Lab 2 Report
Image Segmentation (EM algorithm)

Colin Tenorio C.G., Ulin Briseño E.Y.

[1] EPS, University of Girona, Spain.
carmencolinten@gmail.com, eulinbriseno@gmail.com

Key words: Segmentation, EM, Maximization, Expectation, MRI

Introduction

The primary objective of medical image segmentation is to partition an image into distinct and non-overlapping sets of voxels, each of which corresponds to a specific anatomical structure within the patient. In the context of magnetic resonance (MR) imaging, the segmentation of cerebrospinal fluid (CSF), gray matter (GM), and white matter (WM) presents a formidable challenge.

EM is employed in brain segmentation and can handle complex and intertwined structures in brain imaging. Brain tissue, such as cerebrospinal fluid (CSF), gray matter (GM), and white matter (WM), often exhibits intricate patterns and overlaps. The EM algorithm excels at modeling the underlying probabilistic distributions of these tissues, making it a valuable tool for precise brain segmentation.

Objectives

The objective of this lab is to implement an EM algorithm and to apply it to our different modalities brain images The main goals are:

1. To understand, design, analyse and implement the EM segmentation algorithm in Matlab.
2. Provide the segmented the brain tissue classes into a single image (.nii file).
3. Evaluate the performance of the algorithm using the provided images: study the problems and possible improvements
4. Evaluate the results with the provided ground-truth of the WM, GM and CSF using Dice.

Methodology

Expectation Maximization algorithm (EM)

The Expectation Maximization algorithm is a multiple pass batch algorithm for parameter estimation where some part of the measurements are unknown. The algorithm can either be used for maximum likelihood estimation or in a Bayesian framework to obtain maximum a posteriori estimates. EM algorithm is iterative technique designed for probabilistic models. It is often used for finding the unknown parameters of a mixture model. Likelihood is the probability of a sample of belong to a class, given the parameters of that class. EM maximizes the log-likelihood of the sample as represented by the mixture model: maximizing the likelihood we can fit a mathematical model to the data

A finite mixture model with K components can be defined as:

$$p\left(\underline{x} \mid \Theta\right) = \sum_{k=1}^{K} \alpha_k \, p_k\left(\underline{x} \mid z_k, \theta_k\right) \tag{1}$$

Where $p_k\left(\underline{x} \mid z_k, \theta_k\right)$ are mixture components. Each is a density or distribution defined over $p(\underline{x})$

with parameters θ_k

The α_k are the mixture weights, representing the probability that a randomly selected $\underline{\mathrm{x}}$ was generated by component k

Membership Weights

The membership weight of data point x_i in cluster k given parameters Θ are defined as:

$$w_{ik} = p\left(z_{ik} = 1 \middle| x_i, \Theta\right) = \frac{p_k\left(x_i \middle| z_k, \theta_k\right) \cdot \alpha_k}{\sum_{m=1}^{K} p_m\left(x_i \middle| z_m, \theta_m\right) \cdot \alpha_m} \tag{2}$$

Gaussian Mixture Models

The Gaussian mixture model is defined by a Gaussian density for each component K, We can defined the multivariate Gaussian density as:

$$p_k\left(x \middle| \theta_k\right) = \frac{1}{\left(2\pi\right)^{d/2} \cdot \left|\Sigma_k\right|^{1/2}} e^{-\frac{1}{2}\left(x-\mu_k\right)^t \Sigma_k^{-1}\left(x-\mu_k\right)} \tag{3}$$

with its own parameters θ_k.

The EM Algorithm for Gaussian Mixture Models

The algorithms is an iterative algorithm that starts from some initial estimates of Θ, and then proceeds to iterative update Θ until convergence is detected. Each iteration consists of an E-step and a M-step.

E-Step:

Denote the current parameter values as Θ. Compute the weights using equation 2 for all data and components K.

M-Step:

Now we use the membership weights from previous step and the data to calculate new parameter values. Let $N_k = w_i k$ the sum of the membership weights for the k th component.

$$\alpha_k^{new} = \frac{N_k}{N}, \quad 1 \leq k \leq K \tag{4}$$

These are the new mixture weights. We also update the mean as:

$$\mu_k^{new} = \left(\frac{1}{N_k}\right) \sum_{i=1}^{N} w_{ik} \cdot x_i, \quad 1 \leq k \leq K \tag{5}$$

And the covariance as follows:

$$\Sigma_k^{new} = \left(\frac{1}{N_k}\right) \sum_{i=1}^{N} w_{ik} \cdot \left(x_i - \mu_k^{new}\right) \left(x_i - \mu_k^{new}\right)^t , \quad 1 \leq k \leq K \tag{6}$$

After we computed all the new parameters, we can go to the E-Step to recompute the membership weights. One E-step and M-step are consider one iteration.

Initialization and convergence

The EM algorithm can be started by initializing with a set of initial parameters and then conducting an E-Step. The initial parameters or weights can be chosen randomly or by a heuristic method (K-means).

The convergence will be detected by computing the value of the log-likelihood after each iteration and measure the difference in a significance manner from one iteration to the next one. The log-likelihood is defined as:

$$logl(\Theta) = \sum_{i=1}^{N} logp(x_i|\Theta) = \sum_{i=1}^{N} \left(\log \sum_{k=1}^{K} \alpha_k p_k(x_i|z_k, \theta_k) \right) \tag{7}$$

Data pre-processing

The data provided was 5 volumes, subject 0,1,2,3,4. Each one included the modality T1 and T2Flair. Also, the correspondent labels (Groundtruth) for each subject. In order to correctly perform the segmentation, we need some pre-processing to the original images.

1. Skull Stripping: Generate a mask with the ground truth labels for the desired issues in this case: Gray Matter (GM), White Matter (WM) and CSF. Then we used this Mk in order to remove the non desired voxels (Skull and background).
2. Normalization: We need to scale the pixel values in an image to a specific range.In this case we used min-max normalization to have all the pixels in the specific range.

EM Algorithm implementation

The GMM-EM algorithm alternates between two main steps: the Expectation (E) step and the Maximization (M) step. These steps are designed to iterative refine the estimates of the model's parameters. We show an example in Algorithm 1.

1. **Initialization:**
    (a) Choose the number of Gaussian components (clusters) K that you believe the data can be represented by.In this case is 3, since we have three classes we want to segment (WM,GM and CSF)
    (b) Initialize the parameters of the K Gaussian components: Mean vectors (μ) for each component. Covariance matrices (Σ) for each component. Mixing coefficients that represent the probability of each component. in this step we create two types of initialization: Random and Kmeans. In this case it will create an array size number of components by length of the data. The covariance matrices are created of each of the clusters independently. The probabilities are started by assigning equal probabilities to all components.

2. **Expectation:**

   The Expectation step is the first step in the iterative process of refining the parameters of the Gaussian Mixture Model. It the probabilities that each data point belongs to each of the Gaussian components in the mixture (posterior probabilities) for each data point and each Gaussian component. That membership weight represents the probability that the k-th Gaussian component generated the i-th data point.This calculation assigns a probability value to each data point, expressing how likely it is to belong to each of the Gaussian components (Eq. 2). For a given data point x, we compute its likelihood with respect to each component using the multivariate Gaussian probability density function. We multiply the likelihood values by the prior probabilities (priors) associated with each component. To ensure that the posterior probabilities sum to 1 for each data point, we perform normalization. We then compute the posterior probabilities by dividing each data point's weighted likelihood by this denominator. The result is an array of posterior probabilities for each data point, indicating the probability of each data points association with each component.

3. **Maximization:**

   The Maximization step is the second part of the iterative process and is responsible for updating the parameters of the Gaussian components to maximize the likelihood of the data. In the Maximization step, we update the mean, covariance and priors vectors of each Gaussian component. Following equations 5, 4 and 6. We assign a label of 1 to the component with the highest posterior probability for each data point.

4. **Convergence:**

   The algorithm iterative updates the model parameters in the Maximization step (M-step) to maximize the log-likelihood. By maximizing the log-likelihood, the algorithm is effectively trying to find the best set of parameters.

   During each iteration, we calculate the log-likelihood of the data given the current model parameters. Then, we compare this log-likelihood to the log-likelihood from the previous iteration. If the change in log-likelihood is smaller than a predefined threshold or if the algorithm reaches a maximum number of iterations without significant improvement, it is considered to have converged.

5. **Output:**

   The desired output is the calculated posterior probabilities for each data voxel, indicating the probability that the data point belongs to each Gaussian component. In this case, we get the maximum posterior probability and we assign our data to that component.

Evaluation

In order to perform the evaluation of the segmentation with GMM-EM algorithm. We use the DICE score metric. The final output of the EM algorithm is the label of each data point. Since the initialization of the algorithm will affect the way how the labels are assign, at the end our labels are not matching the original ground truth labels. We implement a function to perform this and match the correct labels with the EM output in order to evaluate our results. Also in other to compare with other clustering algorithms we use the K-Means algorithm and we compared the results.

Algorithm 1: Expectation-Maximization Algorithm

```
   Input  : Initialize model parameters randomly
1  repeat
       // Expectation step
2      Compute labels for all the dataset given the current cluster parameters;
       // Maximization step
3      Use that classification to re-estimate the parameters;
       // Check for convergence
4      ;
5      if not significant changes on the parameters then
6          break;
7  until convergence is achieved;
   Output: Final model parameters
```

Results: Random vs K-means

In Figure 1, we display an example of the pre-processing step applied to each subject before employing the EM-Algorithm.

In Figure 2, we present an example of T1 images, the EM segmentation results with K-Means initialization, and the ground truth images. The qualitative results demonstrate strong performance with this initialization, as the segmentation closely resembles the ground truth labels.

Figure 3 showcases the segmentation with EM initialized using K-Means with both T1 and T2 volumes. Visually, the results are superior to using only the T1 volume. In Figure 4, we observe the results of the EM algorithm initialized with random values and using T1 and T2 volumes. As expected, in some areas, the segmentation lacks accuracy. In some instances, only a few pixels are assigned to certain classes, with the white matter class having the fewest pixels in some cases.

Figure 5 compares different initialization methods for the EM-Algorithm: Random and K-Means, using both the T1 volume and the T1 and T2 volumes. K-Means initialization outperforms random initialization in both scenarios. Random initialization faces challenges in segmenting white and gray matter in some subjects.

In Figure 6, we showcase the segmentation results for each tissue (CSF, WM, and GM) compared to the ground truth for subject 1. The random initialization encounters difficulties in differentiating between gray and white matter in the center and in the borders of the CSF tissue.

To quantitatively assess the different initialization methods, we evaluate the Dice scores, execution time, and mean Dice scores per tissue for all patients.

Table 1 presents the Dice scores per tissue for each patient under two methods: Random and Kmeans, utilizing the T1 volume. Generally, Kmeans outperforms Random for all tissues, although the results are very close.

When considering execution time, Kmeans proves to be the more efficient choice for all subjects.

In Table 2, we present the Dice scores per tissue for each patient in both methods: Random and Kmeans, employing the T1 and T2 volumes. Here, the values are remarkably similar, but Kmeans excels in terms of segmentation time.

Table 3 provides the mean Dice scores per tissue. In general, Kmeans consistently delivers superior results.

The comparison reveals Kmeans as the favored initialization method, as it not only yields better segmentation but also does so more efficiently.

CSF	GM	WM	Mod	subject	model	init	t(s)
0.789	0.776	0.862	T1	1	EM	kmeans	7.746
0.789	0.776	0.862	T1	1	EM	random	13.889
0.864	0.790	0.803	T1	2	EM	kmeans	5.474
0.864	0.793	0.800	T1	2	EM	random	9.873
0.829	0.786	0.852	T1	3	EM	kmeans	7.352
0.779	0.768	0.847	T1	3	EM	random	20.331
0.841	0.819	0.875	T1	4	EM	kmeans	7.517
0.841	0.819	0.875	T1	4	EM	random	12.815
0.825	0.855	0.895	T1	5	EM	kmeans	7.985
0.817	0.852	0.892	T1	5	EM	random	10.892

Table 1: Dice Scores for each tissue in all patients with different initialization for EM: Random and Kmeans

CSF	GM	WM	Mod.	subject	model	init	time
0.9099	0.8269	0.8605	T1+T2	1	EM	kmeans	11.85
0.9099	0.8269	0.8605	T1+T2	1	EM	random	13.89
0.8045	0.7455	0.8026	T1+T2	2	EM	kmeans	11.01
0.8045	0.7448	0.8028	T1+T2	2	EM	random	9.87
0.8573	0.7784	0.8549	T1+T2	3	EM	kmeans	15.50
0.8573	0.7784	0.8549	T1+T2	3	EM	random	20.33
0.9034	0.8403	0.8739	T1+T2	4	EM	kmeans	10.36
0.9034	0.8403	0.8739	T1+T2	4	EM	random	12.81
0.8708	0.8509	0.8947	T1+T2	5	EM	kmeans	10.32
0.8710	0.8511	0.8946	T1+T2	5	EM	random	10.89

Table 2: Dice Scores for each tissue in all patients with different initialization for EM: Random and Kmeans using both volumes T1 and T2

CSF	GM	WM	modalities	model	init	t(s)
0.8692	0.8083	0.8573	T1+T2	EM	random	13.56
0.8692	0.8084	0.8573	T1+T2	EM	kmeans	11.81
0.8296	0.8051	0.8574	T1	EM	kmeans	7.21
0.8180	0.8017	0.8552	T1	EM	random	13.56

Table 3: Mean Dice Scores for each tissue with different modalities, models, and initialization for EM

Results: EM-Algorithm vs K-means

To compare the performance of the EM algorithm, we have implemented the K-Means clustering algorithm to assess the quality of segmentations produced by each method. In Figure 7, we present a visual comparison between the EM method and K-Means. However, visual assessment alone may not clearly indicate the superior method.

For a more quantitative comparison, we provide Table 4, which offers a detailed assessment of both methods.

CSF	GM	WM	Modalities	Model	Init	Time (s)
0.9315	0.7429	0.8879	T1	EM	kmeans$_EM$	5.595
0.9234	0.7872	0.7260	T1	Kmeans	none	1.555
0.8135	0.8436	0.8947	T1+T1	EM	kmeans$_EM$	9.341
0.9452	0.7670	0.7042	T1+T2	Kmeans	none	1.651

Table 4: Mean Dice Scores for each tissue with different modalities, models, and initialization for EM

Discussion

In the table 3, is observed that the random calculations take more time, than the k-means initialization. The difference in time can be attributed since the initial weights for the expected maximization lacks from a particular distribution making more difficult to calculated the values for encountering the minimum difference. In the Dice values are consistent across the different initialization, demonstrating that the algorithm can take different distributions as initial values can obtain in a very similar area as the ground truth.

In the next phase of the testing of our algorithm we tested the performance between using the EM and using a K-means for segmentation of the MRI images. In the table 4 we observed that the K-means is faster in comparison of our EM algorithm, however this impact on the Dice values since we are predicting only once without any optimization. In the other hand, our algorithm is slower since it is a optimization of the original weights from the k-means initialization. The maximization process is the one that lest us get higher values from the k-means only.

In summary, the EM algorithm is a valuable tool for brain tissue segmentation, but the choice of initialization method, particularly K-Means, can significantly enhance its performance. Using multiple modalities (T1+T2) also improved segmentation accuracy. However, it's important to note that there is no one-size-fits-all solution, and the choice of initialization may depend on specific imaging data and clinical requirements. Further research and fine-tuning of the algorithm may lead to even better results.

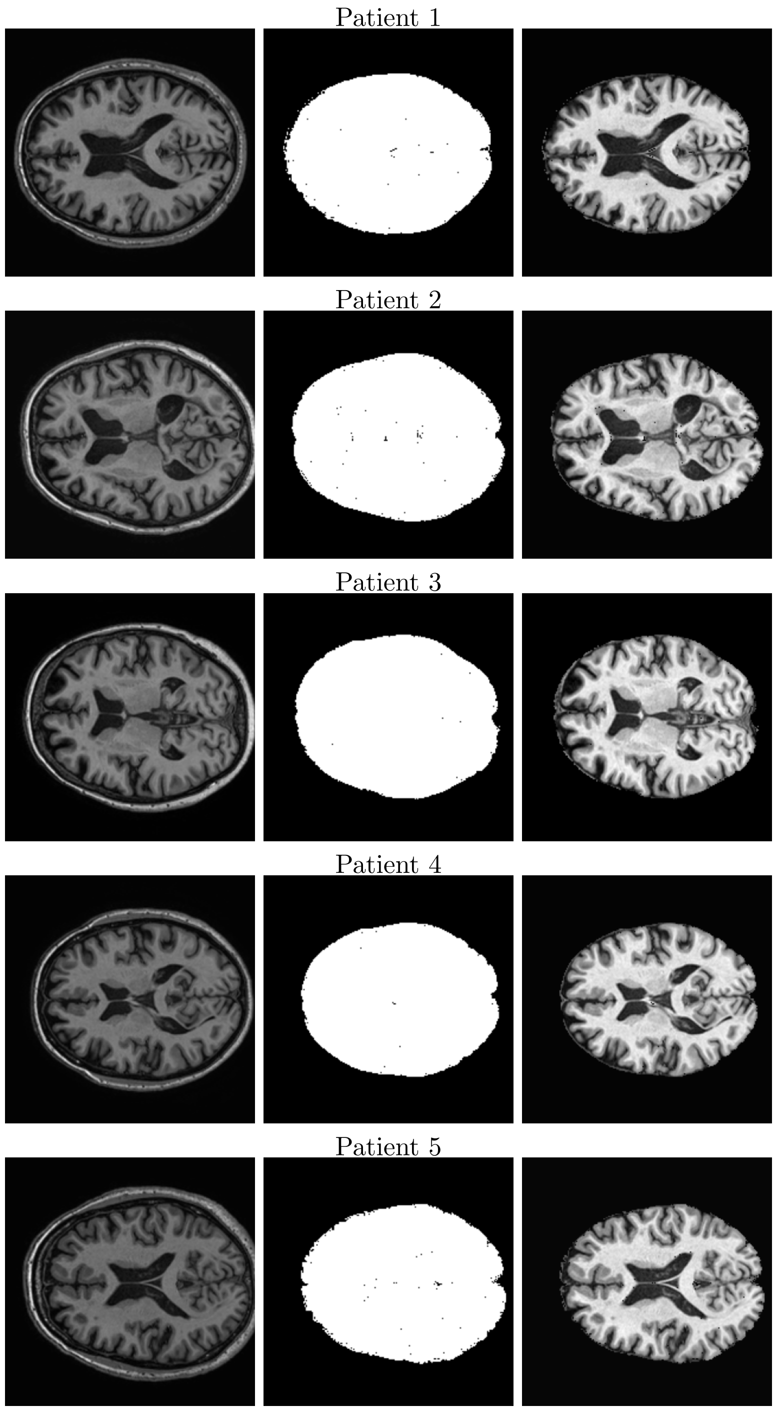

Figure 1: Example of Skull Stripping for each patient. Left to right: Original image T1. Brain Mask. Skull Stripping Result

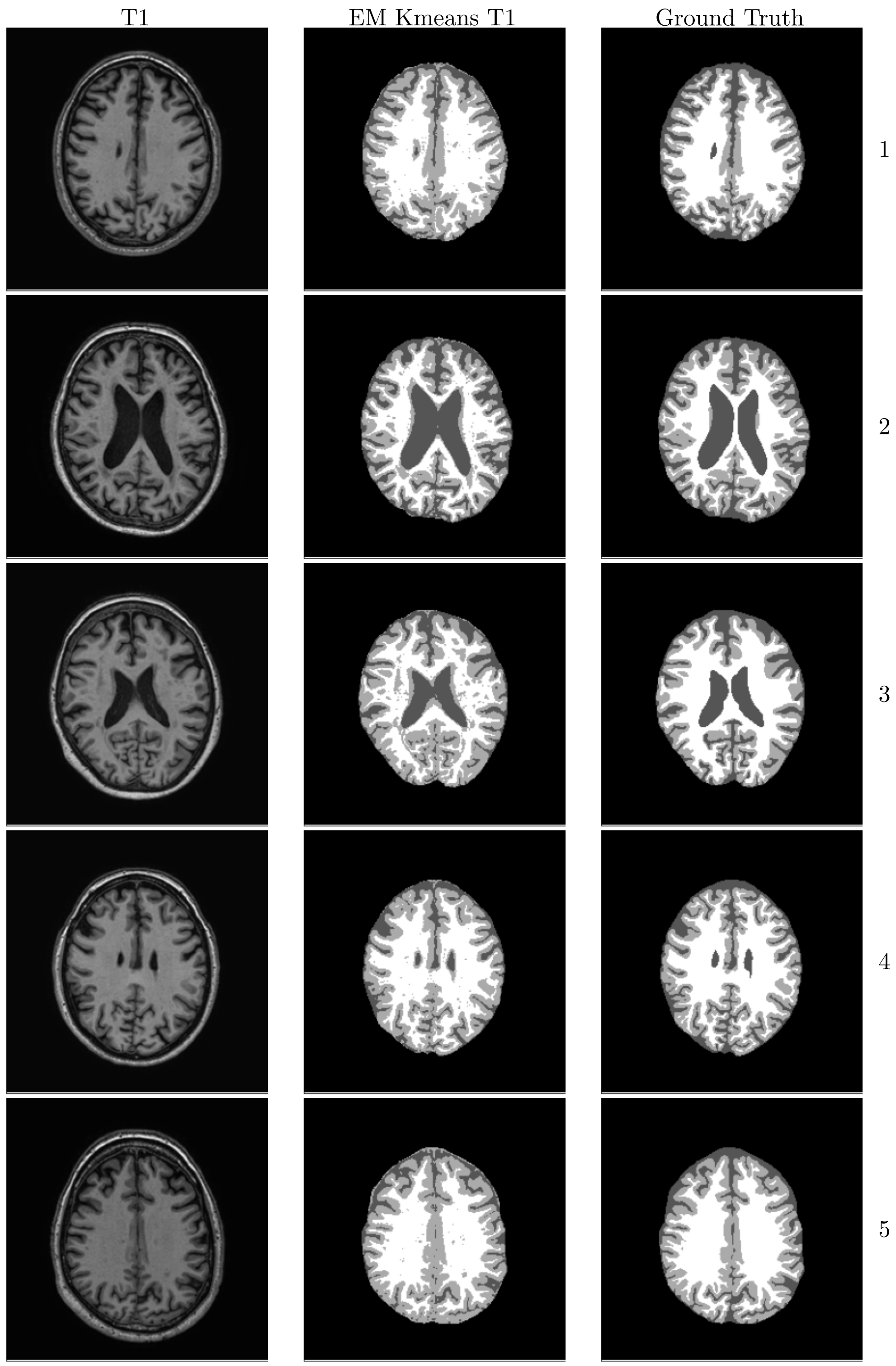

Figure 2: Example of EM algorithm initialization with K means for each patient using T1 volume. Left to right: Original image T1. EM K means Results. Ground truth labels

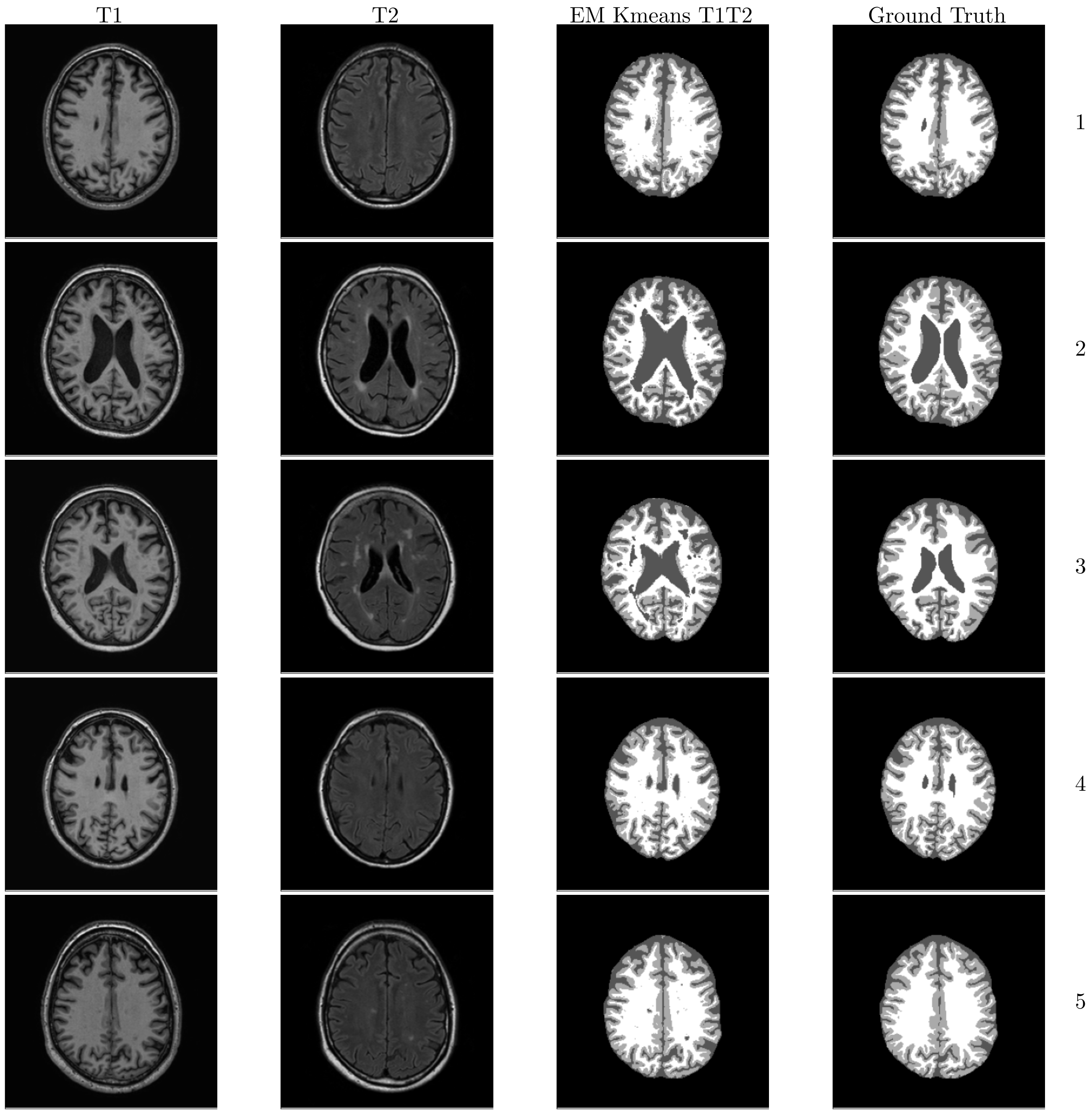

Figure 3: Example of EM algorithm initialization with K means for each patient using T1T2 volumes. Left to right: Original image T1. EM K means Results. Ground truth labels

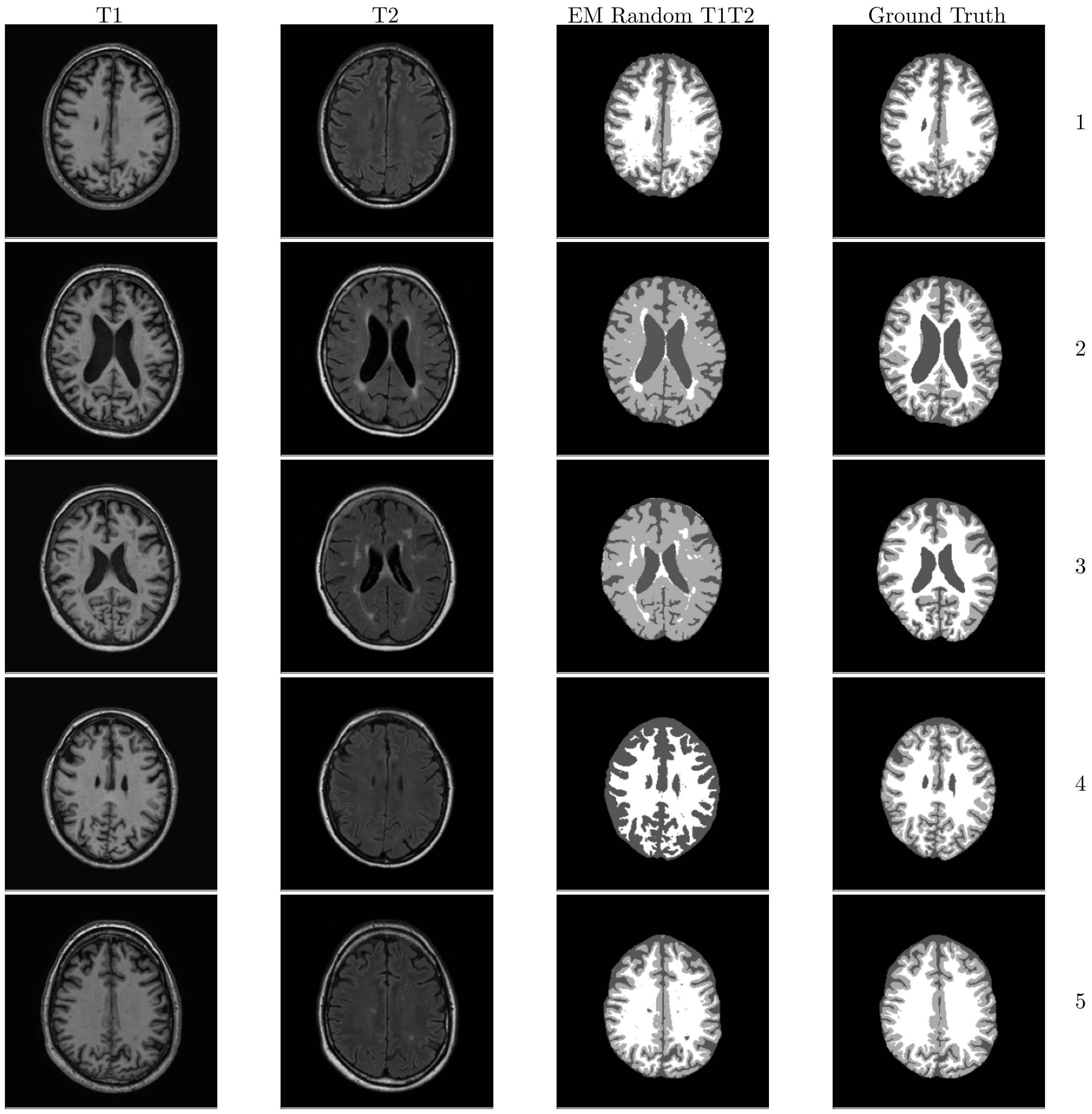

Figure 4: Example of EM algorithm randomly initialized using T1 and T2 volume. Left to right: Original image T1. Original T2 image. EM Random initialization using T1 and T2. Ground truth labels

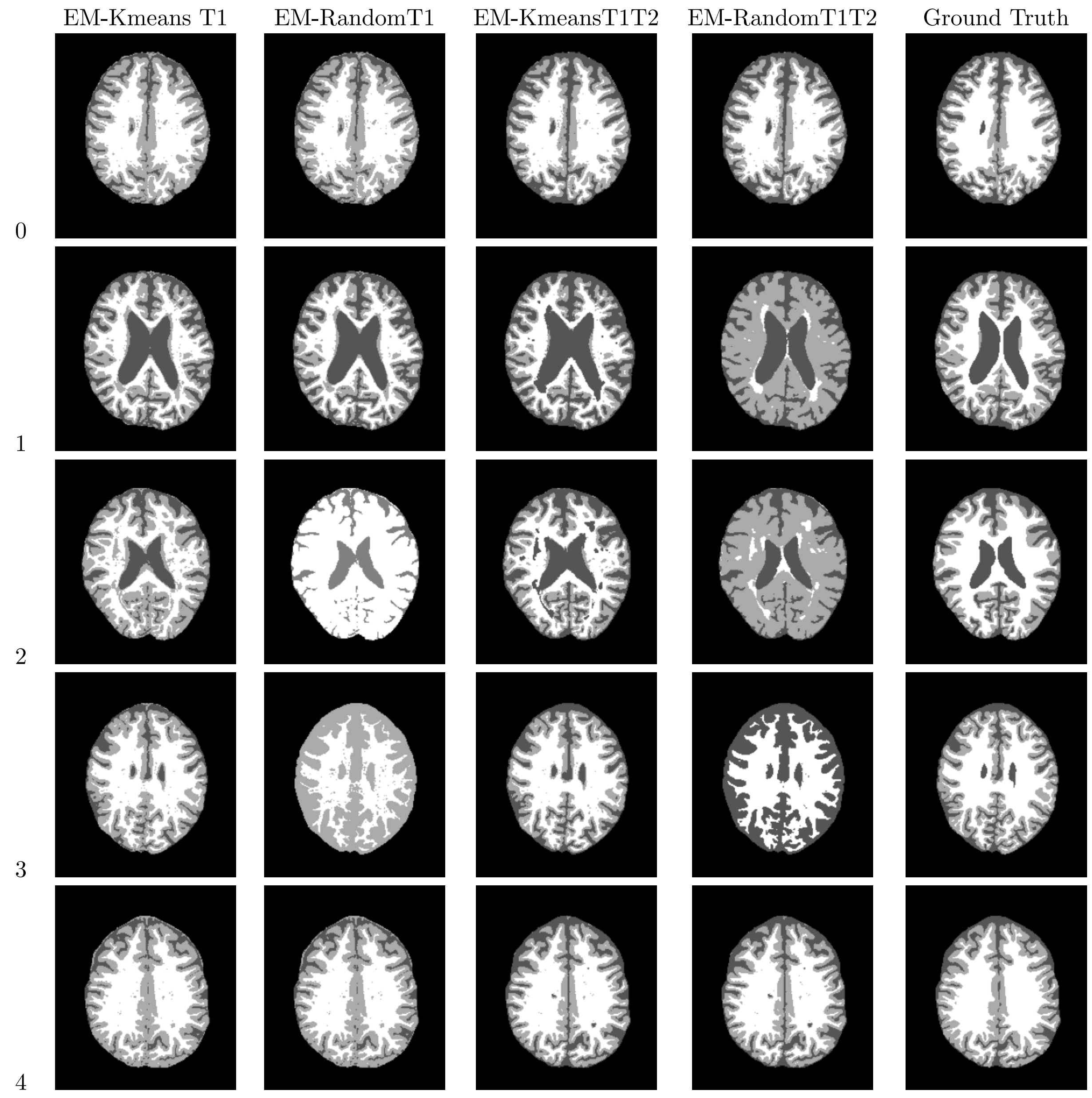

Figure 5: Comparison of different initialization for EM-Algorithm: Random and K means. Using different volumes: T1 and T1T2.

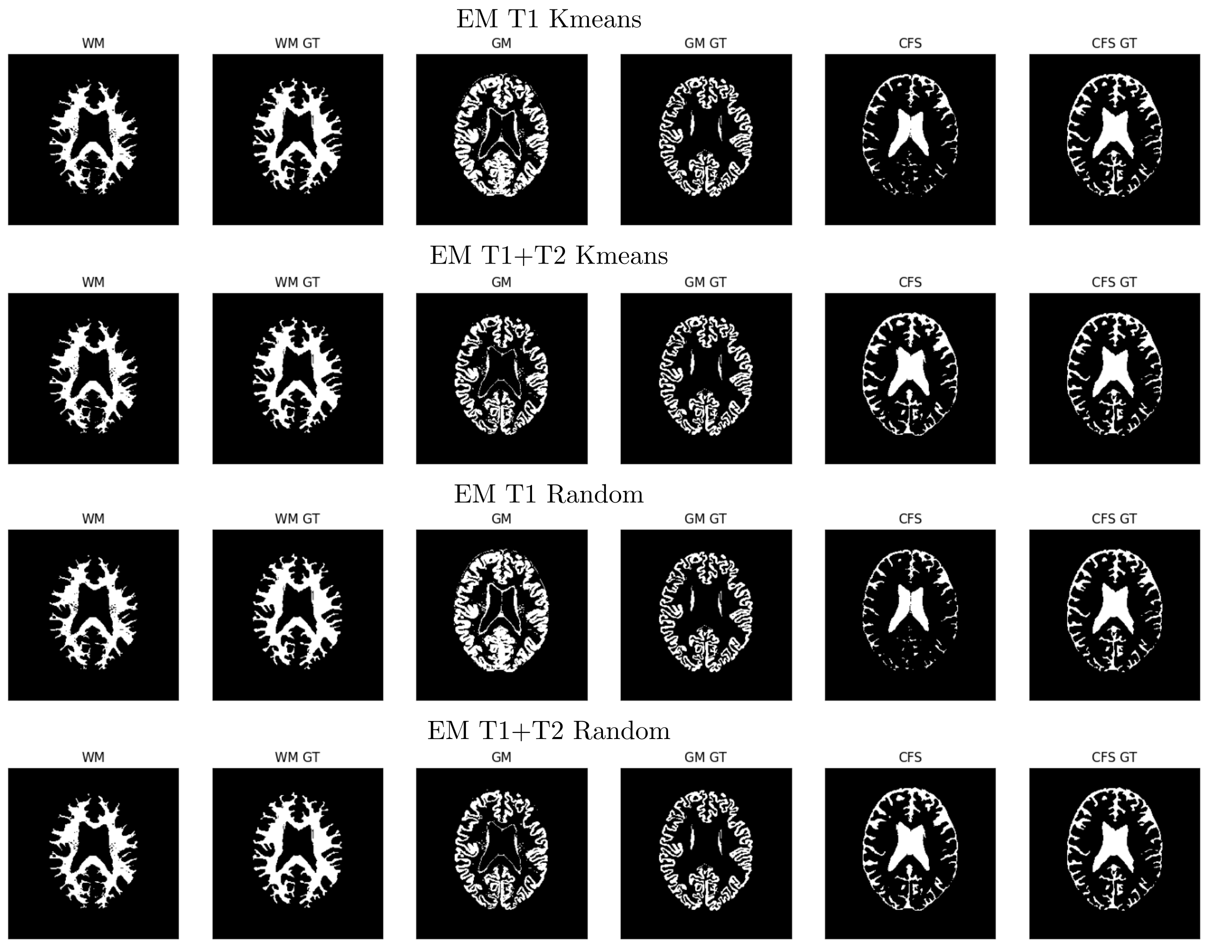

Figure 6: Example of segmentation for each tissue Result using different initialization of the EM-Algorithm. Top to bottom: EM T1 Kmeans. EM T1+T2 Kmeans. EM T1 Random. EM T1+T2 Random

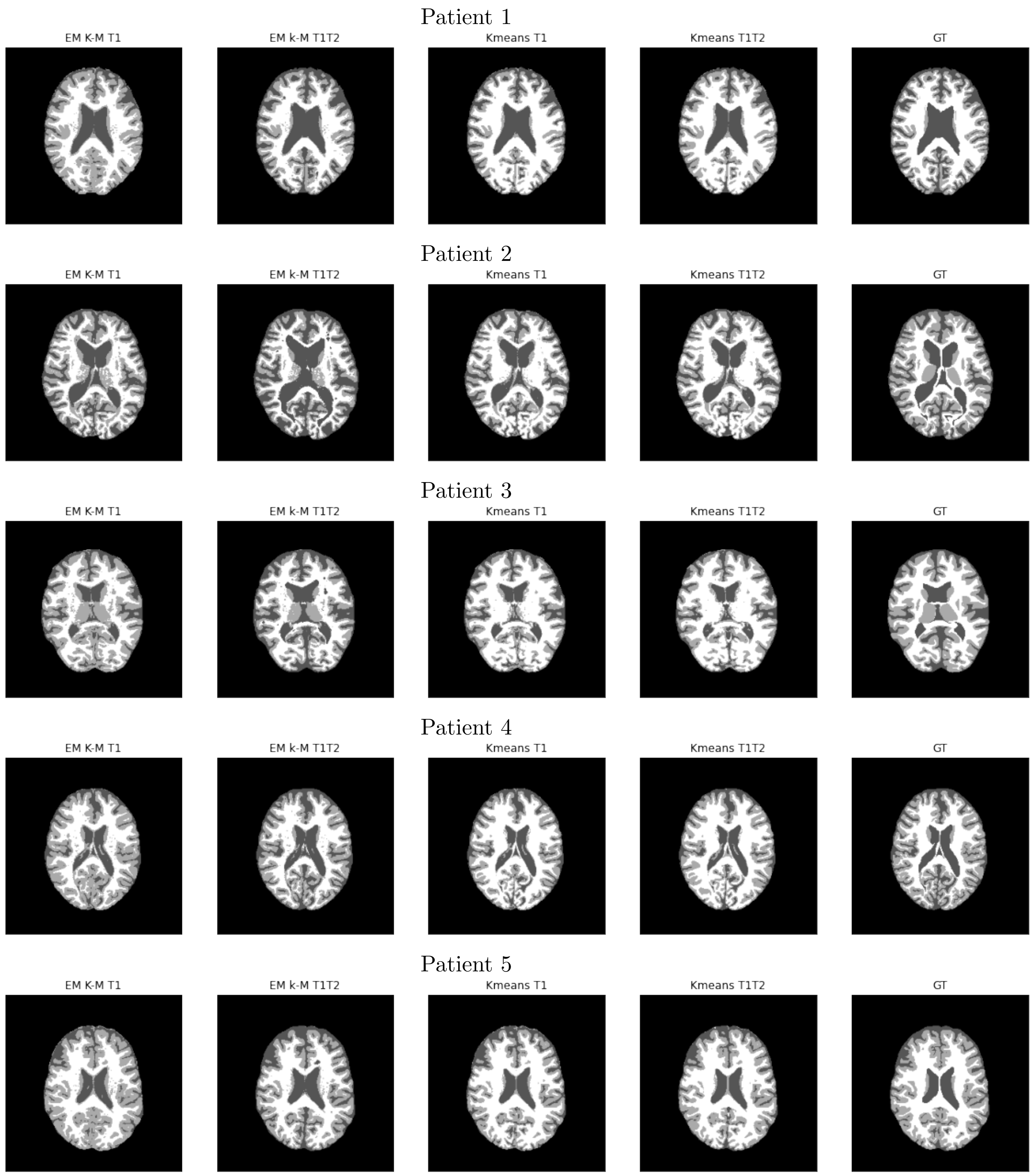

Figure 7: Comparison of EM-Algorithm and Kmeans algorithm using T1 volume and T1 and T2 volumes

Apendix

Python: Example EM code

```
import numpy as np
import cv2 as cv
import matplotlib.pyplot as plt
import os
import nibabel as nib
from tqdm import tqdm
from scipy.stats import multivariate_normal
from sklearn.cluster import KMeans
import time

# import warnings filter
from warnings import simplefilter
# ignore all future warnings
simplefilter(action='ignore', category=FutureWarning)

class maximum_expectation_algorithm:
    def __init__(self, expected_components=3, maximum_iteration=100, change_tolerance=1e-6, s
        self.expected_components = expected_components
        self.maximum_iteration = maximum_iteration
        self.change_tolerance = change_tolerance
        self.seed = seed

    def maximation_phase(self, x, posteriors):
        labels = np.zeros((x.shape[0], self.expected_components))
        labels[np.arange(x.shape[0]), np.argmax(posteriors, axis=1)] = 1

        posteriors = posteriors * labels
        counts = np.sum(posteriors, 0)
        weighted_avg = np.dot(posteriors.T, x)

        means = weighted_avg / counts[:, np.newaxis]

        sigmas = np.zeros((self.expected_components, x.shape[1], x.shape[1]))
        for i in range(self.expected_components):
            difference = x - means[i, :]
            weighted_difference = posteriors[:, i][:, np.newaxis] * difference
            sigmas[i] = np.dot(weighted_difference.T, difference) / counts[i]

        priors = counts / len(x)
        return means, sigmas, priors

    def expectation_phase(self, x, means, sigmas, priors):
        expected_components, _ = means.shape
        likelihood = [multivariate_normal.pdf(x, means[i, :], sigmas[i, :, :], allow_singular
                      for i in range(expected_components)]
        likelihood = np.asarray(likelihood).T
        num = np.asarray([likelihood[:, j] * priors[j] for j in range(expected_components)]).
        denom = np.sum(num, 1)

        posteriors = np.asarray([num[:, j] / denom for j in range(expected_components)]).T

        return posteriors, likelihood

    def get_initial_values(self, x, labels, covenge_regression=1e-9):
        expected_components = labels.shape[1]
        number_features = x.shape[1]
        minimum_value = 10 * np.finfo(labels.dtype).eps
        counts = np.sum(labels, axis=0) + minimum_value
        means = np.dot(labels.T, x) / counts[:, np.newaxis]
        sigmas = np.zeros((expected_components, number_features, number_features))
        for i in range(expected_components):
            difference = x - means[i, :]
            sigmas[i] = np.dot((labels[:, i][:, np.newaxis] * difference).T, difference) / co
            sigmas[i].flat[::number_features + 1] += covenge_regression
            if np.ndim(sigmas) == 1:
                sigmas = (sigmas[:, np.newaxis])[:, np.newaxis]
        return means, sigmas

    def tissue_segmentation(self, T1, T2, brain_mask, type='knn', operation='EM'):
        t1_array = self.min_max_normalization(T1)
        t2_array = self.min_max_normalization(T2)

        t1_vector = t1_array[brain_mask == 255].flatten()
        t2_vector = t2_array[brain_mask == 255].flatten()

        data_vector = np.array(t1_vector)[:, np.newaxis]

        if operation == 'EM':
            number_samples = len(data_vector)
            labels = np.zeros((number_samples, self.expected_components))

            if type == 'knn':
                kmeans = KMeans(n_clusters=self.expected_components, random_state=self.seed).
                labels[np.arange(number_samples), kmeans.labels_] = 1
            elif type == 'random':
                rng = np.random.default_rng(seed=self.seed)
                idx = rng.choice(self.expected_components, size=number_samples)
                labels[np.arange(number_samples), idx] = 1

            means, sigmas = self.get_initial_values(data_vector, labels)

            priors = np.ones((self.expected_components, 1)) / self.expected_components

            previous_log_likelihood = 0

            start_time = time.time()

            for it in tqdm(range(self.maximum_iteration),desc=f'T1: Expectation Maximination
                posteriors, likelihood = self.expectation_phase(data_vector, means, sigmas,

                for i in range(self.expected_components):
                    likelihood[:, i] = likelihood[:, i] * priors[i]

                log_likelihood = np.sum(np.log(np.sum(likelihood, 1)), 0)
                difference = abs(previous_log_likelihood - log_likelihood)
                previous_log_likelihood = log_likelihood

                if difference < self.change_tolerance:
                    break

                means, sigmas, priors = self.maximation_phase(data_vector, posteriors)

            posteriors, likelihood = self.expectation_phase(data_vector, means, sigmas, prio

            predictions = np.argmax(posteriors, 1)

            predictions_image = brain_mask.flatten()
            predictions_image[predictions_image == 255] = predictions + 1

            t1_segmentation_result = predictions_image.reshape(T1.shape)

            t1_time = time.time() - start_time

        if operation == 'Kmeans':
            start_time = time.time()
            model = KMeans(n_clusters=self.expected_components, random_state=self.seed).fit(
            predictions = model.predict(data_vector)
```

```
            predictions_image = brain_mask.flatten()
            predictions_image[predictions_image == 255] = predictions + 1
            t1_segmentation_result = predictions_image.reshape(T1.shape)
            t1_time = time.time() - start_time

        data_vector = np.array([t1_vector, t2_vector]).T

        if operation == 'EM':
            number_samples = len(data_vector)
            labels = np.zeros((number_samples, self.expected_components))

            if type == 'knn':
                kmeans = KMeans(n_clusters=self.expected_components, random_state=self.seed).
                labels[np.arange(number_samples), kmeans.labels_] = 1
            elif type == 'random':
                rng = np.random.default_rng(seed=self.seed)
                idx = rng.choice(self.expected_components, size=number_samples)
                labels[np.arange(number_samples), idx] = 1

            means, sigmas = self.get_initial_values(data_vector, labels)

            priors = np.ones((self.expected_components, 1)) / self.expected_components

            previous_log_likelihood = 0

            start_time = time.time()

            for it in tqdm(range(self.maximum_iteration),desc=f'T1+T2: Expectation Maximinati
                posteriors, likelihood = self.expectation_phase(data_vector, means, sigmas, 

                for i in range(self.expected_components):
                    likelihood[:, i] = likelihood[:, i] * priors[i]

                log_likelihood = np.sum(np.log(np.sum(likelihood, 1)), 0)
                difference = abs(previous_log_likelihood - log_likelihood)
                previous_log_likelihood = log_likelihood

                if difference < self.change_tolerance:
                    break

                means, sigmas, priors = self.maximation_phase(data_vector, posteriors)

            posteriors, likelihood = self.expectation_phase(data_vector, means, sigmas, prior
            predictions = np.argmax(posteriors, 1)

            predictions_image = brain_mask.flatten()
            predictions_image[predictions_image == 255] = predictions + 1

            t2_segmentation_result = predictions_image.reshape(T1.shape)

            t2_time = time.time() - start_time

        if operation == 'Kmeans':
            start_time = time.time()
            model = KMeans(n_clusters=self.expected_components, random_state=self.seed).fit(
            predictions = model.predict(data_vector)
            predictions_image = brain_mask.flatten()
            predictions_image[predictions_image == 255] = predictions + 1
            t2_segmentation_result = predictions_image.reshape(T1.shape)
            t2_time = time.time() - start_time

        return t1_segmentation_result, t2_segmentation_result, t1_time, t2_time

    def min_max_normalization(self, image):
        image = (image - image.min()) / (image.max() - image.min()) * 255
        return image

    def match_pred_with_gt(self, prediction, ground_truth):
        wm = np.zeros_like(prediction)
        gm = np.zeros_like(prediction)
        cfs = np.zeros_like(prediction)

        for tets_prob in range(self.expected_components):
            probs = []
            for prob in range(self.expected_components):
                gt_layer = np.where(ground_truth == prob + 1, 1, 0)
                test = np.where(prediction == tets_prob + 1, 1, 0)
                intersection = np.sum(np.logical_and(gt_layer, test))
                union = np.sum(np.logical_or(gt_layer, test))
                iou = intersection / union
                probs.append(iou)

            max_location = probs.index(max(probs))

            if (max_location + 1) == 1:
                cfs[prediction == tets_prob + 1] = max_location + 1
            if (max_location + 1) == 2:
                gm[prediction == tets_prob + 1] = max_location + 1
            if (max_location + 1) == 3:
                wm[prediction == tets_prob + 1] = max_location + 1

        return wm, gm, cfs

    def dice_score(self, ground_truth, prediction):
        classes = np.unique(ground_truth[ground_truth != 0])
        dice = np.zeros((len(classes)))
        for i in classes:
            binary_prediction = np.where(prediction == i, 1, 0)
            binary_ground_truth = np.where(ground_truth == i, 1, 0)
            dice[i - 1] = np.sum(binary_prediction[binary_ground_truth == 1]) * 2.0 / (
                    np.sum(binary_prediction) + np.sum(binary_prediction))
        return dice.tolist()

    def create_masks(self):
        for i in tqdm(range(5),desc='Creating masks....'):
            brain_mask_img = nib.load(os.path.join('..', 'P2_Data', f'{i}', 'LabelsForTesting
            brain_mask_array = brain_mask_img.get_fdata()
            brain_mask_array = np.where(brain_mask_array > 0, 255, 0).astype('uint8')
            mask_image = nib.Nifti1Image(brain_mask_array, brain_mask_img.affine)
            nib.save(mask_image, os.path.join('..', 'P2_Data', f'{i}', 'brainMask.nii'))

    def get_plots(self, volumes, names, slice_to_display: int = 25):
        n = len(volumes)
        fig, ax = plt.subplots(1, n, figsize=(20, 5))
        for i in range(n):
            ax[i].set_title(names[i])
            ax[i].imshow(volumes[i][slice_to_display, :, :], cmap='gray')
            ax[i].set_xticks([])
            ax[i].set_yticks([])
        plt.show()
```