CEE 498 Applied Machine Learning - HW5

Rini Jasmine Gladstone (rjg7) and Maksymillian Podraza(podraza3)

April 17, 2020

1 Problem 1

1.1 Part a

We used the following image for our analysis.

We normalized the image so that the darkest pixel takes the value (0,0,0) and the lightest pixel takes the value (1,1,1).

```
image_paths = ["RobertMixed03.jpg", "smallstrelitzia.jpg", "
    smallsunset.jpg"]

def load_images():
    image_array = []
    try:
        for image_path in image_paths:
            img = Image.open(image_path)
            image_array.append(img)
    except IOError:
        pass
    return image_array

```

```
images = load_images()

working_image = images[0]

def image_matrix_create(image):
    image_matrix = []
    img_w, img_h = image.size
    image_data = list(image.getdata())
    for y in range(img_h):
        image_matrix.append(image_data[y*img_w:(y+1)*img_w])
    image_df = pd.DataFrame(image_matrix)
    return image_df

def rgb_extract(image):
    rgb = []
    for i in range(3):
        rgb.append(image.apply(lambda x: [y[i] for y in x]))
    return rgb

def rgb_normalize(image):
    normalized = []
    for i in range(3):
        x = image[i].values
        min_max_scaler = preprocessing.MinMaxScaler()
        x_scaled = min_max_scaler.fit_transform(x)
        df = pd.DataFrame(x_scaled)
        normalized.append(df)
    return normalized

r = image_matrix_create(working_image)
rgb_matrix_list = rgb_extract(r)
rgb_normalized_list = rgb_normalize(rgb_matrix_list)
```

Then, we wrote the function for EM. We used K means to initialize the mean vector.

```
def EM_function(X,cov,k,niter):
        weights = np.ones((k)) / k
        kmeans = KMeans(n_clusters=k)
        # fit kmeans object to data
        kmeans.fit(X)
        means = kmeans.cluster_centers_
        eps=1e-8
        likelihood = []
        log_likelihoods = []
        for step in range(niter):
            likelihood = []
            # Expectation step
            for j in range(k):
                likelihood.append(multivariate_normal.pdf(x=X, mean
    =means[j], cov=cov[j]))
            likelihood = np.array(likelihood)
            assert likelihood.shape == (k, len(X))
            b = []
            # Maximization step
            for j in range(k):
                # use the current values for the parameters to
    evaluate the posterior
```

```
                # probabilities of the data have been generated by each gaussian
                b.append((likelihood[j] * weights[j]) / (np.sum([likelihood[i] * weights[i] for i in range(k)], axis=0)+eps))
                # update mean and variance
                means[j] = np.sum(b[j].reshape(len(X),1) * X, axis=0) / (np.sum(b[j]+eps))
                # update the weights
                weights[j] = np.mean(b[j])
                assert cov.shape == (k, X.shape[1], X.shape[1])
                assert means.shape == (k, X.shape[1])
            log_likelihoods.append(np.log(np.sum([k*multivariate_normal(means[i],cov[j]).pdf(X) for k,i,j in zip(weights,range(len(means)),range(len(cov)))])))
            if (step+1)%100==0 and (step+1)>=100:
                print(step+1)
        return means,weights,b,log_likelihoods,likelihood
```

We set the covariance matrix (cov) to be identity matrix and ran the following code setting k = 10, 20 and 50.

```
rvalues=rgb_normalized_list[0].values.flatten()
gvalues=rgb_normalized_list[1].values.flatten()
bvalues=rgb_normalized_list[2].values.flatten()
list_of_tuples = list(zip(rvalues, gvalues, bvalues))
data=pd.DataFrame(list_of_tuples, columns = ['R', 'G', 'B'])
#Number of clusters
k=50
#Initializing the covariance matrix
cov = []
for i in range(k):
    cov.append(np.eye(data.shape[1]))
cov = np.array(cov)
means,weights,b,log_likelihoods,likelihood=EM_function(data,cov,k,200)
#Plotting the log-likelihood
plt.plot(log_likelihoods)
likelihood = np.array(likelihood)
#Finding the closest cluster for each pixel
predictions = np.argmax(likelihood, axis=0)
data['cluster']=predictions
data['mean_R']=np.zeros(data.shape[0])
data['mean_G']=np.zeros(data.shape[0])
data['mean_B']=np.zeros(data.shape[0])
for i in range(data.shape[0]):
    data.at[i,'mean_R']=means[data.at[i,'cluster']][0]
    data.at[i,'mean_G']=means[data.at[i,'cluster']][1]
    data.at[i,'mean_B']=means[data.at[i,'cluster']][2]
for i in range(k):
    data['weight_c'+str(i)]=b[i]
```

We ran the code for 200 iterations.

The following are the images obtained by replacing each pixel with the mean of its cluster center.

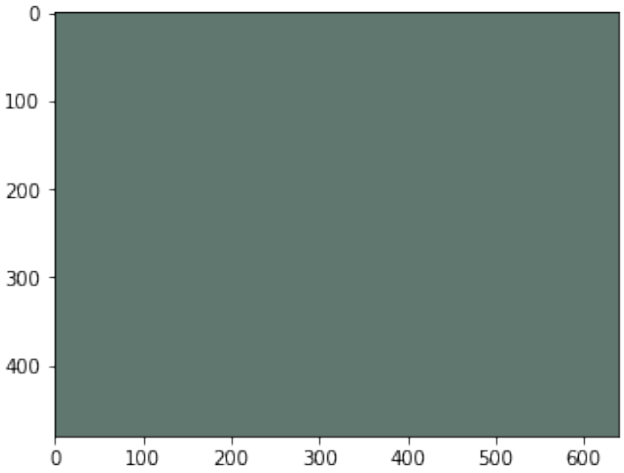

Figure 1: 10 Clusters

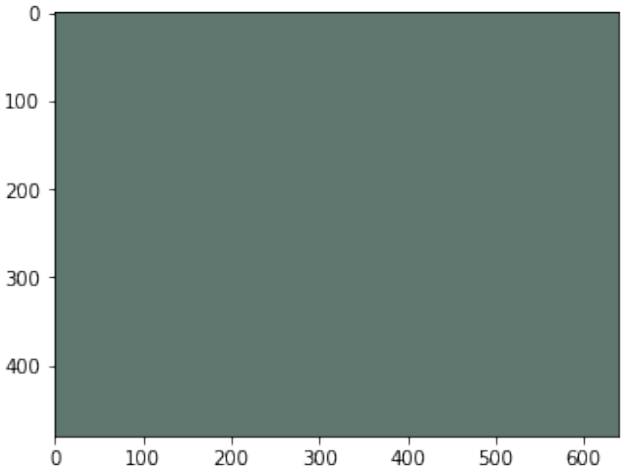

Figure 2: 20 Clusters

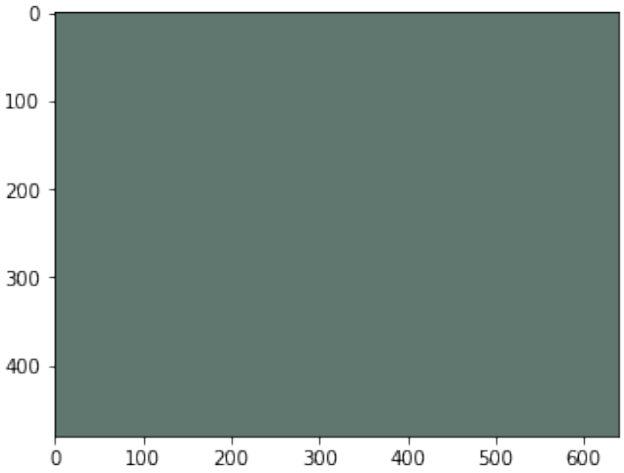

Figure 3: 50 Clusters

The code for generating the above images is as shown below.

```
def rescale_matrix(matrix, mode="int"):
    new_matrix = matrix
    for col in range(working_w):
        new_matrix.loc[:, col] *= 255

    if mode == "int":
        new_matrix.astype(int)
    return new_matrix


def print_matrix_image(original_image, image):
    im2 = Image.new(original_image.mode, original_image.size)
    img = image.values.flatten()
    im2.putdata(img)
    plt.imshow(im2)

def matricize_list(image, img_w=working_w, img_h=working_h):
    image_matrix = []
    for y in range(img_h):
        image_matrix.append(image[y*img_w:(y+1)*img_w])
    image_df = pd.DataFrame(image_matrix)
    return image_df
    return image_df


matrix_mean_R = matricize_list(list(data["mean_R"]))
matrix_mean_G = matricize_list(list(data["mean_G"]))
matrix_mean_B = matricize_list(list(data["mean_B"]))

matrix_mean_rescaled_R = rescale_matrix(matrix_mean_R)
matrix_mean_rescaled_G = rescale_matrix(matrix_mean_G)
matrix_mean_rescaled_B = rescale_matrix(matrix_mean_B)

```

```
matrix_mean_R = matrix_mean_R.astype(int)
matrix_mean_G = matrix_mean_G.astype(int)
matrix_mean_B = matrix_mean_B.astype(int)
matrix_mean_RGB = pd.DataFrame(np.rec.fromarrays((matrix_mean_R.
    values, matrix_mean_G.values, matrix_mean_B.values)).tolist(),
                      columns=matrix_mean_R.columns,
                      index=matrix_mean_R.index)

print_matrix_image(working_image, matrix_mean_RGB)
```

We observe from the figures that all the cluster centers have same mean, i.e. they drift together and hence the mean image is just a single color. Its the same for k=10,20 and 50

1.2 Part b

We constructed the figures showing the weights linking each pixel to each cluster center. Following is the code which executes this.

```
fig = plt.figure(figsize=(12, 12))
gs = gridspec.GridSpec(5, 2)
gs.update(wspace=0.25, hspace=0.25)

for i in range(10):
    ax = plt.subplot(gs[i])
    a = data["weight_c"+str(i)].values.reshape(working_h,working_w)
    plt.axis('off')
    plt.title("Cluster "+ str(i+1))
    ax.set_aspect('equal')
    plt.imshow(a)
```

The images generated are as shown below.

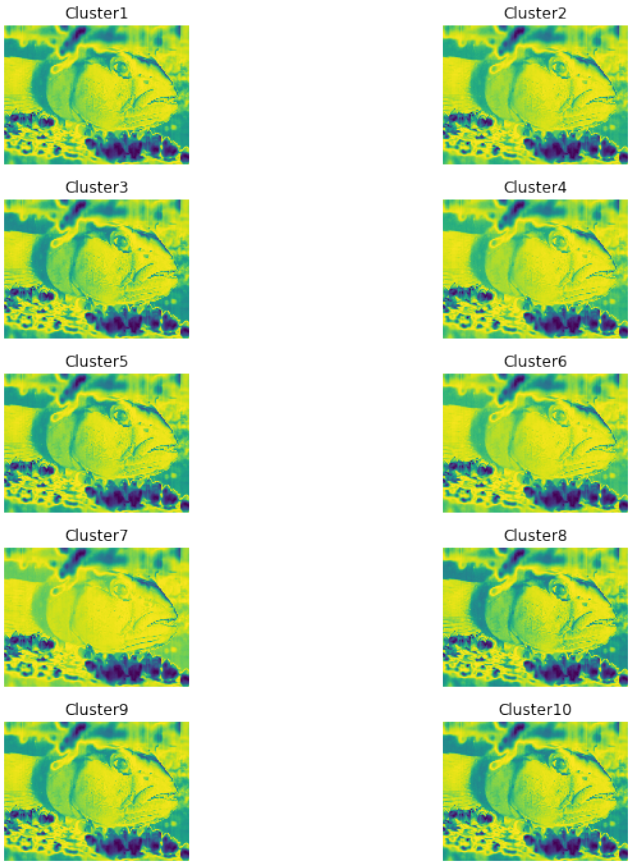

Figure 4: Cluster 1

We can observe that all the weight maps look similar, which is the expected behavior as at this scale, all the pixels are essentially one cluster and hence the weights are the same.

1.3 Part c

We changed the covariance to $0.1I$ as well as $0.0025I$ and ran the code.

```
rvalues=rgb_normalized_list[0].values.flatten()
gvalues=rgb_normalized_list[1].values.flatten()
bvalues=rgb_normalized_list[2].values.flatten()
list_of_tuples = list(zip(rvalues, gvalues, bvalues))
data=pd.DataFrame(list_of_tuples, columns = ['R', 'G', 'B'])
#Number of clusters
k=50
#Initializing the covariance matrix
cov = []
for i in range(k):
```

```
    cov.append(0.1*np.eye(data.shape[1]))
cov = np.array(cov)
means,weights,b,log_likelihoods,likelihood=EM_function(data,cov,k
    ,200)
#Plotting the log-likelihood
plt.plot(log_likelihoods)
likelihood = np.array(likelihood)
#Finding the closest cluster for each pixel
predictions = np.argmax(likelihood, axis=0)
data['cluster']=predictions
data['mean_R']=np.zeros(data.shape[0])
data['mean_G']=np.zeros(data.shape[0])
data['mean_B']=np.zeros(data.shape[0])
for i in range(data.shape[0]):
    data.at[i,'mean_R']=means[data.at[i,'cluster']][0]
    data.at[i,'mean_G']=means[data.at[i,'cluster']][1]
    data.at[i,'mean_B']=means[data.at[i,'cluster']][2]
for i in range(k):
    data['weight_c'+str(i)]=b[i]
```

We ran the code for 200 iterations.

The following are the images obtained by replacing each pixel with the mean of its cluster center.

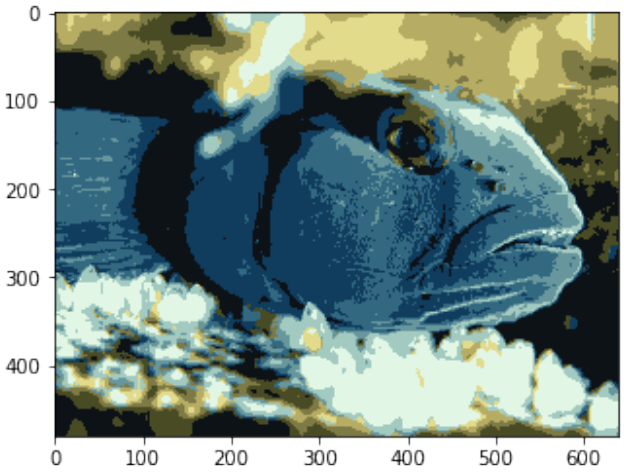

Figure 5: 10 Clusters (0.0025 I)

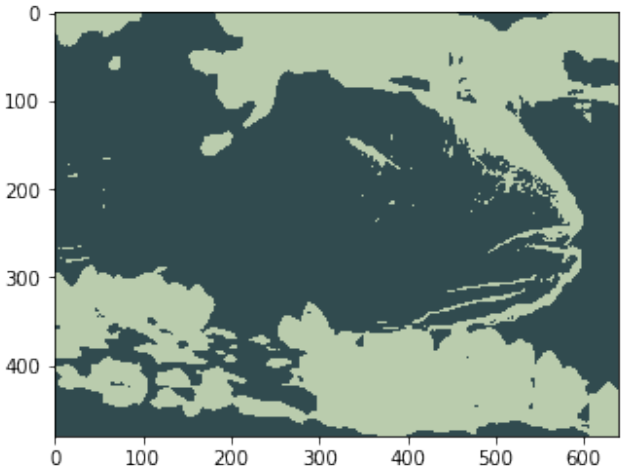

Figure 6: 10 Clusters (0.1 I)

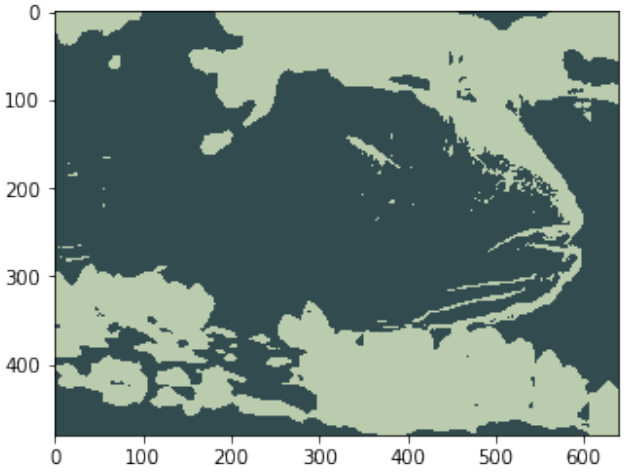

Figure 7: 20 Clusters (0.1 I)

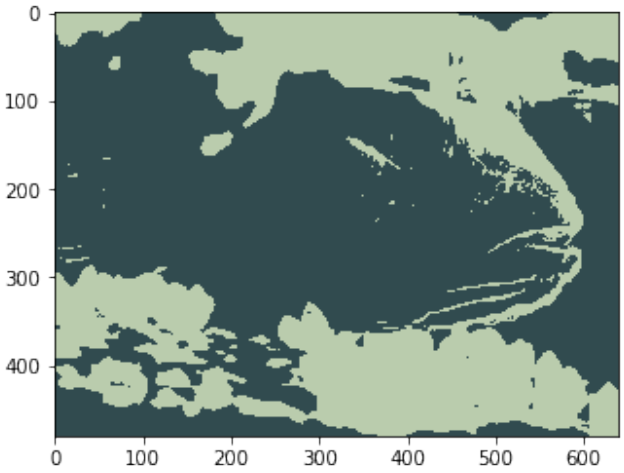

Figure 8: 50 Clusters (0.1 I)

Following are the new set of weight maps.

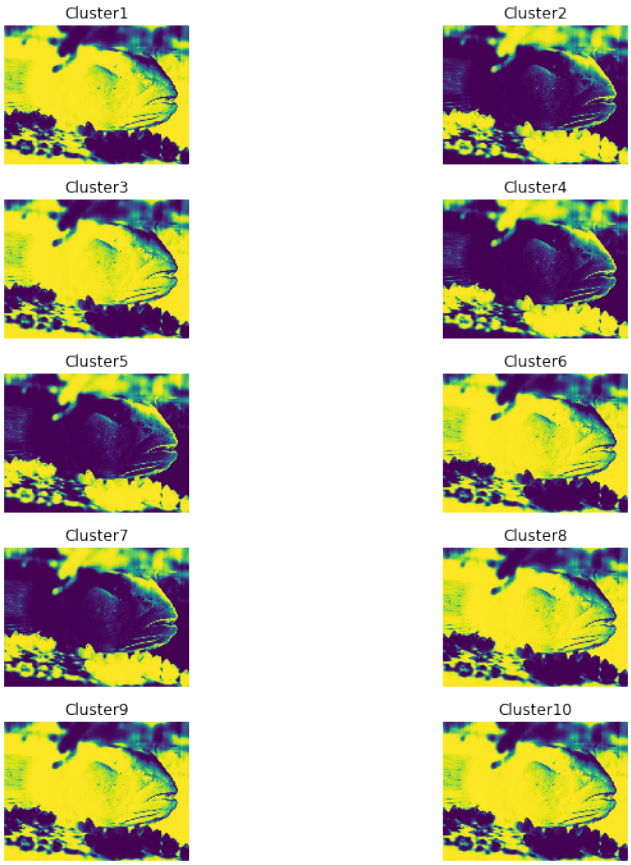

Figure 9: Covariance = 0.1 I

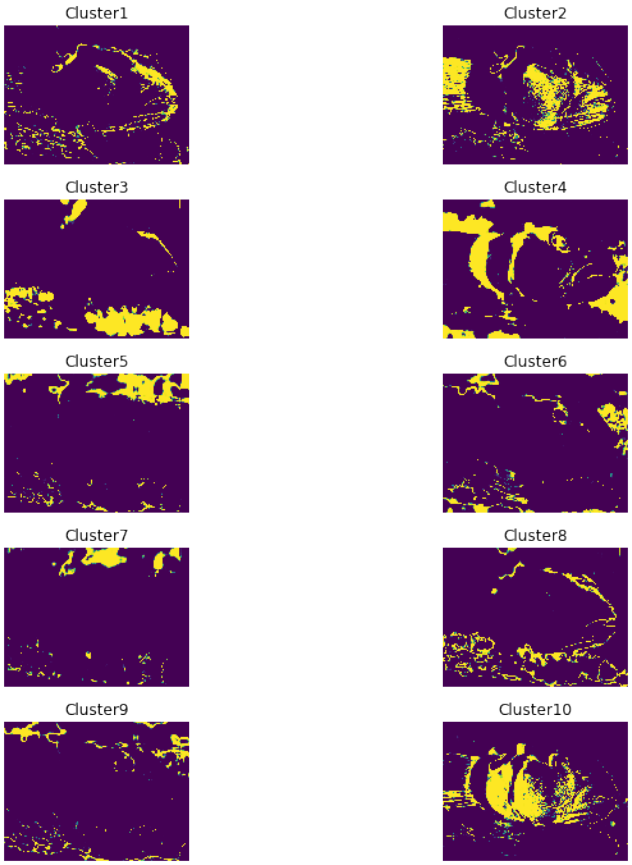

Figure 10: Covariance = 0.0025 I

We can observe that for covariance = 0.1I, the weights are not so different. But as we change the covariance to 0.0025I, we observe variation in weights image across the clusters, which means that the cluster centers do not drift together for covariance = 0.0025I. This also resulted in a better recovery of image by replacing the pixels with the mean value (Figure 5).

1.4 Part d

We estimated the covariance of pixel values by assuming that pixels are normally distributed and clustered them using EM. The code for this is as given below.

```
rvalues=rgb_normalized_list[0].values.flatten()
gvalues=rgb_normalized_list[1].values.flatten()
bvalues=rgb_normalized_list[2].values.flatten()
list_of_tuples = list(zip(rvalues, gvalues, bvalues))
data=pd.DataFrame(list_of_tuples, columns = ['R', 'G', 'B'])
#Number of clusters
```

```
k=50
#Initializing the covariance matrix
cov = []
for i in range(k):
    cov.append(np.array(data.cov()))
cov = np.array(cov)
means,weights,b,log_likelihoods,likelihood=EM_function(data,cov,k
    ,200)
#Plotting the log-likelihood
plt.plot(log_likelihoods)
likelihood = np.array(likelihood)
#Finding the closest cluster for each pixel
predictions = np.argmax(likelihood, axis=0)
data['cluster']=predictions
data['mean_R']=np.zeros(data.shape[0])
data['mean_G']=np.zeros(data.shape[0])
data['mean_B']=np.zeros(data.shape[0])
for i in range(data.shape[0]):
    data.at[i,'mean_R']=means[data.at[i,'cluster']][0]
    data.at[i,'mean_G']=means[data.at[i,'cluster']][1]
    data.at[i,'mean_B']=means[data.at[i,'cluster']][2]
for i in range(k):
    data['weight_c'+str(i)]=b[i]
```

We ran the code for 200 iterations.

The following are the images obtained by replacing each pixel with the mean of its cluster center.

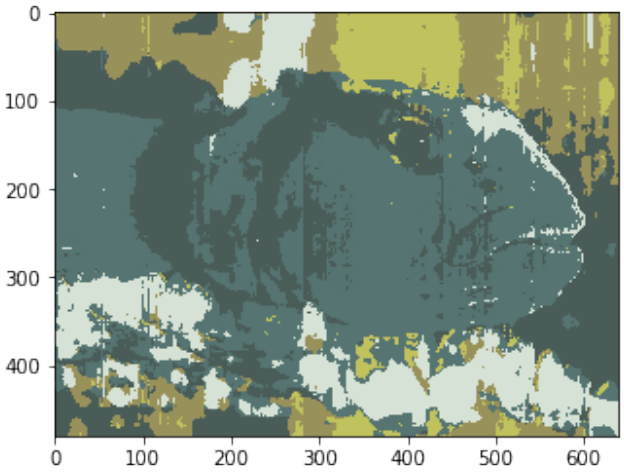

Figure 11: 10 Clusters

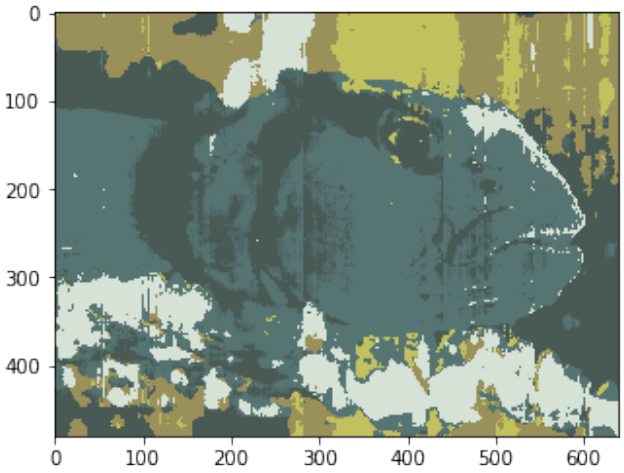

Figure 12: 20 Clusters

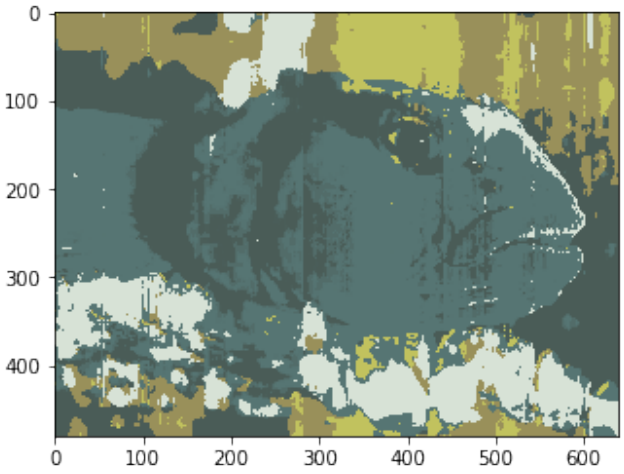

Figure 13: 50 Clusters

Following are the new set of weight maps.

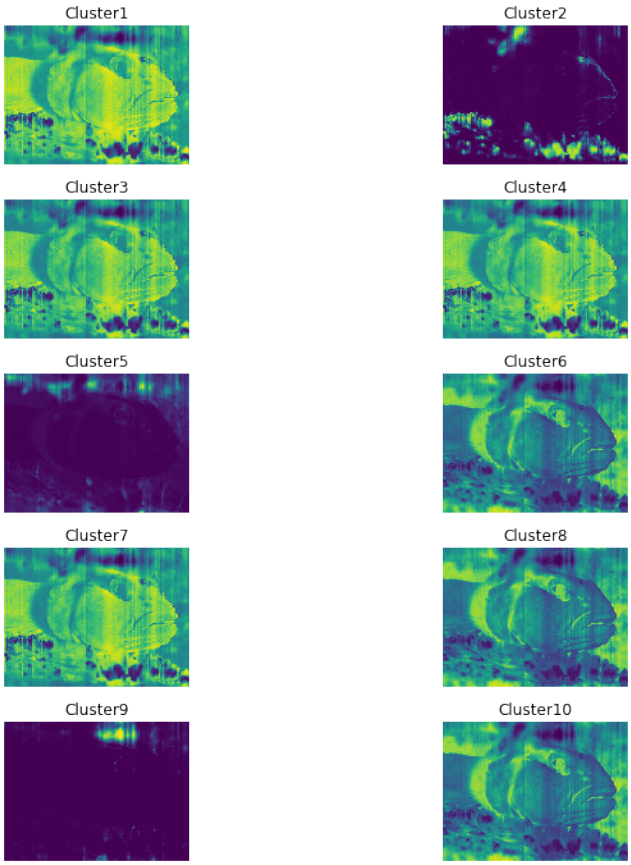

Figure 14:

We observe that he centers do not drift, and the maps are different from each other. The mean image aloso has many colors and could recover the original image to an extent. This is very different from part a and b, where the centers drift together and the mean image is a single color.