: Resultado del algoritmo isodata

3.2.1. Ventajas

Adaptabilidad del Umbral: Ajusta automáticamente el umbral tau basado en las características de la imagen, lo que puede resultar en una segmentación más precisa.
Iterativo: Mejora continuamente el umbral hasta alcanzar la convergencia, asegurando una segmentación óptima.
Independencia del Umbral Inicial: Aunque comienza con un valor inicial de tau, ajusta este valor a lo largo de las iteraciones para reflejar mejor las diferencias entre el fondo y el primer plano.

3.2.2. Desventajas

Sensibilidad a la Variabilidad: Puede no funcionar bien si hay poca diferencia entre el fondo y el primer plano, o si hay mucha variabilidad dentro de una misma región.
Requiere Convergencia: El tiempo de ejecución puede ser largo si la convergencia no se alcanza rápidamente y si no se coloca un número máximo de iteraciones a realizar.
Dependencia de la Media: Asume que las medias de las intensidades del fondo y el primer plano son representativas, lo cual puede no ser cierto en todas las imágenes médicas.

3.3. Region Growing

Este algoritmo necesita un conjunto de etiquetas las cuales usa para a partir de ellas crecer a sus vecinos si la diferencia de intensidades entre ellos no supera un umbral establecido.

```
def algoRegionGrowing(inputData: np.ndarray, seeds: list, intensity_input: int=50) -> np.ndarray:
    # Ensure the data is a valid 3D array
    if len(inputData.shape) != 3:
        raise ValueError("Input data must be a 3D array")

    # Create a mask to keep track of visited voxels
    mask = np.zeros_like(inputData, dtype=np.uint8)

    # Define 8-connectivity for region growing
    connectivity = [
        (1, 0, 0), (0, 0, 1), (0, 1, 0),(-1, 0, 0), (0, -1, 0),  (0, 0, -1),
    ]

    for seed in seeds:

        intensity_difference = intensity_input

        mean_intensity = inputData[seed[0], seed[1], seed[2]]

        number_of_elements_mean_intensity = 0
        intensities_accumulator = 0

        queue = Queue()
        queue.put(seed)

        while not queue.empty():
```

```
            x, y, z = queue.get()
            # Check if the voxel is within the data bounds and hasn't been visited
            if (
                0 <= x < inputData.shape[0] and
                0 <= y < inputData.shape[1] and
                0 <= z < inputData.shape[2] and
                mask[x, y, z] == 0
            ):
                # Check the intensity difference with the seed
                if abs(inputData[x, y, z] - mean_intensity) <= intensity_difference:
                    # Mark the voxel as visited and assign the region label
                    mask[x, y, z] = 255

                    # upadte the mean_intensity
                    number_of_elements_mean_intensity += 1
                    intensities_accumulator += inputData[x, y, z]
                    mean_intensity = intensities_accumulator / number_of_elements_mean_intensity

                    # Enqueue neighboring voxels
                    for dx, dy, dz in connectivity:
                        queue.put((x + dx, y + dy, z + dz))
                else:
                    # this voxel wont take part in the segmentation
                    mask[x, y, z] = 175

    data = np.zeros_like(mask, dtype=np.uint8)
    data[mask == 255] = 255
    return data
```

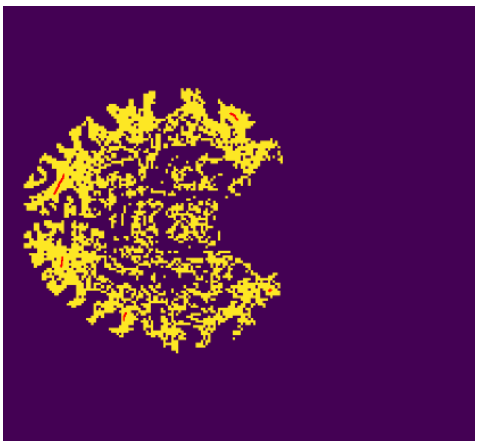

Figura 6: Resultado del algoritmo de region growing a a partir de unas anotaciones en color rojo

3.3.1. Ventajas

Adaptabilidad: El algoritmo puede adaptarse bien a las formas y tamaños de las regiones de interés, ya que se basa en la similitud de intensidad de los píxeles vecinos.
Precisión en Bordes: Es preciso en la detección de bordes y contornos, ya que crece desde semillas específicas y se detiene cuando se encuentra un cambio significativo en la intensidad.
Interactividad: Permite la selección interactiva de semillas, lo que puede mejorar la precisión de la segmentación en aplicaciones prácticas.

3.3.2. Desventajas

Sensibilidad al Ruido: Es susceptible al ruido en la imagen, lo que puede llevar a segmentaciones incorrectas si el ruido no se filtra adecuadamente.
Dependencia de Semillas: La elección de semillas adecuadas es crucial; una selección incorrecta puede resultar en segmentaciones pobres o incompletas.
Complejidad Computacional: Puede ser computacionalmente intensivo, especialmente en imágenes

3D grandes, debido a la necesidad de verificar muchos píxeles vecinos.
Variabilidad en Intensidad: Puede tener dificultades para segmentar regiones con variabilidad en la intensidad, lo que puede ser común en algunas imágenes médicas.

3.4. K-means

La idea principal de este algoritmo es clasificar cada voxel en k clusters. Siendo el criterio para agruparlos en cada cluster, la cercanía que tienen entre si con la intensidad del centroide del cluster. Lo que primero se hace es hallar todas las intensidades, y empezar asignando los centroides de forma aleatoria según ese array de intensidades. Después iterar para encontrar una mejor aproximación de cuales deberían de ser los centroides. Por último, asignar a cada voxel de la data original una intensidad igual para cada conjunto de clusters.

```
def algoKMeans(data: np.ndarray,k: int) -> np.ndarray:

    #Get the one-dimensional array of all intensities.
    oneDimensionArray = data.flatten()

    #Delete all duplicates
    oneDimensionSet = set(oneDimensionArray)

    #Convert to array again
    oneDimensionArray = list(oneDimensionSet)

    # Use np.random.choice to select k unique integer centroids from the data
    centroids = np.random.choice(oneDimensionArray, k, replace=False)

    #Here will be store all the centroids with the closest items to them
    clusterStorage = dict()

    #maximum algo iterations
    maxIterations = 3

    for _ in range(maxIterations):

        clusterStorage.clear() # all the items putted in relation to the previuos centroids are worhtless

        #initialize the clusterStorage keys, there will be k keys
        for i in centroids:
            clusterStorage[i] = []

        #fill the items to the keys: it will put each item of data to its centroid
        for i in oneDimensionArray:
            min_difference = math.inf
            closest_centroid = None
            for j in centroids:
                if min_difference > abs(j-i):
                    min_difference = abs(j-i)
                    closest_centroid = j
            clusterStorage[closest_centroid] = clusterStorage[closest_centroid] + [i]

        #clear the array with the current centroids
        centroids = []

        #set the new centroids
        for i in clusterStorage.keys():
            try:
              centroids.append(int(np.mean(clusterStorage[i])))
            except:
              print("error- cluster empty")

    #having the final centroids edit the data of the nifti file accordignly
    for i in range(data.shape[0]):
      for j in range(data.shape[1]):
        for k in range(data.shape[2]):
          min_difference = math.inf
          closest_centroid = None
```

```
            for c in centroids:
                if min_difference > abs(c-data[i][j][k]):
                    min_difference = abs(c-data[i][j][k])
                    closest_centroid = c
            data[i][j][k] = closest_centroid

    return data
```

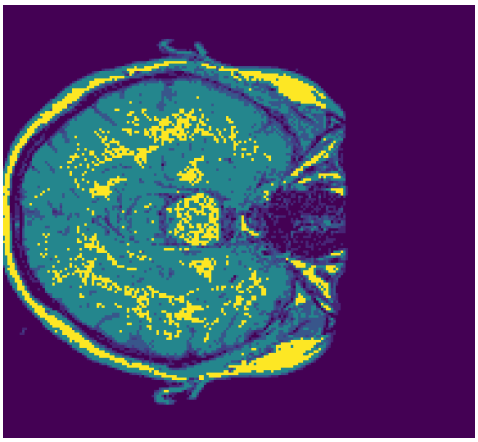

Figura 7: Un ejemplo de un resultado del algoritmo de k-means cuando el numero de clusters es 6

3.4.1. Ventajas

Simplicidad y Popularidad: El algoritmo K-means es simple de entender e implementar y es ampliamente utilizado en la segmentación de imágenes.
Eficiencia: Es computacionalmente eficiente para conjuntos de datos grandes y converge rápidamente en la mayoría de los casos.
Visualizar clusters: Permite obtener clusters de voxeles con intensidades cercanas, lo cual puede ser útil para detectar zonas que se diferencian de otras por tener diferentes niveles de intensidades.

3.4.2. Desventajas

Número de Clusters: Requiere que el número de clusters (k) se especifique de antemano, lo cual puede no ser obvio o adecuado para todos los problemas.
Sensibilidad a Centroides Iniciales: Los resultados pueden depender fuertemente de la selección inicial de los centroides, lo que puede llevar a resultados subóptimos.
No siempre maneja bien formas arbitrarias: No es adecuado para detectar clusters con formas arbitrarias.
No garantiza la solución óptima: Puede que la solución que arroje no sea la mejor posible.
Sensibilidad a Outliers: Es sensible a outliers, que pueden afectar significativamente los centroides y la calidad de los clusters.

4. Aplicación de filtros

4.1. Filtro de promedio - Mean filter

Este filtro suaviza la imagen haciendo borrosos los bordes y reduciendo el ruido de alta frecuencia, asignando el promedio de intensidades de sus vecinos a cada voxel (se tienen en cuenta en este caso solo 4 vecinos para reducir el costo computacional).

```
def mean_filter(inputData: np.ndarray) -> np.ndarray:

    #only 4 neighbors for speed - results are not very different with more neighbors
```

```
neighbors = [
    (1, 0, 0), (0, 0, 1), (0, 1, 0),(0, -1, 0)
]

# Create a mask to not edit the original data
data = np.zeros_like(inputData, dtype=np.uint8)

for i in range(inputData.shape[0]):
  for j in range(inputData.shape[1]):
    for k in range(inputData.shape[2]):
        cell_intensity_sum = 0
        # get the mean of neighbors
        for dx, dy, dz in neighbors:
            x, y, z = i + dx, j + dy, k + dz
            if (0 <= x < inputData.shape[0] and 0 <= y < inputData.shape[1] and 0 <= z < inputData.shape[2]):
                cell_intensity_sum += inputData[x][y][z]
        cell_mean = cell_intensity_sum / len(neighbors)
        data[i][j][k] = cell_mean

return data
```

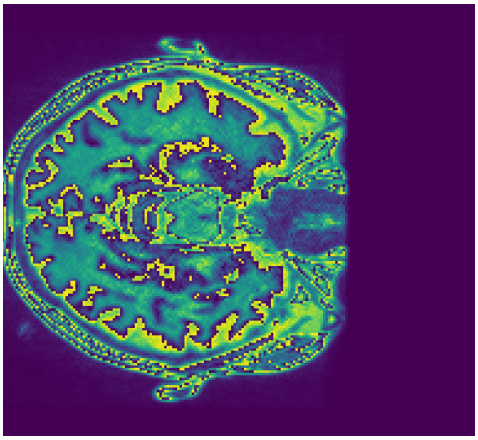

Figura 8: Un ejemplo de un resultado del filtro de promedio

4.1.1. Ventajas

Simplicidad y Facilidad de Implementación: El filtro de media es sencillo de entender y fácil de implementar.
Reducción de Ruido: Ayuda a suavizar la imagen y reducir el ruido, mejorando la calidad visual de la imagen.

4.1.2. Desventajas

Eficiencia Computacional: Su complejidad computacional es elevada al usar tantos ciclos for. Se podría mejorar usando kernels.
Suavizado de Detalles: Puede suavizar o eliminar detalles importantes en la imagen, afectando la precisión en la detección de características.
Sensibilidad a Bordes: No preserva bien los bordes, lo que puede resultar en imágenes borrosas.
Dependencia de Vecinos: La calidad del resultado depende de la cantidad y selección de vecinos; usar solo 4 puede no ser suficiente en algunas aplicaciones.

4.2. Filtro de mediana - Median filter

Este filtro reduce el ruido de sal y pimienta y mantiene los bordes de la imagen mejor que el filtro de promedio, asignando la mediana de intensidades de sus vecinos a cada voxel (se tienen en cuenta en este caso solo 4 vecinos para reducir el costo computacional).

```python
def median_filter(inputData: np.ndarray) -> np.ndarray:

    #only 4 neighbors for speed - results are not very different with more neighbors
    neighbors = [
        (1, 0, 0), (0, 0, 1), (0, 1, 0),(0, -1, 0)
    ]

    # Create a mask to not edit the original data
    data = np.zeros_like(inputData, dtype=float)

    for i in range(inputData.shape[0]):
      for j in range(inputData.shape[1]):
        for k in range(inputData.shape[2]):
            intensities = np.array([])
            # get the median of neighbors
            for dx, dy, dz in neighbors:
                x, y, z = i + dx, j + dy, k + dz
                if (0 <= x < inputData.shape[0] and 0 <= y < inputData.shape[1] and 0 <= z < inputData.shape[2]):
                   intensities = np.append(intensities, [inputData[x][y][z]])
            data[i][j][k] = np.median(intensities)

    return data
```

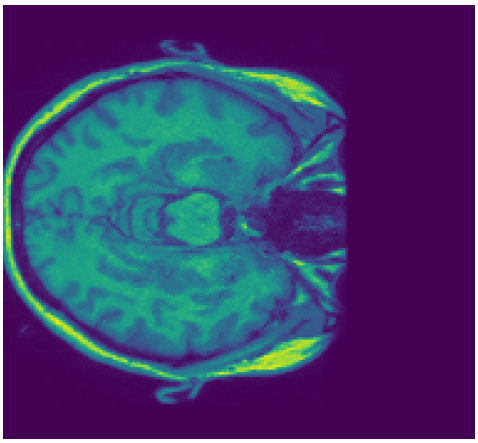

Figura 9: Un ejemplo de un resultado del filtro de mediana

4.2.1. Ventajas

Reducción de Ruido Impulsivo: El filtro de mediana es muy efectivo para eliminar ruido impulsivo (salt and pepper noise) de las imágenes.
Preservación de Bordes: A diferencia del filtro de media, el filtro de mediana preserva mejor los bordes y detalles finos de la imagen.
Simplicidad: Es relativamente sencillo de implementar y entender.

4.2.2. Desventajas

Costo Computacional: Puede ser más computacionalmente intensivo que el filtro de media, especialmente para imágenes grandes, debido a la necesidad de ordenar los valores de los vecinos. Al usar también tantos ciclos for, y por la cantidad de inserciones sobre listas, este algoritmo es lento, pero se puede mejorar con el uso de kernels.
Efectividad Reducida con Vecinos Limitados: Usar solo 4 vecinos puede no ser suficiente para lograr un filtrado efectivo en todas las situaciones.

5. Algoritmos de estandarización

5.1. Rescaling std.

Este algoritmo sirve para rescalar el rango de las intensidades entre 0 y 1.

```
def rescaling_std(inputData: np.ndarray) -> np.ndarray:
    flattenInputData = inputData.flatten()
    minIntensity = np.min(flattenInputData)
    maxIntensity = np.max(flattenInputData)

    # Create a mask
    data = np.zeros_like(inputData, dtype=float)

    for i in range(inputData.shape[0]):
        for j in range(inputData.shape[1]):
            for k in range(inputData.shape[2]):
                data[i][j][k] = (inputData[i][j][k]-minIntensity)/(maxIntensity - minIntensity)

    return data
```

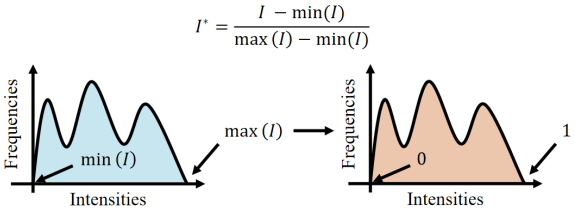

Figura 10: Rescalado de intensidades

5.1.1. Ventajas

Normalización de Intensidades: Escala los valores de intensidad de los píxeles a un rango común (0 a 1), lo que puede mejorar la comparabilidad y el procesamiento posterior.
Simplicidad: El algoritmo es simple y fácil de implementar.

5.1.2. Desventajas

Sensibilidad a Valores Extremos: Los valores mínimos y máximos pueden ser outliers, lo que puede distorsionar la escala.
Pérdida de Información Relativa: La transformación puede perder información sobre la distribución original de intensidades si hay una gran variabilidad.
No Considera Desviación Estándar: Solo utiliza los valores mínimo y máximo, sin tener en cuenta la desviación estándar o la media, lo que puede ser insuficiente para imágenes con distribución no uniforme.

5.2. Zscore std.

La estandarización z-score transforma los datos para que tengan una media de 0 y una desviación estándar de 1.

```
def z_score_std(inputData: np.ndarray) -> np.ndarray:
    flattenInputData = inputData.flatten()
    # Supongamos que "mi_array" es un array
    mi_array = flattenInputData
    # 1. Calcular la media
    media = sum(mi_array) / len(mi_array)
    # 2. Restar la media y elevar al cuadrado
    diferencias_cuadrado = [(x - media) ** 2 for x in mi_array]
```

```
    # 3. Calcular la media de las diferencias al cuadrado
    media_diferencias_cuadrado = sum(diferencias_cuadrado) / len(mi_array)
    # 5. Tomar la raíz cuadrada de la media de las diferencias al cuadrado
    desviacion_estandar = math.sqrt(media_diferencias_cuadrado)

    # Create a mask
    data = np.zeros_like(inputData, dtype=float)

    for i in range(inputData.shape[0]):
        for j in range(inputData.shape[1]):
            for k in range(inputData.shape[2]):
                data[i][j][k] = (inputData[i][j][k]-media)/desviacion_estandar


    return data
```

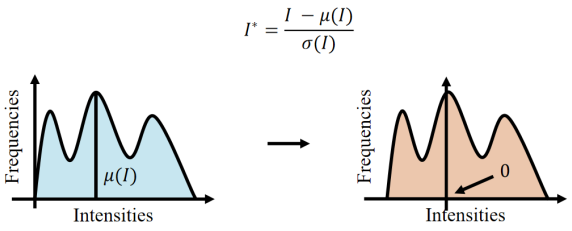

Figura 11: Transformación z-score

5.2.1. Ventajas

Normalización Efectiva: El algoritmo de estandarización Z-score transforma los datos para que tengan una media de 0 y una desviación estándar de 1, facilitando la comparabilidad entre diferentes conjuntos de datos.
Manejo de Datos Anómalos: Reduce el impacto de outliers, ya que estos serán menos prominentes después de la normalización.
Preparación para Modelos Estadísticos: Los datos estandarizados son esenciales para muchos algoritmos de aprendizaje automático y estadísticos que asumen distribuciones normales.

5.2.2. Desventajas

Pérdida de Interpretabilidad: Después de la estandarización, los valores transformados pueden ser menos interpretables en el contexto original de los datos.

5.3. Histogram matchng std.

Este algoritmo permite estandarizar una archivo nii (test) para que su distribución de intensidades se mas parecida a la distribución de otro archivo nii (train) resultando en otro archivo nii (result). Este algoritmo sirve para hacer un mejor análisis del archivo test, al estandarizarlo con respecto a un archivo que inicialmente se toma como un archivo bueno o sin tanto ruido.

```
import numpy as np

#Se usa para llamar a la funcion que crea las funciones de
#transformacion y llamar a la funcion que realiza la transormacion
#con las funciones lineales halladas
def h_matching(trainData, testData, k):
    funcs = training(trainData, k)
    standardized_data = testing(testData, funcs)
    return standardized_data

#Usando las funciones de transformacion, en el testData se halla el percentil
#de cada voxel para hallar la nueva intensidad de ese voxel segun ese percentil hallado
```

```python
def testing(testData, functions):

    standardized_data = np.zeros(testData.shape)

    sorted_testData = np.sort(testData.flatten())
    len_sorted_testData = len(sorted_testData)

    for i in range(testData.shape[0]):
        for j in range(testData.shape[1]):
            for k in range(testData.shape[2]):
                index = np.searchsorted(sorted_testData, testData[i,j,k])
                percentile = (index + 1) / len_sorted_testData * 100
                percentile = percentile if 5 <= percentile <= 95 else 0
                for func in functions:
                    if func['startPercentile'] <= percentile < func['endPercentile']:
                        standardized_data[i, j, k] = func['func'](percentile)

    return standardized_data

#Se crean las funciones con las que se van a transformar el testData (nuevo data)
def training(trainData, k):
    percentiles = np.linspace(5, 95, k) #X
    y = np.percentile(trainData.flatten(), percentiles) #Y
    functions = []

    for i in range(k - 1):
        start = percentiles[i]
        end = percentiles[i + 1]
        m_i = (y[i + 1] - y[i]) / (percentiles[i+1] - percentiles[i])
        b_i = y[i] - m_i * percentiles[i]
        functions.append({'startPercentile': start, 'endPercentile': end, 'func': lambda x, m=m_i, b=b_i: m * x + b})

    return functions
```

```python
myTrainData = np.random.randint(1, 1001, size=(100, 100, 100))
myTestData = np.random.randint(1, 100, size=(20, 20, 20))
```

Figura 12: Ejemplo de entrada de los datos de entrenamiento y prueba

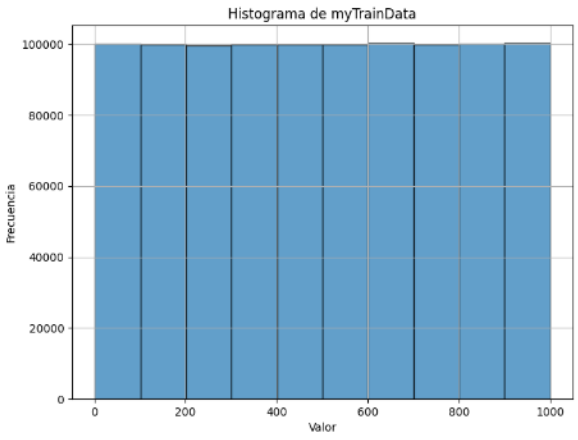

Figura 13: Ejemplo del histograma de entrenamiento

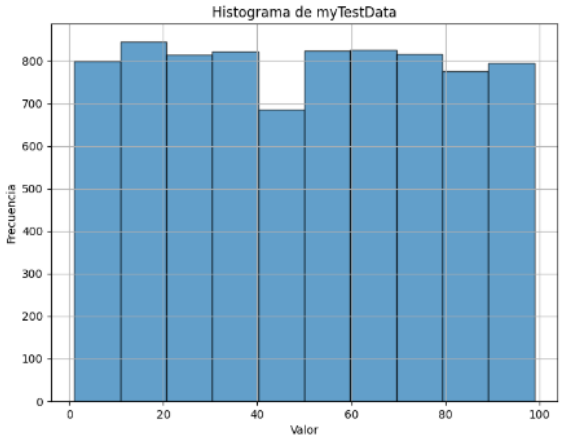

Figura 14: Ejemplo del histograma de prueba

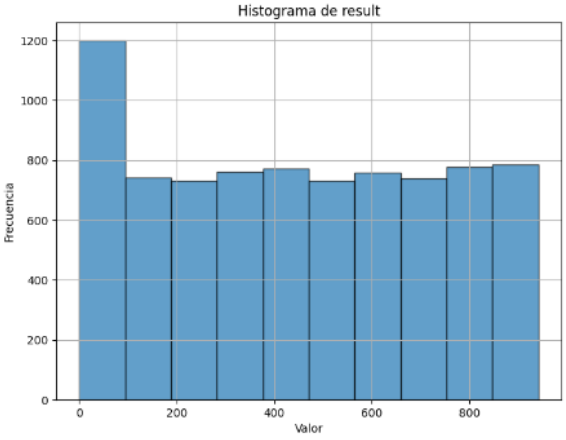

Figura 15: Ejemplo del histograma del resultado

5.3.1. Ventajas

Normalización Basada en Percentiles: Este enfoque asegura que los datos estén distribuidos de manera uniforme, eliminando efectos de valores extremos y distribuciones sesgadas.
Adaptabilidad: La técnica se adapta bien a diferentes distribuciones de datos, ya que se basa en percentiles, lo que es útil para imágenes médicas con variabilidad de intensidades.
Flexibilidad de Transformación: La creación de funciones de transformación basadas en intervalos percentiles permite una transformación flexible y ajustada a las características específicas del conjunto de datos.

5.3.2. Desventajas

Complejidad Computacional: El cálculo de percentiles y la búsqueda de índices en grandes volúmenes de datos pueden ser intensivos en tiempo y recursos.

5.4. White stripe std.

Transforma todas las intensidades, dividendo cada una por el valor de la intensidad que esta en el último pico del histograma de intensidades. Para esto se hallan todos los picos, y se toma el último.

Un pico se define cuando la frecuencia de una intensidad es mayor a la frecuencia de las intensidades a la izquierda y a la derecha de la misma.

```
def white_stripe_std(inputData: np.ndarray) -> np.ndarray:

    # Obtener el array de intensidades
    intensity_values = inputData.flatten()

    # Calcular el histograma y los bins de intensidades
    hist, bins = np.histogram(intensity_values, bins=50)

    # Due that there is one more bin than elements in hist:
    bins = bins[:-1]

    all_peaks_bin_value = []

    #append all the peaks to all_peaks
    for i in range(0,len(hist)):
        if i - 1 >= 0 and i + 1 < len(hist):
            if hist[i - 1] < hist[i] and hist[i] > hist[i + 1]:
                print("peak: ", hist[i - 1], hist[i], hist[i+1])
                all_peaks_bin_value.append(bins[i])


    #take the last element
    ws = all_peaks_bin_value[len(all_peaks_bin_value) - 1]

    # Create a mask to set the result
    data = np.zeros_like(inputData, dtype=float)

    #edit the mask to create the result
    for i in range(inputData.shape[0]):
        for j in range(inputData.shape[1]):
            for k in range(inputData.shape[2]):
                    data[i][j][k] = inputData[i][j][k]/ws

    return data
```

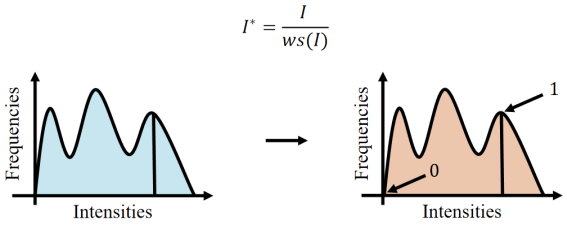

Figura 16: White strype

5.4.1. Ventajas

Normalización Basada en Picos: Utiliza el pico más alto del histograma para normalizar las intensidades, lo que puede ser efectivo para imágenes médicas con distribuciones específicas.
Reducción de Outliers: El método puede ayudar a reducir el impacto de valores atípicos al centrarse en el valor de pico del histograma.
Simplicidad: Es relativamente fácil de implementar y entender.

5.4.2. Desventajas

Dependencia del Histograma: El éxito del algoritmo depende en gran medida de la precisión del histograma y la identificación correcta de los picos.
Sensibilidad a la Distribución: Puede no funcionar bien si la distribución de intensidades no tiene picos claros o tiene múltiples picos relevantes.

6. Bordes y Curvaturas

6.1. Bordes

Para obtener los bordes, se usa el filtro Sobel, con el fin de hallar una aproximación a la primera derivada en las direcciones x, y, y z usando convoluciones y luego se computa la magnitud del gradiente de cada punto usando la fórmula de la distancia euclidiana. Esta magnitud representa la fuerza de los bordes en los datos de entrada.

```
import numpy as np
import scipy
def edges_algo(inputData):

    kernel_x = np.array([-1, 1]) / 2

    Dx = scipy.ndimage.convolve(inputData, kernel_x.reshape((2, 1, 1)))
    Dy = scipy.ndimage.convolve(inputData, kernel_x.reshape((1, 2, 1)))
    Dz = scipy.ndimage.convolve(inputData, kernel_x.reshape((1, 1, 2)))

    data = np.sqrt(Dx ** 2 + Dy ** 2 + Dz ** 2)

    return data
```

6.1.1. Ventajas

Detección de Bordes: Es útil para identificar características y contornos importantes.
Simplicidad y Eficiencia: Es relativamente sencillo y eficiente, ya que utiliza convoluciones básicas con pequeños kernels para calcular las derivadas.
Aplicabilidad en 3D: Funciona con datos tridimensionales, lo que es ideal para aplicaciones en imágenes médicas.

6.1.2. Desventajas

Sensibilidad al Ruido: Como la mayoría de los detectores de bordes, puede ser sensible al ruido en la imagen, lo que puede resultar en falsos positivos en la detección de bordes.
No es Escalable: Utiliza un kernel fijo, lo que puede no ser adecuado para todas las resoluciones y tipos de imágenes.

6.2. Curvaturas

Para hallar curvaturas, se usa un kernel de tres dimensiones, el cual se usa una primera vez para hallar una aproximación a la primera derivada en un dirección, y luego se usa una segunda vez para hallar una aproximación a la segunda derivada que es la permite obtener curvaturas. Este procedimiento se hace para las direcciones x, y, y z para posteriormente hallar la magnitud de la curvatura en cada punto usando la fórmula de la distancia euclidiana.

```
import scipy
import numpy as np
def curvature_algo(inputData):

    kernel_x = np.array([-1, 1]) / 2

    Dx = scipy.ndimage.convolve(inputData, kernel_x.reshape((2, 1, 1)))
    Dxx = scipy.ndimage.convolve(Dx, kernel_x.reshape((2, 1, 1)))
    Dy = scipy.ndimage.convolve(inputData, kernel_x.reshape((1, 2, 1)))
    Dyy = scipy.ndimage.convolve(Dy, kernel_x.reshape((1, 2, 1)))
    Dz = scipy.ndimage.convolve(inputData, kernel_x.reshape((1, 1, 2)))
    Dzz = scipy.ndimage.convolve(Dz, kernel_x.reshape((1, 1, 2)))

    data = np.sqrt(Dxx ** 2 + Dyy ** 2 + Dzz ** 2)
    return data
```

6.2.1. Ventajas

Detección de Curvatura: Este algoritmo calcula la curvatura en una imagen 3D, proporcionando información sobre las características geométricas de las superficies.
Análisis Completo: Utiliza derivadas de segundo orden para detectar cambios en la pendiente, lo que es útil para identificar características sutiles en la imagen.
Aplicación en 3D: Funciona con datos tridimensionales, siendo ideal para imágenes volumétricas como las médicas.

6.2.2. Desventajas

Sensibilidad al Ruido: Las derivadas de segundo orden son más sensibles al ruido, lo que puede resultar en detección de curvatura incorrecta.
Requiere Suavizado: Puede requerir un preprocesamiento de suavizado para reducir el impacto del ruido antes de aplicar el algoritmo.

7. Registro de imágenes médicas

En este caso se implementó el registro afín, el cual sirve para alinear dos imágenes o conjuntos de datos para que estén en la misma geometría espacial. Como ejemplo se tomó una imagen nifti como imagen base o fija, esta luego se rotó 10 grados para obtener una imagen rotada, y con estas dos imágenes se llamó a la función de registro afín, el cual da como resultado la imagen rotada más acorde espacialmente a la imagen fija o base. Se puede observar que la imagen resultado, no tiene la misma calidad que las imágenes usadas para llamar a la función.

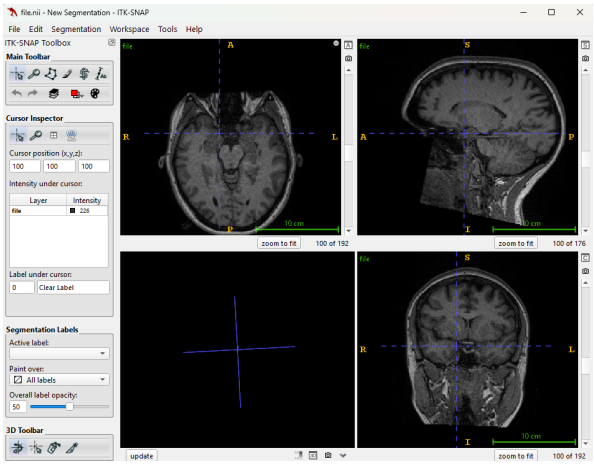

Figura 17: Imagen base

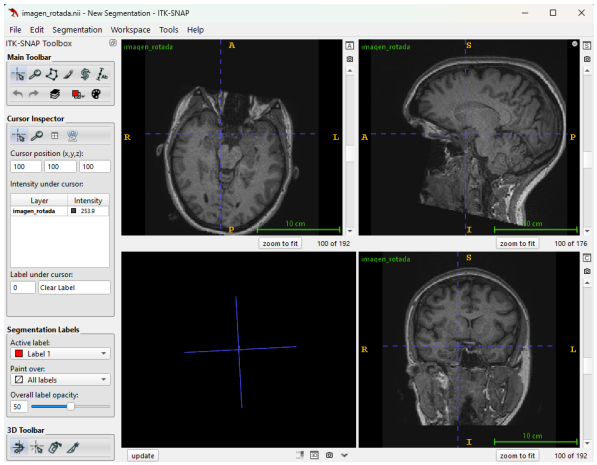

Figura 18: Imagen rotada

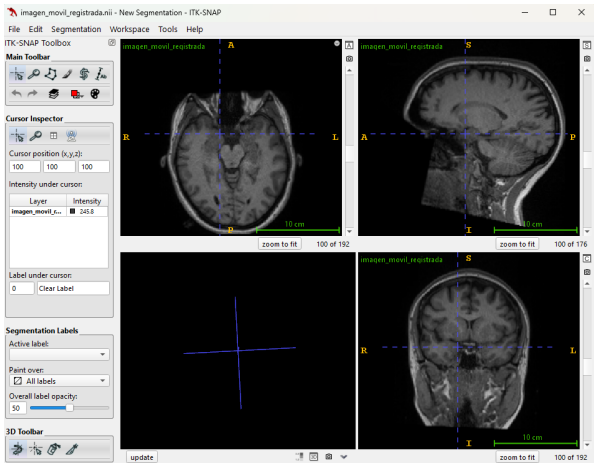

Figura 19: Imagen resultado

```
def affine_registration_algo(data_fixed_path, data_rotated_path):
    # Cargar imágenes
    data_fixed = sitk.ReadImage(data_fixed_path, sitk.sitkFloat32)
    data_rotated = sitk.ReadImage(data_rotated_path, sitk.sitkFloat32)

    # Crear registro
    register = sitk.ImageRegistrationMethod()

    # Configurar transformación (afín)
    initial_transformation = sitk.CenteredTransformInitializer(data_fixed,
                                                               data_rotated,
                                                               sitk.AffineTransform(3),
                                                               sitk.CenteredTransformInitializerFilter.GEOMETRY)
    register.SetInitialTransform(initial_transformation)
    register.SetMetricFixedMask(data_fixed>0)

    # Configurar métrica de similitud
    register.SetMetricAsMattesMutualInformation(numberOfHistogramBins=50)
    register.SetMetricSamplingStrategy(register.RANDOM)
    register.SetMetricSamplingPercentage(0.01)

    # Configurar optimizador
    register.SetOptimizerAsGradientDescent(learningRate=1.0, numberOfIterations=100,
    convergenceMinimumValue=1e-6, convergenceWindowSize=10)
    register.SetOptimizerScalesFromIndexShift()

    # Realizar registro
```

```
transformation = register.Execute(data_fixed, data_rotated)

# Aplicar transformación a la imagen móvil
data_rotated_registered = sitk.Resample(data_rotated, data_fixed, transformation,
sitk.sitkLinear, 0.0, data_rotated.GetPixelID())

# Muestrear la imagen móvil registrada en el espacio de la imagen móvil original
data_rotated_registered_resample = sitk.Resample(data_rotated_registered, data_rotated)

# Calcular la diferencia entre la imagen móvil original y la imagen móvil registrada muestreada
difference = sitk.Abs(data_rotated_registered_resample - data_rotated)

np_data_rotated_registered = sitk.GetArrayFromImage(data_rotated_registered).T
np_difference = sitk.GetArrayFromImage(difference)
return np_data_rotated_registered
```

7.1. Ventajas

Precisión: El algoritmo de registro afín puede ajustar imágenes médicas con gran precisión, alineando correctamente las estructuras anatómicas.
Flexibilidad: Soporta transformaciones afines, lo que permite rotaciones, escalados, traslaciones y sesgos.

7.2. Desventajas

Computacionalmente Intensivo: Puede ser lento debido al uso de métricas complejas y la optimización de múltiples parámetros.
Dependencia de Parámetros: Requiere ajuste fino de los parámetros de optimización y métricas para obtener resultados óptimos.

8. Coordenadas Laplacianas (versión original, 2D)

Para implementar el algoritmo de Coordenadas Laplacianas, se pueden seguir los siguientes pasos tal como se expresa en Casaca et al. [2014]:

1. **Crear la matriz pesos W:** Esta matriz la cual tiene una dimensión de (width*height,width*height), donde width es el ancho de la imagen en px y height es el alto de la imagen en px, es donde se guardan los ponderados asignados a las artistas que conectan cualquier voxel (píxel (i,j) en done i < height y j < width) con otro voxel (pixel de coordenadas (i,j) en done i < height y j < width). Para calcular la matriz, a su vez se puede seguir estos pasos:

   1.1 **Constante Beta:** Es una constante de afinación. En este documentó se usa Beta = 0.1

   ```
   ...
   beta=0.1
   ...
   ```

   1.2 **Hallar la raíz cuadrada del valor absoluto de la diferencia al cuadrado de las intensidades.** Para cada para de voxeles vecinos, se halla la raíz cuadrada del valor absoluto de la diferencia al cuadardo de las intensidades.

$$\|I_i - I_j\|_\infty^2$$

Figura 20: Raíz cuadrada del valor absoluto de la diferencia al cuadrado de las intensidades

```
...
np.sqrt(np.abs(image[i, j] - image[i - 1, j])**2)
...
np.sqrt(np.abs(image[i, j] - image[i + 1, j])**2)
...
```

```
np.sqrt(np.abs(image[i, j] - image[i, j - 1])**2)
...
np.sqrt(np.abs(image[i, j] - image[i, j + 1])**2)
...
```

1.3 **Hallar sigma:** Este valor es el valor máximo de la norma infinito calculada entre los pares de píxeles que comparten una arista. Este valor se usa para normalizar las diferencias de intensidad entre píxeles en la imagen para asegurar que los pesos calculados no sean excesivamente pequeños o grandes.

$$\sigma = \max_{(P_i,P_j)\in E} ||I_i - I_j||_\infty$$

Figura 21: Formula de Sigma

```python
# Primero encontrar sigma, la máxima diferencia de intensidad entre vecinos
sigma = 0
for i in range(height):
    for j in range(width):
        if i > 0:  # Arriba
            sigma = max(sigma, np.abs(image[i, j] - image[i - 1, j]))
        if i < height - 1:  # Abajo
            sigma = max(sigma, np.abs(image[i, j] - image[i + 1, j]))
        if j > 0:  # Izquierda
            sigma = max(sigma, np.abs(image[i, j] - image[i, j - 1]))
        if j < width - 1:  # Derecha
            sigma = max(sigma, np.abs(image[i, j] - image[i, j + 1]))

# Asegurarse de que sigma no sea cero para evitar división por cero
if sigma == 0:
    sigma = 1
```

1.4 **Matriz de pesos W completa** Una vez hechos los pasos anteriores, se usa cada paso para crear la matriz de pesos W.

$$w_{ij} = \exp\left(-\frac{\beta||I_i - I_j||_\infty^2}{\sigma}\right)$$

Figura 22: Matriz de pesos W

```python
def calculate_weights(image, beta=0.1):
    height, width = image.shape
    num_pixels = height * width
    indices = np.arange(num_pixels).reshape(height, width)
    W = np.zeros((num_pixels, num_pixels))

    # Primero encontrar sigma, la máxima diferencia de intensidad entre vecinos
    sigma = 0
    for i in range(height):
        for j in range(width):
            if i > 0:  # Arriba
                sigma = max(sigma, np.abs(image[i, j] - image[i - 1, j]))
            if i < height - 1:  # Abajo
                sigma = max(sigma, np.abs(image[i, j] - image[i + 1, j]))
            if j > 0:  # Izquierda
                sigma = max(sigma, np.abs(image[i, j] - image[i, j - 1]))
            if j < width - 1:  # Derecha
                sigma = max(sigma, np.abs(image[i, j] - image[i, j + 1]))

    # Asegurarse de que sigma no sea cero para evitar división por cero
    if sigma == 0:
        sigma = 1

    # Calcular los pesos con el sigma encontrado
    for i in range(height):
```

```
            for j in range(width):
                index = indices[i, j]
                if i > 0:  # Arriba
                    W[index, indices[i - 1, j]] =
                    np.exp(-beta * (np.sqrt(np.abs(image[i, j] - image[i - 1, j])**2) / sigma))
                if i < height - 1:  # Abajo
                    W[index, indices[i + 1, j]] =
                    np.exp(-beta * (np.sqrt(np.abs(image[i, j] - image[i + 1, j])**2) / sigma))
                if j > 0:  # Izquierda
                    W[index, indices[i, j - 1]] =
                    np.exp(-beta * (np.sqrt(np.abs(image[i, j] - image[i, j - 1])**2) / sigma))
                if j < width - 1:  # Derecha
                    W[index, indices[i, j + 1]] =
                    np.exp(-beta * (np.sqrt(np.abs(image[i, j] - image[i, j + 1])**2) / sigma))

    return W
```

2. **Crear la matriz Laplaciana L:**

   2.1 **Crear el vector D:** Este vector representa la suma de los pesos de todas las conexiones para cada nodo, y es crucial para la formulación de la matriz Laplaciana. También se puede entender como la valencia de los pesos de cada voxel (voxel siendo un píxel de coordenadas (i,j) en done i < height y j < width).

assigned to edge (P_i, P_j) of the graph. The set $N(i) = \{j : (P_i, P_j) \in E\}$ represents the indices of the pixels P_j that share an edge with pixel P_i and $d_i = \sum_{j \in N(i)} w_{ij}$ is the weighted valency of P_i.

Figura 23: Vector D

La matriz W, debido a que en cada fila i, tiene 0's en las columnas en donde no tiene conexión y un valor distinto de 0 cuando si tiene conexión y por tanto un peso, se suman todos los pesos de esa fila para obtener el resultado de la fila i en D. Este vector tiene una dimensión de (width*height), siendo unidimensional.

```
import numpy as np
D = np.sum(W, axis=1)
```

   2.2 **Calcular la Matriz Laplaciana**

$$\mathbf{L} = \mathbf{D} - \mathbf{W}$$

Figura 24: Matriz Laplaciana

Para calcular la matriz Laplaciana, haciendo la diferencia de D y W, hay que tener en cuenta que W tiene una dimensión de (width*height,width*height) o sea una arreglo de dos dimensiones, mientras que D tiene una dimensión por ahora de (width*height) o sea unidimensional, por lo que antes de realizar la resta, se debe transformar a D en un array de dos dimensiones (de igual número de filas y columnas que W), y, según el documento base, los valores actuales en D, deben de quedar en la diagonal, mientras que los demás valores que no están en la diagonal, deben ser 0's. Así pues lo que se hace es crear una matriz esparsa de D, lo cual es mucho mas eficiente que convertir D a una matriz de dimensiones (width*height,width*height) en donde muchas de esas celdas serán 0's.

```
import scipy.sparse as sp
L = sp.diags(D) - W
```

3. **Crear el sistema**

   3.1 **Crear L^2:** Para crear la matriz Laplaciana al cuadrado, se multiplica la matriz Laplaciana hallada por ella misma.

```
import numpy as np
L2 = L.dot(L)
```

3.2 **Entender la matriz de índices de los Píxeles:** Esta matriz de ejemplo tiene 9 píxeles, con nombres P1 hasta P9. Los números del 1 al 9 son importantes para transformar posiciones (i,j) en esos números del 1 al 9 (útiles por ejemplo para transformar valores de semillas a valores en la matriz Is). Cabe resaltar que i puede tomar valores desde 0 hasta height-1 y j puede tomar valores desde 0 hasta width-1 en esta matriz de nueve Píxeles.

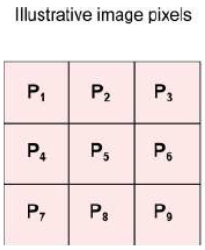

Figura 25: Matriz de índices de Píxeles

3.3 **Entender y manejar el arreglo de semillas y el arreglo de anotaciones, y los escalares xB y xF:**

```
seeds = [
# Fondo
(5, 5), (5, 55), (30, 60), (0, 15), (10,0),(20,0),
(30,0),(20,40),(60,50),(60,0),(70,0),(60,80), (2,20), (40,0),
# Perro izquierdo
(15, 20), (40, 20), (60, 20), (65, 20), (10,20),
(65,25),(55,20),(40,25),(35,20),(70,30),(25,20),(15,20),(45,19)
]
labels = [
'B', 'B', 'B', 'B','B', 'B', 'B', 'B','B', 'B', 'B', 'B','B', 'B', 'B', 'B','B','B'
'F', 'F', 'F', 'F','F', 'F', 'F', 'F','F','F','F','F','F'
]
xB = 1
xF = 0
```

El arreglo de seeds es un arreglo de tuplas de 2 elementos, cada tupla es una posición en la imagen. En seeds, en este caso, primero se especifican algunas anotaciones del fondo (Background) y algunas anotaciones de la parte que se quiere obtener (Foreground). Después se encuentra el arreglo de labels, que es donde se escriben las anotaciones de cada tupla en seeds. En este arreglo labels, se asigna 'B' si la tupla en esa misma posición en seeds es Background, o 'F', si la tupla en esa misma posición en seeds es Foreground. Por último se asigna un valor tanto a xB como a xF, en donde xB debe de ser mayor a xF.

3.4 **Crear la matriz Is:** Para hallar la matriz Is, se debe crear una matriz de dimensiones (width*height,width*height) en donde cada semilla b ya sea de Background o Foreground, se debe transformar a una coordenada en la matriz de indicies. Una vez obtenido el número del Píxel, por ejemplo un número i, se debe poner en la posición (i,i) en la matriz Is, un 1. Recordar que se debe hacer esto para todas las semillas, tanto de Background como de Foreground. Las demás posiciones deben de ser 0.

where $\mathbf{I_S}$ is a diagonal matrix such that $I_S(i,i) = 1, i \in S = B \cup F$, and zero, otherwise, $\mathbf{b}$ is the vector where $b(i) = x_B, i \in B$, $b(i) = x_F, i \in F$, and zero, otherwise,

Figura 26: Matriz Is

```
import numpy as np
# Preparar I_s, donde sólo los elementos de las semillas son 1
I_s = sp.lil_matrix((num_pixels, num_pixels))
for (i, j), label in zip(seeds, labels):
    idx = indices[i, j]
    I_s[idx, idx] = 1
```

3.5 **Entender y crear el arreglo b:** El arreglo b, es un arreglo unidimensional de longitud width*height en donde según el número del Píxel en la matriz de índices que representa tupla en seeds, irá el valor de xB si ese Píxel a sido escogido como Background o xF si ese Píxel a sido escogido como Foreground, o 0 si no está relacionado con ninguna semilla.

```
import numpy as np
b = np.zeros(num_pixels)
for (i, j), label in zip(seeds, labels):
    idx = indices[i, j]
    b[idx] = xB if label == 'B' else xF
```

4. **Resolver el sistema usando Factorización de Cholesky** Finalimente con todo lo obtenido, se puede formar un sistema el cual puede ser resuelto por medio de la factorización de Cholesky. Esto da un resultado x, el cual es un arreglo, es único y tiene una longitud de width*height. En x está contenida la solución al sistema.

```
import scipy.sparse as sp
# Usando factorización Cholesky para resolver el sistema
A = I_s + L2
A = sp.csr_matrix(A)
solve = factorized(A)  # Factorización Cholesky
#asignar a x la solución del sistema
x = solve(b)
```

5. **Asignar etiquetas al resultado de resolver el sistema** Ya con el arreglo x, se procede a clasificar cada elemento de x en Background o Foreground. Para obtener una mejor representación visual, se hizo pequeño cambio en la implementación a como se describe en el documento guía (en vez de colocar xF, se colocó el valor de la intensidad en la imagen original, con el fin de ver en la imagen original que fue lo que se segmentó).

$$y_i = \begin{cases} x_B, & \text{if } x_i \geq \frac{x_B + x_F}{2} \\ x_F, & \text{otherwise} \end{cases} \qquad (3)$$

Figura 27: Final Value Assignation

```
import numpy as np
def apply_labels(segmented_values,image, xB, xF):
    # Calcular el umbral basado en xB y xF
    threshold = (xB + xF) / 2

    # Asignar etiquetas basado en el umbral
    labels = np.where(segmented_values >= threshold, xB, image)

    return labels
```

6. **Código completo de Coordinadas Laplacianas** Finalmente se puede unir todo, para obtener el código de la implementación del algoritmo de coordenadas Laplacianas. Cabe destacar que en lo posible se trata de usar matrices dispersas para tratar de mejorar el rendimiento del algoritmo. Para que funcione todo con solo copiar y pegar, pegar el siguiente código en google colab, y añadir en drive de la cuenta una carpeta con nombre 'IPDI' y dentro de ella una imagen llamada 'dogs.png'.

```
import numpy as np
import matplotlib.pyplot as plt
from google.colab import drive
import os
# Mount Google Drive
drive.mount('/content/drive')
my_drive_path = '/content/drive/MyDrive/IPDI'

import numpy as np
import scipy.sparse as sp
from scipy.sparse.linalg import spsolve, factorized
def calculate_weights(image, beta=0.1):
    height, width = image.shape
```

```
    num_pixels = height * width
    indices = np.arange(num_pixels).reshape(height, width)
    W = np.zeros((num_pixels, num_pixels))

    # Primero encontrar sigma, la máxima diferencia de intensidad entre vecinos
    sigma = 0
    for i in range(height):
        for j in range(width):
            if i > 0:  # Arriba
                sigma = max(sigma, np.abs(image[i, j] - image[i - 1, j]))
            if i < height - 1:  # Abajo
                sigma = max(sigma, np.abs(image[i, j] - image[i + 1, j]))
            if j > 0:  # Izquierda
                sigma = max(sigma, np.abs(image[i, j] - image[i, j - 1]))
            if j < width - 1:  # Derecha
                sigma = max(sigma, np.abs(image[i, j] - image[i, j + 1]))

    # Asegurarse de que sigma no sea cero para evitar división por cero
    if sigma == 0:
        sigma = 1

    # Calcular los pesos con el sigma encontrado
    for i in range(height):
        for j in range(width):
            index = indices[i, j]
            if i > 0:  # Arriba
                W[index, indices[i - 1, j]] =
                np.exp(-beta * (np.sqrt(np.abs(image[i, j] - image[i - 1, j])**2) / sigma))
            if i < height - 1:  # Abajo
                W[index, indices[i + 1, j]] =
                np.exp(-beta * (np.sqrt(np.abs(image[i, j] - image[i + 1, j])**2) / sigma))
            if j > 0:  # Izquierda
                W[index, indices[i, j - 1]] =
                np.exp(-beta * (np.sqrt(np.abs(image[i, j] - image[i, j - 1])**2) / sigma))
            if j < width - 1:  # Derecha
                W[index, indices[i, j + 1]] =
                np.exp(-beta * (np.sqrt(np.abs(image[i, j] - image[i, j + 1])**2) / sigma))

    return W


def segment_image(image, seeds, labels, xB, xF, beta):
    height, width = image.shape
    num_pixels = height * width
    indices = np.arange(num_pixels).reshape(height, width)

    W = calculate_weights(image, beta)
    D = np.sum(W, axis=1)
    print("D shape: ",D.shape)
    L = sp.diags(D) - W

    L2 = L.dot(L)

    # Preparar I_s, donde sólo los elementos de las semillas son 1
    I_s = sp.lil_matrix((num_pixels, num_pixels))

    b = np.zeros(num_pixels)
    for (i, j), label in zip(seeds, labels):
        idx = indices[i, j]
        I_s[idx, idx] = 1
        b[idx] = xB if label == 'B' else xF

    A = I_s + L2

    # Usando factorización Cholesky para resolver el sistema
    A = sp.csr_matrix(A)
    solve = factorized(A)  # Factorización Cholesky

    x = solve(b)

    segmented_image = x.reshape((height, width))
```

```
    return segmented_image

from PIL import Image
import numpy as np

def load_image_to_numpy(path):
    # Open the image file
    with Image.open(path) as img:
        # Convert the image to grayscale if needed
        img_gray = img.convert('L')  # Use 'RGB' to keep it in color

        # Convert the image to a NumPy array
        image_array = np.array(img_gray)
        print(image_array)
        return image_array


data = load_image_to_numpy(os.path.join(my_drive_path,'dog.jpeg'))
#image = np.random.rand(100, 100)
image = data
print(image.shape)
plt.imshow(image)
seeds = [
    # Fondo
    (5, 5), (5, 55), (30, 60), (0, 15), (10,0),(20,0),
    (30,0),(20,40),(60,50),(60,0),(70,0),(60,80), (2,20), (40,0),
    # Perro izquierdo
    (15, 20), (40, 20), (60, 20), (65, 20), (10,20),
    (65,25),(55,20),(40,25),(35,20),(70,30),(25,20),(15,20),(45,19)
]
labels = [
    'B', 'B', 'B', 'B','B', 'B', 'B', 'B','B', 'B', 'B', 'B','B',
    'B', 'B', 'B','B','B'
    'F', 'F', 'F', 'F','F', 'F', 'F', 'F','F','F','F','F','F'
]
xB = 1
xF = 0
beta = 1
def apply_labels(segmented_values,image, xB, xF):
    # Calcular el umbral basado en xB y xF
    threshold = (xB + xF) / 2

    # Asignar etiquetas basado en el umbral
    labels = np.where(segmented_values >= threshold, xB, image)

    return labels
final = segment_image(image, seeds, labels, xB, xF, beta)
plt.imshow(apply_labels(final,image,xB,xF))
```

7. **Ejemplo de resultado** La siguiente imagen es una imagen de 75px*100px, en donde hay 3 perros, y un fondo casi uniforme. La idea, es obtener el perro a la izquierda (se puede ver esto en el arreglo seeds y en el arreglo labels.

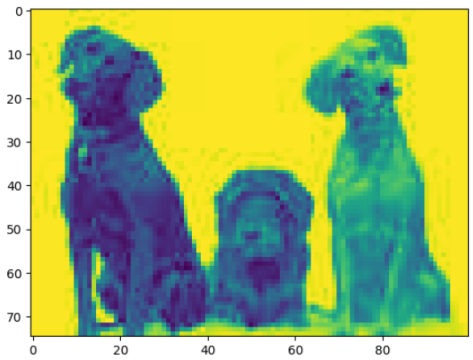

Figura 28: Input Image

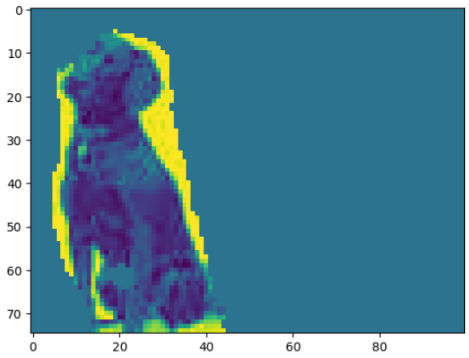

Figura 29: Result Image (using some seeds and labels)

8. **Conclusiones sobre el algoritmo Coordenadas Laplacianas** Se tiene que aunque se usaron métodos para optimizar el código como por ejemplo el uso de matrices dispersas, el algoritmo es muy complejo computacional mente, siendo útil solo para imágenes 2D y que estas tengan una resolución menor a 100px*100px, de lo contrario el tiempo y lo recursos que este puede tomar no serán viables. También se concluye que cuando este algoritmo logra terminar, el resultado es muy bueno en relación con lo que se espera, necesitando solo algunas pocas seeds y aún así logrando segmentar lo que se quiere. Por último, recalcar la importancia de crear buenas seeds, debido que las seeds tanto de Background como de Foreground son esenciales en la creación de la matriz Is, debido a que tanto las semillas de Background como de Foreground en esa matriz se transformarán en 1's y las celdas que no estén relacionadas con alguna seed tomarán valores de 0.

8.1. Ventajas

Precisión en Segmentación: Puede proporcionar segmentaciones muy precisas y detalladas.
Flexibilidad: Permite definir semillas y etiquetas, lo que ofrece gran control sobre la segmentación deseada.
Robustez: Utiliza la matriz de pesos y una métrica robusta (Laplaciana) para calcular la segmentación, lo que puede manejar bien variaciones en la intensidad de la imagen.

8.2. Desventajas

Computacionalmente Intensivo: El cálculo de la matriz de pesos y la solución del sistema de ecuaciones puede ser costoso en términos de tiempo y recursos cuando se requieren matrices moderadamente grandes, tanto que para imágenes médicas, se debe de realizar un subescalado para poder correr este algoritmo.
Sensibilidad a Parámetros: Requiere ajuste fino de parámetros como beta y las posiciones de las semillas para obtener resultados óptimos.
Complejidad de Implementación: La implementación y el ajuste de este tipo de algoritmo puede ser más complejo en comparación con métodos más simples.

9. Código de Coordenadas Laplacianas (3D para imágenes médicas)

9.1. Conclusiones

Al modificar el código para que acepte imágenes nifti (3D), se encuentra que este algoritmo no es factible, al necesitar demasiado almacenamiento. Para mitigar el problema del excesivo consumo de memoria, se encontró la "solución"de no trabajar con el archivo original, sino con una muestra del

mismo. Esto se hace usando un factor, el cual solo toma un dato cada "factor"veces. El factor mínimo que se tiene como factible es 5, el cual daña mucho la imagen. Cabe aclarar por último, que el factor de una imagen original es 1 (se cogen todos los datos cada 1 vez).

9.2. Ejemplo de muestras usando factor 5

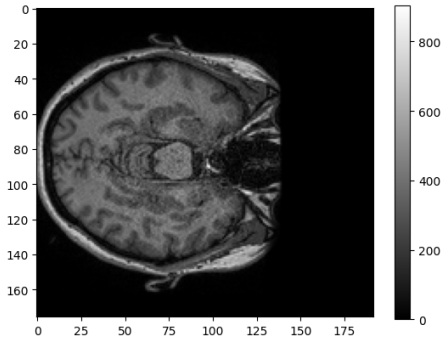

Figura 30: Factor 1

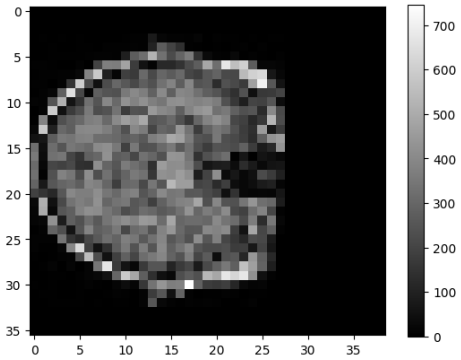

Figura 31: Factor 5

9.3. Código

```
import numpy as np
import scipy.sparse as sp
from scipy.sparse.linalg import factorized
import nibabel as nib
import matplotlib.pyplot as plt

def calculate_weights_3d(image, beta=0.1):
    depth, height, width = image.shape
    num_voxels = depth * height * width
    indices = np.arange(num_voxels).reshape(depth, height, width)
    W = sp.dok_matrix((num_voxels, num_voxels))

    diffs = [
        np.abs(np.diff(image, axis=0)),
        np.abs(np.diff(image, axis=1)),
        np.abs(np.diff(image, axis=2))
    ]
    sigma = np.max([np.max(diff) for diff in diffs])
    if sigma == 0:
        sigma = 1
    beta_scaled = -beta / (2 * sigma**2)

    for k in range(depth):
```

```
        for i in range(height):
            for j in range(width):
                index = indices[k, i, j]
                neighbor_offsets = [(-1, 0, 0), (1, 0, 0), (0, -1, 0), (0, 1, 0), (0, 0, -1), (0, 0, 1)]
                for dk, di, dj in neighbor_offsets:
                    nk, ni, nj = k + dk, i + di, j + dj
                    if 0 <= nk < depth and 0 <= ni < height and 0 <= nj < width:
                        neighbor_index = indices[nk, ni, nj]
                        weight = np.exp(beta_scaled * (image[k, i, j] - image[nk, ni, nj])**2)
                        W[index, neighbor_index] = weight

    return W.tocsr()

def segment_image_3d(image, seeds, labels, xB, xF, beta):
    depth, height, width = image.shape
    num_voxels = depth * height * width
    indices = np.arange(num_voxels).reshape(depth, height, width)

    W = calculate_weights_3d(image, beta)
    D = np.array(W.sum(axis=1)).flatten()
    L = sp.diags(D) - W
    L2 = L.dot(L)

    I_s = sp.lil_matrix((num_voxels, num_voxels))
    b = np.zeros(num_voxels)
    for (k, i, j), label in zip(seeds, labels):
        idx = indices[k, i, j]
        I_s[idx, idx] = 1
        b[idx] = xB if label == 'B' else xF

    A = I_s + L2
    print("Creacion del sistema")
    A = A.tocsc()  # Convert to CSC format for factorization
    print("Sistema convertido a csc")
    solve = factorized(A)
    print("Sistema factorizado")
    x = solve(b)
    print("Sistema resuelto")

    segmented_image = x.reshape((depth, height, width))
    return segmented_image

def apply_labels_3d(segmented_values, xB, xF):
    threshold = (xB + xF) / 2
    labels = (segmented_values >= threshold).astype(int)
    labels = labels * xB + (1 - labels) * xF
    return labels

# Load and subsample the image data
img = nib.load(os.path.join(my_drive_path,'file.nii'))
full_data = img.get_fdata()

# Subsampling the data
factor = 5  # Subsampling factor, choose based on your data size and memory constraints
sampled_data = full_data[::factor, ::factor, ::factor]

# Parameters and seeds setup
seeds = [(0, 0, 0), (0, 0, 1), (9, 9, 9), (9, 9, 8)]
labels = ['B', 'B', 'F', 'F']
xB = 100
xF = 255
beta = 1.0

# Adjust seed positions according to the subsampling factor
adjusted_seeds = [(k//factor, i//factor, j//factor) for k, i, j in seeds]

# Perform segmentation on the subsampled data
segmented_values = segment_image_3d(sampled_data, adjusted_seeds, labels, xB, xF, beta)
final_labels = apply_labels_3d(segmented_values, xB, xF)
```

9.4. Ventajas

Flexibilidad: Permite definir semillas y etiquetas, lo que ofrece gran control sobre la segmentación deseada.
Robustez: Utiliza la matriz de pesos y una métrica robusta (Laplaciana) para calcular la segmentación, lo que puede manejar bien variaciones en la intensidad de la imagen.

9.5. Desventajas

Computacionalmente Intensivo: El cálculo de la matriz de pesos y la solución del sistema de ecuaciones puede ser costoso en términos de tiempo y recursos cuando se requieren matrices moderadamente grandes, tanto que para imágenes médicas, se debe de realizar un subescalado para poder correr este algoritmo.
Sensibilidad a Parámetros: Requiere ajuste fino de parámetros como beta y las posiciones de las semillas para obtener resultados óptimos.
Complejidad de Implementación: La implementación y el ajuste de este tipo de algoritmo puede ser más complejo en comparación con métodos más simples.
Imprecisiones subsecuentes al subescalado: Al tener que subescalar la imágen médica para correr el algoritmo, se pierden demasiados detalles, resultando imposible usar esta implementación para algo más que segmentar el cerebro del fondo.

Referencias

Wallace Casaca, Luis Gustavo Nonato, and Gabriel Taubin. Laplacian coordinates for seeded image segmentation. In *2014 IEEE Conference on Computer Vision and Pattern Recognition*, pages 384–391, 2014. doi: 10.1109/CVPR.2014.56.