Universidad de Málaga

Imágenes Biomédicas

Práctica 3

Segmentación de imágenes

Autor:
Mario Pascual González

Profesores asignados:
Dr. Francisco Sendra Portero
Dr. Enrique Nava Baro
Dr. Ignacio Rodríguez Rodríguez

1. Segmentación

La segmentación desempeña un papel crucial en la interpretación y el análisis de imágenes biomédicas. Permitiendo dividir una imagen en múltiples segmentos o regiones, la segmentación facilita la identificación de áreas de interés, tales como tumores o vasos sanguíneos, lo que es crucial para diagnósticos más precisos y para la planificación de tratamientos eficaces.

En el caso de mi imagen asignada (Figura 1), la región de interés es el tejido óseo, el cual puede visualizarse en la imagen como la zona con niveles de grises más claros dentro de la estructura anatómica principal. En el histograma, esta zona corresponde a una meseta que se encuentra a continuación del tercer pico del histograma, abarcando un rango de valores desde ≈ 90 hasta ≈ 110 niveles de gris.

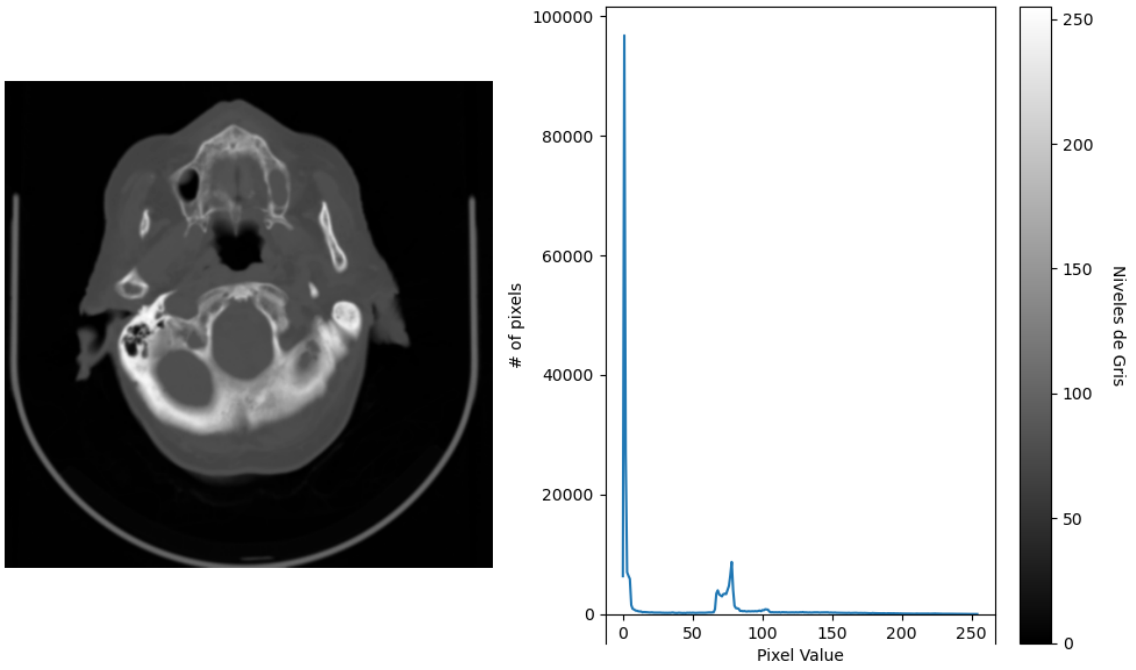

Figura 1: Mi imagen con su respectivo histograma de grises

Después de poder tener una toma de contacto con estos algoritmos he podido asimilar la vital necesidad de acompañar los resultados de cada algoritmo con una visualización completa.

Debido a esto, he implementado una función que permite visualizar **la imagen original** -junto con cambios que se puedan aplicar a esta como filtros paso bajo-, **la máscara** -generada durante la umbralización- y **la superposición de la máscara sobre la imagen original** -adquiriendo la máscara un color rojizo para poder visualizar los segmentos extraídos mediante la umbralización-.

1.1. Superposición de una máscara

En Python, una máscara es un tipo de dato `numpy.ndarray` bidimensional con una serie de píxeles marcados en blanco sobre un fondo negro homogéneo.

Debido a que he decidido colorear la máscara en rojo, el primer paso es crear tres dimensiones más en la imagen original, una para cada color **RGB**, de esto se puede encargar la función `cv2.cvtColor(image, cv2.COLOR_GRAY2BGR)`, aunque podría simplemente realizarse un `np.stack([im]*3, axis=2)`, pero intentaré usar otras librerías en este documento.

Seguido, identificamos las coordenadas de los píxeles de la máscara cuyo valor es 255, es decir, son píxeles activos. Finalmente, a los píxeles de la imagen original a color correspondientes a esas posiciones se asigna el color elegido (rojo) (ver Listing 1).

```
def overlay_mask_on_image(image, mask, color=(255, 0, 0)):
    color_image = cv2.cvtColor(image, cv2.COLOR_GRAY2BGR)
    active_pixels = (mask == 255)
    color_image[active_pixels] = color
    return color_image
```

Listing 1: Superposición de una máscara sobre una imagen

1.2. Visualización completa

Finalmente utilicé herramientas de visualización básicas de Python para componer una imagen con las características definidas previamente (ver Listing 2).

```
def visualize(image, mask, color=(255, 0, 0)):
    plt.figure(figsize=(15, 5))
    plt.subplot(1, 3, 1)
    plt.imshow(image, cmap='gray')
    plt.title('Original Image')
    plt.axis('off')

    plt.subplot(1, 3, 2)
    plt.imshow(mask, cmap='gray')
    plt.title('Mask')
    plt.axis('off')

    overlay = overlay_mask_on_image(image, mask, color=color)
    plt.subplot(1, 3, 3)
    plt.imshow(overlay, cmap='gray')
    plt.title('Overlayed mask on image')
    plt.axis('off')

    plt.show()
```

Listing 2: Composición de imagen, máscara y máscara superpuesta a imagen

1.3. Preprocesamiento de imagen

Algunos apartados de esta práctica requieren aplicar un filtro paso bajo para poder reducir el ruido de una imagen. Con la finalidad de darle un uso al código de mi práctica anterior he definido una clase `Preprocess` que incorpora tres diferentes métodos:

Un **constructor** (Listing 3) que almacena una imagen e inicializa un filtro gaussiano a `None`.

```
def __init__(self, image):
        """
        Inicializamos la clase pasando una imagen
        :param image: Imagen (ndarray 2D)
        """
        self.image = image
        self.gaussian = None
```

Listing 3: Constructor de la clase Preprocess

Un método para crear un **filtro Gaussiano** (Listing 4) reciclado de la práctica anterior

```
def gaussian_filter(self, kernel_size=3, sigma=1):
        offset = int(kernel_size // 2)
        x, y = np.mgrid[-offset:offset + 1, -offset:offset + 1]
        self.gaussian = (1 / (2 * np.pi * sigma ** 2)) *
                        np.exp(-(x ** 2 + y ** 2) / (2 * sigma ** 2))
```

Listing 4: Método para crear un filtro Gaussiano

Un método para **aplicar una convolución usando OpenCV** (Listing 5), ya que mi definición del `im2col` era correcta, pero quería probar con OpenCV.

```
def convolve(self, kernel):
    return cv2.filter2D(self.image, -1, kernel)
```

Listing 5: Método para aplicar la operación de convolución entre un kernel y una imagen

Finalmente, un método para **normalizar la imagen a un tipo de datos específico** (Listing 6)

```
def normalize(self, dtype=None):
        max = np.max(self.image)
        min = np.min(self.image)
        self.image = (self.image - min)/(max - min)
        if dtype is not None:
            max_val = np.iinfo(dtype).max
            self.image = (self.image * max_val).astype(dtype)
```

Listing 6: Método para normalizar la imagen

Se procede entonces a explicar, implementar, y analizar los diferentes algoritmos de umbralización propuestos en el documento de guía de la práctica.

2. Umbralización

La umbralización es una técnica de procesamiento de imágenes que se utiliza para segmentar una imagen en diferentes regiones, con el objetivo de resaltar áreas de interés y facilitar su posterior análisis.

Consiste en definir un valor umbral que sirve como criterio para clasificar los píxeles de una imagen en dos o más categorías, generalmente correspondientes al objeto de interés y al fondo.

Al aplicar este valor umbral, se crean imágenes binarias o etiquetadas que son más fáciles de interpretar y analizar.

En este trabajo se describirán dos principales tipos de umbralización del histograma de la imagen: *umbralización manual* y *umbralización automática*.

2.1. Umbralización manual

La umbralización manual es uno de los métodos más simples y directos para segmentar imágenes. En este enfoque, se selecciona manualmente un valor umbral para distinguir entre las diferentes regiones de una imagen, en el caso de mi imagen, el tejido óseo (objeto) y el fondo (resto).

Aunque la umbralización manual puede ser efectiva en ciertos contextos donde las condiciones de iluminación y contraste son consistentes, su precisión depende en gran medida del juicio del operador, y no es óptima para imágenes con variaciones de intensidad como por ejemplo las Imágenes por Resonancia Mágnética (MRI), que se centran en tejidos blandos con intensidades muy parecidas.

Implementación

Para implementar este tipo de umbralización he comenzado por observar qué región del histograma pertenece al objeto de interés. Una vez he identificado que esta región se encuentra aproximadamente entre las intensidades 85 y 200 (correspondientes a tonos de gris claros), he implementado la función 7, la cual toma como parámetros de entrada la imagen, un umbral inferior y, opcionalmente, un umbral superior.

```
def umbralizacion_manual(img, lower_bound, upper_bound=255):
    thresholded_image = np.zeros_like(img)
    thresholded_image[(img >= lower_bound) &
                      (img <= upper_bound)] = 255
    return thresholded_image
```

Listing 7: Umbralización manual

Resultados

Después he probar diferentes ventanas, he podido extraer una umbralización manual prácticamente óptima (ver Figura 2). El tejido óseo ha quedado -desde mi ojo inexperto- recogido en su totalidad dentro de la máscara, pudiendo haberse pasado por algo algunas partes correspondientes a la mandíbula (ver Listing 8).

Este riesgo que se ha aceptado relacionado con dejar fuera parte de la mandíbula está relacionado con mi intento por dejar fuera de la máscara la camilla del paciente.

```
masked_im1 = umbralizacion_manual(pixel_data_uint8, 110, upper_bound=250)
visualize(pixel_data_uint8, masked_im1)
```

Listing 8: Valores de la umbralización manual

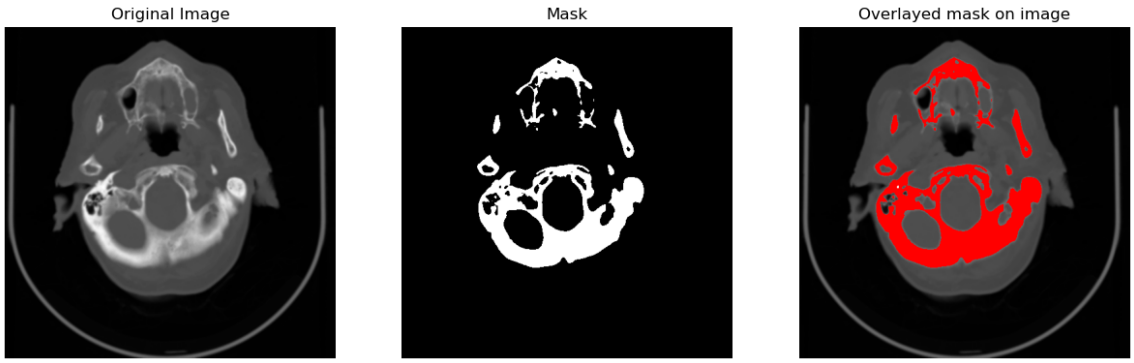

Figura 2: Umbralización manual

Conclusión

La umbralización manual, como se ha podido observar, es una herramienta potente para la segmentación de imágenes, ya que es fácil de implementar y suele expulsar buenos resultados.

Por otra parte, tiene un gran inconveniente, y es que debe ser realizada por alguien experto en este tipo de imágenes, ya que de otra forma, parte del objeto de interés podría quedar fuera de la máscara. Además, este tipo de umbralizaciones no suele responder bien a imágenes con intensidades muy similares.

2.2. Umbralización automática o iterativa

A diferencia de la umbralización manual, la umbralización automática busca determinar de forma adaptativa el valor umbral óptimo para segmentar una imagen. Esto se realiza a través de algoritmos que analizan las propiedades estadísticas de la imagen, como el histograma de intensidades, o que toman en cuenta la variación espacial de las intensidades de los píxeles.

Durante esta sección explicaré las diferentes formas de resolver este problema que he abordado.

2.2.1. Algoritmo de umbralización de las trasparencias

Este tipo de algoritmos caen dentro de la categoría de **umbralización global**, ya que opera con todos los píxeles de la imagen en conjunto. Este algoritmo fue propuesto por el profesor Enrique Nava durante una de las clases teóricas de la asignatura Imágenes Biomédicas. Para explicar este algoritmo recomiendo el refuerzo visual de la Figura 3.

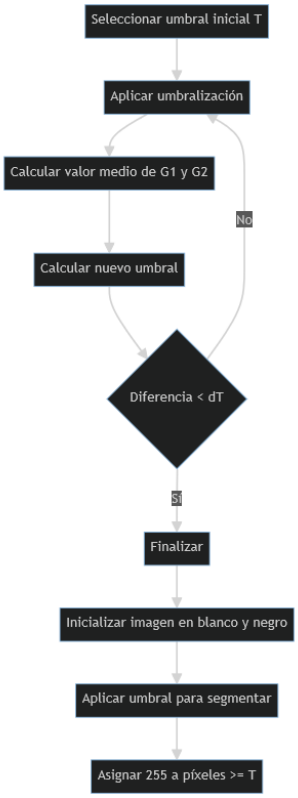

Figura 3: Diagrama de flujo para este algoritmo

Implementación

```
def iterative_thresholding(image, dt=1e-5):
    t = np.mean(image)
    while True:
        g1 = image[image < t]
        g2 = image[image >= t]
        m1 = np.mean(g1)
        m2 = np.mean(g2)
        new_t = (m1 + m2) / 2
        if abs(new_t - t) < dt:
            break
        t = new_t
    thresholded_image = np.zeros_like(image)
    thresholded_image[image >= t] = 255
    return thresholded_image, t
```

Listing 9: Función de umbralización iterativa

Descripción general

Se inicia con un umbral que es la media de los valores de intensidad de la imagen. A continuación, divide iterativamente los píxeles de la imagen en dos grupos, basándose en este umbral, y calcula la media de intensidad para cada grupo. El umbral se ajusta entonces a la media de estas dos medias de intensidad. El proceso se repite hasta que el cambio en el umbral entre iteraciones sea menor que un valor dado (`dt`).

En el bucle `while`, se divide la imagen en dos grupos, `g1` y `g2`, basándose en el umbral actual `t`. Luego se calculan las medias `m1` y `m2` de estos grupos. Se promedian estos valores medios para obtener un nuevo umbral `new_t`. Si la diferencia entre el umbral actual y el nuevo umbral es menor que `dt`, el bucle se detiene. Si no, el umbral se actualiza y el proceso continúa.

Una vez determinado el umbral óptimo, se inicializa una imagen `thresholded_image` con valores de intensidad cero (negro). Se asigna un valor de 255 (blanco) a los píxeles en la imagen original que tienen un valor de intensidad mayor o igual al umbral. Finalmente, la imagen segmentada y el umbral se devuelven como salida.

Resultados

Como se puede observar en la Figura 4, los resultados de este algoritmo distan mucho del resultado óptimo. Debido a que el histograma consta de dos principales "picos", uno que corresponde al fondo, y otro que corresponde a la estructura anatómica principal, el algoritmo encuentra un umbral perfecto para separar el fondo de la estructura anatómica principal, que es la cabeza en su totalidad.

Un paso que podríamos dar ahora es volver a aplicar el algoritmo, pero solo a los píxeles que pertenezcan a la región delimitada por esta máscara -es decir, a la cabeza del paciente-. Esto equivaldría a hacer un "zoom.[en] el histograma a las intensidades de grises que no corresponden al fondo.

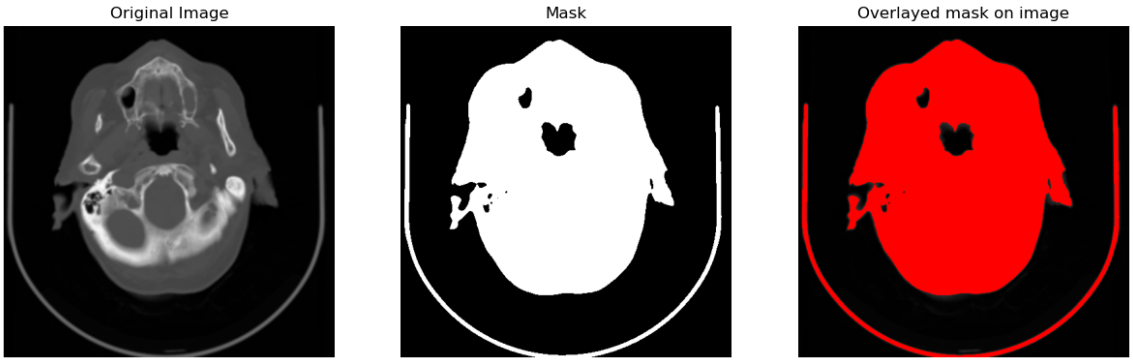

Figura 4: Umbralización iterativa, algoritmo de transparencias

Para proceder con este paso, solo he agregado un condicional al principio de la función, este condicional define el subconjunto de píxeles de la imagen pasada como argumento que pertenecen a la máscara pasada también como argumento.

```
def iterative_thresholding_with_mask(image, mask, dt=1e-5):
    image_region = image[mask == 255]

    mascara, t = iterative_thresholding(image_region)

    thresholded_image = np.zeros_like(image)
    thresholded_image[mask == 255] = (image[mask == 255] >= t) * 255
    return thresholded_image, t
```

Listing 10: Iterative thresholding aplicado a la máscara anteriormente obtenida

Los resultados de esta iteración, mostrados en la Figura 5, muestran una significativa mejora en los resultados, teniendo una máscara muy parecida a la de la umbralización manual. Estos resultados son tan buenos ya que el algoritmo consigue distinguir entre dos diferentes intensidades dentro de la cabeza del paciente: Una que corresponde con el músculo, tejido conectivo, etc. -de un gris más oscuro- y otra correspondiente al tejido óseo -de un gris más claro-. Pudiendo encontrar el umbral que separa estas dos intensidades en el histograma.

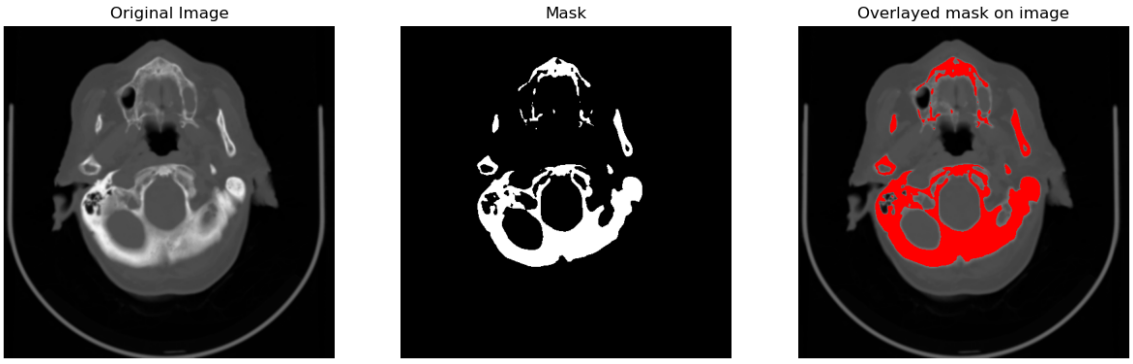

Figura 5: Umbralización iterativa, algoritmo de transparencias (II)

Después de haber obtenido estos resultados me pregunté: *¿Cómo podría hacerse esto más automático?*, al fin y al cabo, este es el objetivo de la umbralización automática. Finalmente me propuse programar un algoritmo basado en este, pero que fuese capaz de generalizar para un número N de regiones pre-seleccionadas -cosa que realmente lo convertiría en un algoritmo semiautomático, pero debido a que el nivel de conocimiento del campo es mucho menos exigente que en la segmentación automática, se procederá con este enfoque-.

Mi algoritmo: 'K-Umbrales'; Un enforque más general

```
def k_umbrales(image, N, dt=1e-5):
    # Paso 1
    ts = np.linspace(np.min(image), np.max(image), N+1)[1:-1]
    means = np.zeros(N)
    new_ts = np.zeros_like(ts)

    while True:
        # Paso 2
        segments = [image[(image >= (ts[i-1]
                           if i > 0
                           else np.min(image)))
                           & (image < ts[i])]
                              for i in range(N-1)]
        segments.append(image[image >= ts[-1]])
        # Paso 3
        for i in range(N):
            means[i] = np.mean(segments[i])
        # Paso 4
        for i in range(N-1):
            new_ts[i] = (means[i] + means[i+1]) / 2
        if np.all(np.abs(new_ts - ts) < dt):
            break

        ts = new_ts.copy()
    # Paso 5
    thresholded_image = np.zeros_like(image)
    for i in range(N-1):
        thresholded_image[(image >= (ts[i-1]
                           if i > 0
                           else np.min(image)))
                           & (image < ts[i])]
                           = 255 * (i+1) // N
    thresholded_image[image >= ts[-1]] = 255

    return thresholded_image, ts
```

Listing 11: Mi implementación de un iterative thresholding más general

Vamos a desglosar cada uno de los pasos del Listing 11:

Paso 1: Inicializar N-1 umbrales

En este paso inicializo un conjunto de `N-1` umbrales (`ts`) que son distribuidos uniformemente entre el valor mínimo y máximo de las intensidades de la imagen. Estos umbrales dividirán la imagen en `N` segmentos o regiones de intensidades.

Paso 2: Segmentar la imagen en N grupos

Aquí, segmento la imagen en `N` grupos diferentes basados en los `N-1` umbrales definidos en el paso anterior. Cada segmento consiste en un conjunto de píxeles cuyas intensidades están entre dos umbrales consecutivos. Por ejemplo, el primer segmento consiste en píxeles con intensidades entre el mínimo de la imagen y el primer umbral. El último segmento consiste en píxeles con intensidades que son mayores o iguales al último umbral.

Paso 3: Calcular la media de cada grupo

Una vez que la imagen está segmentada en diferentes grupos, este paso calcula la intensidad media de los píxeles dentro de cada segmento. Las medias se almacenan en el vector `means`.

Paso 4: Actualizar cada umbral

Cada umbral se actualiza basándose en las intensidades medias de los segmentos adyacentes. Específicamente, un umbral se actualiza para que sea el promedio de las intensidades medias de sus dos segmentos vecinos. Esta actualización permite que los umbrales se adapten y refinen basándose en las características de intensidad de la imagen.

Comprobar si la actualización de todos los umbrales es menor que dT

Si la diferencia entre los valores actuales de los umbrales y sus valores actualizados es menor que un valor `dt` para todos los umbrales, el algoritmo se detiene. Si no, los umbrales se actualizan y el proceso se repite desde el Paso 2.

Paso 5: Umbralizar la imagen en función de los umbrales actualizados

Una vez que los umbrales cumplen el criterio de convergencia, este paso umbraliza la imagen original basándose en estos umbrales. Los píxeles en la imagen resultante son asignados a diferentes niveles de intensidad basados en qué segmento de intensidad pertenecen en la imagen original. Por ejemplo, si hay 3 segmentos, los píxeles del primer segmento podrían ser asignados a una intensidad de 85 $\frac{255}{3}$, los del segundo segmento a 170 $\frac{2\times255}{3}$ y los del tercer segmento a 255.

Para implementar este algoritmo me he basado mucho en el algoritmo de Machine Learn (ML) **K-Means**, estudiado en la asignatura *Intelligent Systems*, el cual ya había programado a bajo nivel previamente.

K-umbrales: Resultados

Como se puede observar en la Figura 6, los resultados de mi algoritmo para un valor $N = 3$ son idénticos a los resultados del algoritmo de umbralización manual (Figura 2), sin embargo, el algoritmo *K-Umbrales* ofrece una flexibilidad que el otro no.

Si nos fijamos en la máscara, y teniendo en cuenta la explicación de la página anterior, podemos visualizar 3 niveles de grises, uno correspondiente al fondo, otro correspondiente al tejido musculoso, conectivo, etc. de la cabeza y el último, el más claro, correspondiente al tejido óseo. Que la superposición de la máscara haya coincidido con el tejido óseo es porque al definir la función `overlay_mask_on_image` (Listing 1), se estableció que se marcasen en rojo los píxeles que en la máscara fuesen igual a 255 -el tono de gris más blanco-. Sin embargo, al contener más grises en la máscara, podríamos ajustar esta función para mostrar otro umbral detectado por *K-Umbrales*, como la cabeza o el fondo incluso.

Figura 6: Resultados del algoritmo *K-Umbrales* con $K = 3$

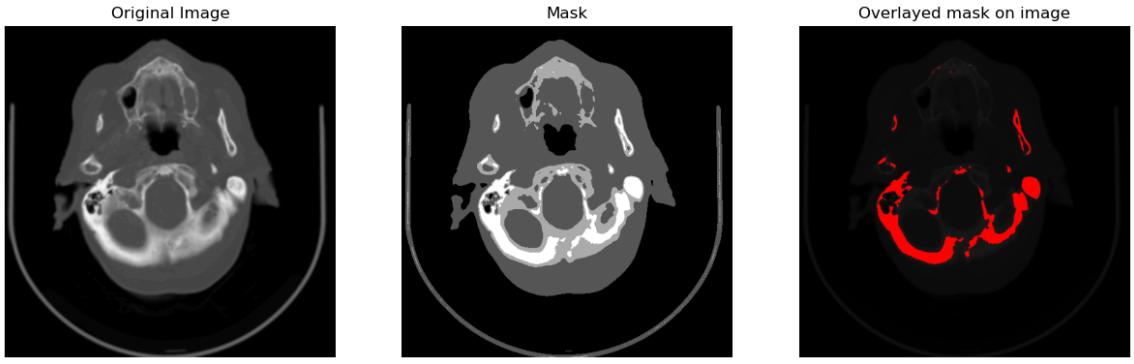

Figura 7: Resultados del algoritmo *K-Umbrales* con $K = 4$

Conclusion

Estos algoritmos de umbralización iterativa parecen proveer buenos resultados, una de sus ventajas es su baja complejidad a la hora de comprenderlos, ya que se basan en la segmentación del histograma dependiendo de una serie de umbrales y no incorpora métricas estadísticas más complicadas de la media.

Por otra parte, aunque el conocimiento para aplicarlos es bajo, se deben tener unas nociones mínimas para elegir el número de umbrales. No parece que respondan demasiado buen a objetos de interés con poco píxeles en la imagen o con tonos de gris similares a otras zonas -cosa que ocurre en los MRI-. En el histograma este tipo de imágenes tendría una representación demasiado uniforme, siendo complicado umbralizar el histograma dependiendo de sus "picos".

2.2.2. Umbralización adaptativa

Un segundo enfoque que decidí adaptar para resolver este problema de manera iterativa fue mediante el uso de la *umbralización adaptativa*.

La umbralización adaptativa es un método avanzado de segmentación de imágenes que ajusta el valor umbral de manera local en lugar de aplicar un único valor umbral a toda la imagen. El objetivo es adaptar el umbral a las variaciones de iluminación y contraste que puedan existir en diferentes parches de la imagen.

Para esta sección he decidido implementar un **algoritmo de umbralización adaptativa gaussiana**, donde el valor umbral se calcula como una suma ponderada de los valores de los píxeles vecinos, utilizando una función gaussiana como peso.

Implementación

Para implementar de manera ordenada y efectiva al umbralización adaptativa gaussiana he decidido crear una clase en Python '`CustomGaussianThresholding`'. Esta clase se ha basado bastante en el método 4 de la clase `Preprocess`. Se incluyen a continuación las clases que he implementado:

Constructor

```
def __init__(self, image, sigma=1, window_size=11, c=2):
    self.image = image
    self.window_size = window_size
    self.c = c
    self.sigma = sigma
    self.mask = self.create_gaussian_mask()
```

Listing 12: Constructor de la clase `CustomGaussianThresholding`

He implementado este constructor con los siguientes parámetros:

- image: matriz numpy 2D que representa una imagen en escala de grises.
- window_size: entero impar que representa el tamaño de la ventana gaussiana.
- C: Constante restada del umbral calculado.

El significado del valor C se explicará en detalle más adelante.

Se implementa un kernel gaussiano, **create_gaussian_mask**. La clase `Preprocess` ya incluía una definición del kernel Gaussiano (ver Listing 4), sin embargo, para evitar dependencias circulares entre las librerías, he decidio volver a definirlo para esta clase.

Finalmente, se implementa la operación de la umbralización adaptativa gaussiana con el método `apply_threshold`.

```
def apply_threshold(self):
    result = np.zeros_like(self.image)
    padded_image = np.pad(self.image,
                          (self.window_size // 2,
                          self.window_size // 2),
                          mode='constant')

    for i in range(self.image.shape[0]):
        for j in range(self.image.shape[1]):
            window = padded_image[i:i+self.window_size,
                                  j:j+self.window_size]
            threshold = np.sum(window * self.mask) - self.c
            result[i, j] = 255 if self.image[i, j]
                           > threshold else 0

    return result
```

Listing 13: Operación de la umbralización adaptativa gaussiana

El umbral que define a una sección particular de la imagen del mismo tamaño que el kernel gaussiano implementado se puede observar en 8. Como se puede observar es la convolución entre el parche y el kernel menos un valor C, el cual podemos elegir nosotros con la finalidad de penalizar valores demasiado altos.

$$T(x,y) = \sum_{i=-a}^{a} \sum_{j=-b}^{b} W(i,j) \cdot I(x+i, y+j) - C$$

$$W(i,j) = G(i,j;\sigma) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{i^2+j^2}{2\sigma^2}\right)$$

Figura 8: Cálculo del threshold adaptativo

Finalmente, se decide la intensidad del píxel dentral del parche de la imagen comparando su intensidad original con la calculada al aplicar la operación definida en 8. Este threshold entonces queda definido en 9.

$$O(x,y) = \begin{cases} 1, & \text{if } I(x,y) \geq T(x,y) \\ 0, & \text{if } I(x,y) < T(x,y) \end{cases}$$

Figura 9: Aplicación de la umbralización: Diferentes casos

Resultados

Como podemos observar en la imagen 10, para los valores `window_size=11, sigma=6, c=3`, los resultados son los esperados. La máscara refleja los cambios en intensidad en la imagen con una presición igual al tamaño de ventana del kernel gaussiano, aunque pueda parecer que un valor de C algo no nos interesa para nuestra región de interés -ya que C penaliza las zonas brillantes de la imagen y el tejido oseo es brillante-, C también penaliza el ruido de la imagen.

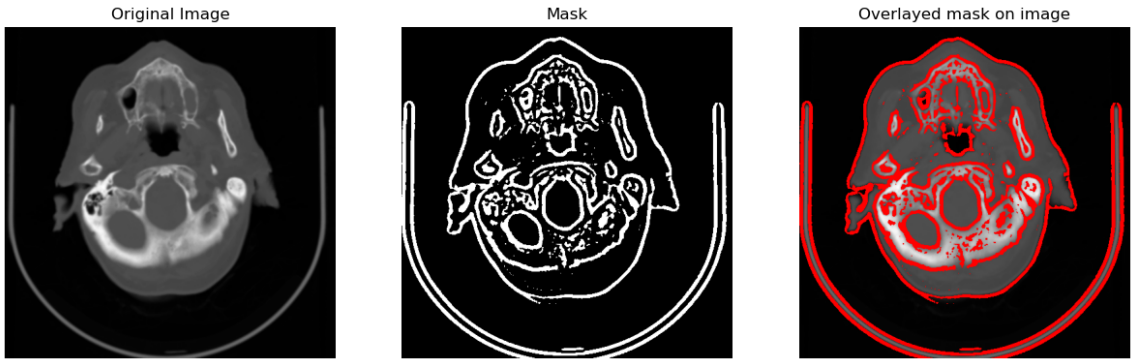

Figura 10: Resultados de la umbralización adaptativa

Para poder rellenar los huecos he decidido implementar un concepto de morfología matemática visto recientemente en clase, el *cierre*, que es una dilatación seguida de una erosión (ver Figura 11). Sin embargo, como este tema no es el principal objetivo de la práctica, he decidido implementarlo utilizando la librería OpenCV[1] (Listing 14).

```
def morphological_closing(image, kernel_size):
    # Crear el kernel
    kernel = np.ones((kernel_size, kernel_size), np.uint8)
    # Aplicar la operación morfológica de cierre
    closed_img = cv2.morphologyEx(image, cv2.MORPH_CLOSE, kernel)
    return closed_img
```

Listing 14: Operación de cierre

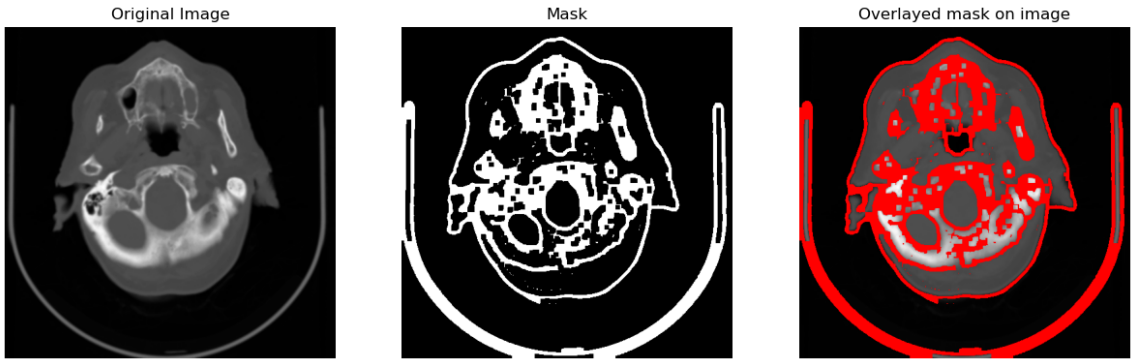

Figura 11: Umbralización adaptativa + Operación de cierre morfológico

Conclusión

Como se puede observar en la imagen 11, los resultados de la umbralización adaptativa no satisfacen al completo el principal objetivo de la práctica. Si bien es verdad que gran parte de la estructura ósea queda marcada, también se incluyen otro tipo de estructuras que no están relacionadas.

2.2.3. Filtrado paso bajo y algoritmo de umbralización iterativa

Cumpliendo con los puntos estipulados en el documento de la práctica, se añade una sección en la que se se va a realizar **un filtrado paso bajo -Gaussiano- para reducir el ruido y una umbralización iterativa**. Para esto se utilizará la librería `Preprocess`, descrita al principio del documento.

Implementación

```
from image_preproc import Preprocess

prep = Preprocess(pixel_data_uint8)
prep.normalize(dtype=np.uint8)
prep.gaussian_filter(kernel_size=3, sigma=1)
denoised = prep.convolve(prep.gaussian)
masked_im4, umbral = k_umbrales(denoised, 3)
visualize(pixel_data_uint8, masked_im4)
```

Listing 15: Código usado para el filtrado y umbralización iterativa

Resultados

Como se puede observar en la imagen 12, los resultados han empeorado ligeramente. Este resultado es lógico, ya que al aplicar un filtrado paso bajo gaussiano estamos realizando un promedio con pesos gaussianos de cada zona de la imagen, atenuando los picos del histograma y dificultando que el algoritmo pueda encontrar los umbrales que mejor separan los contenedores con mayor cantidad de muestras de intensidad en el histograma.

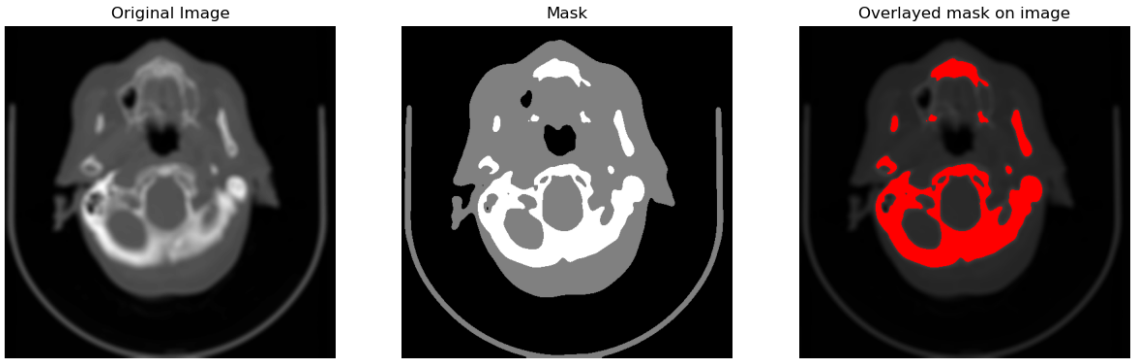

Figura 12: Filtrado paso bajo gaussiano + Umbralización adaptativa

Conclusión

Parece que un filtrado paso bajo no es la mejor opción para los algoritmos de umbralización basados en al histograma, ya que pueden reducir el ruido pero también atenúan las diferencias entre picos del histograma. Como en todos los algoritmos, puede ser útil en algunos casos, y la clave sería encontrar el equilibrio entre la cantidad de ruido reducido y las zonas que la umbralización iterativa pueden saltarse en el tejido objetivo.

3. Simple Linear Iterative Clustering (SLIC)

El algoritmo SLIC es un método de segmentación de imágenes que divide una imagen en segmentos homogéneos llamados superpíxeles.

Este algoritmo inicia seleccionando centros de superpíxeles distribuidos uniformemente en la imagen, la distancia entre los centros se aproxima mediante $S = \sqrt{N/K}$, siendo N el número de píxeles en la imagen y K el número de superpíxeles deseados. El número N de superpíxeles se considera un hiperparámetro que se debe optimizar.

A continuación, cada píxel de la imagen se asigna al centro de superpíxel más cercano según una métrica de distancia que combina la similitud en el espacio de color como la proximidad espacial (ver Ecuación 13).

$$D = \sqrt{\left(\frac{d_c}{m}\right)^2 + (d_s)^2}$$

Figura 13: Ecuación para el cálculo de la distancia de un píxel al centro del superpíxel. Aquí, d_c es la distancia en el espacio de color, d_s es la distancia espacial entre el píxel y el centro del superpíxel, y m es un parámetro que controla la importancia relativa de la similitud de color frente a la proximidad espacial

Los centros de los superpíxeles se actualizan tomando el promedio de todas las posiciones y colores de los píxeles asignados a cada superpíxel.

El algoritmo itera a través de los pasos de asignación y actualización hasta que los centros de los superpíxeles convergen, resultando en una segmentación final en superpíxeles que son coherentes tanto en el espacio de color como en la ubicación espacial.

Implementación

Para facilitar la implementación del algoritmo y centrarnos sobre todo en el ajuste del hiperparámetro N -número de superpíxeles-, se utilizará la implementación de la librería `skimage`[2].

```
from skimage.segmentation import slic
from skimage.color import label2rgb
```

Listing 16: Clases usadas de la librería `skimage`

La clase `slic` del módulo `skimage.segmentation` implementa el algoritmo SLIC para la segmentación de superpíxeles en imágenes. Esta clase toma una imagen de entrada y varios parámetros como el número de superpíxeles deseados -N-, la compactación y el parámetro sigma para el preprocesamiento gaussiano. Devuelve una imagen donde cada valor representa un superpíxel en la imagen original.

Por otro lado, la función `label2rgb` del módulo `skimage.color` se utiliza para convertir una imagen etiquetada -como la que devuelve `slic` con los N superpíxeles- en una imagen RGB para fines de visualización. Esta función toma una imagen etiquetada y la imagen original como entradas y aplica un mapeo de color para generar una imagen RGB donde cada superpíxel tiene un color único.

Ajustando el número de superpíxeles, N

Para poder visualizar cómo el número de superpíxeles afecta a la imagen de entrada, y poder tratar con estos datos he implementado la función mostrada en el Listing

```
def slic_images(image, n_segments_list):
    segmented_images = {}
    for n_segments in n_segments_list:
        segments = slic(image, n_segments=n_segments, sigma=1)
        segmented_image = label2rgb(segments, image, kind='avg')
        segmented_images[n_segments] = segmented_image
    return segmented_images
```

Listing 17: Guardado de información de las imágenes para diferentes valores de N

Con una implementación básica usando la librería `matplotlib`, podemos visualizar las imágenes para cada valor N (ver Figura 14).

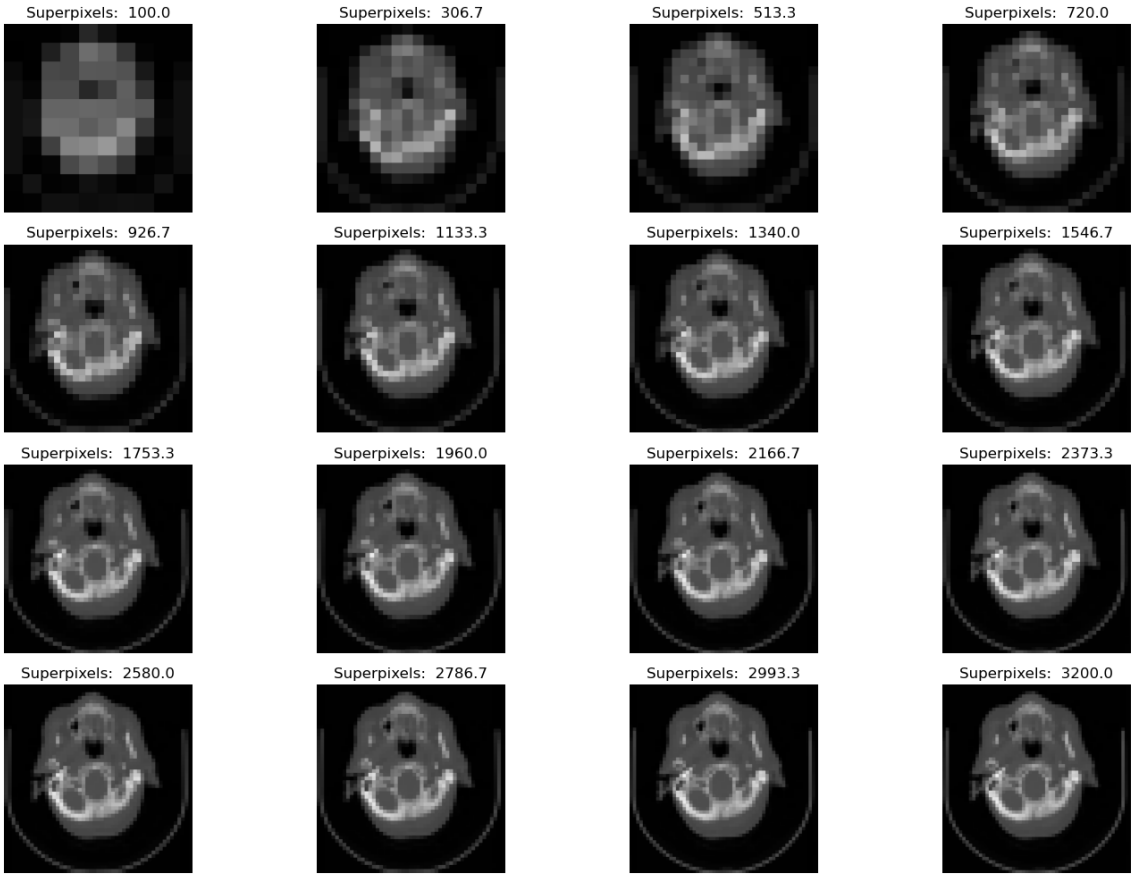

Figura 14: Visualización de las imágenes generadas para cada valor de N

Como podemos observar en la figura 14, un menor número de superpíxeles hace que a figura se deba dividir en secciones S de mayor tamaño, sintetizando de manera más bruta las características de color y textura de la imagen.

Por otra parte, un mayor número de superpíxeles hace que las secciones$S$ sean más pequeñas, pudiendo capturar de una manera más fina las características de cada sección. Esto tiene dos contrapartes, la primera es que esto conlleva un mayor coste computacional, la segunda es que si se aumenta demasiado el número de superpíxeles el algoritmo no conseguirá generalizar de manera efectiva la imagen, llegando a poder ser prácticamente igual que la imagen original.

El principal objetivo entonces es encontrar un buen compromiso entre el número de superpíxeles, N, mientras aseguramos generalizar lo suficiente la información de la imagen.

Con la finalidad de **cuantificar la semejanza entre superpíxeles**, he decidido programar una función (Listing 18) para calcular la varianza entre todos estos. El comportamiento esperado es que para un número bajo de superpíxeles haya una varianza alta, ya que todos los superpíxeles deben tener valores de intensidad de gris únicos y diferentes. A medida que la cantidad de superpíxeles aumenta, la varianza debería disminuir, esto es debido a que las intensidades de gris de los superpíxeles, al ser únicas y haber una gran cantidad, cada vez son similares para tejidos iguales.

```
def calculate_variance_of_superpixels(image, labels):
    unique_labels = np.unique(labels)
    variances = []

    for label in unique_labels:
        mask = (labels == label)
        superpixel_values = image[mask]
        variance = np.var(superpixel_values)
        variances.append(variance)

    return np.mean(variances)

# Main

# Inicializamos un diccionario de varianzas
average_variances = {}
# Almacenamos la varianza para cada segmento con
# la finalidad de visualizarlo
for n_segments, segmented_image in segmented_images.items():
    segments = slic(pixel_data_uint8, n_segments=n_segments, sigma=1)
    average_variance =
            calculate_variance_of_superpixels(pixel_data_uint8,
                                              segments)
    average_variances[n_segments] = average_variance
```

Listing 18: Cálculo de la varianza para cada imagen con un número N de superpíxeles asociado

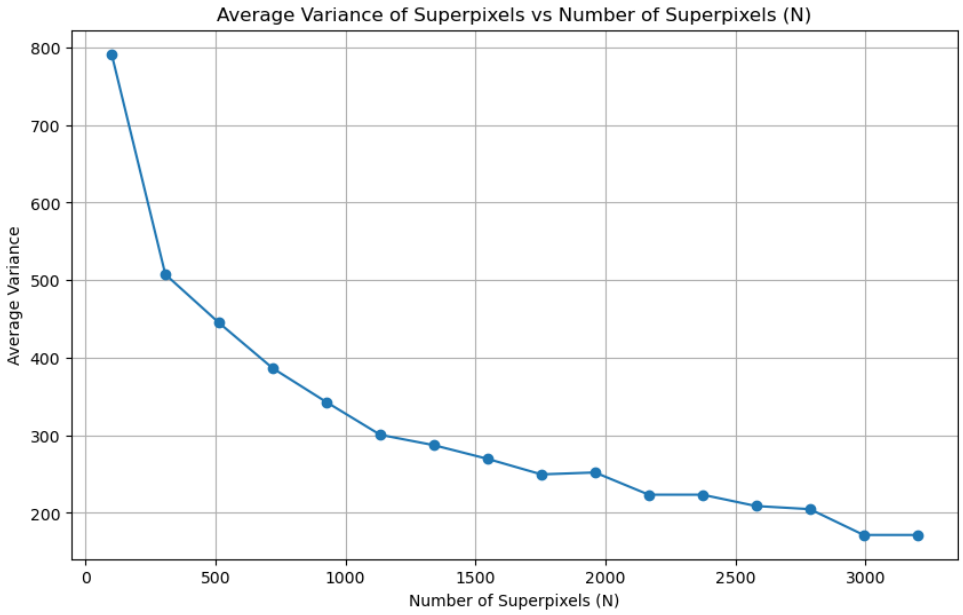

Figura 15: Variación de la varianza para cada imagen con un número N de superpíxeles asociado

Como se puede observar en la figura 15, el comportamiento real de la varianza es el esperado. La varianza decae de forma exponencial para saltos en valores de superpíxeles bajos, a partir de un valor, empieza a estabilizarse. Después de tener en cuenta todas estas consideraciones y observar la imagen 15, he decidido que utilizaré un número de superpíxeles $N = 1750$.

Debido a que esta práctica trata sobre **umbralización**, he decidido programar una pequeña función para encontrar la Region Of Interest (ROI) en la imagen con $N = 1750$ superpíxeles y usarlo como máscara sobre la imagen original.

```python
def identify_roi(image, segments, intensity_threshold):
    unique_labels = np.unique(segments)
    roi_mask = np.zeros_like(image, dtype=np.uint8)

    for label in unique_labels:
        mask = (segments == label)
        superpixel_values = image[mask]
        mean_intensity = np.mean(superpixel_values)

        if mean_intensity > intensity_threshold:
            roi_mask[mask] = 255

    return roi_mask
```

Listing 19: Cálculo sencillo de la Region Of Interest dado un umbral

La función 19 identifica las ROI en una imagen basándose en nuestra segmentación previa en superpíxeles -representada por el argumento `segments`-.

La función utiliza un umbral de intensidad, `intensity_threshold`, para determinar qué superpíxeles se consideran como parte de la ROI. Para cada superpíxel único en `segments`, esta función calcula la intensidad media de los píxeles dentro de ese superpíxel. Si esta intensidad media supera el umbral establecido, todos los píxeles dentro de ese superpíxel se marcan como parte de la ROI en una máscara binaria -`roi_mask`-.

Resultados

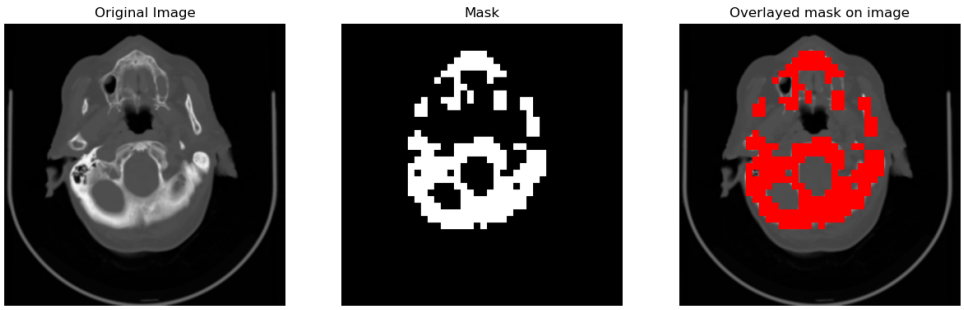

Figura 16: Resultado de aplicar una máscara basada en 1750 superpíxeles con un umbral ROI de 90

Conclusion

Como se puede observar en la imagen 16, parece que este método ha dado buenos resultados a la hora de marcar las zonas pertenecientes al tejido óseo, sin embargo, lo hace de una manera poco exacta.

Esto podría ajustarse aumentar el número de superpíxeles dedicados a la creación de la máscara, intentando como siempre mantener el compromiso entre carga computacional y no usar un número tan elevado de superpíxeles como para que la umbralización con ROI sea demasiado similar a una umbralización manual.

4. Conclusión final

A lo largo de este documento he tenido la oportunidad de explorar diferentes algoritmos de umbralización de imagen biomédica y he podido enfrentarme al reto de programar uno desde cero que cumpliese con los requisitos preestablecidos.

Si bien ya se han comentado las ventajas y desventajas de cada algoritmo. En mi opinión, si se quiere adoptar un enfoque más general, creo que se debe optar por un enfoque relacionado con los superpíxeles o la umbralización adaptativa. Por otra parte, si se tiene experiencia médica, una umbralización manual podría ser una buena opción.

Nota del Autor:

Me gustaría aclarar que el código y todo el contenido asociado presentado en este documento son de mi propia autoría y han sido creados con el propósito de cumplir con una tarea académica. Este trabajo es el resultado de mi propio entendimiento y esfuerzos en el contexto de los requisitos de la asignación.

Código Fuente

El código fuente y los recursos adicionales para este proyecto están disponibles en mi repositorio de GitHub. Puedes acceder a él a través del siguiente enlace: GitHub Repository.

Mario Pascual González

31 de octubre de 2023

Referencias

[1] O. Team, *OpenCV (Open Source Computer Vision Library) Documentation*, OpenCV.org, 2022. [Online]. Available: https://docs.opencv.org/master/

[2] scikit-image developers, *scikit-image (skimage): Image processing in Python*, scikit-image.org, 2022. [Online]. Available: https://scikit-image.org/docs/stable/