Project1

Image Segmentation Using K-means Clustering, Gaussian Mixture model and Expectation Maximization

ENCS 6161

Concordia University

Group Members:

40232982 Zunao Hu

40258305 Qian Sun

I. Abstract

K-means clustering and Gaussian Mixture Models (GMM) are two popular methods for data clustering. K-means divides data into K clusters based on distance between data points, while GMM models the data as a combination of Gaussian distributions. The GMM method uses the Expectation-Maximization (EM) algorithm for parameter estimation. The EM algorithm alternates between an estimation step, which assigns data to clusters or distributions, and a maximization step, which updates the cluster/distribution parameters to better fit the data. Convergence is checked by evaluating the log likelihood, and the process is repeated until convergence is achieved.

This study investigates the performance of the K-means algorithm and Gaussian Mixture Model (GMM) with Expectation Maximization (EM) for image segmentation. The K-means algorithm was implemented for grayscale and color images with and without intensity histograms. A custom function was developed for fitting data to Gaussian Mixture, which was used for grayscale and color image segmentation, and compared with the Gaussian Mixture class from sklearn library. The function was also used for fitting Gaussian Mixture to 2D datasets and compared with the input signals. The study provides a comprehensive analysis of the strengths and limitations of K-means and GMM-EM for image segmentation.

II. Table of contents

III. List of abbreviations in alphabetical order

EM - Expectation Maximization

E-step - Expectation step

GMM - Gaussian Mixture Model

GMM-EM - Gaussian Mixture Model with Expectation Maximization

M-step - Maximization step

IV. List of symbols

D - Dimension

d - Distance

K - Number of clusters

l - Likelihood of Gaussian Mixture Model

r_{nk} - The probability that the k^{th} mixture component generated the n^{th} data point

μ_k^{new} - Determinant of Mean of the k^{th} mixture component

Σ_k^{new} - Determinant of Covariance of the k^{th} mixture component

π_k^{new} - Mixing coefficient of the k^{th} mixture component

V. List of Figures

1 Introduction

Clustering is a common technique in data analysis, used to group similar data points together. One of the traditional clustering methods is the K-means algorithm, which involves a division approach. K-means calculates the similarity between a data object and the cluster center, and then divides the distance from the center into a cluster. The process repeats until the criterion function converges. While K-means is highly efficient, it can only handle numeric data and is sensitive to exception data, making it unsuitable for categorical data and non-convex shapes.

Another widely used tool for parameter estimation in mixture models is the Expectation-Maximization (EM) algorithm, which is applied to the Gaussian Mixture Model (GMM). EM is an iterative procedure that serves as a maximum-likelihood estimator. However, EM has some well-documented limitations, such as the need for good initial values and the possibility of being trapped in local optima. Nonetheless, EM continues to play a crucial role in parameter estimation for mixture models.

This study explored the performance of K-means and GMM-EM in image segmentation tasks for both grayscale and color images. The results obtained using K-means with and without intensity histograms were compared, and a custom function was developed for fitting data to GMMs. This function was applied to segment grayscale and color images, and the results were compared with the GaussianMixture class from sklearn. Additionally, the function was utilized for fitting GMMs to 2D datasets and compared with the input signals. Through this in-depth analysis, insights into the capabilities and limitations of these clustering methods for image segmentation were provided.

2 Scope and objectives of the project

The scope of this project is to perform image segmentation using concepts, ideas, and techniques covered in the course. The project objectives are as follows:

1) Learning the K-means algorithm: The K-means algorithm is a clustering technique used to partition a set of data points into K clusters based on similarity. The project aims to implement the K-means algorithm and its variations to gain proficiency in unsupervised learning techniques.
2) Learning Gaussian Mixture Model (GMM) and Expectation Maximization (EM): Gaussian Mixture Model (GMM) is a statistical model that uses a mixture of Gaussian distributions to represent a given set of data points. Expectation Maximization (EM) is an iterative algorithm used to estimate the parameters of a GMM. The project objective is to learn how to implement GMM and EM algorithms.
3) Implementing Image Segmentation: Image segmentation is the process of dividing an image into multiple segments or regions based on similar characteristics such as color, texture, or intensity. The project aims to apply the K-means algorithm and GMM/EM algorithms learned in objectives 1 and 2 for image segmentation.
4) Fitting 2-D Gaussian Mixture Data: The objective of this project is to learn how to fit 2-D Gaussian mixture data using the GMM and EM algorithms learned in objective 2.

By achieving these objectives, this project aims to provide a comprehensive understanding of the unsupervised learning techniques used in image segmentation and their practical applications, as taught in the course. It also aims to provide hands-on experience in implementing these techniques using Python and relevant libraries such as scikit-learn and OpenCV.

3 Detailed methodology and implementation

3.1 K-means methodology

The K-means algorithm belongs to the partition-based clustering algorithms. It defines the initial centroid value to determine the number of groups [1]. Determining the number of clusters (k) precisely is crucial for the K-means algorithm as the initial cluster center may change, leading to unstable data grouping. The output of the K-means algorithm depends on the selected center values for clustering. The algorithm determines the clusters based on the initial value of the cluster's center point. Randomly assigning the initial cluster centroid can have an impact on the performance of the cluster [2]. K-means is a distance-based clustering algorithm that partitions set data into K clusters. It works well for numerical attributes.

The K-means algorithm involves the following steps:

1) Determine the number of clusters (K) and the maximum number of iterations.

2) Initialize K cluster centroids randomly from the data points or using other initialization methods [3].

3) Assign each observation data to the nearest cluster using the Euclidean distance metric, which can be calculated using the following equation.

$$\mathrm{d} = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \tag{3.1.1}$$

where x and y are two data points, and x_1 and y_1 are the feature values of x and y for the first feature, respectively.

4) Update the centroids of each cluster based on the mean position of the data points assigned to that cluster.

5) Repeat steps 3 and 4 until the algorithm converges, which is defined as either the centroids do not change or the maximum number of iterations is reached.

6) Return the final clustering result, which is the assignment of each data point to its nearest cluster centroid.

3.2 K-means implementation

The code implementation employs the Python programming language and relevant libraries such as scikit-learn and OpenCV.

1. Grayscale image segmentation with histogram

Necessary libraries such as OpenCV, numpy and matplotlib were imported for image processing, numerical operations and visualization.

```
import cv2
import numpy as np
from matplotlib import pyplot as plt
```

Figure 3.2.1 Libraries to import for K-means

The first step is to load a grayscale image and compute its intensity histogram. The histogram is then reshaped into a 2D array of pixel intensities, and a user-defined number of clusters (K) is chosen.

```
# Load grayscale image
img = cv2.imread('street_view_10_grey.jpg', cv2.IMREAD_GRAYSCALE)

# Compute intensity histogram
hist, bins = np.histogram(img.ravel(), 256, [0, 256])

# Reshape histogram into a 2D array of pixel intensities
pixel_values = np.column_stack((bins[:-1], hist))

# Choose the number of clusters
K = 8
```

Figure 3.2.2 Loading the grayscale image and Generating histogram

Cluster centroids are initialized using the intensity histogram, and pixels are assigned to clusters based on their intensity values. The centroids are then updated by computing the mean of each cluster.

```
# Assign pixels to clusters
labels = np.zeros((img.shape[0], img.shape[1]))
for i in range(img.shape[0]):
    for j in range(img.shape[1]):
        pixel_intensity = img[i, j]
        distances = np.abs(pixel_intensity - centroids.flatten())
        cluster_index = np.argmin(distances)
        labels[i, j] = cluster_index

# Update centroids
for k in range(K):
    cluster_pixels = img[labels == k]
    if len(cluster_pixels) > 0:
        centroids[k] = np.mean(cluster_pixels)
```

Figure 3.2.3 Assigning pixels and Updating centroids

K-means clustering was performed on the grayscale image using numpy and OpenCV libraries in Python. The algorithm assigned each pixel to its nearest centroid based on the pixel intensity and updated the centroids until convergence. Finally, the image was segmented and visualized using matplotlib.

```
# Repeat steps 5-6 until convergence
max_iterations = 100
for iteration in range(max_iterations):
    old_centroids = centroids.copy()

    # Assign pixels to clusters
    for i in range(img.shape[0]):
        for j in range(img.shape[1]):
            pixel_intensity = img[i, j]
            distances = np.abs(pixel_intensity - centroids.flatten())
            cluster_index = np.argmin(distances)
            labels[i, j] = cluster_index

    # Update centroids
    for k in range(K):
        cluster_pixels = img[labels == k]
        if len(cluster_pixels) > 0:
            centroids[k] = np.mean(cluster_pixels)

    # Check for convergence
    if np.all(old_centroids == centroids):
        break

# Segment the image
segmented_values = np.zeros_like(img)
for k in range(K):
    segmented_values[labels == k] = centroids[k]

# Visualize the original and segmented images
fig, axs = plt.subplots(1, 2)
axs[0].imshow(img, cmap='gray')
axs[0].set_title('Original Image')
axs[1].imshow(segmented_values, cmap='gray')
axs[1].set_title('Segmented Image')
plt.show()
```

Figure 3.2.4 Looping till convergence and Visualization

2. Grayscale image segmentation without histogram

This implementation performs grayscale image segmentation without using histograms. Instead of using histograms, the pixel values are reshaped into a 2D array and assigned to clusters using the K-means algorithm.

```
# Load grayscale image
img = cv2.imread('street_view_10_grey.jpg', cv2.IMREAD_GRAYSCALE)

# Reshape image into a 2D array of pixels
pixel_values = img.reshape(-1, 1)

# Choose the number of clusters
K = 8

# Initialize cluster centroids
centroids = np.random.randint(0, 256, size=K)

# Assign pixels to clusters
labels = np.zeros_like(pixel_values)
for i in range(pixel_values.shape[0]):
    pixel_intensity = pixel_values[i, 0]
    distances = np.abs(pixel_intensity - centroids)
    cluster_index = np.argmin(distances)
    labels[i, 0] = cluster_index

# Update centroids
for k in range(K):
    cluster_pixels = pixel_values[labels == k]
    if len(cluster_pixels) > 0:
        centroids[k] = np.mean(cluster_pixels)
```

Figure 3.2.5 Loading the grayscale image without using histogram

The algorithm iteratively updates the centroids until convergence is achieved.

The segmented image is then reconstructed from the assigned cluster labels and their respective centroids. This approach differs from histogram-based segmentation in that it does not rely on the distribution of pixel intensities, but rather on the clustering of individual pixel values.

3. Color image segmentation without histogram

The RGB image was reshaped into a 2D array of pixels, where each pixel was represented by a vector of its RGB intensities.

```
# Load RGB image
img = cv2.imread('street_view_10_cut.jpg')

# Reshape image into a 2D array of pixels
pixel_values = img.reshape(-1, 3)
```

Figure 3.2.6 Loading the color image

The number of clusters was chosen, and cluster centroids were initialized randomly. Pixels were then assigned to clusters based on the Euclidean distance between the pixel vector and each cluster centroid.

```
# Choose the number of clusters
K = 4

# Initialize cluster centroids
centroids = np.random.randint(0, 256, size=(K, 3))

# Assign pixels to clusters
labels = np.zeros((pixel_values.shape[0],))
for i in range(pixel_values.shape[0]):
    pixel_intensity = pixel_values[i, :]
    distances = np.sum(np.abs(pixel_intensity - centroids)**2, axis=1)
    cluster_index = np.argmin(distances)
    labels[i] = cluster_index
```

Figure 3.2.7 Initialization and Assignment of pixel values

The centroids were updated as the mean of the pixels in each cluster, and the process was repeated until convergence or a maximum number of iterations was reached, which was similar to the previous implementations.

Finally, the segmented image was reconstructed by assigning the mean value of each cluster to its corresponding pixels, and the original and segmented images were displayed using Matplotlib.

Compared to the grayscale image segmentation without histogram, the main difference in this code was that the pixel values were represented as 3D vectors instead of 1D vectors, and the distance between a pixel vector and a cluster centroid was calculated as the Euclidean distance instead of the absolute difference between the pixel intensity and the centroid intensity. The update of centroids and segmentation of the image was similar to the grayscale implementation.

3.3 GMM-EM methodology

The Gaussian Mixture Model (GMM) is a probabilistic model that uses a combination of Gaussian (Normal) probability distributions to estimate the density of the data. In GMM, the parameters of the Gaussian distributions, such as mean and standard deviation, are estimated using the Expectation-Maximization (EM) algorithm[4]. The EM algorithm consists of two steps: the estimation step and the maximization step. This iterative process continues until a good set of latent values and a maximum likelihood that fit the data are achieved:

1) Initialization

To start, the number of Gaussian distributions (k) is selected to fit the data. The means, covariances, and mixture weights of the k Gaussian distributions are then randomly initialized.

2) Expectation step

In the Expectation step (E-step), the responsibility of each Gaussian distribution for generating each data point x_i is calculated. The responsibility is the probability that the data point x_i belongs to each of the k Gaussian distributions. Bayes' rule is used to compute these probabilities based on the current estimates of the means, covariances, and mixture weights [5].

$$\mathrm{r}_{nk} = \frac{\pi_k \mathrm{N}(x_n \mid \mu_k, \Sigma_k)}{\sum_{j=1}^{K} \pi_j \mathrm{N}(x_n \mid \mu_j, \Sigma_j)} \tag{3.3.1}$$

3) Maximization step

In the Maximization step (M-step), the parameters of the k Gaussian distributions are updated to maximize the likelihood of the data given the current probabilities of each data point belonging to each Gaussian. This involves calculating new estimates for the means, covariances, and mixture weights using the current probabilities.

$$\mu_k^{new} = \frac{\sum_{n=1}^{N} r_{nk} x_n}{\sum_{n=1}^{N} r_{nk}} \tag{3.3.2}$$

$$\Sigma_{\mathrm{k}}^{new} = \frac{1}{N_k} \sum_{n=1}^{N} r_{nk} (x_n - \mu_k^{new})(x_n - \mu_k^{new})^T \tag{3.3.3}$$

$$\pi_{\mathrm{k}}^{new} = \frac{N_k}{N}, \quad where\ N_k = \sum_{n=1}^{N} r_{nk} \tag{3.3.4}$$

4) Evaluation and Convergence check

The log-likelihood of the data given the current estimates of the parameters is evaluated, and it is checked if the algorithm has converged. If the convergence criterion is not met, the E and M steps are repeated until convergence is achieved [6].

$$Ln\left(l(x1, x2, \ldots x_N; \mu, \Sigma, \pi)\right) = \sum_{n=1}^{N} Ln\left(\sum_{k=1}^{K} \pi_k N\left(x_n \mid \mu_k, \Sigma_k\right)\right) \quad (3.3.5)$$

3.4 GMM-EM implementation

1. Self-defined fit function

The implemented code of the fit function using GMM and EM incorporates the "gaussian_pdf()" function, which is responsible for computing the probability density function of a multivariate Gaussian distribution. To ensure invertibility, a small positive constant has been added to the diagonal of the covariance matrix. Furthermore, the "np.nan_to_num()" function has been employed to replace any nan or inf values in the output array with 0.

In addition, a 'GMM' class was created to perform Gaussian mixture modeling. It includes methods to compute the likelihood and posterior probability of each component given a set of data points, as well as to predict the most likely component for each data point.

```
class GMM:
    def __init__(self, n_components, means, covars, weights):
        self.n_components = n_components
        self.means = means
        self.covars = covars
        self.weights = weights

    def predict_proba(self, x):
        # Compute the likelihood of each data point given each component
        likelihoods = np.zeros((x.shape[0], self.n_components))
        for i in range(self.n_components):
            likelihoods[:, i] = gaussian_pdf(x, self.means[i, :], self.covars[i, :, :])
        # Compute the posterior probability of each component given each data point
        responsibilities = likelihoods * self.weights
        responsibilities /= np.sum(responsibilities, axis=1, keepdims=True)
        return responsibilities

    def predict(self, x):
        return np.argmax(self.predict_proba(x), axis=1)
```

Figure 3.4.1 Self-defined GMM class

The function “fit_gmm()” was defined to fit a Gaussian Mixture Model (GMM) to data using the Expectation-Maximization (EM) algorithm.

```
def fit_gmm(data, n_components, init_means=None, init_covars=None, max_iter=100, tol=1e-4):
```

Figure 3.4.2 Self-defined function fit_gmm()

The steps involved in this algorithm are:

1) Initialization: The means and covariances of the Gaussian components and mixing coefficients are initialized. If no initial values are provided, the means are randomly selected from the data, and the covariances are initialized as diagonal matrices with random variances.

```
# Step 1: Initialization
if init_means is None:
    # Initialize means randomly from data points
    init_means = data[np.random.choice(data.shape[0], size=n_components, replace=False)]
if init_covars is None:
    # Initialize covariances as diagonal matrices with random variances
    init_variances = np.var(data, axis=0)
    init_covars = np.zeros((n_components, data.shape[1], data.shape[1]))
    for i in range(n_components):
        init_covars[i, :, :] = np.diag(np.random.rand(data.shape[1]) * init_variances)

# Initialize mixing coefficients as uniform probabilities
mixing_coefs = np.ones(n_components) / n_components

# Initialize log-likelihood and iteration counter
log_likelihood = -np.inf
n_iter = 0
```

Figure 3.4.3 Initializing parameters

2) E-step: The responsibilities of each component are computed for each data point based on the current parameter estimates. The likelihood of each data point given each component is computed, and then, the posterior probability of each component given each data point is calculated. Finally, the log-likelihood of the data is computed and compared to the previous iteration to check for convergence.

```
while n_iter < max_iter:

    # Step 2: E-step - Evaluate responsibilities for each data point
    # Compute the likelihood of each data point given each component
    likelihoods = np.zeros((data.shape[0], n_components))
    for i in range(n_components):
        likelihoods[:, i] = gaussian_pdf(data, init_means[i, :], init_covars[i, :, :])
    # Compute the posterior probability of each component given each data point
    responsibilities = likelihoods * mixing_coefs
    responsibilities /= np.sum(responsibilities, axis=1, keepdims=True)
    # Compute the log-likelihood of the data
    log_likelihood_new = np.sum(np.log(np.sum(likelihoods * mixing_coefs, axis=1)))
    # Check for convergence
    if np.abs(log_likelihood_new - log_likelihood) < tol:
        break
    log_likelihood = log_likelihood_new
```

Figure 3.4.4 Expectation Step

3) M-step: The parameters of the model are updated using the responsibilities computed in the E-step. The mixing coefficients, means, and covariances of each component are updated based on the total responsibility of each component.

```
# Step 3: M-step - Re-estimate parameters
# Compute the total responsibility of each component
total_resp = np.sum(responsibilities, axis=0)
# Update the mixing coefficients
mixing_coefs = total_resp / data.shape[0]
# Update the mean of each component
for i in range(n_components):
    init_means[i, :] = np.sum(responsibilities[:, i, np.newaxis] * data, axis=0) / total_resp[i]
# Update the covariance of each component
for i in range(n_components):
    diff = data - init_means[i, :]
    weighted_diff = (responsibilities[:, i, np.newaxis] * diff).T
    init_covars[i, :, :] = np.dot(weighted_diff, diff) / total_resp[i]

n_iter += 1
```

Figure 3.4.5 Maximization Step

4) Repeat steps 2 and 3 until convergence or maximum iterations are reached.

5) A Gaussian mixture model object is created with the final parameter estimates, and it is returned as the output of the function.

```
# Create the Gaussian mixture model object
gmm = GMM(n_components=n_components, means= init_means, covars= init_covars, weights=
mixing_coefs )
return gmm
```

Figure 3.4.6 A Guassian mixture model object

2. Grayscale image segmentation using GMM and EM.

This code segment implements grayscale image segmentation using Gaussian Mixture Model (GMM) and Expectation Maximization (EM) algorithm. The input grayscale image is first reshaped into a large vector and then the GMM is fitted to this vector data using the “fit_gmm()” function. The GMM is initialized with 4 clusters and initial means and covariances.

```
# Reshape the image to a large vector
gray_img_vector = gray_img.reshape((-1, 1))

# Initialize means and covariances for GMM
K = 4
init_means = np.array([[0], [50], [100], [150]])
init_covars = np.array([[[100]], [[100]], [[100]], [[100]]])

# Fit GMM to image vector data using fit_gmm function
gmm = fit_gmm(gray_img_vector, K, init_means, init_covars)
```

Figure 3.4.7 Greyscale image segmentation using self-defined fit_gmm function

Once the GMM is fitted, cluster labels are assigned to each pixel in the image, and the pixel intensities are mapped to the mean values of the clusters. Finally, the segmented image is displayed and saved with a filename that includes the number of clusters used.

```
# Assign cluster labels to pixels in the image
labels = gmm.predict(gray_img_vector)

# Map pixel intensities to mean values of clusters
means = gmm.means.astype(int)
imseg = means[labels].reshape(gray_img.shape)

# Display segmented image
imseg_disp = cv2.convertScaleAbs(imseg)
cv2.imshow('Segmented Image', imseg_disp)
cv2.waitKey(0)
cv2.destroyAllWindows()

# Save the segmented image with filename as segmented_image_k={k}.jpg
cv2.imwrite('D:\encs6161\git_project\Segmented
Images\Scenario4\street_view_10_cut_segmented_greyscale_image_k={}.jpg'.format(K),
        imseg_disp)
```

Figure 3.4.8 Display greyscale segmented image

3. Color image segmentation using GMM and EM

Color image segmentation using GMM and EM started with loading a color image using OpenCV's “imread()” function. The image was then reshaped into a large vector, with each pixel being represented by three values (RGB). The means and covariances for GMM were initialized using numpy arrays.

```
# Load color image
img = cv2.imread(r'D:\encs6161\git_project\Image\street_view_10_cut.jpg')

# Reshape the image to a large vector
img_vector = img.reshape((-1, 3))

# Initialize means and covariances for GMM
K = 10
init_means = np.array(
    [[50, 0, 0], [0, 50, 0], [0, 0, 50], [25, 25, 25], [50, 50, 50], [75, 75, 75], [100, 0, 0], [0, 100, 0],
     [0, 0, 100], [100, 100, 100]])
init_covars = np.array([[[50, 0, 0], [0, 50, 0], [0, 0, 50]]] * K)
```

Figure 3.4.9 Color image segmentation using self-defined fit_gmm function

The GMM was then fit to the image vector data using the "fit_gmm()" function. The cluster labels were assigned to pixels in the image, and the mean values of clusters were mapped to pixel intensities. The resulting segmented image was displayed using OpenCV's "imshow()" function, and saved using "imwrite()" function. The code can be easily modified to experiment with different values of K and initialization parameters.

4. 2-D dataset segmentation using GMM and EM

First, a 2-D dataset was generated using numpy's "random.multivariate_normal()" function[7]. The dataset consisted of 4 clusters, each with a different mean and covariance matrix. The means and covariances were defined as follows:

```
# Define means and covariances of the Gaussians
mu1 = [1, -2.5]
sigma1 = [[2, 0], [0, 0.4]]
mu2 = [2, 1]
sigma2 = [[0.5, 0], [0, 1.5]]
mu3 = [-2, 1]
sigma3 = [[1, -0.5], [-0.5, 1]]
mu4 = [-3, 0]
sigma4 = [[0.09, 0], [0, 0.09]]

# Generate the 2D dataset
x1 = np.random.multivariate_normal(mu1, sigma1, 200)
x2 = np.random.multivariate_normal(mu2, sigma2, 200)
x3 = np.random.multivariate_normal(mu3, sigma3, 200)
x4 = np.random.multivariate_normal(mu4, sigma4, 80)
x = np.vstack((x1, x2, x3, x4))
```

Figure 3.4.10 2-D dataset generation

The dataset was generated by stacking 200 samples from each cluster, except for the fourth cluster which only had 80 samples. The resulting dataset x was then used to fit a GMM using the “fit_gmm()” function. The GMM was specified to have 4 components using the “n_components” argument.

After fitting the GMM, the “predict()” method was used to assign cluster labels to each data point in the dataset x. These labels were then used to plot the original signal distribution and the segmented signal distribution. In the segmented signal distribution plot, each cluster was assigned a different color for visualization purposes.

```
# Plot the segmented signal distribution
colors = ['r', 'g', 'b', 'c']
plt.figure(figsize=(8, 6))
for i in range(x.shape[0]):
    plt.scatter(x[i, 0], x[i, 1], color=colors[y[i]], alpha=0.5)
plt.title('Segmented Signal Distribution')
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.show()
```

Figure 3.4.11 Segmented 2-D dataset plot

4 Experimental results

1. Grayscale image segmentation using K-means with histogram

When K was set to 8, the resulting segmented image was obtained, indicating successful segmentation of the original image into 8 clusters.

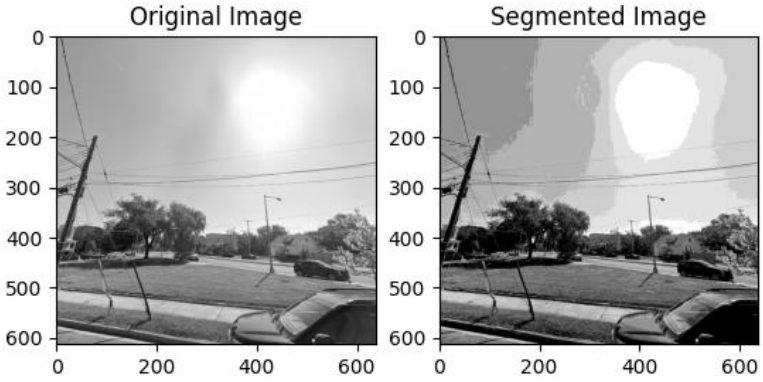

Figure 4.1 Grayscale image segmentation result using K-means with histogram

2. Grayscale image segmentation using K-means without histogram

Similar to the previous section, when K was set to 8, successful segmentation of the original image into 8 clusters was achieved, resulting in a segmented image.

However, the segmentation differed in the area surrounding the sun. Compared to segmentation based solely on the intensity value of each pixel, where the sun blended in with another cluster, segmentation with histogram preserved the shape of the sun. This difference in segmentation can be attributed to the fact that histogram-based segmentation takes into account the frequency distribution of pixel intensities, which helps distinguish between the sun and the surrounding pixels with similar intensity values. Segmentation based solely on the intensity value of each pixel may not be able to differentiate between the sun and the surrounding pixels, resulting in less accurate segmentation in that area.

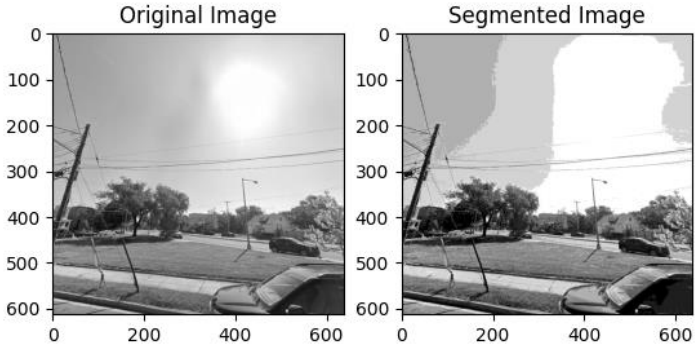

Figure 4.2 Grayscale image segmentation result using K-means without histogram

3. Color image segmentation using K-means without histogram

When setting K to 4, successful segmentation of the original image into 4 clusters was achieved.

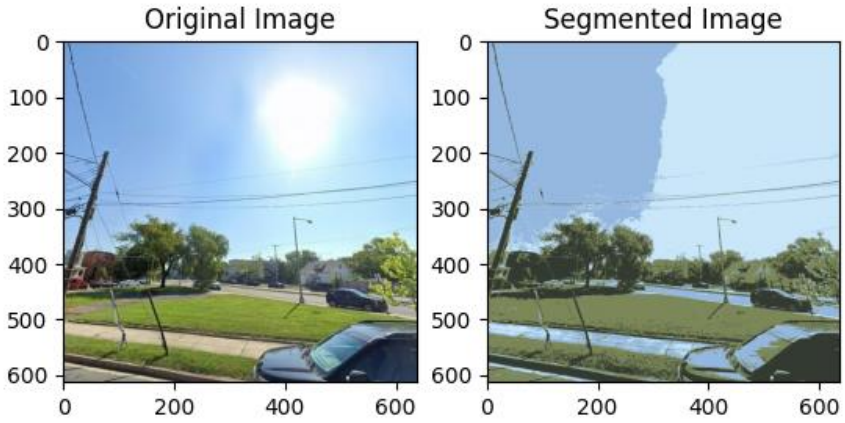

Figure 4.3 Color image segmentation result using K-means without histogram

4. Grayscale image segmentation using GMM and EM

When K was set to 4 and GMM was initialized with random means and covariances, successful segmentation of the original image into 4 classes was achieved.

Figure 4.4 Grayscale image segmentation result using self-defined GMM

To evaluate the performance of our self-defined GMM and EM implementation, a comparison to the "GaussianMixture" class from the "sklearn.mixture" library was performed. Both methods were run on the same original image and with the same settings. The comparison of the resulting segmented images suggests that our implementation shows promising performance.

The segmented results obtained using GMM and EM demonstrate a better performance

compared to K-means clustering, as evident from the clear boundaries of distinct clusters and close-to-life segmentations of different objects. This improvement in segmentation accuracy can be attributed to the more sophisticated modeling approach employed by GMM and EM. Unlike K-means, GMM and EM can model each cluster as a probability distribution, which allows for a more nuanced characterization of the data distribution. This can lead to a more precise clustering of data points based on the underlying probability distribution of the data.

Figure 4.5 Grayscale image segmentation result using GMM in sklearn

5. Color image segmentation using GMM and EM

When K was set to 10 and GMM was initialized with random means and covariances, the original image was successfully segmented into 10 classes.

Figure 4.6 Color image segmentation result using self-defined GMM

Similarly, a comparison to the "GaussianMixture" class from the "sklearn.mixture" library was performed, with both methods being run on the same original image and with the same settings. The comparison of the resulting segmented images suggests that our implementation shows promising performance

Figure 4.7 Color image segmentation result using GMM in sklearn

6. 2-D dataset segmentation using GMM and EM

The original dataset distribution was obtained and plotted as follows.

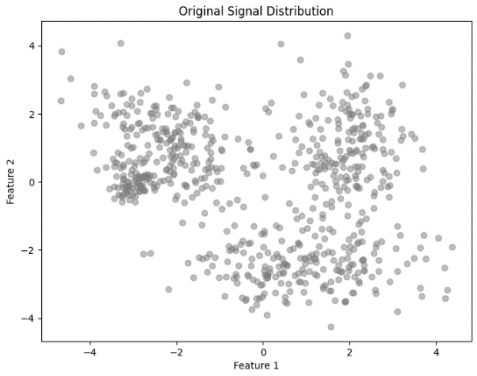

Figure 4.8 2-D dataset distribution

Knowing that K equals 4, we have obtained the segmented distribution as follows, which is the same as the four Gaussian signals that we used to generate the input dataset. The segmentation successfully regained each of the four input Gaussian signals, indicating the effectiveness of our approach in accurately identifying the underlying signal components of the input data.

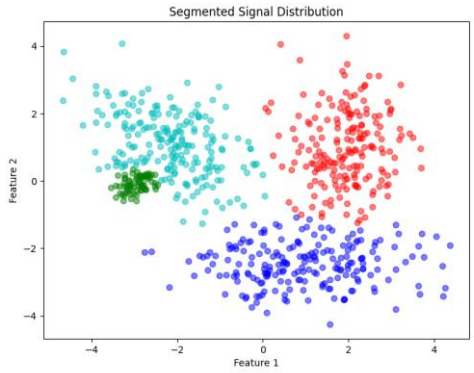

Figure 4.9 2-D dataset segmentation result using self-defined GMM

5. Conclusion

In conclusion, this report has demonstrated the implementation of K-means and GMM/EM algorithms for image segmentation on grayscale and color images, as well as 2D datasets. The obtained results suggest the effectiveness of these techniques in clustering pixels, which can contribute to the development of computer vision applications.

The report's results are promising, but there are potential limitations to consider. The experiments were conducted only on a single image, and the performance of these techniques could vary with more complex images. In addition, the algorithms require the selection of the appropriate number of clusters, which can be challenging in some cases. These limitations suggest the need for further research to explore the use of these techniques in a wider range of scenarios and to investigate possible modifications or extensions to overcome these challenges.

References

[1] Eltibi M F and Ashour W M 2011 Initializing K-Means Clustering Algorithm using Statistical Information. K-means clustering algorithm is one of the best known, XXIX 7 p51

[2] Cosmin M P, Marian C M, Mihai M An Optimized Version of the K-Means Clustering Algorithm, Proceedings of the 2014 Federated Conference on Computer Science and Information Systems (ACSIS) 2 p695

[3] H. Palus and M. Frackiewicz, "Deterministic vs. Random Initializations for K-Means Color Image Quantization," 2019 15th International Conference on Signal-Image Technology & Internet-Based Systems (SITIS), Sorrento, Italy, 2019, pp. 50-55, doi: 10.1109/SITIS.2019.00020.

[4] M. Fujimoto and Y. A. Riki, "Robust speech recognition in additive and channel noise environments using GMM and EM algorithm," 2004 IEEE International Conference on Acoustics, Speech, and Signal Processing, Montreal, QC, Canada, 2004, pp. I-941, doi: 10.1109/ICASSP.2004.1326142.

[5] A. Kundu, S. Chatterjee, A. Sreenivasa Murthy and T. V. Sreenivas, "GMM based Bayesian approach to speech enhancement in signal / transform domain," 2008 IEEE International Conference on Acoustics, Speech and Signal Processing, Las Vegas, NV, USA, 2008, pp. 4893-4896, doi: 10.1109/ICASSP.2008.4518754.

[6] Y. Tang and X. Wei, "Existence of maximum likelihood estimation for three-parameter log-normal distribution," 2009 8th International Conference on Reliability, Maintainability and Safety, Chengdu, China, 2009, pp. 305-307, doi: 10.1109/ICRMS.2009.5270184.

[7] J. Zhang and M. L. Huang, "2D approach measuring multidimensional data pattern in big data visualization," 2016 IEEE International Conference on Big Data Analysis (ICBDA), Hangzhou, China, 2016, pp. 1-6, doi: 10.1109/ICBDA.2016.7509823.