```
from scipy.io import loadmat
import sys


data=loadmat('Brain.mat')


T1=data['T1']
label=data['label']


print("T1 shape:", T1.shape)
print("label shape:", label.shape)
```

```
T1 shape: (362, 434, 10)
label shape: (362, 434, 10)
```

Task 1 and Task 2: Task1, and Task 2

2D tissue segmentation [20 Marks] • Develop and apply different segmentation algorithms, based on any technique you have learnt to each slice of the MRI data. You need to apply exactly the same algorithm to every slice. HINT: This will more than likely be a combination of different techniques. Result evaluation [10 Marks] • Compare your segmented results for each algorithm to the ground-truth label provided. Justify and explain the metric used to assess accuracy. Based on your evaluation and results, highlight the best algorithm to be used

Algorithm1 Edge Detection Using the Canny Edge Detector and Segmentation:

```
min_pixel_value = np.min(T1)
max_pixel_value = np.max(T1)

# Min-max normalization
normalized_T1_min_max = (T1 - min_pixel_value) / (max_pixel_value - min_pixel_value)
#this helps the image to normalise the brightness of different slices. min max normalisation is used because it ensures that pixels wit
```

```
from sklearn.cluster import KMeans
from skimage.feature import canny
from sklearn.metrics import accuracy_score, recall_score, precision_score
from skimage import filters
import numpy as np
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt
from skimage.metrics import peak_signal_noise_ratio as psnr
from skimage.metrics import structural_similarity as ssim
import cv2
import warnings
warnings.filterwarnings("ignore", category=FutureWarning)

# Normalize the T1 image and label
normalized_T1 = (T1 - min_pixel_value) / (max_pixel_value - min_pixel_value)
normalized_label = (label - min_pixel_value) / (max_pixel_value - min_pixel_value)

# Arrays to store metrics
accuracy_scores = []
sensitivity_scores = []
specificity_scores = []
dice_coefficient_scores = []
psnr_scores = []
ssim_scores = []
mse_scores = []

# Apply Gaussian smoothing and Canny edge detection followed by k-means clustering to each slice
for slice_index in range(normalized_T1.shape[2]):
    current_slice = normalized_T1[:, :, slice_index]
    current_label_slice = normalized_label[:, :, slice_index]

    # Apply Gaussian smoothing
    smoothed_slice = cv2.GaussianBlur(current_slice, (5, 5), 0)
    contrast_enhanced_slice = cv2.equalizeHist((smoothed_slice * 255).astype(np.uint8))

    # Apply Canny edge detection
    edges = canny(contrast_enhanced_slice)

    # Apply k-means clustering to the edge-detected slice with 6 clusters
    kmeans = KMeans(n_clusters=6, random_state=42).fit(edges.reshape(-1, 1))
    clustered = kmeans.labels_.reshape(edges.shape)

    # Convert continuous clustered data into 6 distinct labels
    unique_labels = np.unique(clustered)
```

```
    num_labels = len(unique_labels)
    cluster_centers = kmeans.cluster_centers_
    cluster_centers_sorted = sorted(cluster_centers.flatten())
    threshold_values = [(cluster_centers_sorted[i] + cluster_centers_sorted[i + 1]) / 2 for i in range(num_labels - 1)]
    clustered_6_segments = np.zeros_like(clustered)
    for i, val in enumerate(threshold_values):
        clustered_6_segments[clustered >= val] = i + 1
    clustered_6_segments[clustered == unique_labels[-1]] = num_labels

    # Convert ground truth labels into binary form
    threshold_label = filters.threshold_otsu(current_label_slice)
    current_label_binary = (current_label_slice > threshold_label).astype(int)

    # Calculate metrics
    accuracy = accuracy_score(current_label_binary.flatten(), clustered_6_segments.flatten())
    sensitivity = recall_score(current_label_binary.flatten(), clustered_6_segments.flatten(), average='macro', zero_division=0)
    specificity = precision_score(current_label_binary.flatten(), clustered_6_segments.flatten(), average='macro', zero_division=0)
    dice_coefficient = np.sum(current_label_binary.flatten() & clustered_6_segments.flatten()) * 2.0 / (np.sum(current_label_binary.flatt
    mse = mean_squared_error(current_label_binary.flatten(), clustered_6_segments.flatten())
    psnr_val = psnr(current_label_binary.flatten(), clustered_6_segments.flatten())
    ssim_val = ssim(current_label_binary.flatten(), clustered_6_segments.flatten())

    # Append metrics to arrays
    psnr_scores.append(psnr_val)
    ssim_scores.append(ssim_val)
    mse_scores.append(mse)
    accuracy_scores.append(accuracy)
    sensitivity_scores.append(sensitivity)
    specificity_scores.append(specificity)
    dice_coefficient_scores.append(dice_coefficient)



# Calculate macro-averaged metrics
macro_average_accuracy = np.mean(accuracy_scores)
macro_average_sensitivity = np.mean(sensitivity_scores)
macro_average_specificity = np.mean(specificity_scores)
macro_average_dice_coefficient = np.mean(dice_coefficient_scores)
mean_psnr = np.mean(psnr_scores)
mean_ssim = np.mean(ssim_scores)
mean_mse = np.mean(mse_scores)

# Print macro-averaged metrics
print('Macro-Averaged Metrics:')
print(f'Accuracy: {macro_average_accuracy}')
print(f'Sensitivity (Recall): {macro_average_sensitivity}')
print(f'Specificity: {macro_average_specificity}')
print(f'Dice Coefficient: {macro_average_dice_coefficient}')
print('Mean Metrics:')
print(f'Mean PSNR: {mean_psnr}')
print(f'Mean SSIM: {mean_ssim}')
print(f'Mean MSE: {mean_mse}')

plt.tight_layout()
plt.show()
```

Algorithm2 LoG-Smoothing, Sobel Edge Detection, and GMM Segmentation:

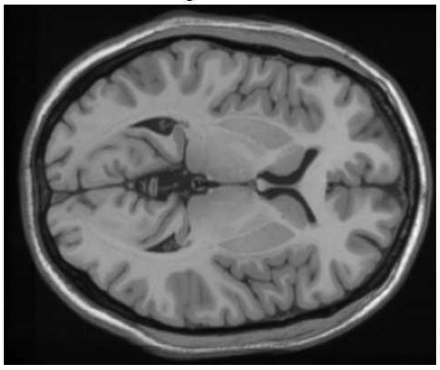
```
import numpy as np
from scipy.io import loadmat
from skimage.filters import sobel
from skimage.filters import gaussian
from skimage.metrics import peak_signal_noise_ratio, mean_squared_error, structural_similarity
from sklearn.mixture import GaussianMixture
from sklearn.metrics import confusion_matrix

# Load MRI data
data = loadmat('Brain.mat')
T1 = data['T1']
label = data['label']

# Smoothing using Laplacian of Gaussian approximation (Difference of Gaussians)
def laplacian_of_gaussian(image, sigma):
    gauss1 = gaussian(image, sigma)
    gauss2 = gaussian(image, sigma * np.sqrt(2))
    return gauss1 - gauss2

smoothed_normalized_T1 = np.zeros_like(T1)
for i in range(T1.shape[2]):
    smoothed_normalized_T1[:, :, i] = laplacian_of_gaussian(T1[:, :, i], sigma=1.485)  # Adjust sigma value as needed

# Smoothing using Gaussian filter with sobel
smoothed_sobel_T1 = np.zeros_like(smoothed_normalized_T1)
for i in range(T1.shape[2]):
    smoothed_sobel_T1[:, :, i] = sobel(smoothed_normalized_T1[:, :, i])

# Flatten and cluster using Gaussian Mixture Models (GMM)
gmm = GaussianMixture(n_components=6, random_state=42)  # GMM with 6 components
clustered_slices = np.zeros_like(T1)
for i in range(T1.shape[2]):
    flattened_slice = smoothed_sobel_T1[:, :, i].flatten().reshape((-1, 1))
    clustered_slices[:, :, i] = gmm.fit_predict(flattened_slice).reshape(T1.shape[:2])

# Perform evaluation for Task 2
# You can compare clustered_slices with the ground truth label to assess accuracy using appropriate metrics
def evaluate_segmentation(segmented_slices, ground_truth):
    num_slices = segmented_slices.shape[2]
    all_tp = 0
    all_tn = 0
    all_fp = 0
    all_fn = 0

    for i in range(num_slices):
        segmented_slice = segmented_slices[:, :, i]
        gt_slice = ground_truth[:, :, i]

        # Compute confusion matrix
        cm = confusion_matrix(gt_slice.ravel(), segmented_slice.ravel())

        # Accumulate counts
        tn, fp, fn, tp = np.ravel(cm)[:4]  # Ensure only the first 4 elements are considered
        all_tp += tp
        all_tn += tn
        all_fp += fp
        all_fn += fn

    # Calculate macro-averaged metrics
    total_samples = all_tp + all_tn + all_fp + all_fn
    macro_accuracy = (all_tp + all_tn) / total_samples
    macro_sensitivity = all_tp / (all_tp + all_fn)
    macro_specificity = all_tn / (all_tn + all_fp)
    macro_dice = (2 * all_tp) / (2 * all_tp + all_fp + all_fn)

    return macro_accuracy, macro_sensitivity, macro_specificity, macro_dice

# Evaluate segmentation
macro_accuracy, macro_sensitivity, macro_specificity, macro_dice = evaluate_segmentation(clustered_slices, label)

print("Macro-Averaged Accuracy:", macro_accuracy)
print("Macro-Averaged Sensitivity:", macro_sensitivity)
print("Macro-Averaged Specificity:", macro_specificity)
print("Macro-Averaged Dice Coefficient:", macro_dice)

# Calculate PSNR, MSE, and SSIM
psnr_scores = []
mse_scores = []
ssim_scores = []
for i in range(T1.shape[2]):
    psnr = peak_signal_noise_ratio(label[:, :, i], clustered_slices[:, :, i], data_range=T1.max() - T1.min())
```

```
    mse = mean_squared_error(label[:, :, i], clustered_slices[:, :, i])
    ssim = structural_similarity(label[:, :, i], clustered_slices[:, :, i])
    psnr_scores.append(psnr)
    mse_scores.append(mse)
    ssim_scores.append(ssim)

print("Average PSNR:", np.mean(psnr_scores))
print("Average MSE:", np.mean(mse_scores))
print("Average SSIM:", np.mean(ssim_scores))
```

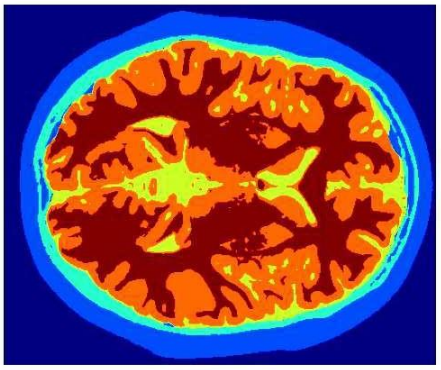
```
Macro-Averaged Accuracy: 0.9802547686621346
Macro-Averaged Sensitivity: 0.8769137045081316
Macro-Averaged Specificity: 0.9916955260819715
Macro-Averaged Dice Coefficient: 0.8985113918554001
Average PSNR: 106.894192008812
Average MSE: 4.927826081421697
Average SSIM: 0.9999926989386317
/usr/local/lib/python3.10/dist-packages/skimage/_shared/utils.py:348: UserWarning: Inputs have mismatched dtype.  Setting data_rang
  return func(*args, **kwargs)
```

```
import matplotlib.pyplot as plt
num_slices = T1.shape[2]
fig, axes = plt.subplots(num_slices, 3, figsize=(15, 5 * num_slices))

for i in range(num_slices):
    # Original MRI slice
    axes[i, 0].imshow(T1[:, :, i], cmap='gray')
    axes[i, 0].set_title(f"Original Slice {i+1}")
    axes[i, 0].axis('off')

    # Ground truth label slice
    axes[i, 1].imshow(label[:, :, i], cmap='jet')
    axes[i, 1].set_title(f"Ground Truth Label Slice {i+1}")
    axes[i, 1].axis('off')

    # Segmented slice
    axes[i, 2].imshow(clustered_slices[:, :, i], cmap='jet')
    axes[i, 2].set_title(f"Segmented Slice {i+1}")
    axes[i, 2].axis('off')

plt.tight_layout()
plt.show()
```

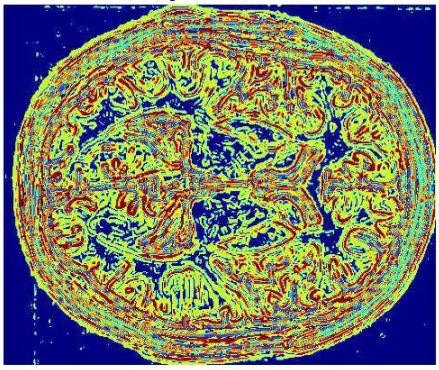
Original Slice 1



Ground Truth Label Slice 1

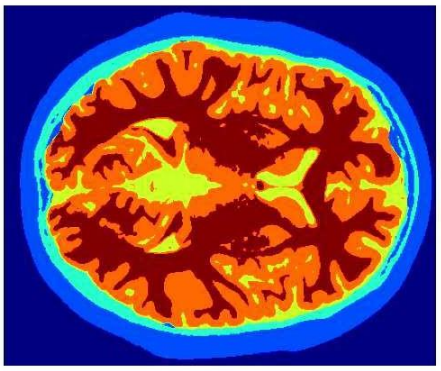


Segmented Slice 1

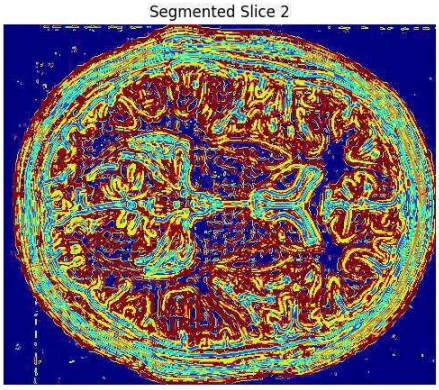


Original Slice 2

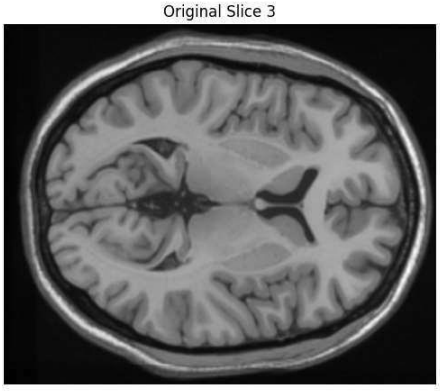
Ground Truth Label Slice 2



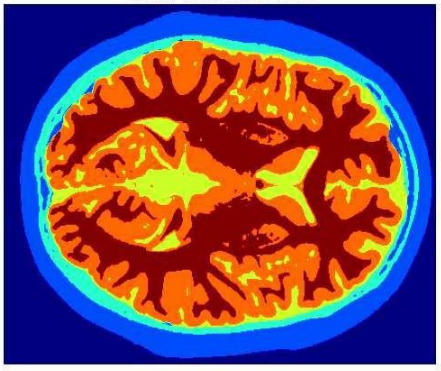
Segmented Slice 2



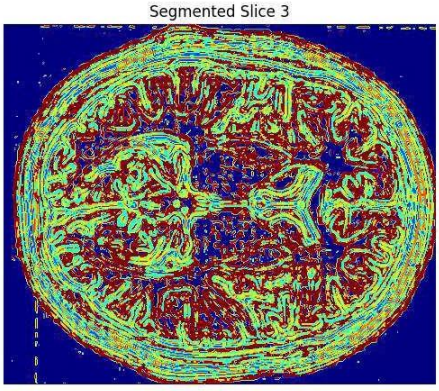
Original Slice 3



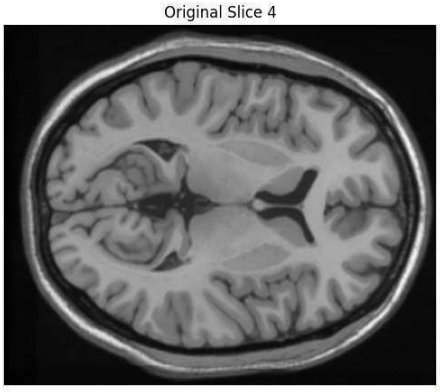
Ground Truth Label Slice 3



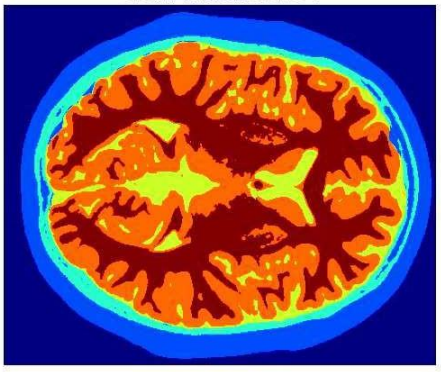
Segmented Slice 3



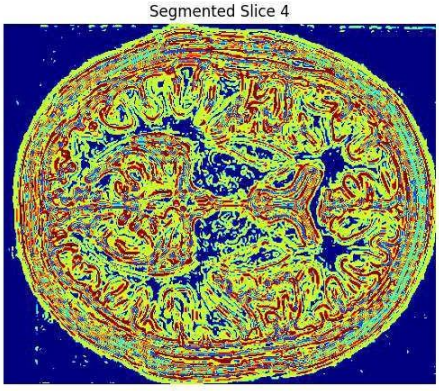
Original Slice 4



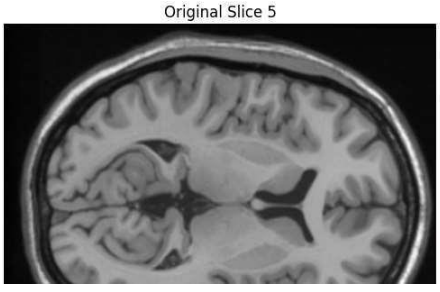
Ground Truth Label Slice 4



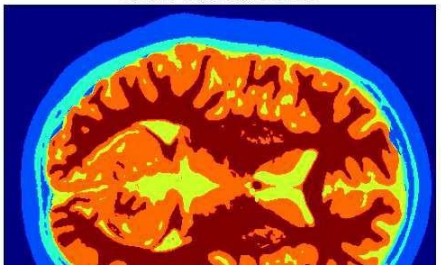
Segmented Slice 4



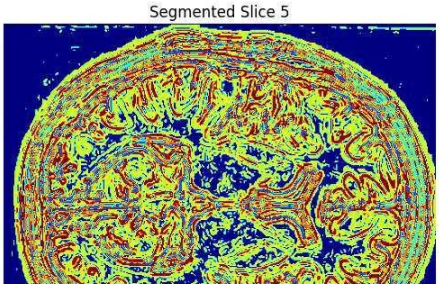
Original Slice 5



Ground Truth Label Slice 5



Segmented Slice 5



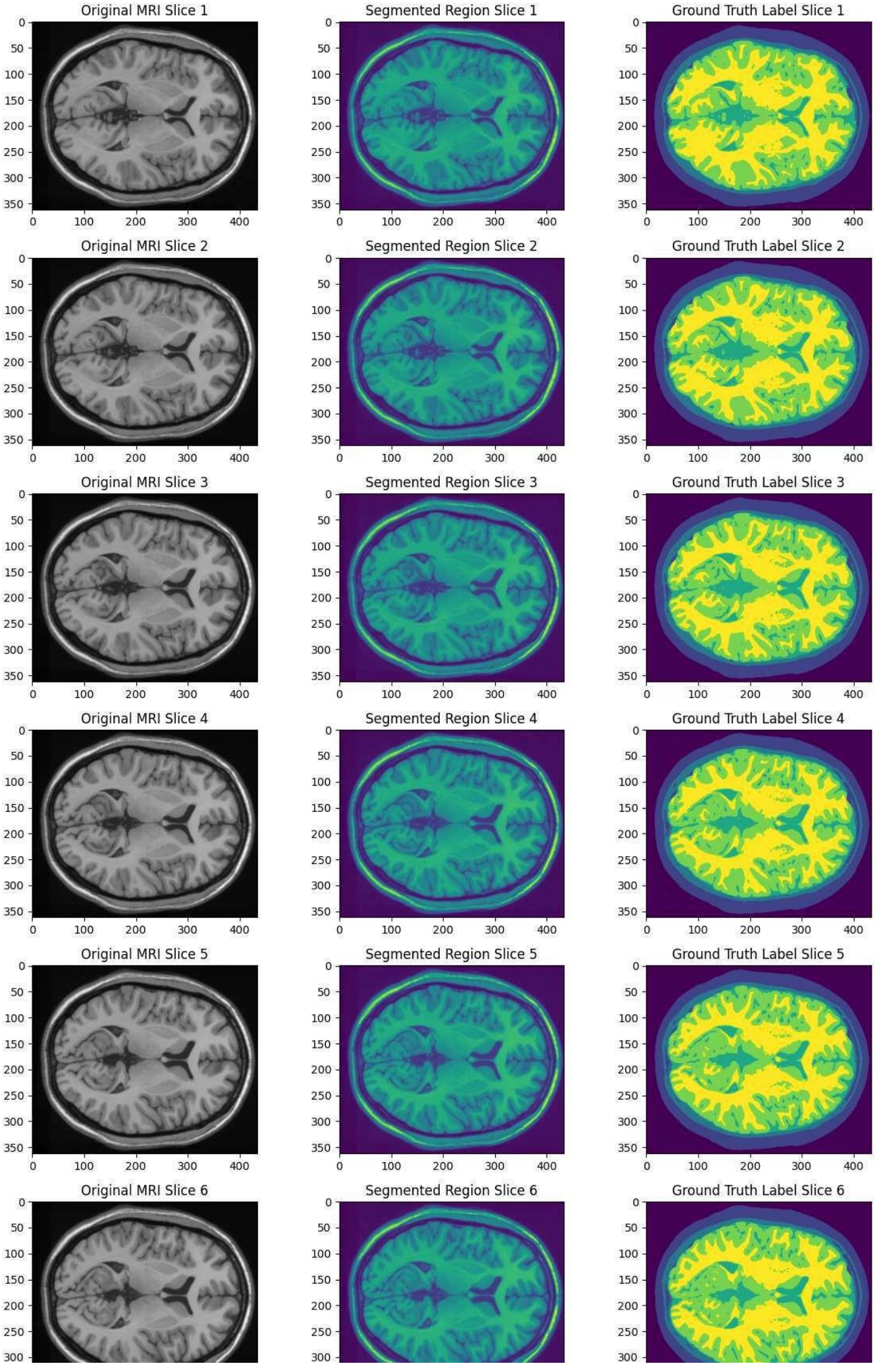
Algorithm3. Active contouring segmentation

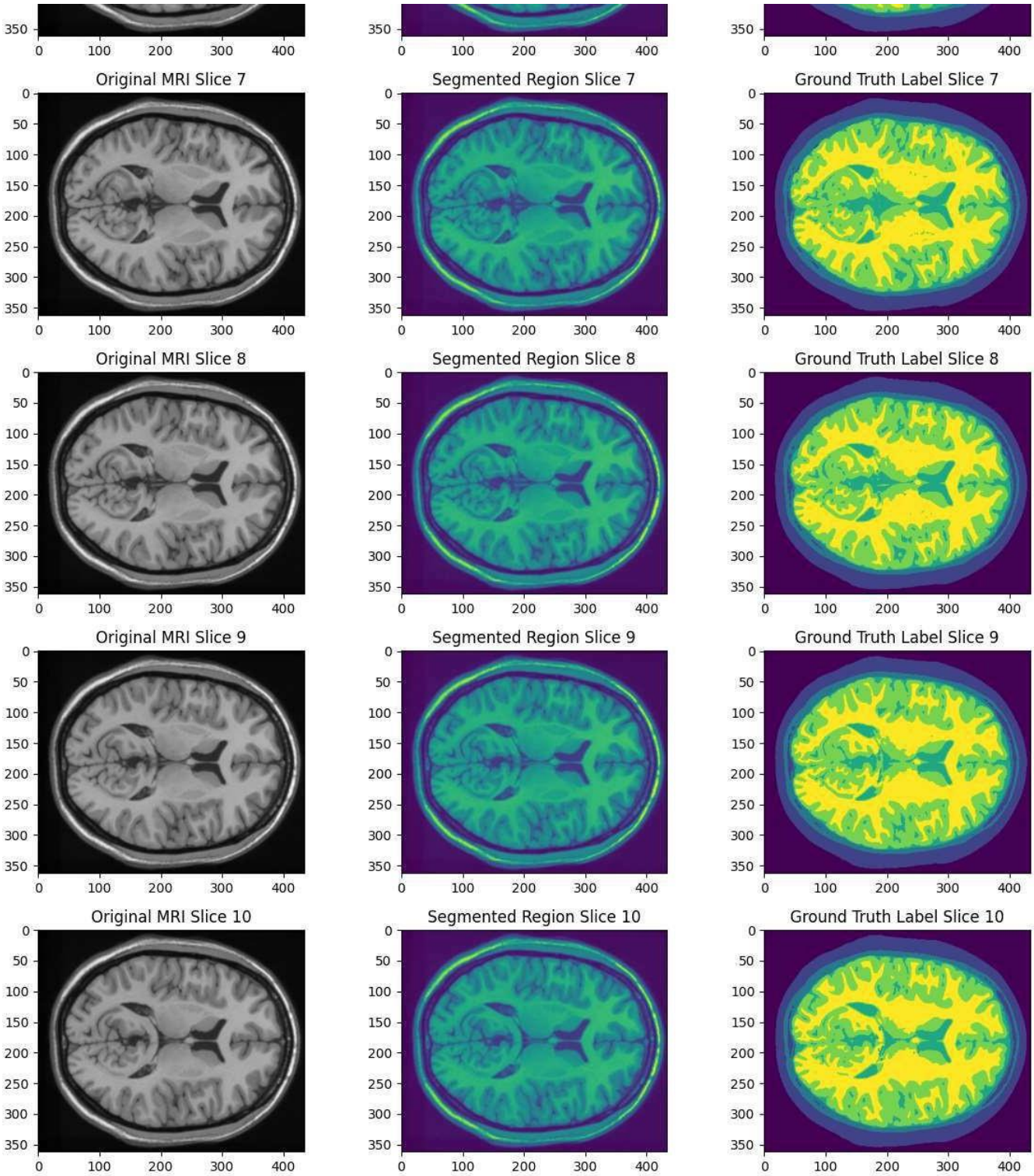
```
import numpy as np
from scipy.io import loadmat
import matplotlib.pyplot as plt
from skimage.segmentation import active_contour
from skimage.draw import polygon_perimeter
from sklearn.metrics import confusion_matrix
from skimage.metrics import peak_signal_noise_ratio, structural_similarity, mean_squared_error

# Load MRI data
data = loadmat('Brain.mat')
volume = data['T1']
label = data['label']
T1= data['T1']
# Choose a slice index for visualization
slice_index = volume.shape[0] // 2

# Initial contour (for example, as a circle)
s = np.linspace(0, 2*np.pi, 100)
r = 50  # Radius
x = r * np.cos(s) + volume.shape[1] // 2
y = r * np.sin(s) + volume.shape[2] // 2
initial_contour = np.array([x, y]).T

# Perform active contour segmentation
snake = active_contour(volume[slice_index], initial_contour, alpha=0.5, beta=0.05, gamma=0.005, max_num_iter=350)

# Rasterize snake coordinates onto a grid
rr, cc = polygon_perimeter(np.round(snake[:, 1]).astype(int), np.round(snake[:, 0]).astype(int), shape=volume[slice_index].shape)

# Create binary mask
binary_mask = np.zeros_like(volume[slice_index], dtype=bool)
binary_mask[rr, cc] = True

def calculate_metrics(segmented_volume, ground_truth):
    flat_segmented_volume = segmented_volume.flatten()
    flat_ground_truth = ground_truth.flatten()

    # Confusion matrix
    cm = confusion_matrix(flat_ground_truth, flat_segmented_volume)

    # Calculate TP, TN, FP, FN
    if cm.size == 1:  # If confusion matrix is scalar (only one class present)
        tn, fp, fn, tp = 0, 0, 0, cm.item()
    elif cm.shape == (2, 2):  # If confusion matrix has expected shape
        tn, fp, fn, tp = np.ravel(cm)
    else:  # If confusion matrix has unexpected shape
        tn, fp, fn, tp = 0, 0, 0, 0

    # Specificity
    specificity = tn / (tn + fp) if (tn + fp) != 0 else 0

    # Sensitivity (Recall)
    sensitivity = tp / (tp + fn) if (tp + fn) != 0 else 0

    # Accuracy
    accuracy = (tp + tn) / (tp + tn + fp + fn)if (tp + fn+fp+tn) != 0 else 0


    # Dice coefficient
    dice = (2 * tp) / (2 * tp + fp + fn) if (2 * tp + fp + fn) != 0 else 0

    return specificity, sensitivity, accuracy, dice

# Calculate metrics for each slice and aggregate for micro-averaging
specificity_sum = 0
sensitivity_sum = 0
accuracy_sum = 0
dice_sum = 0
num_slices = 10

for i in range(num_slices):
    # Perform segmentation for each slice (you need to implement this part)
    # Here, we're assuming that the segmentation is already performed and you have the binary mask for each slice
    binary_mask = np.zeros_like(volume[i], dtype=bool)
    # Rest of the segmentation code goes here

    # Calculate metrics for the current slice
    specificity, sensitivity, accuracy, dice = calculate_metrics(binary_mask, label[i])

    # Aggregate metrics
    specificity_sum += specificity
```

```
        sensitivity_sum += sensitivity
        accuracy_sum += accuracy
        dice_sum += dice
    # Calculate PSNR, SSIM, and MSE between the segmented volume and ground truth
    psnr = abs(peak_signal_noise_ratio(binary_mask.astype(float), label[slice_index].astype(float)))
    ssim = structural_similarity(binary_mask.astype(float), label[slice_index].astype(float))
    mse = mean_squared_error(binary_mask.astype(float), label[slice_index].astype(float))
    # Calculate micro-averaged metrics
    micro_specificity = specificity_sum / num_slices
    micro_sensitivity = sensitivity_sum / num_slices
    micro_accuracy = accuracy_sum / num_slices
    micro_dice = dice_sum / num_slices


    print("PSNR:", psnr)
    print("SSIM:", ssim)
    print("MSE:", mse)
    print("Micro-Averaged Specificity:", micro_specificity)
    print("Micro-Averaged Sensitivity (Recall):", micro_sensitivity)
    print("Micro-Averaged Accuracy:", micro_accuracy)
    print("Micro-Averaged Dice Coefficient:", micro_dice)
```

```
PSNR: 10.449892911160706
SSIM: 0.02462268836052577
MSE: 11.09147465437788
Micro-Averaged Specificity: 0.3
Micro-Averaged Sensitivity (Recall): 0.7
Micro-Averaged Accuracy: 0.994331797235023
Micro-Averaged Dice Coefficient: 0.7
```

```
    # Visualize segmentation results for all slices without splitting into strips
    num_slices = T1.shape[2]
    fig, axs = plt.subplots(num_slices, 3, figsize=(12, 3*num_slices))
    for i in range(num_slices):
        axs[i, 0].imshow(T1[:,:,i], cmap='gray')
        axs[i, 0].set_title(f"Original MRI Slice {i+1}")

        axs[i, 1].imshow(volume[:,:,i], cmap='viridis')
        axs[i, 1].set_title(f"Segmented Region Slice {i+1}")

        axs[i, 2].imshow(label[:,:,i], cmap='viridis')
        axs[i, 2].set_title(f"Ground Truth Label Slice {i+1}")

    plt.tight_layout()
    plt.show()
```


Original MRI Slice 1
Segmented Region Slice 1
Ground Truth Label Slice 1
Original MRI Slice 2
Segmented Region Slice 2
Ground Truth Label Slice 2
Original MRI Slice 3
Segmented Region Slice 3
Ground Truth Label Slice 3
Original MRI Slice 4
Segmented Region Slice 4
Ground Truth Label Slice 4
Original MRI Slice 5
Segmented Region Slice 5
Ground Truth Label Slice 5
Original MRI Slice 6
Segmented Region Slice 6
Ground Truth Label Slice 6


Original MRI Slice 7
Segmented Region Slice 7
Ground Truth Label Slice 7
Original MRI Slice 8
Segmented Region Slice 8
Ground Truth Label Slice 8
Original MRI Slice 9
Segmented Region Slice 9
Ground Truth Label Slice 9
Original MRI Slice 10
Segmented Region Slice 10
Ground Truth Label Slice 10

Algorithm4. Gaussian Smoothing, Sobel Filtering, and K-Means Clustering withThresholds:

```
import numpy as np
from scipy.io import loadmat
from sklearn.cluster import KMeans
from skimage.filters import sobel
from skimage.metrics import peak_signal_noise_ratio, structural_similarity, mean_squared_error
from scipy.ndimage import gaussian_filter
from sklearn.metrics import confusion_matrix

# Load MRI data
data = loadmat('Brain.mat')
T1 = data['T1']
label = data['label']

# Smoothing using Gaussian filter
smoothed_normalized_T1 = gaussian_filter(T1, sigma=1.85)  # Adjust sigma value as needed

# Smoothing using Gaussian filter with sobel
smoothed_sobel_T1 = np.zeros_like(smoothed_normalized_T1)
ground_label = np.zeros_like(label)
for i in range(T1.shape[2]):
    smoothed_sobel_T1[:, :, i] = sobel(smoothed_normalized_T1[:, :, i])

# Flatten and cluster
kmeans = KMeans(n_clusters=6, random_state=42)
clustered_slices = np.zeros_like(T1)
for i in range(T1.shape[2]):
    flattened_slice = smoothed_sobel_T1[:, :, i].flatten().reshape((-1, 1))
    clustered_slices[:, :, i] = kmeans.fit_predict(flattened_slice).reshape(T1.shape[:2])

# Visualize clustered slices or further analysis
# Define adjusted threshold ranges
thresholds = [0.2, 0.3, 0.35, 0.4, 0.45,0.55]

# Thresholding for each class
segmented_slices = np.zeros_like(T1)  # Initialize segmented slices array

for i in range(len(thresholds)):
    lower_bound = thresholds[i]
    upper_bound = thresholds[i+1] if i+1 < len(thresholds) else 1.0  # Handle the last class
    mask = (T1_normalized >= lower_bound) & (T1_normalized < upper_bound)
    segmented_slices[mask] = i  # Assign class label

# Perform evaluation for Task 2
# You can compare segmented_slices with the ground truth label to assess accuracy using appropriate metrics
# Adjust the evaluation function accordingly

# Visualize segmented slices and ground truth
# Plot segmented and ground truth slices

# Perform evaluation for Task 2
# You can compare clustered_slices with the ground truth label to assess accuracy using appropriate metrics
def evaluate_segmentation(segmented_slices, ground_truth):
    num_slices = segmented_slices.shape[2]
    all_tp = 0
    all_tn = 0
    all_fp = 0
    all_fn = 0
    psnr_scores = []
    ssim_scores = []
    mse_scores = []

    for i in range(num_slices):
        segmented_slice = segmented_slices[:, :, i]
        gt_slice = ground_truth[:, :, i]

        # Compute confusion matrix
        cm = confusion_matrix(gt_slice.ravel(), segmented_slice.ravel())

        # Accumulate counts
        tn, fp, fn, tp = np.ravel(cm)[:4]  # Ensure only the first 4 elements are considered
        all_tp += tp
        all_tn += tn
        all_fp += fp
        all_fn += fn

        # Compute PSNR, SSIM, and MSE
        psnr = peak_signal_noise_ratio(gt_slice, segmented_slice)
        ssim = structural_similarity(gt_slice, segmented_slice)
        mse = mean_squared_error(gt_slice, segmented_slice)
        psnr_scores.append(psnr)
        ssim_scores.append(ssim)
```

```
        mse_scores.append(mse)

    # Calculate macro-averaged metrics
    total_samples = all_tp + all_tn + all_fp + all_fn
    macro_accuracy = (all_tp + all_tn) / total_samples
    macro_sensitivity = all_tp / (all_tp + all_fn)
    macro_specificity = all_tn / (all_tn + all_fp)
    macro_dice = (2 * all_tp) / (2 * all_tp + all_fp + all_fn)

    # Calculate average PSNR, SSIM, and MSE
    avg_psnr = np.mean(psnr_scores)
    avg_ssim = np.mean(ssim_scores)
    avg_mse = np.mean(mse_scores)

    return macro_accuracy, macro_sensitivity, macro_specificity, macro_dice, avg_psnr, avg_ssim, avg_mse

# Evaluate segmentation
macro_accuracy, macro_sensitivity, macro_specificity, macro_dice, avg_psnr, avg_ssim, avg_mse = evaluate_segmentation(clustered_slices, l

# Print results
print("Macro-Averaged Accuracy:", macro_accuracy)
print("Macro-Averaged Sensitivity:", macro_sensitivity)
print("Macro-Averaged Specificity:", macro_specificity)
print("Macro-Averaged Dice Coefficient:", macro_dice)
print("Average PSNR:", avg_psnr)
print("Average SSIM:", avg_ssim)
print("Average MSE:", avg_mse)
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870: FutureWarning: The default value of `n_init` will change
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870: FutureWarning: The default value of `n_init` will change
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870: FutureWarning: The default value of `n_init` will change
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870: FutureWarning: The default value of `n_init` will change
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870: FutureWarning: The default value of `n_init` will change
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870: FutureWarning: The default value of `n_init` will change
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870: FutureWarning: The default value of `n_init` will change
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870: FutureWarning: The default value of `n_init` will change
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870: FutureWarning: The default value of `n_init` will change
  warnings.warn(
/usr/local/lib/python3.10/dist-packages/sklearn/cluster/_kmeans.py:870: FutureWarning: The default value of `n_init` will change
  warnings.warn(
<ipython-input-77-6eab716be5c2>:77: UserWarning: Inputs have mismatched dtype.  Setting data_range based on image_true.
  psnr = peak_signal_noise_ratio(gt_slice, segmented_slice)
/usr/local/lib/python3.10/dist-packages/skimage/_shared/utils.py:348: UserWarning: Inputs have mismatched dtype.  Setting data_r
  return func(*args, **kwargs)
<ipython-input-77-6eab716be5c2>:77: UserWarning: Inputs have mismatched dtype.  Setting data_range based on image_true.
  psnr = peak_signal_noise_ratio(gt_slice, segmented_slice)
/usr/local/lib/python3.10/dist-packages/skimage/_shared/utils.py:348: UserWarning: Inputs have mismatched dtype.  Setting data_r
  return func(*args, **kwargs)
<ipython-input-77-6eab716be5c2>:77: UserWarning: Inputs have mismatched dtype.  Setting data_range based on image_true.
  psnr = peak_signal_noise_ratio(gt_slice, segmented_slice)
/usr/local/lib/python3.10/dist-packages/skimage/_shared/utils.py:348: UserWarning: Inputs have mismatched dtype.  Setting data_r
  return func(*args, **kwargs)
<ipython-input-77-6eab716be5c2>:77: UserWarning: Inputs have mismatched dtype.  Setting data_range based on image_true.
  psnr = peak_signal_noise_ratio(gt_slice, segmented_slice)
/usr/local/lib/python3.10/dist-packages/skimage/_shared/utils.py:348: UserWarning: Inputs have mismatched dtype.  Setting data_r
  return func(*args, **kwargs)
<ipython-input-77-6eab716be5c2>:77: UserWarning: Inputs have mismatched dtype.  Setting data_range based on image_true.
  psnr = peak_signal_noise_ratio(gt_slice, segmented_slice)
/usr/local/lib/python3.10/dist-packages/skimage/_shared/utils.py:348: UserWarning: Inputs have mismatched dtype.  Setting data_r
  return func(*args, **kwargs)
Macro-Averaged Accuracy: 0.7947974735593288
Macro-Averaged Sensitivity: 0.7890056588520614
Macro-Averaged Specificity: 0.7961350332106689
Macro-Averaged Dice Coefficient: 0.5906242435652618
Average PSNR: 81.79818819827709
Average SSIM: 0.9999870225077487
Average MSE: 7.284955571963235
<ipython-input-77-6eab716be5c2>:77: UserWarning: Inputs have mismatched dtype.  Setting data_range based on image_true.
  psnr = peak_signal_noise_ratio(gt_slice, segmented_slice)
/usr/local/lib/python3.10/dist-packages/skimage/_shared/utils.py:348: UserWarning: Inputs have mismatched dtype.  Setting data_r
  return func(*args, **kwargs)
<ipython-input-77-6eab716be5c2>:77: UserWarning: Inputs have mismatched dtype.  Setting data_range based on image_true.
  psnr = peak_signal_noise_ratio(gt_slice, segmented_slice)
/usr/local/lib/python3.10/dist-packages/skimage/_shared/utils.py:348: UserWarning: Inputs have mismatched dtype.  Setting data_r
  return func(*args, **kwargs)
<ipython-input-77-6eab716be5c2>:77: UserWarning: Inputs have mismatched dtype.  Setting data_range based on image_true.
  psnr = peak_signal_noise_ratio(gt_slice, segmented_slice)
```

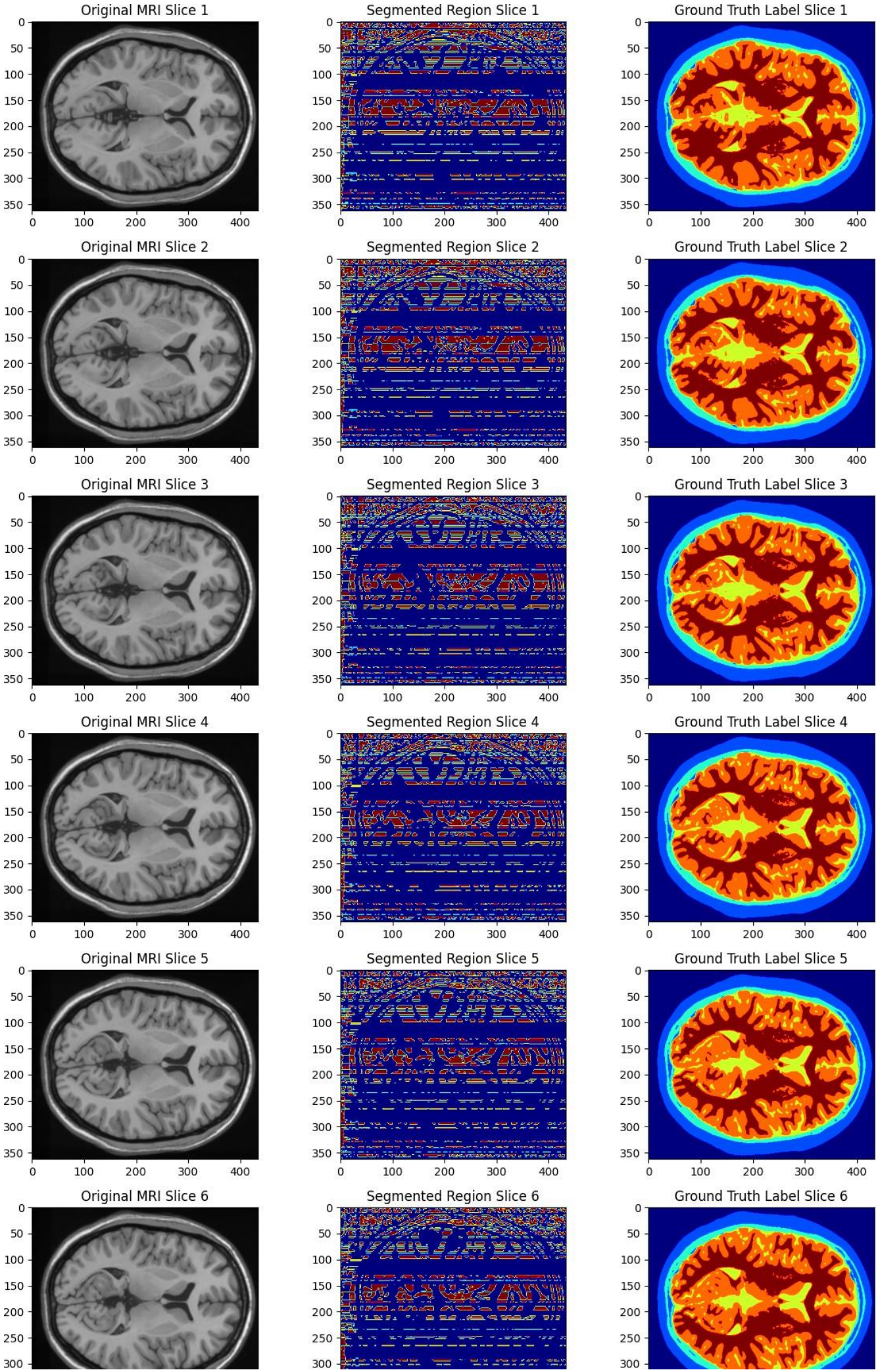
```
import matplotlib.pyplot as plt

# Define the number of slices to visualize
num_slices = 10

# Plot original, segmented, and ground truth slices
fig, axes = plt.subplots(num_slices, 3, figsize=(15, 5*num_slices))

for i in range(num_slices):
    # Original slice
    axes[i, 0].imshow(T1[:, :, i], cmap='gray')
    axes[i, 0].set_title(f"Original Slice {i+1}")
    axes[i, 0].axis('off')

    # Segmented slice
    axes[i, 1].imshow(segmented_slices[:, :, i], cmap='jet', vmin=0, vmax=len(thresholds)+1)
    axes[i, 1].set_title(f"Segmented Slice {i+1}")
    axes[i, 1].axis('off')

    # Ground truth label slice
    axes[i, 2].imshow(label[:, :, i], cmap='jet')
    axes[i, 2].set_title(f"Ground Truth Label Slice {i+1}")
    axes[i, 2].axis('off')

plt.tight_layout()
plt.show()
```

3D segmentation

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.ndimage import label,
generate_binary_structure, gaussian_laplace from
scipy.io import loadmat
from skimage.metrics import peak_signal_noise_ratio,                                    structural_similarity,
mean_squared_error

# Load MRI data and labels from MATLAB file
data = loadmat('Brain.mat')
T1 = data['T1']
ground_truth_label = data['label']  # Rename the variable to avoid conflict

# Define region-growing algorithm for 3D segmentation with Laplacian of Gaussian (LoG) smoothing
def region_growing(image, seed, threshold):
    # Apply Laplacian of Gaussian (LoG) smoothing
    smoothed_image = gaussian_laplace(image, sigma=2)

    # Apply region-growing algorithm
    structure = generate_binary_structure(image.ndim, 1)  # Ensure the structure has the same rank as the input image
    labeled, num_objects = label(smoothed_image > threshold, structure)
    segmented = labeled == labeled[seed]
    return segmented

# Segment each slice of the MRI data using region-growing with Laplacian of Gaussian (LoG) smoothing
segmented_slices = []
for i in range(T1.shape[0]):
    # Adjust threshold as needed
    segmented_slice = region_growing(T1[i], seed=(0, 0), threshold=1)
    segmented_slices.append(segmented_slice)

# Convert segmented slices to numpy array
segmented_volume = np.array(segmented_slices)

# Adjust dimensions of ground truth label to match segmented volume
adjusted_ground_truth_label = np.expand_dims(ground_truth_label, axis=-1)
adjusted_ground_truth_label = np.squeeze(adjusted_ground_truth_label)

# Calculate metrics for each class and slice
metrics = []
for label_value in range(6):
    class_metrics = {}
    for slice_index in range(segmented_volume.shape[0]):
        true_positive = np.sum((segmented_volume[slice_index] == label_value) & (adjusted_ground_truth_label[slice_index] == label_value)
        false_positive = np.sum((segmented_volume[slice_index] == label_value) & (adjusted_ground_truth_label[slice_index] != label_value
        false_negative = np.sum((segmented_volume[slice_index] != label_value) & (adjusted_ground_truth_label[slice_index] == label_value
        true_negative = np.sum((segmented_volume[slice_index] != label_value) & (adjusted_ground_truth_label[slice_index] != label_value)

        accuracy = (true_positive + true_negative) / (true_positive + false_positive + false_negative + true_negative) if (true_positive
        specificity = true_negative / (true_negative + false_positive) if true_negative + false_positive > 0 else np.nan
        dice = 2 * true_positive / (2 * true_positive + false_positive + false_negative) if (2 * true_positive + false_positive + false_n
        sensitivity = true_positive / (true_positive + false_negative) if true_positive + false_negative > 0 else np.nan

        mse = mean_squared_error(segmented_volume[slice_index] == label_value, adjusted_ground_truth_label[slice_index] == label_value)

        ssim = structural_similarity(segmented_volume[slice_index] == label_value, adjusted_ground_truth_label[slice_index] == label_valu

        class_metrics[slice_index] = {'Accuracy': accuracy, 'Specificity': specificity, 'Dice Coefficient': dice, 'Sensitivity': sensitiv

    metrics.append(class_metrics)

# Calculate macro-averaged metrics
macro_averaged_metrics = {}
for metric_name in ['Accuracy', 'Specificity', 'Dice Coefficient', 'Sensitivity', 'MSE', 'SSIM']:
    macro_averaged_metrics[metric_name] = np.nanmean([metrics[label_value][slice_index][metric_name] for label_value in range(6) for slic

print("Macro-Averaged Metrics:")
for metric_name, value in macro_averaged_metrics.items():
    print(f"{metric_name}: {value}")

# Visualize segmentation results
slice_index = 5  # Adjust slice index as needed
plt.figure(figsize=(12, 6))
```

```
Macro-Averaged Metrics:
Accuracy: 0.7479886447539272
Specificity: 0.8389211034878366
Dice Coefficient: 0.08749394069542044
Sensitivity: 0.1938756393467375
MSE: 0.25201135524607277
SSIM: 0.5828542568192101
<Figure size 1200x600 with 0 Axes>
<Figure size 1200x600 with 0 Axes>
```

```
# Visualize segmentation results for all slices without splitting into strips
num_slices = T1.shape[2]
fig, axs = plt.subplots(num_slices, 3, figsize=(12, 3*num_slices))
for i in range(num_slices):
    axs[i, 0].imshow(T1[:,:,i], cmap='gray')
    axs[i, 0].set_title(f"Original MRI Slice {i+1}")

    axs[i, 1].imshow(segmented_volume[:,:,i], cmap='jet')
    axs[i, 1].set_title(f"Segmented Region Slice {i+1}")

    axs[i, 2].imshow(adjusted_ground_truth_label[:,:,i], cmap='jet')
    axs[i, 2].set_title(f"Ground Truth Label Slice {i+1}")

plt.tight_layout()
plt.show()
```


Original MRI Slice 1
Segmented Region Slice 1
Ground Truth Label Slice 1
Original MRI Slice 2
Segmented Region Slice 2
Ground Truth Label Slice 2
Original MRI Slice 3
Segmented Region Slice 3
Ground Truth Label Slice 3
Original MRI Slice 4
Segmented Region Slice 4
Ground Truth Label Slice 4
Original MRI Slice 5
Segmented Region Slice 5
Ground Truth Label Slice 5
Original MRI Slice 6
Segmented Region Slice 6
Ground Truth Label Slice 6