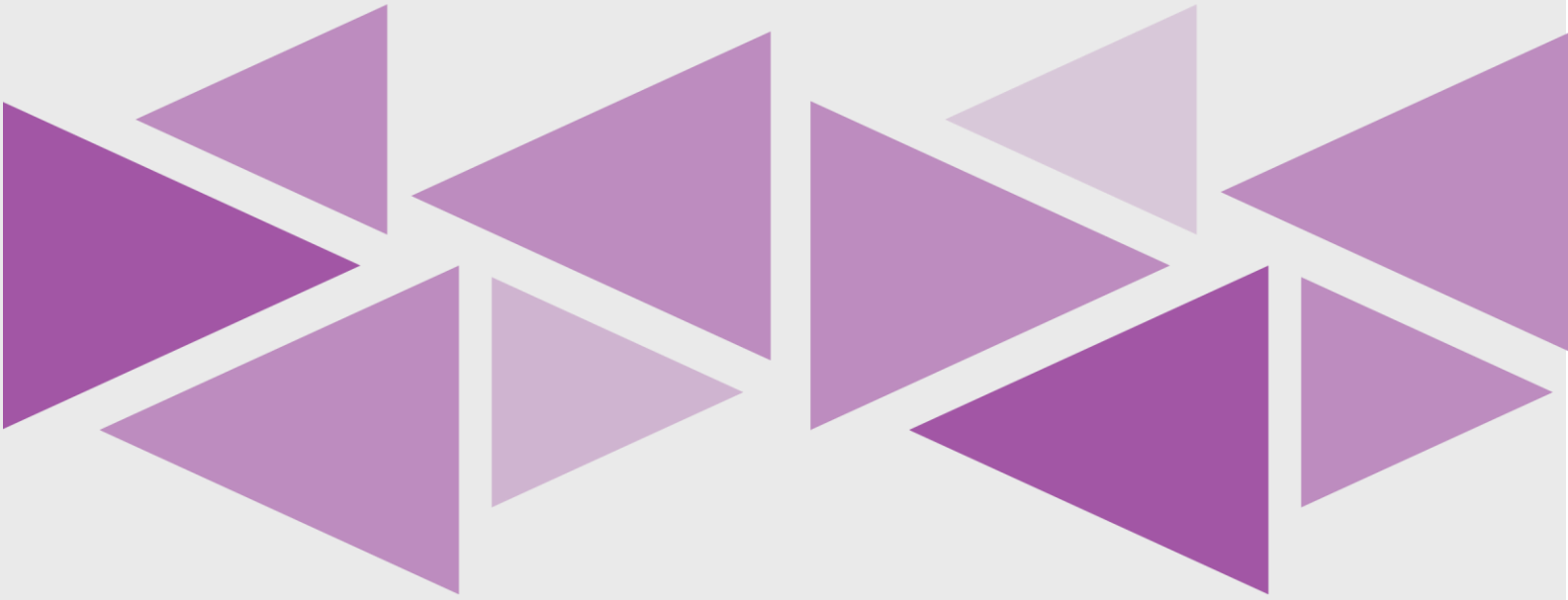
Assignment 4

# Computer Vision

DR. Ahmed M. Badawi
T.A. Laila Abbas
T.A. Peter Salah



| STUDENT NAME | SEC | BN. |
|---|---|---|
| GHOFRAN MOHAMED | 2 | 8 |
| KAREMAN YASER | 2 | 9 |
| MAYAR FAYEZ | 2 | 42 |
| NADA AHMED | 2 | 46 |
| NAIRA YOUSSEF | 2 | 48 |

## Description:

A small web application based app developed with python and streamlit, to apply different image processing techniques.

## Requirements:

- Python 3.
- Streamlit 1.13.0
- Numpy 1.23.4
- Matplotlib 3.6.2

## Running command:

Streamlit run server.py

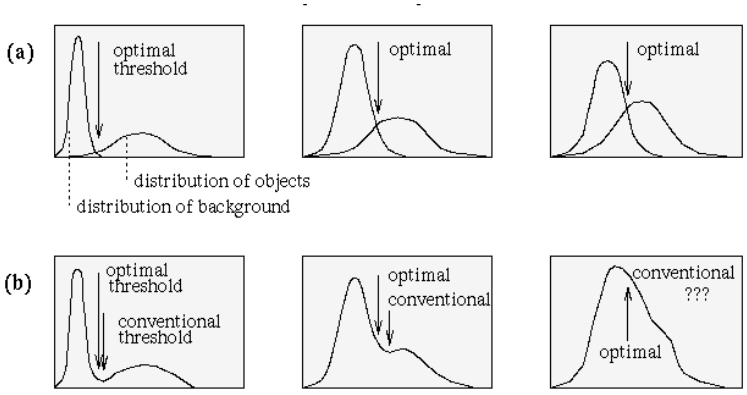
- The UI contains two main tabs Thresholding, Segmentation

## Tab1:

- Optimal Thresholding
- Otsu Thresholding
- Spectral Thresholding
- Local Thresholding

❖ **Optimal thresholding:**

Minimize number of misclassified pixels if we have some prior knowledge about distribution of the gray levels values that make up the object and the background. Is the one who divide histogram into two parts given that distribution of values at the same segment has minimum variance.




**Basic steps for Optimal thresholding:**

1. First approximation that the four corners of the image contain background pixels only and the reminder contains object pixels.
2. At every step "t" compute mean of background and object gray level.
3. Determine threshold value "T" at every step as the mid-point between the two means.
4. Update the value of threshold until T(t+1) =T(t)

**Algorithm:**

```
def optimal_thresholding(gray_image):

     # Maximum number of rows and cols for image
    max_x = gray_image.shape[1] - 1
    max_y = gray_image.shape[0] - 1
    first_corner = (gray_image[0, 0]).astype(int)
    second_corner = (gray_image[0, max_x]).astype(int)
    third_corner = (gray_image[max_y, 0]).astype(int)
    forth_corner= (gray_image[max_x, max_y]).astype(int)

    # Mean Value of Background Intensity, Calculated From The Four Corner Pixels
    background_mean = ( first_corner + second_corner + third_corner +forth_corner ) / 4
    Sum = 0
    Length = 0

    # Loop To Calculate Mean Value of object Intensity
    for i in range(0, gray_image.shape[1]):
```

```
        for j in range(0, gray_image.shape[0]):
            # Skip The Four Corner Pixels
            if not ((i == 0 and j == 0) or (i == max_x and j == 0) or (i == 0 and j ==
max_y) or (i == max_x and j == max_y)):
                Sum += gray_image[j, i]
                Length += 1
    object_mean = Sum / Length

    OldThreshold = (background_mean + object_mean) / 2
    NewThreshold = new_threshold(gray_image, OldThreshold)

    # Iterate untill the old and new threshold is equal
    while OldThreshold.any() != NewThreshold.any():
        OldThreshold = NewThreshold
        NewThreshold = new_threshold(gray_image, OldThreshold)
    thresh_img = np.zeros(gray_image.shape)
    thresh_img = np.uint8(np.where(gray_image >= NewThreshold, 255, 0))
    return thresh_img
def new_threshold(gray_image, Threshold):

    # Get Background Array, Consisting of All Pixels With Intensity Lower Than The
Given Threshold
    new_background = gray_image[np.where(gray_image < Threshold)]
    # Get object Array, Consisting of All Pixels With Intensity Higher Than The Given
Threshold
    new_object = gray_image[np.where(gray_image > Threshold)]

    new_background_mean = np.mean(new_background)
    new_object_mean = np.mean(new_object)
    # Calculate Optimal Threshold
    OptimalThreshold = (new_background_mean + new_object_mean) / 2
    return OptimalThreshold
```

**Results:**

Input image

Algorithm output

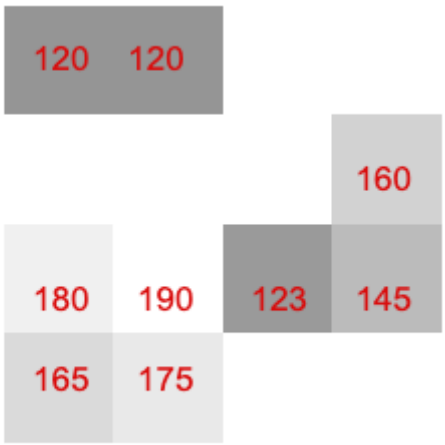
Library output

❖ **Otsu thresholding:**

The algorithm iteratively searches for the threshold that minimizes the within-class variance, defined as a weighted sum of variances of the two classes (background and foreground). The colors in grayscale are usually between 0-255 (0-1 in case of float). So, if we choose a threshold of 100, then all the pixels with values less than 100 becomes the background and all pixels with values greater than or equal to 100 becomes the foreground of the image.

The formula for finding the within-class variance at any threshold *t* is given by:

$$\sigma^2(t) = \omega_{bg}(t)\sigma^2_{bg}(t) + \omega_{fg}(t)\sigma^2_{fg}(t) \quad (1)$$

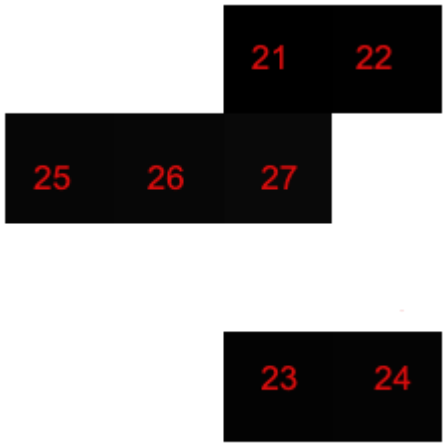




Image with pixel color values  foreground pixels  background pixels

**Algorithm:**

```
def otsu_thresholding(gray_image):
    HistValues = plt.hist(gray_image.ravel(), 256)[0]
    # print(hist)
    background, foreground = np.split(HistValues,[1])

    within_variance = []
    between_variance = []
    d = 0
    for i in range(len(HistValues)):
        background, foreground = np.split(HistValues,[i])
```

```
        c1 = np.sum(background)/(gray_image.shape[0]*
gray_image.shape[1])
        c2 =
np.sum(foreground)/(gray_image.shape[0]*gray_image.shape[1])

        background_mean = np.sum([ intensity*frequency for
intensity,frequency in enumerate(background)])/np.sum(background)
        background_mean = np.nan_to_num(background_mean)
        foreground_mean = np.sum([ (intensity + d)*(frequency) for
intensity,frequency in enumerate(foreground)])/np.sum(foreground)

        background_variance = np.sum([(intensity -
background_mean)**2*frequency for intensity,frequency in
enumerate(background)])/np.sum(background)
        background_variance = np.nan_to_num(background_variance)
        foreground_variance = np.sum([(((intensity + d -
foreground_mean)*(intensity + d - foreground_mean))*frequency) for
intensity,frequency in enumerate(foreground)])/np.sum(foreground)

        d = d +1
        within_variance.append((c1*background_variance) +
(c2*foreground_variance))
        between_variance.append(c1*c2*(background_mean-
foreground_mean)*(background_mean-foreground_mean))

    min =np.argmin(within_variance)
    max=np.argmax(background_variance)

    thresh_img = np.uint8(np.where(gray_image >min, 255, 0))
    return thresh_img
```

**result:**

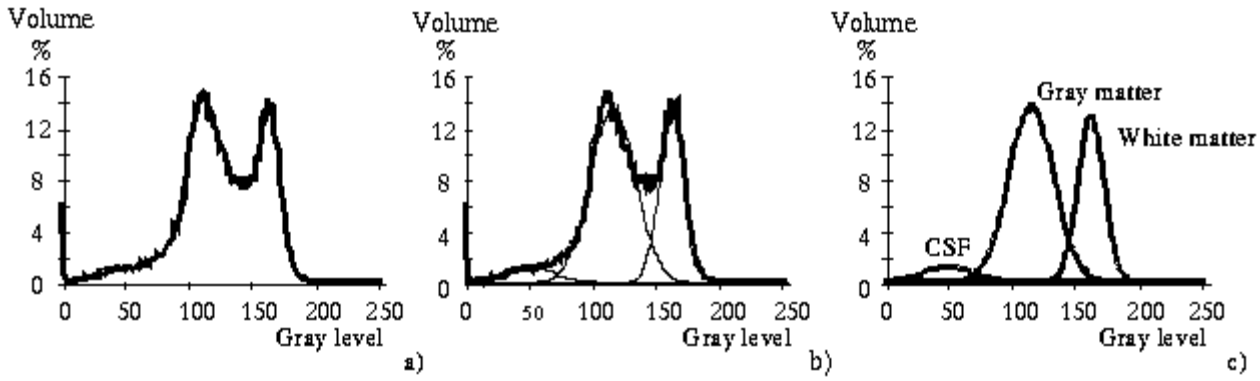
Input image

Algorithm output

Library output

❖ **Spectral thresholding:**

It is proposed for segmentation an image into multiple levels using the mean and variance starting from the extreme pixel values at both ends of histogram plot.




**Basic steps for Spectral thresholding:**

1. Smooth image using gaussian filter.
2. Calculate histogram and normalize it by dividing each bin by the total number of pixels in the image.
3. The nested loop to iterates over all possible threshold values from 0 to 255.
4. Divide the image pixels to 3 groups: background, foreground and midground.
5. For each group calculate weights and mean intensity values then use these values to calculate variance for the current threshold value.

**Algorithm:**

```
def spectral_thresholding(gray_image):
    blur = cv2.GaussianBlur(gray_image,(5,5),0)
    hist = cv2.calcHist([blur],[0],None,[256],[0,256])
    hist /= float(np.sum(hist))
    ClassVarsList = np.zeros((256, 256))
    for bar1 in range(len(hist)):

        for bar2 in range(bar1, len(hist)):
            ForegroundLevels = []
            BackgroundLevels = []
            MidgroundLevels = []
            ForegroundHist = []
            BackgroundHist = []
            MidgroundHist = []
            for level, value in enumerate(hist):
                if level < bar1:
                    BackgroundLevels.append(level)
                    BackgroundHist.append(value)
                elif level > bar1 and level < bar2:
                    MidgroundLevels.append(level)
```

```
                    MidgroundHist.append(value)
                else:
                    ForegroundLevels.append(level)
                    ForegroundHist.append(value)

            FWeights = np.sum(ForegroundHist) / float(np.sum(hist))
            BWeights = np.sum(BackgroundHist) / float(np.sum(hist))
            MWeights = np.sum(MidgroundHist) / float(np.sum(hist))
            FMean = np.sum(np.multiply(ForegroundHist, ForegroundLevels)) /
float(np.sum(ForegroundHist))
            BMean = np.sum(np.multiply(BackgroundHist, BackgroundLevels)) /
float(np.sum(BackgroundHist))
            MMean = np.sum(np.multiply(MidgroundHist, MidgroundLevels)) /
float(np.sum(MidgroundHist))
            BetClsVar = FWeights * BWeights * np.square(BMean - FMean) + \
                                                FWeights * MWeights * np.square(FMean -
MMean) + \
                                                    BWeights * MWeights *
np.square(BMean - MMean)
            ClassVarsList[bar1, bar2] = BetClsVar
        max_value = np.nanmax(ClassVarsList)
    threshold = np.where(ClassVarsList == max_value)[0][0]
    output_image = np.zeros_like(gray_image)
    output_image[gray_image > threshold] = 255
    return output_image
```

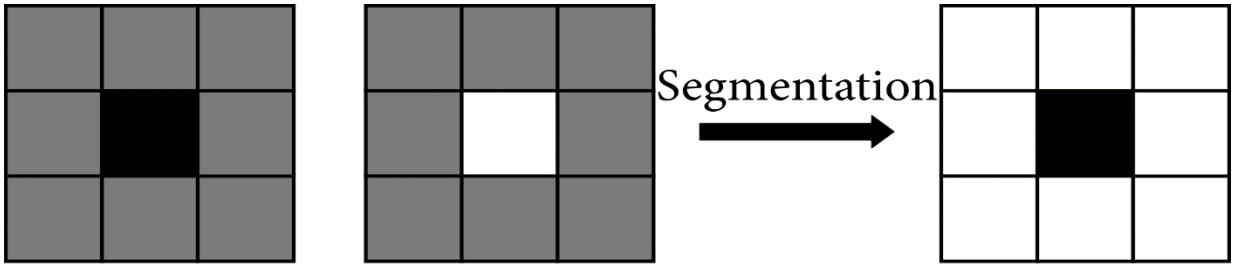
**Results:**

Input image

Algorithm output

❖ **Local thresholding:**

Local thresholding algorithms can enable local image regions such as brightness, contrast, and texture to have corresponding optimal thresholds. Common local threshold segmentation algorithms include the mean value of the local neighborhood blocks and the Gaussian weighted sum of the local neighborhood blocks.




Before the split

After the split

**Parameters that user can enter them:**

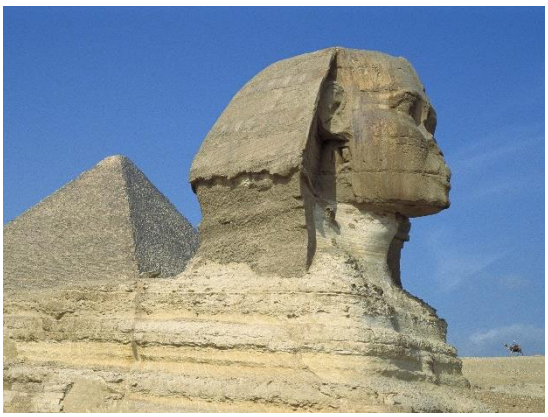
- Window size (it must be an odd number).

**Algorithm:**

```
def Local_threshold(gray_image, win_size ):
    # Create an empty binary image
    binary_img = np.zeros_like(gray_image)

# Loop through each pixel in the image
    for i in range(gray_image.shape[0]):
        for j in range(gray_image.shape[1]):
            # Calculate the local threshold using the Otsu method
            i_min = max(0, i - win_size // 2)
            i_max = min(gray_image.shape[0] - 1, i + win_size // 2)
            j_min = max(0, j - win_size // 2)
            j_max = min(gray_image.shape[1] - 1, j + win_size // 2)
            local_img = gray_image[i_min:i_max+1, j_min:j_max+1]
            threshold = np.mean(local_img) + 0.5 *
(np.std(local_img) / 128 - 1)

            # Binarize the pixel based on the local threshold
            if gray_image[i, j] > threshold:
                binary_img[i, j] = 255
    return binary_img
```

**result:**

Input image



Algorithm output

## Tab2:

- map RGB to LUV
- K-Means
- Region Growing
- Agglomerative
- Mean Shift Methods

### ❖ map RGB to LUV:

1. Convert RGB To XYZ using the following formulas.

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \leftarrow \begin{bmatrix} 0.412453 & 0.357580 & 0.180423 \\ 0.212671 & 0.715160 & 0.072169 \\ 0.019334 & 0.119193 & 0.950227 \end{bmatrix} \cdot \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

2. Convert XYZ to LUV as follows.

un = 0.19793943
vn = 0.46831096
u` = 4*x / (x + 15*y + 3*z)
v` = 9*y / (x + 15*y + 3*z)
l = 116*(y**(1/3))-16  for l> 0.008856
OR l= 903.3*y  for  l<= 0.008856
u = 13 * l * (u` - un)
v = 13 * l * (v`- vn)

3. Scaling is performed as follows.

8U data type:

L = l * FW_MAX_8U / 100
U = (u + 134) * FW_MAX_8U / 354
V = (v + 140) * FW_MAX_8U / 256

**Basic steps for map RGB to LUV:**

1. Convert RGB to XYZ
2. Convert XYZ to LUV
3. Scale L, U, and V

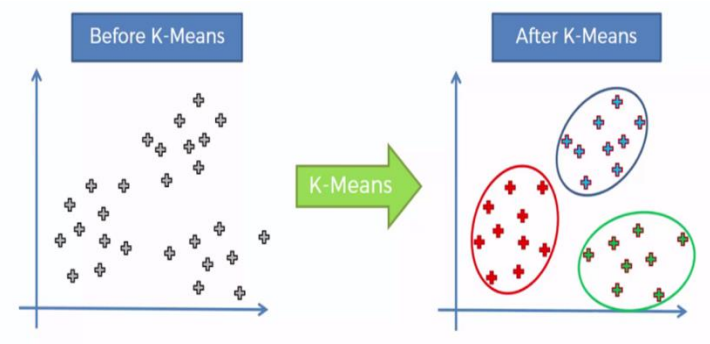
**Algorithm :**

```
def rgb_to_luv(rgb_img):
    # Convert RGB to XYZ
    rgb_normalized = rgb_img.astype(np.float32) / 255.0
    r, g, b = np.split(rgb_normalized, 3, axis=2)
    x = 0.412453 * r + 0.35758 * g + 0.180423 * b
    y = 0.212671 * r + 0.71516 * g + 0.072169 * b
    z = 0.019334 * r + 0.119193 * g + 0.950227 * b
    xyz_img = np.concatenate([x, y, z], axis=2)
    # Convert XYZ to LUV
    un = 0.19793943
    vn = 0.46831096
    x, y, z = np.split(xyz_img, 3, axis=2)
    u = 4*x / (x + 15*y + 3*z)
    v = 9*y / (x + 15*y + 3*z)
    delta = 0.008856
    k = 903.3
    y_linear = np.where(y > delta,116*(y**(1/3))-16, (k*y))
    l =  y_linear
    u_ = 13 * l * (u - un)
    v_ = 13 * l * (v - vn)
    # Scale L, U, and V
    l_scaled = np.clip(l * 255 / 100, 0, 255).astype(np.uint8)
    u_scaled = np.clip((u_ + 134) * 255 / 354, 0, 255).astype(np.uint8)
    v_scaled = np.clip((v_ + 140) * 255 / 256, 0, 255).astype(np.uint8)
    # Stack LUV channels and return
    luv_img = np.concatenate([l_scaled, u_scaled, v_scaled], axis=2)
    return luv_img
```

**Results :**

| Input Image | Algorithm Output | Libraries Output |
| --- | --- | --- |
|  |  |  |

## ❖ Segmentation Using K-Means:




K means clustering Initially assumes random cluster centers in feature space. Data are clustered to these centers according to the distance between them and centers. Now we can update the value of the center for each cluster, it is the mean of its points. Process is repeated and data are re-clustered for each iteration, new mean is calculated till convergence. Finally we have our centers and its related data points.

### Basic steps for k means segmentation:

1. Pick n data points that will act as the initial centroids.
2. Calculate the Euclidean distance of each data point from each of the centroid points selected in step 1.
3. Form data clusters by assigning every data point to whichever centroid it has the smallest distance from.
4. Take the average of each formed cluster. The mean points are our new centroids.

- Repeat steps 2 through 4 until there is no longer a change in centroids.

### Parameters that user can enter them:

- K-Means Number
- Initial Points for initial tests

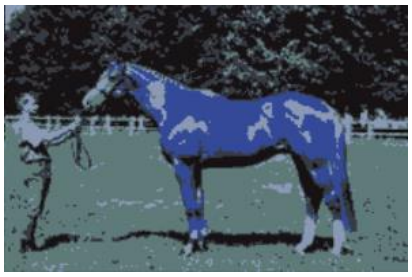
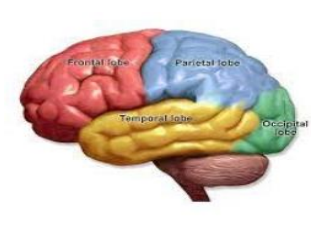
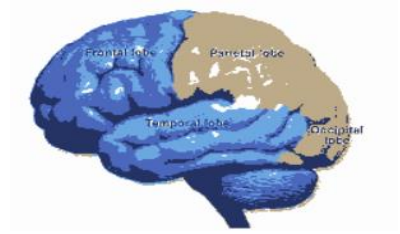
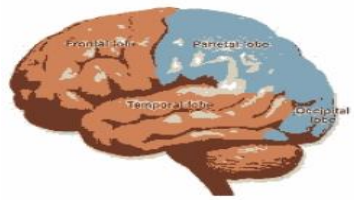
### Algorithm :

```
#.............................. Segmentation Using K-Means
................................................
def kmeans(image, k, means):
    start = time.time()
    input_image = image/255  #Normalise the image
    input_image = input_image.astype(np.float32)
    #convert the MxNx3 image to a Kx3 image where k = MxN
    vectorised = input_image.reshape((-1,3))
    #Convert the array to a dataframe
    input_image_df = pd.DataFrame(vectorised)
    input_image_df.rename(columns={0:'R', 1:'G', 2: 'B'}, inplace =True)
    #taking random centroids for initial tests
    centroids = input_image_df.sample(means)
```

```
    X = input_image_df
    diff = 1
    j=0
    while(abs(diff)>0.05):
        XD=X
        i=1
        #iterate over each centroid point
        for index1,row_c in centroids.iterrows():
            ED=[]
            #iterate over each data point
            print("Calculating distance")
            for index2,row_d in tqdm(XD.iterrows()):
                #calculate distance between current point and centroid
                d1=(row_c["R"]-row_d["R"])**2
                d2=(row_c["G"]-row_d["G"])**2
                d3=(row_c["B"]-row_d["B"])**2
                d=np.sqrt(d1+d2+d3)
                ED.append(d) #append disstance in a list 'ED'
            X[i]=ED  #append distace for a centroid in original data frame
            i=i+1

        C=[]
        print("Getting Centroid")
        for index,row in tqdm(X.iterrows()):
            min_dist=row[1]  #get distance from centroid of current data point
            pos=1
            #loop to locate the closest centroid to current point
            for i in range(k):
                if row[i+1] < min_dist:  #if current distance is greater than that of
other centroids
                    #the smaller distanc becomes the minimum distance
                    min_dist = row[i+1]
                    pos=i+1
            C.append(pos)
        #assigning the closest cluster to each data point
        X["Cluster"]=C
        #grouping each cluster by their mean value to create new centroids
        centroids_new = X.groupby(["Cluster"]).mean()[["R","G", "B"]]
        if j == 0:
            diff=1
            j=j+1
        else:
            #check if there is a difference between old and new centroids
            diff = (centroids_new['R'] - centroids['R']).sum() + (centroids_new['G'] -
centroids['G']).sum() + (centroids_new['B'] - centroids['B']).sum()
            print(diff.sum())
        centroids = X.groupby(["Cluster"]).mean()[["R","G","B"]]
```

```
    centroids = centroids.to_numpy()
    labels = X["Cluster"].to_numpy()
    #overwritting the pixels values
    segmented_image = centroids[labels-1]
    segmented_image = segmented_image.reshape(input_image.shape)

    exe_time = str(time.time() - start)  # Ouput the execution time

    return segmented_image, exe_time
```

**Results :**

| Input Image | Algorithm Output | Libraries Output |
|---|---|---|
|  |  |  |
|  |  |  |
|  |  |  |

## ❖ Segmentation Using Region Growing:

Region growing method. this method is based on choosing seeds, and then for each seeds, finding his "similar" neighbors, and creating a region of those similar pixels, the tricky part of the method is which pixels to choose as seeds, and how many of those is necessary to choose in order to get a robust segmentation in this algorithm we use fully automated seeds pick - can be chosen using the mathemtical features of a image. i.e finding local minimas of an image. After picking the seeds, all is to find spread around the seed and to find similar pixels. We defined similiar pixels by euclidean distance of the RGB pixels between a seed and his neighbors, to wrap the

algorithm up, to return the optimal segmentation, we assigned automated procedure on picking number of local minimas and threshold on the euclidean distance. by iterating, it finds the best yielded segmentation ( best measured by highest F-score).

**Basic steps for Region Growing segmentation:**

1. Find the $30^{th}$ (or less) minimal local minimas of the image. (30 is a chosen threshold by trials and errors)
2. For each seed, apply BFS(Breadth-First Search) using Queue data structure, to find nearest similar neighbors, and add them to region:
    - q = Queue()
    - q.Enqueue(seed)
    - While q is not empty
        - p = q.Dequeuq()
        - For each neighbor Ƥ of p:
          if Ƥ  and p similar(Euclidean RGB dist < Threshold) and Ƥ not already labeled then:
          q.Enqueue(Ƥ) , label(Ƥ)
3. Start with initial threshold = 3, Apply region growing iterativly, with different threshold to find optimal threshold that yields robust segmentation

**Parameters that user can enter them:**

- Number of local minimas
- Threshold on the euclidean distance

**Algorithm:**

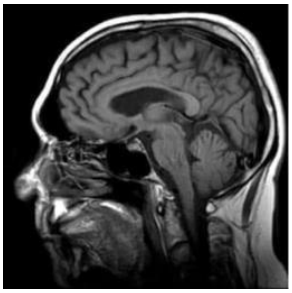
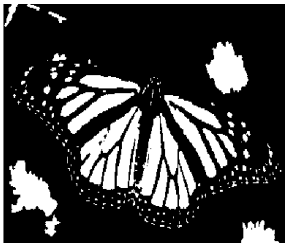
```
    class Queue:

      def __init__(self):
        self.items = []

      def isEmpty(self):
        return self.items == []

      def enqueue(self, item):
        self.items.insert(0,item)

      def dequeue(self):
        return self.items.pop()

      def size(self):
        return len(self.items)

def f_evaluataion(img, GT): # enter two thresholded gray images
```

```
    img[img > 0] = 1  # binary map of image
    GT[GT > 0] = 1  # binary map of GT
    e_gt = np.sum(img + GT == 2)
    e = np.sum(img)
    gt = np.sum(GT)
    if e == 0:
        return 0
    p = round(e_gt / e, 3)
    r = round(e_gt / gt, 3)
    f = round(2 * p * r / (p + r), 3)
#     return p, r, f
    if np.isnan(f):
        return 0
    return f

def img_Resize(path,size): #accepts path! not image
    im = cv2.imread(path, cv2.IMREAD_GRAYSCALE)
    im = cv2.resize(im, size, interpolation = cv2.INTER_AREA)
    return im

def generate_seeds(img_samp): #input: a gray image
    img_samp = cv2.GaussianBlur(img_samp ,(3,3),cv2.BORDER_DEFAULT) #apply some
smoothing
    dist = 10
    flag = False
    #get local minimas of image intensity and use them as seeds
    while flag is False:
        seeds = peak_local_max(img_samp.max() - img_samp, min_distance=dist,
indices=True)
        #iterate until the seeds filtered to equal or less to 30 seeds (found this
value by trials and errors)
        if seeds.shape[0]<= 30:
            flag = True
        dist+=10
    return seeds #return seeds which is list of coordinates


def region_growing_BFS(img_samp,img_samp_color,T,connectivity=8): #USING BFS and Queue
DataStruct to find the growing region of a seed
    seeds = generate_seeds(img_samp)
     #img_samp IS GrayScale IMAGE!!!
    #img_samp_color IS RGB IMAGE!!!
    rg_img = np.zeros(img_samp_color[:, :, 0].shape)
    ngbrs = [(0,-1),(0,1),(-1,0),(1,0),(-1, -1), (-1, 1), (1, -1), (1, 1)] # if
connectivity is 4 than reads only first 4 items, else reads all 8
    height, width = img_samp_color[:, :, 0].shape
    for i in range(seeds.shape[0]):#
        seed = seeds[i]
        q = Queue()
```

```
        q.enqueue(seed)
        ###APPLY BFS:
        count2 =0
        while q.size() > 0:
            p = q.dequeue()
            rg_img[p[0], p[1]] = 255
            for j in range(connectivity): # For each neighbor of the pixel,
connectivity = 4 || 8
                # Compute the neighbor pixel position
                x_new = p[0] + ngbrs[j][0]
                y_new = p[1] + ngbrs[j][1]
                # Boundary Condition - check if the coordinates are inside the image
                check_inside = (x_new >= 0) & (y_new >= 0) & (x_new < height) & (y_new
< width)
                # Check neighbor if inside boundries and not already "labeled" (inside
rg_img)
                if check_inside:
                    if rg_img[x_new, y_new] == 0:
                        #change the threshold to RGB EUCLIDEAN DISTANCE
                        R = int(img_samp_color[p[0], p[1], 0])
                        G = int(img_samp_color[p[0], p[1], 1])
                        B = int(img_samp_color[p[0], p[1], 2])
                        RR = int(img_samp_color[x_new, y_new, 0])
                        GG = int(img_samp_color[x_new, y_new, 1])
                        BB = int(img_samp_color[x_new, y_new, 2])
                        dist = ((RR - R) ** 2 + (GG - G) ** 2 + (BB - B) ** 2) ** 0.5
#euclidean distance between the RGB colors of a pixel and his neighbor
                        if dist < T: # if the pixels similar by defined Threshold, then
label the pixel
                            q.enqueue((x_new, y_new))
                            rg_img[x_new, y_new] = 255
    return rg_img,seeds

def region_growing_optimal(img_samp,img_samp_color,gt,T=3,connectivity=4):
    t = T
    f = 0
    segmat,seeds = region_growing_BFS(img_samp,img_samp_color,t,connectivity)
    f_next = f_evaluataion(segmat, gt)
    while f_next > f:
        f = f_next
        t+=1
        best_segmat = segmat
        best_seeds = seeds
        segmat,seeds = region_growing_BFS(img_samp,img_samp_color,t,connectivity)
        f_next = f_evaluataion(segmat, gt)
    t+=1
    f_new = f_next
    segmat,seeds = region_growing_BFS(img_samp,img_samp_color,t,connectivity)
```

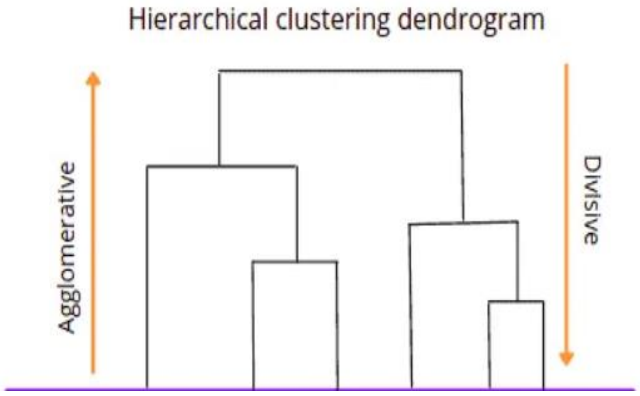
```
    f_next = f_evaluataion(segmat, gt)
    if f_next < f:
        return f,best_segmat,best_seeds

    while f_next > f_new:
        f_new = f_next
        t+=1
        best_segmat = segmat
        best_seeds = seeds
        segmat,seeds = region_growing_BFS(img_samp,img_samp_color,t,connectivity)
        f_next = f_evaluataion(segmat, gt)
        return f_new,best_segmat,best_seeds
```

**Results:**

| Input Image | Algorithm Output | Libraries Output |
|---|---|---|
|  |  |  |
|  |  |  |

❖ **Segmentation Using Agglomerative Methods:**

Hierarchical clustering uses two different approaches to create clustersone of them is Agglomerative which is a bottom-up approach in which the algorithm starts with taking all data points as single clusters and merging them until one cluster is left.



Agglomerative starts joining data points of the dataset that are the closest to each other and repeats until it merges all of the data points into a single cluster containing the entire dataset.

**Basic steps for Agglomerative segmentation:**

1. Take every data point as a separate cluster. If there are N data points, the number of clusters will be N
2. Take the two closest data points or clusters and merge them to form a bigger cluster. The total number of clusters becomes N-1.
3. Subsequent algorithm iterations will continue merging the nearest two clusters until only one cluster is left.
4. Once the algorithm combines all the data points into a single cluster, it can build the dendrogram describing the clusters' hierarchy
5. Measuring distance bewteen two clusters using Euclidean distance
6. Get the Dendrogram plot, where the x-axis shows all data points, and the y-axis shows the distance between them.
7. Once we have the dendrogram for the clusters, we can set a **threshold** (a red horizontal dashed line) to visually see the number of output classes from the dataset after algorithm execution.

**Parameters that user can enter them:**

- Number of clusters For Agglomerative

**Algorithm:**

```
 def euclidean_distance(point1, point2):

    # Computes euclidean distance of point1 and point2.

    return np.linalg.norm(np.array(point1) - np.array(point2))

def clusters_distance(cluster1, cluster2):

    # Computes distance between two clusters.

    return max([euclidean_distance(point1, point2) for point1 in cluster1 for point2 in
cluster2])

def clusters_distance_2(cluster1, cluster2):
```

```
    # Computes distance between two centroids of the two clusters

    cluster1_center = np.average(cluster1, axis=0)
    cluster2_center = np.average(cluster2, axis=0)
    return euclidean_distance(cluster1_center, cluster2_center)


class AgglomerativeClustering:

    def __init__(self, k=2, initial_k=25):
        self.k = k
        self.initial_k = initial_k

    def initial_clusters(self, points):

        # partition pixels into self.initial_k groups based on color similarity

        groups = {}
        d = int(256 / (self.initial_k))
        for i in range(self.initial_k):
            j = i * d
            groups[(j, j, j)] = []
        for i, p in enumerate(points):
            if i%100000 == 0:
                print('processing pixel:', i)
            go = min(groups.keys(), key=lambda c: euclidean_distance(p, c))
            groups[go].append(p)
        return [g for g in groups.values() if len(g) > 0]

    def fit(self, points):

        # initially, assign each point to a distinct cluster
        print('Computing initial clusters ...')
        self.clusters_list = self.initial_clusters(points)
        print('number of initial clusters:', len(self.clusters_list))
        print('merging clusters ...')

        while len(self.clusters_list) > self.k:

            # Find the closest (most similar) pair of clusters
            cluster1, cluster2 = min([(c1, c2) for i, c1 in
enumerate(self.clusters_list) for c2 in self.clusters_list[:i]],
                 key=lambda c: clusters_distance_2(c[0], c[1]))

            # Remove the two clusters from the clusters list
            self.clusters_list = [c for c in self.clusters_list if c != cluster1 and c
!= cluster2]

            # Merge the two clusters
```

```
            merged_cluster = cluster1 + cluster2

          # Add the merged cluster to the clusters list
            self.clusters_list.append(merged_cluster)
            print('number of clusters:', len(self.clusters_list))
        print('assigning cluster num to each point ...')
        self.cluster = {}
        for cl_num, cl in enumerate(self.clusters_list):
            for point in cl:
                self.cluster[tuple(point)] = cl_num
        print('Computing cluster centers ...')
        self.centers = {}
        for cl_num, cl in enumerate(self.clusters_list):
            self.centers[cl_num] = np.average(cl, axis=0)



    def predict_cluster(self, point):
        """
        Find cluster number of point
        """
        # assuming point belongs to clusters that were computed by fit functions
        return self.cluster[tuple(point)]

    def predict_center(self, point):
        """
        Find center of the cluster that point belongs to
        """
        point_cluster_num = self.predict_cluster(point)
        center = self.centers[point_cluster_num]
        return center
```

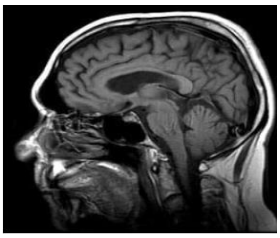
**Results:**

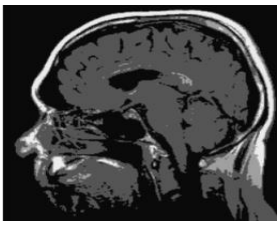
Input Image

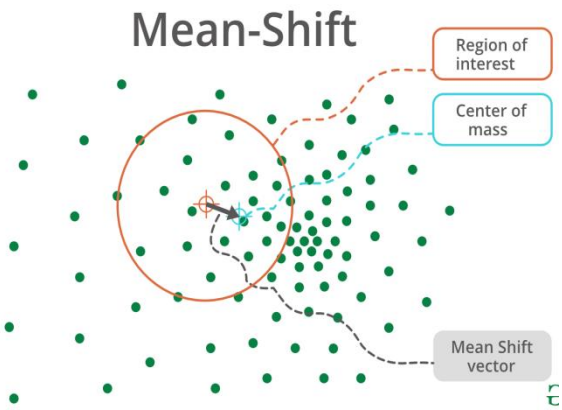
Algorithm Output

## ❖ Segmentation Using Mean Shift Methods:

- Mean Shift is also known as the mode-seeking








algorithm that assigns the data points to the clusters in a way by shifting the data points towards the high-density region. The highest density of data points is termed as the model in the region.

$$\mathbf{m}(\mathbf{x}) = \frac{\sum_{i=1}^{n} K(\mathbf{x} - \mathbf{x}_i)\,\mathbf{x}_i}{\sum_{i=1}^{n} K(\mathbf{x} - \mathbf{x}_i)}$$

## Basic steps for mean-shift segmentation:

1. Getting the feature space of the input-image.
2. Selecting a random row from the feature space and assigning it as the current mean.
3. Finding the eucledian distance of the randomly selected row i.e. current mean with all the other rows.
4. Checking if the distance calculated is within the window. If yes taking those rows and adding them to a list below_threshold_arr.
5. Calculate Euclidean distance between mean color/position and current mean.
6. When found, color all the rows in below_threshold_arr with the color of the row in below_threshold_arr that has i,j nearest to mean_i and mean_j.

7. Remove below-threshold pixels from feature space.

**Parameters that user can enter them:**

- Window Size
- Threshold

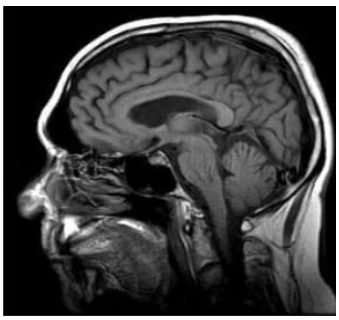
**Algorithm:**

```
def mean_shift(img,window=70,threshold=1.0):
    t1=time.time()

    def euclidean_distance(p1, p2):
        return np.sqrt(np.sum((p1 - p2) ** 2))

    row, col, _ = img.shape
    segmented_image = np.zeros((row,col,3), dtype= np.uint8)
    feature_space   = np.zeros((row * col,5))
    counter=0
    current_mean_random = True
    current_mean_arr = np.zeros((1,5))

    for i in range(0,row):
        for j in range(0,col):
            feature_space[counter]=[img[i][j][0],img[i][j][1],img[i][j][2],i,j]
            counter+=1

    while(len(feature_space) > 0):
        print (len(feature_space))
        #selecting a random row from the feature space and assigning it as the current
mean
        if current_mean_random:
            current_mean_index = random.randint(0, feature_space.shape[0] - 1)
            current_mean_arr[0] = feature_space[current_mean_index]
        below_threshold_arr=[]

        distances = np.zeros(feature_space.shape[0])
        for i in range(0,len(feature_space)):
            distance = 0
            #Finding the eucledian distance of the randomly selected row i.e. current
mean with all the other rows
            for j in range(0,5):
                distance += ((current_mean_arr[0][j] - feature_space[i][j])**2)

            distances[i] = distance**0.5

            #Checking if the distance calculated is within the window. If yes taking
those rows and adding
```

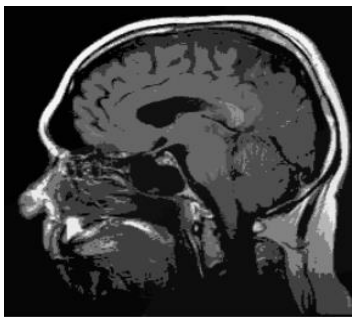
```
            #them to a list below_threshold_arr
        below_threshold_arr = np.where(distances < window)[0]

        mean_color = np.mean(feature_space[below_threshold_arr, :3], axis=0)
        mean_pos = np.mean(feature_space[below_threshold_arr, 3:], axis=0)
        # Calculate Euclidean distance between mean color/position and current mean
        mean_color_distance = euclidean_distance(mean_color, current_mean_arr[0][:3])
        mean_pos_distance = euclidean_distance(mean_pos, current_mean_arr[0][3:])
        mean_e_distance = mean_color_distance + mean_pos_distance

        if(mean_e_distance < threshold):
            new_arr = np.zeros((1,3))
            new_arr[0] = mean_color
            # When found, color all the rows in below_threshold_arr with
            #the color of the row in below_threshold_arr that has i,j nearest to mean_i
and mean_j
            current_mean_random = True
            segmented_image[feature_space[below_threshold_arr, 3].astype(int),
feature_space[below_threshold_arr, 4].astype(int)] = new_arr
            # Remove below-threshold pixels from feature space
            feature_space[below_threshold_arr, :] = -1
            feature_space = feature_space[feature_space[:, 0] != -1]

        else:
            current_mean_random = False
            current_mean_arr[0, :3] = mean_color
            current_mean_arr[0, 3:] = mean_pos

    segmented_image = cv2.cvtColor(segmented_image, cv2.COLOR_BGR2RGB)
    exe_time = str(time.time() - t1)  # Ouput the execution time

    return segmented_image, exe_time
```

**Results:**

Input Image



Algorithm Output



Libraries Output