vclsi-4-Schultz-Ibarra

June 4, 2018

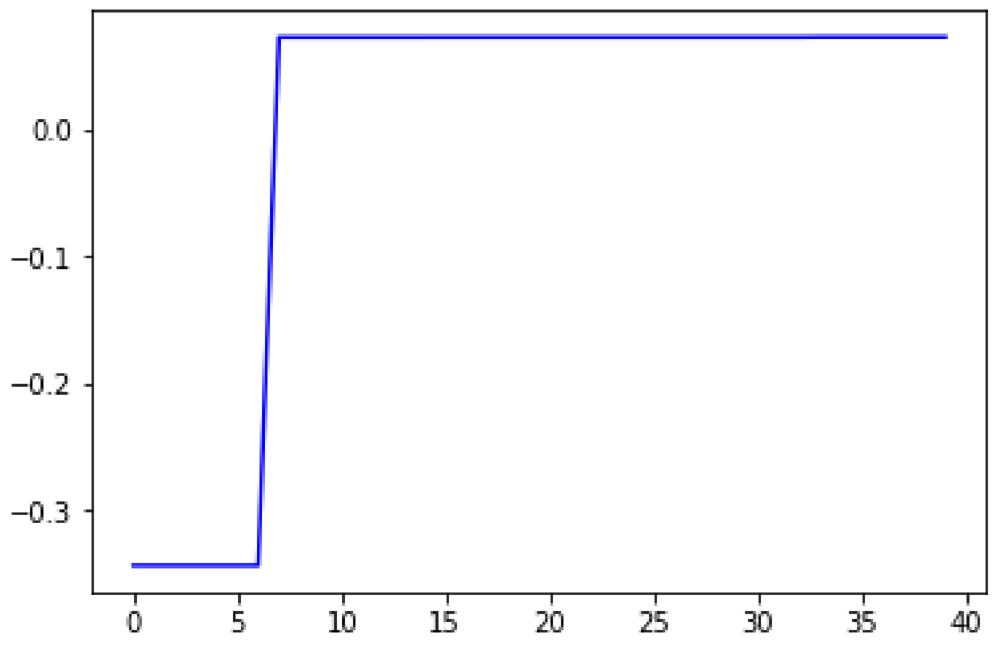
1 Assignment Sheet 4

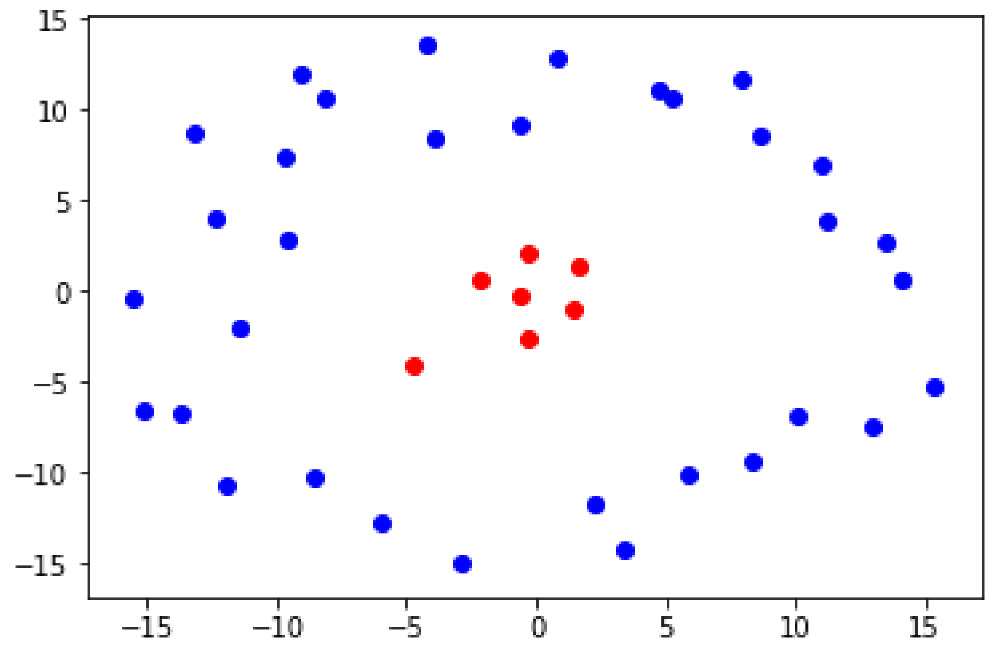
Bruce Schultz
bschultz@uni-bonn.de
Miguel A. Ibarra-Arellano
ibarrarellano@gmail.com

1.1 Exercise 1

```
In [22]: import numpy as np
         import matplotlib
         import matplotlib.pyplot as plt

         def get_point_edges(p, sigma_distance, edge_radius):
             """
             This function constructs a graph describing similarity of points in the given
                array.

             :param p: An array of shape (nPoint,2) where each row provides the
                       coordinates of one of nPoint points in the plane.
             :param sigma_distance: The standard deviation of the Gaussian distribution used
                       to weigh down longer edges.
             :param edge_radius: A positive float providing the maximal length of edges.
             :return:  tuple (edge_weight,edge_indices) where edge_weight is an array of
                        length n_edge providing the weight of all produced edges and
                        EdgeIndices is an integer array of shape (n_edge,2) where each row
                        provides the indices of two pixels which are connected by an edge.
             """
             # Initialize lists
             weights = list()
             indices = list()

             # Iterate over points
             for i in range(len(p)):
                 for j in range(i + 1, len(p)):

                     # If less than edge radius then store indices an weights
```

```
            fx = ((p[i][0] - p[j][0]) ** 2) + ((p[i][1] - p[j][1]) ** 2)
            if fx ** 0.5 < edge_radius:
                c = np.exp(-(fx / (2 * (sigma_distance ** 2))))
                weights.append(c)
                indices.append((i, j))

    return weights, indices


def get_laplacian(n, weights, indices):
    """
    Constructs a matrix providing the Laplacian for the given graph.

    :param n: The number of vertices in the graph (resp. pixels in the image).
    :param weights: A one-dimensional array of nEdge floats providing the weight
             for each edge.
    :param indices: An integer array of shape (nEdge,2) where each row provides
             the vertex indices for one edge.
    :return: A matrix providing the Laplacian for the given graph.
    """
    # Empty matrix filled with zeros
    adjacency = np.zeros((n, n))
    degree = np.zeros((n, n))

    # Iterate over weights
    for k in range(len(weights)):
        adjacency[indices[k][0], indices[k][1]] = adjacency[indices[k][1], indices[k]
        degree[indices[k][0], indices[k][0]] += weights[k]
        degree[indices[k][1], indices[k][1]] += weights[k]

    return degree - adjacency


def get_fiedler_vector(laplacian):
    """
    Given the Laplacian matrix of a graph this function computes the normalized
    Eigenvector for its second-smallest Eigenvalue (the so-called Fiedler vector)
    and returns it.

    :param laplacian: Laplacian matrix
    :return: laplacian matrix normalized by the Fiedler vector
    """

    return np.linalg.eigh(laplacian)[1][:, 1]

if (__name__ == "__main__"):
    # This list of points is to be clustered
    points = np.asarray(
        [(-8.097, 10.680), (-3.902, 8.421), (-9.711, 7.372), (0.859, 12.859), (4.732,
         (-4.224, 13.585), (-9.066, 11.891), (-13.181, 8.663), (-12.374, 3.983), (-11
         (-13.665, -6.667), (-15.521, -0.454), (-15.117, -6.587), (-11.970, -10.621),
         (-2.853, -14.978), (-8.501, -10.217), (2.311, -11.670), (3.441, -14.171), (5
         (15.382, -5.215), (14.091, 0.675), (11.187, 3.903), (8.685, 8.502), (7.879,
         (11.025, 6.888), (13.446, 2.612), (12.962, -7.393), (8.363, -9.330), (-0.594
         (1.424, -1.019), (-0.351, -2.552), (-2.127, 0.675), (-0.271, 2.128), (-4.743

    n_vertex = points.shape[0]

    # Construct the graph for the points
    edge_weight, edge_indices = get_point_edges(points, 1.0, 7.0)

    # Construct the Laplacian matrix for the graph
    laplacian = get_laplacian(n_vertex, edge_weight, edge_indices)

    # Compute the Fiedler vector
    fiedler_vector = get_fiedler_vector(laplacian)

    # Show the results
    plt.plot(list(range(0, len(fiedler_vector))), sorted(fiedler_vector), c="b")
    plt.show()

    for i, point in enumerate(points):
        if fiedler_vector[i] > -0.1:
            plt.scatter(point[0], point[1], color="b")
        else:
            plt.scatter(point[0], point[1], color="r")
#     plt.show()
```





a. Plot the sorted coeffcients of the Fiedler vector. Do they make it easy to define a suitable threshold to obtain a clear clustering?

Yes, they help.

b. What threshold would you choose and why? (3P)

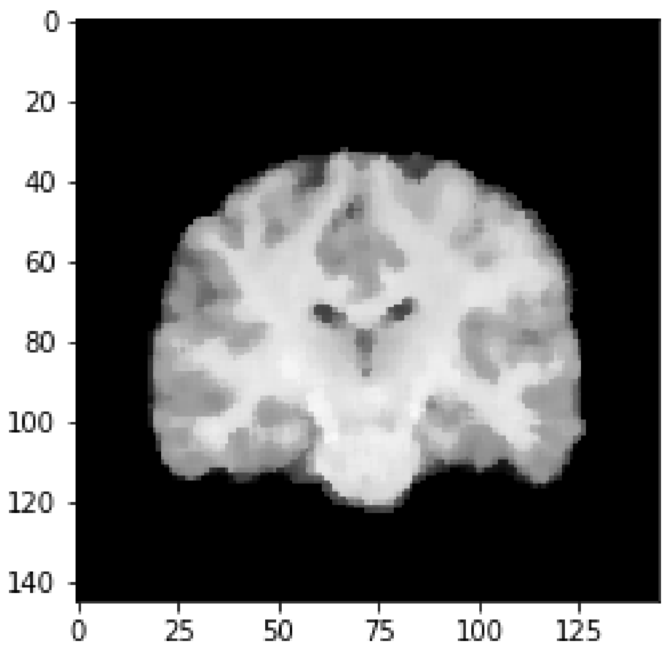
-0.1 divides perfectly the 2 groups

1.2 Exercise 2

```
In [1]: import numpy as np
        import imageio
        from skimage import color
        from sklearn import mixture
        from scipy import ndimage, misc
        import matplotlib.pyplot as plt
        import pandas as pd
        from gaussian_mixture_model import *
        from PIL import Image
```

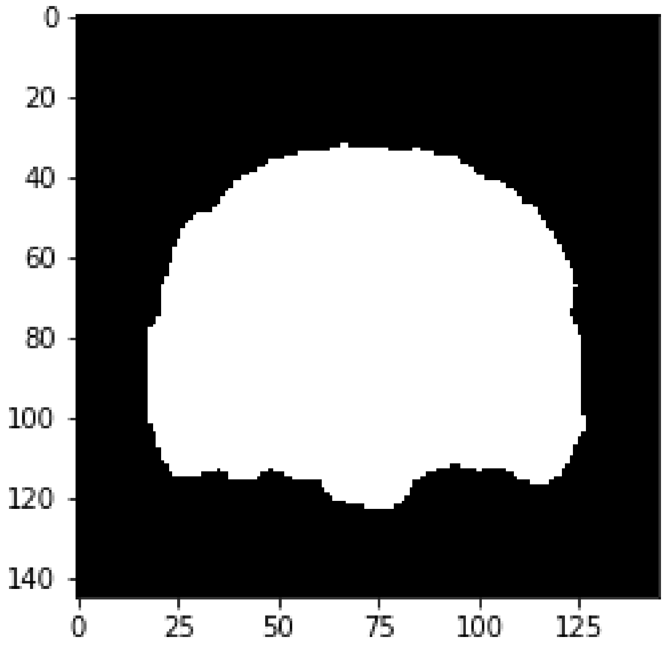
a) Read the grayscale image brain.png, which is provided on the lecture homepage. Reduce the salt and pepper noise in the image using a median filter. (3P)

```
In [23]: # Read image into array data
         raw_img_read = plt.imread("brain-noisy.png", True)

         # Denoising image
         mf_img = ndimage.median_filter(raw_img_read, size=5)

         plt.imshow(mf_img)

Out[23]: <matplotlib.image.AxesImage at 0x23d45f55588>
```



b) Produce a binary mask that marks all pixels with an intensity greater than zero. In all further steps, only treat pixels within that mask. (1P)

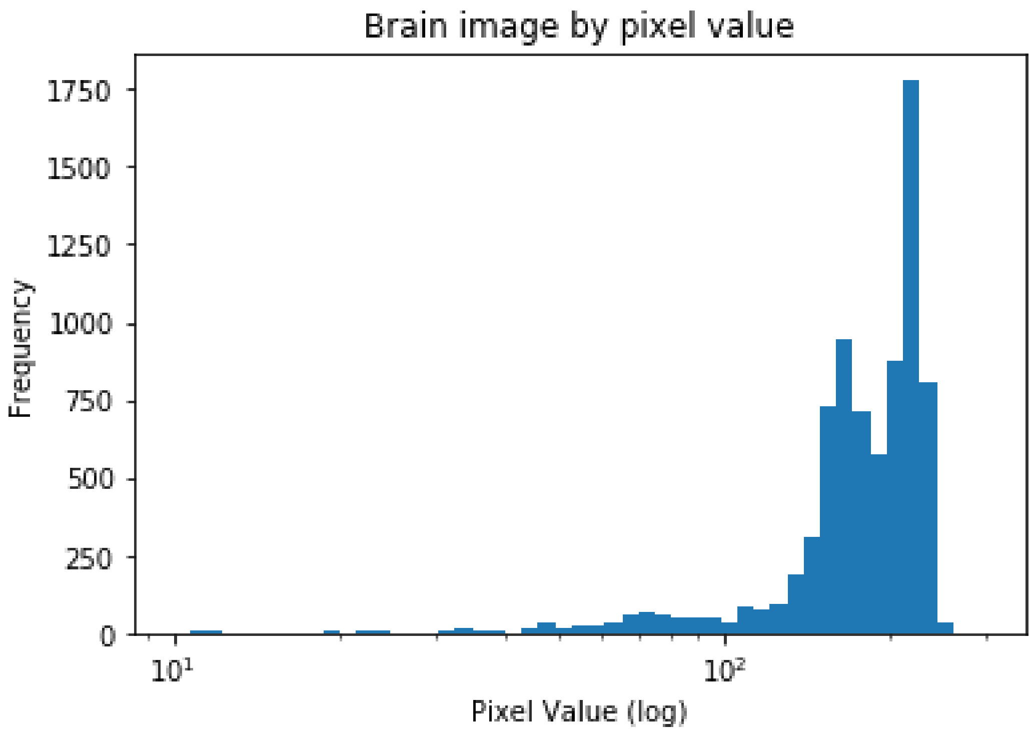
```
In [24]: # Creating binary mask
         binary_mask = mf_img > 0  # Any value greater than 0 (background)
         bin_masked_img = mf_img.copy()
         bin_masked_img[binary_mask] = 255  # 255 == white

         plt.imshow(bin_masked_img)

Out[24]: <matplotlib.image.AxesImage at 0x23d45fb40b8>
```



c) Plot a log-scaled histogram of the pixels within the mask. It should show how frequently different intensity values occur in the image. What do the peaks in this histogram represent? Hint: One way to and out is to create masks that highlight the pixels belonging to each peak. (4P)

```
In [4]: # Plot values from non-background pixels on a log scaled histogram
        bins = 50
        plt.gca().set_xscale("log")
        counts, pixels, bars = plt.hist(mf_img[binary_mask], np.logspace(np.log10(10), np.log1
        plt.xlabel("Pixel Value (log)")
        plt.ylabel("Frequency")
        plt.title("Brain image by pixel value")
        # plt.show()  # Peaks refer to segmentation thresholds, gray/white matter and backgrou
```

```
Out[4]: Text(0.5,1,'Brain image by pixel value')
```

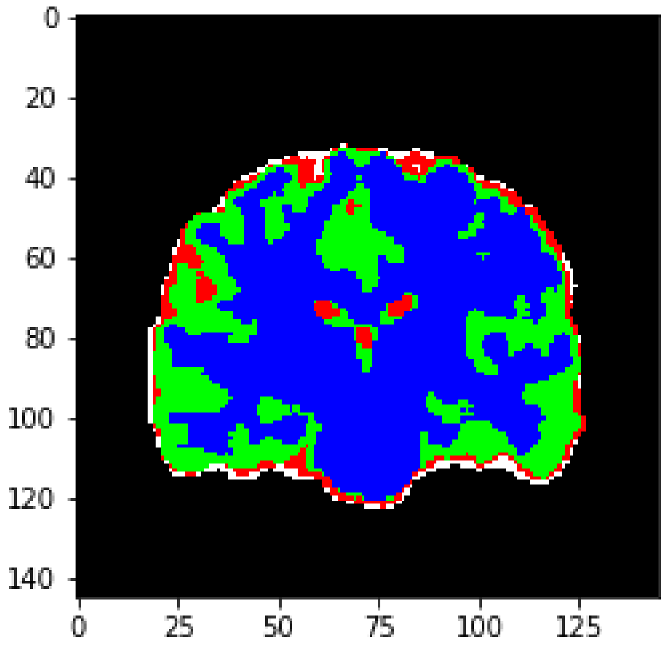


The peaks in this plot represent the different classes within the image, specifically the different parts of the brain. Each peak shows the pixel intensity that is most associated with that brain anatomy.

```
In [5]: plt.close()
        # Create masks for the pixel values surrounding each peak in histogram

        # Determine histogram peaks and the corresponding pixel value
        peak_values = []
        threshold = 75
        for i in range(len(counts)-1):
            if counts[i] > threshold and counts[i] > counts[i-1] and counts[i] > counts[i+1]:
                peak_values.append(pixels[i])

        #  Visualize image with peak pixel value locations after converting to RGB array
        masks = []
        pix_range = 40  # To give an acceptable range for pixel values
        for pix_value in peak_values:
            masks.append(np.logical_and(pix_value+pix_range >= mf_img, mf_img >= pix_value-pix_

        peak_img = bin_masked_img.copy()
        peak_img = color.gray2rgb(peak_img)  # Convert to RGB array
```

```
prime_colors = [[255, 0, 0], [0, 255, 0], [0, 0, 255]]  # Define primary colors [R, G,
for counter, mask in enumerate(masks):
    if counter > 2:
        break
    peak_img[mask] = prime_colors[counter]

plt.imshow(peak_img)
```

```
Out[5]: <matplotlib.image.AxesImage at 0x23d4461f4a8>
```



Note that since I only used pixel intensities that were within 40 units of each peak, some pixels in the image remain white as they were not included in the range

d) Now, we will use a three-compartment Gaussian Mixture Model for image segmentation: Based on their gray level, pixels that fall within the mask from c) should be assigned to one of three Gaussians, capturing corticospinal uid (dark), gray matter (medium), or white matter (bright). To start this process, initialize the parameters of a three-compartment GMM to some reasonable values and use them to compute the responsibilities pik of cluster k for pixel i. (4P)

```
In [6]: # Create some GMM functions

        from random import randint
        from math import pi, exp, sqrt
        import numpy as np


        def GMM_init(data_points, n_distributions, means_init=None):
            """
```

```
    Initializes N gaussian distributions for use in GMM modeling
    :param data_points: 2D array containing data of interesting for GMM
    :param n_distributions: Number of clusters predicted to be found
    :param means_init: Optional, means to start the clustering process
    :return: Separate 1D arrays for mixing coefficients, variance values, and means
    """
    mix_coeff = [1/n_distributions] * n_distributions  # Sum of pi across all clusters
    if means_init is not None:
        means = means_init[:, 1]  # Set initial means to user specified values
    else:
        # Random initialization using y values ([:,1])
        means = [randint(min(data_points[:, 1]), max(data_points[:, 1])) for i in range

    # Initialize variance to sig**2 = sum(X-mu)**2 / N
    init_variance = sum([(data_points[i, 1]-min(means))**2 for i in range(len(data_poi
    sigma = [sqrt(init_variance/len(data_points[:, 1]))] * n_distributions
    return mix_coeff, sigma, means

# TODO implement k-means init

# E-step of GMM algorithm
def GMM_responsibilities(data_points, n_distributions, mix_coeff, sigma, means):

    # Calculate gaussians
    # GMM array has x values in first column and GMM sum value in the last column
    GMM_array = np.empty((len(data_points[:, 1]), n_distributions + 2))
    GMM_array[:, 0] = data_points[:, 1]  # First column is our values
    for i in range(len(data_points[:, 1])):  # Iterate through values
        for k in range(n_distributions):  # Iterate through clusters
            gauss = 1/(sqrt(2*pi)*sigma[k])*exp(-((data_points[i, 1]-means[k])**2)/(2*
            GMM_array[i, k+1] = mix_coeff[k]*gauss
        GMM_array[i][n_distributions + 1] = sum(GMM_array[i][1:n_distributions+1])  #

    # Calculate responsibilities
    responsibilities = np.empty((len(data_points[:, 1]), n_distributions))
    for i in range(len(data_points[:, 1])):
        for k in range(n_distributions):
            responsibilities[i][k] = GMM_array[i, k+1]/GMM_array[i, n_distributions+1]

    # Only responsibilities values! i (sample #) rows by k (cluster #) columns
    return responsibilities


# M-step of GMM algorithm
def GMM_optimize(data_points, n_distributions, mix_coeff, sigma, means):

    rho = GMM_responsibilities(data_points, n_distributions, mix_coeff, sigma, means)
    # Create lists to fill with optimized values
    opt_mix_coeff = [0] * len(mix_coeff)
    opt_means = [0] * len(means)
    opt_sigma = [0] * len(sigma)

    # Optimize parameters
    for k in range(n_distributions):
        cluster_resp_sum = sum(rho[:, k])

    # Mixing Coefficients
        opt_mix_coeff[k] = cluster_resp_sum / len(data_points[:, 1])

    # Means
        mean_numerator = sum([rho[i][k]*data_points[i][1] for i in range(len(data_poin
        opt_means[k] = mean_numerator/cluster_resp_sum

    # Sigma
        sig_numerator = sum([(rho[i][k]*((data_points[i][1]-means[k])**2)) for i in ra
        opt_sigma[k] = sqrt(sig_numerator/cluster_resp_sum)

    return opt_mix_coeff, opt_sigma, opt_means, rho


def GMM_convergence(data_points, n_distributions, iterations=25, means_init=None, only_
    mix_coeff, sigma, means = GMM_init(data_points, n_distributions, means_init)

    if only_init:
        rho = GMM_responsibilities(data_points, n_distributions, mix_coeff, sigma, mean
        return mix_coeff, sigma, means, rho

    # Create list to track changes with every iteration
    mix_coefficient_list = [list(mix_coeff)]
    sigma_list = [list(sigma)]
    means_list = [list(means)]

    i = 0
    while i < iterations:
        mix_coeff, sigma, means, rho = GMM_optimize(data_points, n_distributions, mix_
        mix_coefficient_list.append(mix_coeff)
        sigma_list.append(sigma)
        means_list.append(means)
        i += 1
    return mix_coefficient_list, sigma_list, means_list, rho


def pixel_cluster_matcher(mask_template, cluster_assignment_list, cluster_number):
    """
    Uses a mask template to determine pixel location and iterates over new mask, chang
```

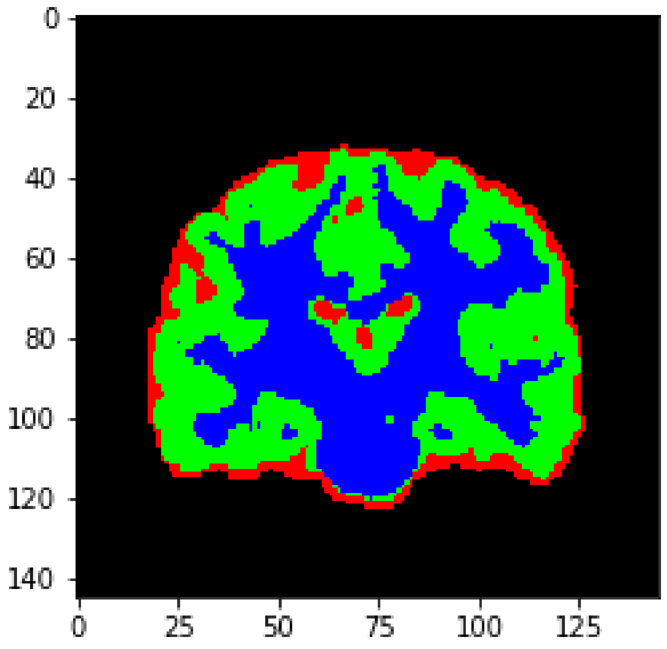
```
        values to false if they don't match cluster_number
        :param mask_template: Mask_template to use to determine pixels of interest to chan
        :param cluster_assignment_list: 1D array with cluster assignment for every pixel t
        :param cluster_number: Which cluster you are building this mask for
        :return: Mask with True values for only pixels at specified cluster_number locatio
        """
        new_mask = mask_template.copy()
        k = 0
        for pixel in np.nditer(new_mask, op_flags=['readwrite']):
            if pixel[...]:
                if cluster_assignment_list[k] != cluster_number:
                    pixel[...] = False
                k += 1
        return new_mask


    # TODO Make this iterate and fix functions to work together better -- use OOP?
```

```
In [7]: # Create 2D array with pixel number (x value) and pixel intensity (y value)
        gmm_data = np.column_stack(enumerate(mf_img[binary_mask])).transpose()
        points_init = np.array([[1, 2, 3], peak_values]).transpose()

        # Use homemade GMM functions to predict pixel clustering using only 1 iteration and no
        mix_coeff, sigma, means, responsibilities = GMM_convergence(gmm_data, 3,
                                                            iterations=iter, means_ini

        cluster_predictions = [np.argmax(sample) for sample in responsibilities]
        cluster_probabilities = [np.amax(sample) for sample in responsibilities]
```

e) Visualize the responsibilities by mapping the probabilities of belonging to the CSF, gray matter, and white matter clusters to the red, blue, and green color channels, respectively. Please submit the resulting image. (3P)

```
In [21]: # Copy the binary mask image and convert to RGB
         gmm_img = bin_masked_img.copy()
         gmm_img = color.gray2rgb(gmm_img)


         # Create masks for CSF, gray/white matter then assign them color layers
         csf_mask = pixel_cluster_matcher(binary_mask, cluster_predictions, 0)
         gray_mask = pixel_cluster_matcher(binary_mask, cluster_predictions, 1)
         white_mask = pixel_cluster_matcher(binary_mask, cluster_predictions, 2)

         gmm_img[csf_mask] = [255, 0, 0]
         gmm_img[gray_mask] = [0, 255, 0]
         gmm_img[white_mask] = [0, 0, 255]

         # Multiply each value by the probability of that pixel belonging to that class (darke
         # gmm_img[binary_mask] = [[value*cluster_probabilities[i] for value in pixel] for i, p
         plt.imsave('GMM_notoptimization.png', gmm_img)
         plt.imshow(gmm_img)
```

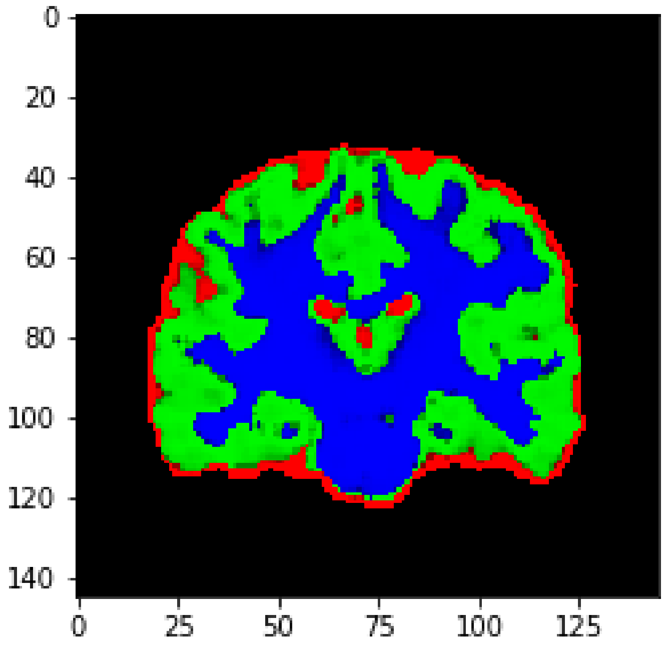
```
Out[21]: <matplotlib.image.AxesImage at 0x23d447fc0b8>
```



f) Use the update rules provided in the lecture to re-compute the parameters muk, sigmak, and pik. (4P)

```
In [19]: # Using my homemade GMM algorithm until convergence
         iter = 30
         mix_coeff, sigma, means, responsibilities = GMM_convergence(gmm_data, 3, iterations=i

         cluster_predictions = [np.argmax(sample) for sample in responsibilities]
         cluster_probabilities = [np.amax(sample) for sample in responsibilities]

         # Create masks for CSF, gray/white matter then assign them color layers
         csf_mask = pixel_cluster_matcher(binary_mask, cluster_predictions, 0)
         gray_mask = pixel_cluster_matcher(binary_mask, cluster_predictions, 1)
         white_mask = pixel_cluster_matcher(binary_mask, cluster_predictions, 2)

         gmm_img[csf_mask] = [255, 0, 0]
         gmm_img[gray_mask] = [0, 255, 0]
         gmm_img[white_mask] = [0, 0, 255]

         # Multiply each value by the probability of that pixel belonging to that class (darke
         gmm_img[binary_mask] = [[value*cluster_probabilities[i] for value in pixel] for i, pi
```

```
        plt.imsave('GMM_convergence.png', gmm_img)
        plt.imshow(gmm_img)

Out[19]: <matplotlib.image.AxesImage at 0x23d448eb860>
```



Here I increased the iterations to 30. When iterations > 0, then the optimizing function from above kicks in and update the parameters

g) Iterate the E and M steps of the algorithm until convergence. Please submit the final parameter values, a visualization of the final responsibilities, and your code. (3P)

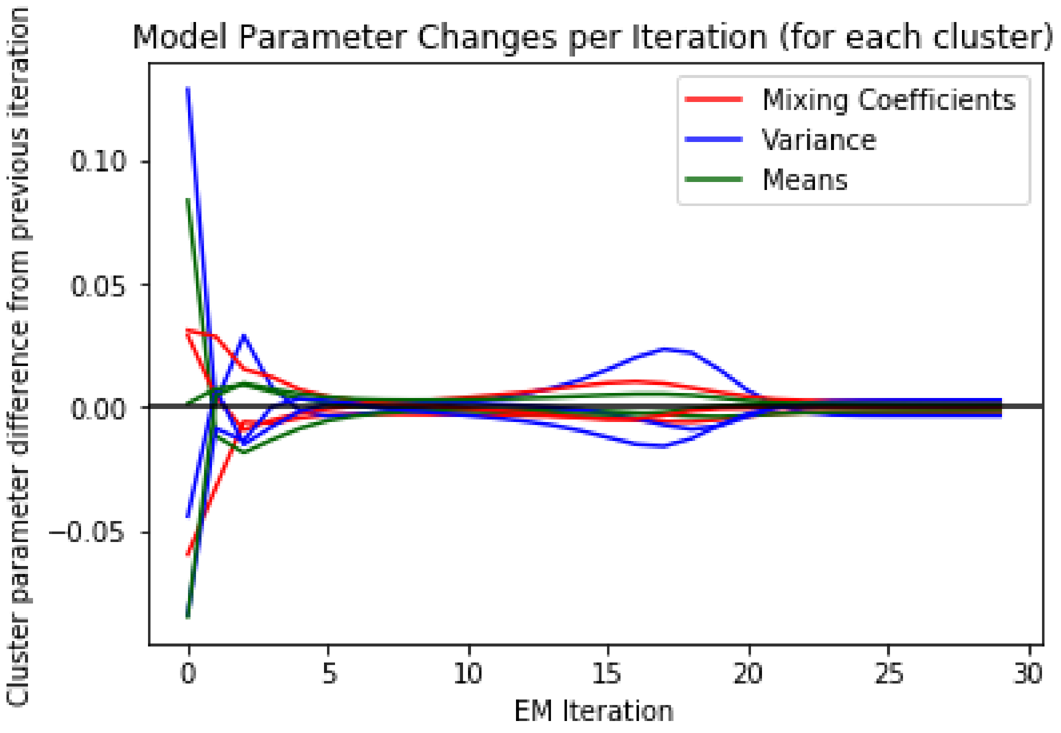
```
In [59]: print('Mixing Coefficients:', mix_coeff[-1], '\nVariances:', sigma[-1], '\nMeans:', me
         respons_df = pd.DataFrame(responsibilities, columns=['Cluster 1', 'Cluster 2', 'Clust
         print(respons_df.round(2))

Mixing Coefficients: [0.1396437862062487, 0.4549239195504564, 0.4054322942432959]
Variances: [42.53345983144453, 19.03189002807373, 9.932208187401812]
Means: [163.86472112236027, 181.14368829352125, 192.53123839690593]

Responsibilities:
      Cluster 1  Cluster 2  Cluster 3
0          1.00       0.00       0.00
1          1.00       0.00       0.00
2          1.00       0.00       0.00
3          1.00       0.00       0.00
4          0.63       0.37       0.00
5          0.11       0.89       0.00
6          0.50       0.50       0.00
```

```
7           0.78        0.22        0.00
8           0.75        0.25        0.00
9           0.95        0.05        0.00
10          1.00        0.00        0.00
11          1.00        0.00        0.00
12          1.00        0.00        0.00
13          1.00        0.00        0.00
14          1.00        0.00        0.00
15          1.00        0.00        0.00
16          1.00        0.00        0.00
17          1.00        0.00        0.00
18          1.00        0.00        0.00
19          1.00        0.00        0.00
20          1.00        0.00        0.00
21          1.00        0.00        0.00
22          1.00        0.00        0.00
23          1.00        0.00        0.00
24          1.00        0.00        0.00
25          1.00        0.00        0.00
26          1.00        0.00        0.00
27          0.44        0.56        0.00
28          0.11        0.89        0.00
29          0.02        0.97        0.01
...          ...         ...         ...
7835        0.01        0.29        0.70
7836        0.02        0.98        0.00
7837        0.02        0.98        0.00
7838        0.02        0.98        0.00
7839        0.03        0.97        0.00
7840        0.04        0.96        0.00
7841        1.00        0.00        0.00
7842        1.00        0.00        0.00
7843        1.00        0.00        0.00
7844        1.00        0.00        0.00
7845        1.00        0.00        0.00
7846        1.00        0.00        0.00
7847        1.00        0.00        0.00
7848        1.00        0.00        0.00
7849        1.00        0.00        0.00
7850        1.00        0.00        0.00
7851        1.00        0.00        0.00
7852        1.00        0.00        0.00
7853        1.00        0.00        0.00
7854        1.00        0.00        0.00
7855        1.00        0.00        0.00
7856        1.00        0.00        0.00
7857        1.00        0.00        0.00
7858        1.00        0.00        0.00
```

```
7859       1.00       0.00       0.00
7860       1.00       0.00       0.00
7861       1.00       0.00       0.00
7862       1.00       0.00       0.00
7863       1.00       0.00       0.00
7864       1.00       0.00       0.00

[7865 rows x 3 columns]
```

h) Create and submit a plot that illustrates the convergence of your algorithm. (3P)

```
In [60]: def track_changes(value_array):
             # Standardize each iteration element
             differences = []
             for i, iteration in enumerate(value_array):
                 value_array[i] = [value/sum(iteration) for value in iteration]
                 if i > 0:
                     differences.append([value_array[i][k]-value_array[i-1][k] for k in range(
             return list(map(list, zip(*differences)))

         # Graph the changes in model parameters with each iteration
         for i in range(len(track_changes(mix_coeff))):
             plt.plot(range(iter), track_changes(mix_coeff)[i], color='red', label='Mixing Coe
             plt.plot(range(iter), track_changes(sigma)[i], color='blue', label='Variance')
             plt.plot(range(iter), track_changes(means)[i], color='darkgreen', label='Means')
             if i == 0:
                 plt.legend(['Mixing Coefficients', 'Variance', 'Means'])
         plt.axhline(y=0, color='black', linestyle='-')
         plt.xlabel('EM Iteration')
         plt.ylabel('Cluster parameter difference from previous iteration')
         plt.title('Model Parameter Changes per Iteration (for each cluster)')

Out[60]: Text(0.5,1,'Model Parameter Changes per Iteration (for each cluster)')
```


Model Parameter Changes per Iteration (for each cluster)
Mixing Coefficients
Variance
Means
Cluster parameter difference from previous iteration
0.10
0.05
0.00
−0.05
0
5
10
15
20
25
30
EM Iteration