Computer vision

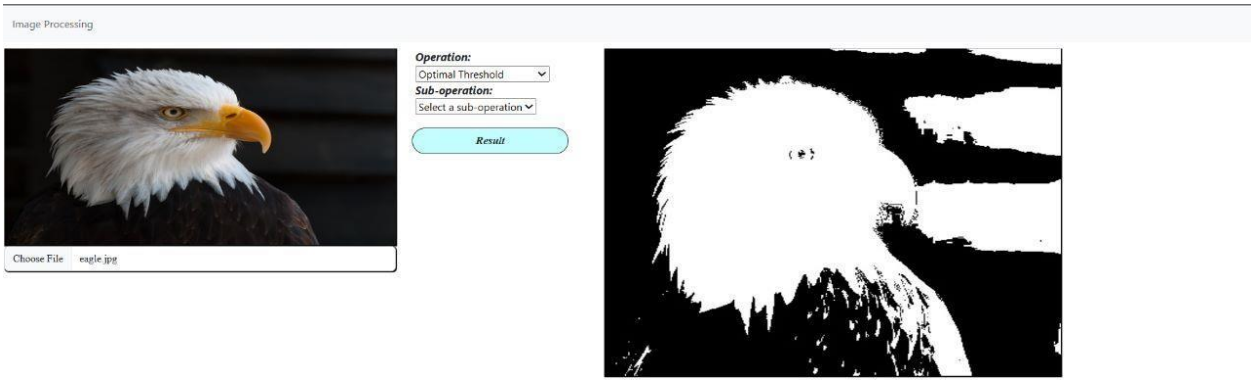
Assessment 4

Team members

Name	Section	B.N
osamah Faisal Abdulatef	1	11
Beshara Safawt	1	22
Sara Tarek Galal Ahmed	1	40
Shuaib Abdulsalam	1	48
Abdelrahman Sameh	1	53
Mariam Mounier AbdElRehim	2	35

- Optimal threshold

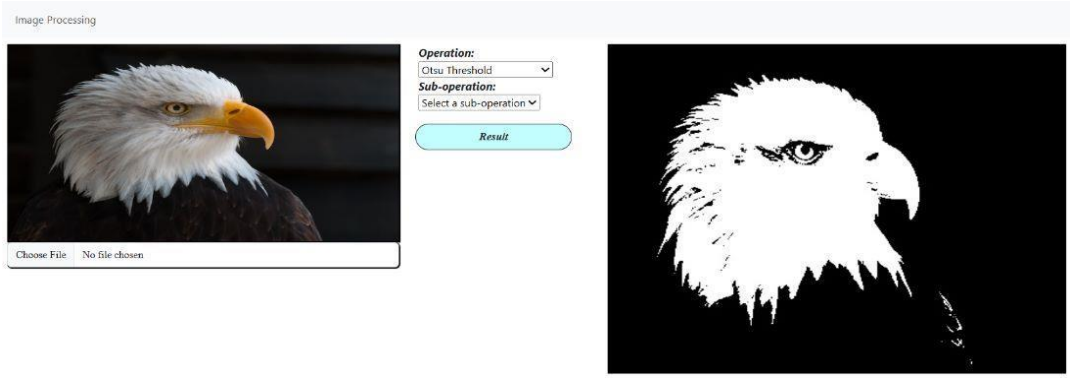
First we take the four corner pixels as background and calculate the mean of them, also calculate the mean of foreground pixels and calculate mean of these two means and this is the new threshold. Then calculate the mean of pixels under this threshold as background, calculates the mean of pixels bigger than this threshold as foreground and also calculate the mean of these two means and this is the new threshold and so on until the threshold value doesn't change.




```python
def optimal_threshold(path):
    img = im.readImg(path,(400,400),'rgb')
    # Convert image to grayscale
    gray_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    threshold = gray_img.mean()
    prev_threshold = -1
    while abs(threshold - prev_threshold) >= 1:
        # Calculate the average of the four corner pixels
        bg_avg = (gray_img[0,0] +
                  gray_img[-1,0] +
                  gray_img[0,-1] +
                  gray_img[-1,-1]) / 4.0
        # Calculate the average of other pixels
        fg_avg = np.mean(gray_img) - bg_avg
        # Calculate the new threshold
        prev_threshold = threshold
        threshold = (bg_avg + fg_avg) / 2.0
        # Modify the image array in place using boolean indexing
        gray_img[gray_img <= threshold] = 0
        gray_img[gray_img > threshold] = 255
        os.remove("static/images/output/optim_thres.jpg")
        pathOFResult= f"static/images/output/optim_thres.jpg"
        cv2.imwrite(pathOFResult,gray_img)

    return pathOFResult
```

- Otsu threshold

  We compute the histogram of the grayscale image. The histogram represents the distribution of pixel intensities in the image. We then calculate the probability of each pixel intensity by dividing the frequency of it by the total number of pixels in the image. Then we calculate the cumulative sum of probabilities. We calculate the mean gray level intensity of image. We iterate over all possible threshold values from 0 to the maximum pixel intensity and for each threshold value, we calculate the between-class variance, which measures the separation between the foreground and background regions, and we select the threshold value that maximizes the between-class variance.




```python
def otsu_threshold(path):
    img = im.readImg(path,(400,400),'rgb')
    gray_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    hist = np.zeros(256)
    for i in range(gray_img.shape[0]):
        for j in range(gray_img.shape[1]):
            pixel_value = gray_img[i,j]
            hist[pixel_value] += 1
    # Normalize the histogram
    hist /= (gray_img.shape[0] * gray_img.shape[1])
    # Calculate the cumulative sum and cumulative mean of the normalized histogram
    cum_sum = np.cumsum(hist)
    cum_mean = np.cumsum(np.arange(256) * hist)
    # Initialize variables for calculating the between-class variance and threshold
    max_var = 0
    threshold = 0
    # Iterate over all possible threshold values and calculate the between-class variance
    for t in range(256):
        w0 = cum_sum[t]
        w1 = 1 - w0
        if w0 == 0 or w1 == 0:
            continue
        mu0 = cum_mean[t] / w0
        mu1 = (cum_mean[-1] - cum_mean[t]) / w1
        var_between = w0 * w1 * (mu0 - mu1)**2
        if var_between > max_var:
            max_var = var_between
            threshold = t
```

- Spectral Thresholding

1- Initialize the whole image as single region

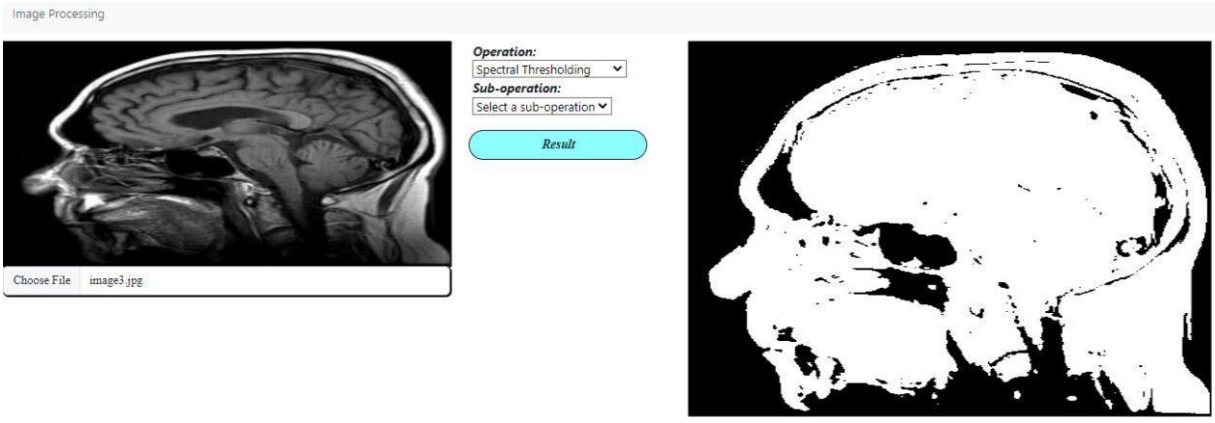
2- Compute a smoothed histogram for each spectral band. Find the most significant peak in each histogram & determine two thresholds as local minima on either sides of this maximum. Segment each region in each spectral band into sub regions according to these thresholds. Each segmentation in each spectral band is projected into multi spectral segmentation. Regions for the next processing steps are those in multi-spectral image.

3- Repeat step 2 for each region of the image until each region's histogram contains only one significant peak minima on either side of this maximum. Segment each region in each spectral band into sub regions according to these thresholds. Each segmentation in each spectral band is projected into multi-spectral segmentation. Regions for next processing steps are those in multi spectral image.

```python
def spectral_thresholding(path):
    gray_image = im.readImg(path,(400,400),'gray')
    blur = cv2.GaussianBlur(gray_image,(5,5),0)
    hist = cv2.calcHist([blur],[0],None,[256],[0,256])
    hist /= float(np.sum(hist))  # type: ignore
    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))
```

```
        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] # type: ignore
output_image = np.zeros_like(gray_image)
output_image[gray_image > threshold] = 255

os.remove("static/images/output/spectral_thresholding.jpg")
pathOFResult= f"static/images/output/spectral_thresholding.jpg"
cv2.imwrite(pathOFResult,output_image)

return pathOFResult
```

- Output



- RGB to Luv

Converting an RGB image to Luv color space involves several steps, which are as follows:

1. Convert the RGB color values to linear RGB values: The sRGB color space is the most commonly used color space for digital images, but it is not suitable for color manipulations because it is nonlinear. Therefore, the first step is to convert the sRGB color values to linear RGB values using a gamma correction formula.

2. Convert the linear RGB values to XYZ values: The next step is to convert the linear RGB values to XYZ values using a matrix multiplication.

3. Normalize the XYZ values: The XYZ values obtained in the previous step need to be normalized to account for differences in luminance between different color spaces

4. Calculate the L*, u*, and v* values: The L*, u*, and v* values are calculated from the normalized XYZ values using the following formulas:

$$L^* = 116 * f(Y/Yn) - 16$$

$$u^* = 13 * L^* * (f(X/Xn) - f(Y/Yn))$$

$$v^* = 13 * L^* * (f(Y/Yn) - f(Z/Zn))$$

where Yn, Xn, and Zn are the reference white values, and f(t) is a nonlinear function that maps the t values to the L* values.

5. Convert the L*, u*, and v* values to Luv values: Finally, the Luv values can be obtained by converting the L*, u*, and v* values using the following formulas:

$$L = L^*$$

$$u = 13 * L * (u' - un)$$

$$v = 13 * L * (v' - vn)$$

- RGB to XYZ

```python
def RGB_To_XYZ(image):
    # Convert the image to a numpy array
    XYZ_Image = np.array(image)
    # Normalize the RGB values to the range [0, 1]
    XYZ_Image = image / 255.0

    # Define the RGB to XYZ transformation matrix
    transMatrix = np.array([[0.412453, 0.357580, 0.180423],
                            [0.212671, 0.715160, 0.072169],
                            [0.019334, 0.119193, 0.950227]])

    XYZ_Image = np.dot(XYZ_Image, transMatrix)
    return XYZ_Image
```

- XYZ To LUV

```python
def xyz_to_luv(xyz, white_point=[0.95047, 1.0, 1.08883]):

    x, y, z = xyz / np.sum(xyz)
    u_green_red = 4 * x / (x + 15 * y + 3 * z)
    v_blue_yellow = 9 * y / (x + 15 * y + 3 * z)

    uw_ = 4 * white_point[0] / (white_point[0] + 15 * white_point[1] + 3 * white_point[2])
    vw_ = 9 * white_point[1] / (white_point[0] + 15 * white_point[1] + 3 * white_point[2])
    yw = y / white_point[1]

    if yw > 0.008856:
        Lightness = 116 * (yw ** (1/3)) - 16
    else:
        Lightness=903.3 * yw

    u=13 * Lightness * (u_green_red - uw_)
    v=13 * Lightness * (v_blue_yellow - vw_)
    return Lightness, u, v
```

- Output

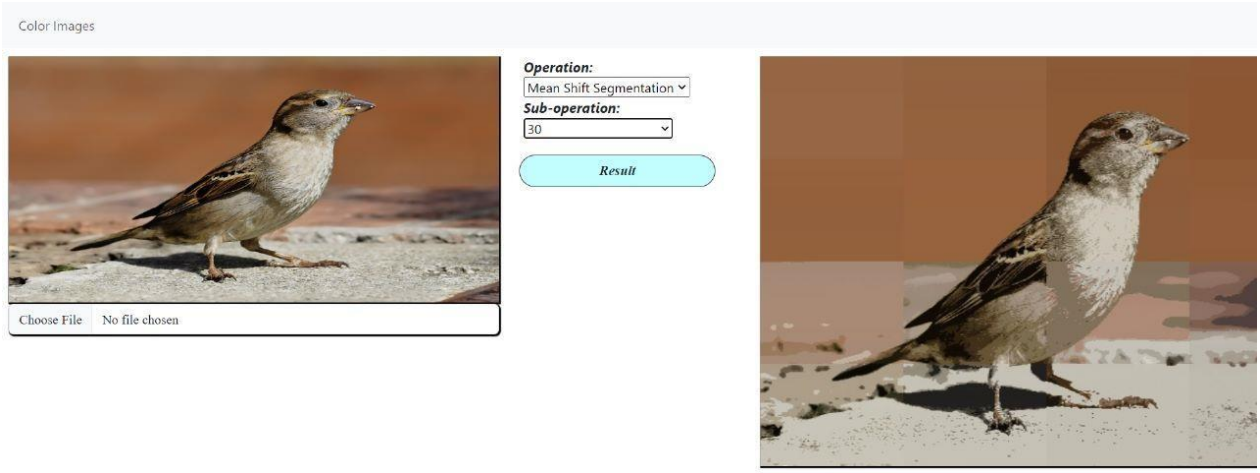
- mean shift segmentation

The algorithm works by iteratively shifting the center of a window (or kernel) towards the mode of the pixel values within that window. The mode represents the highest density of pixels within the window, and it serves as the new center for the next iteration. This process continues until convergence, where the final mode represents a cluster of similar pixels.

```python
# Define a function to perform mean shift on a patch of the image
def mean_shift_patch(patch,bandwidth):
    # Convert the patch to float64
    pixels = np.float64(patch.reshape(-1, 3))
    # Loop over all the pixels in the patch
    for i in range(len(pixels)):
        # Initialize the mean shift vector
        shift = np.array([1, 1, 1])
        # Loop until convergence
        while np.linalg.norm(shift) > 1:
            # Compute the mean of the pixels within the bandwidth
            kernel = pixels - pixels[i]
            kernel_norm = np.linalg.norm(kernel, axis=1)
            within_bandwidth = kernel_norm < bandwidth
            mean = np.mean(pixels[within_bandwidth], axis=0)
            # Compute the mean shift vector
            shift = mean - pixels[i]
            # Shift the pixel
            pixels[i] += shift
    # Convert the pixels back to uint8 and reshape to patch shape
    segmented_patch = np.uint8(pixels.reshape(patch.shape))
    return segmented_patch
```

```python
def mean_shift_segmentation (path,bandwidth):

    # Load the image
    img = im.readImg(path,(400,400),'rgb')

    # Define the patch size
    patch_size = (100,100)

    # Split the image into patches
    patches = []
    for i in range(0, img.shape[0], patch_size[0]):
        for j in range(0, img.shape[1], patch_size[1]):
            patch = img[i:i+patch_size[0], j:j+patch_size[1]]
            patches.append(patch)

    # Perform mean shift segmentation on each patch in parallel
    num_cores = 8  # Change this to the number of CPU cores you want to use
    segmented_patches = Parallel(n_jobs=num_cores)(delayed(mean_shift_patch)(patch,bandwidth) for patch in patches)

    # Combine the segmented patches into a single image
    segmented_img = np.zeros_like(img)
    k = 0
    for i in range(0, img.shape[0], patch_size[0]):
        for j in range(0, img.shape[1], patch_size[1]):
            segmented_img[i:i+patch_size[0], j:j+patch_size[1]] = segmented_patches[k]
            k += 1

    os.remove("static/images/output/mean_segmented.jpg")
    pathOFResult= f"static/images/output/mean_segmented.jpg"
    cv2.imwrite(pathOFResult,segmented_img)
```

- Output



- K-means segmentation

1. Choose the number of clusters: The first step is to choose the number of clusters (K) that the algorithm will create. This is a crucial step as it affects the quality of the final segmentation. The value of K can be chosen based on prior knowledge or by trial and error.

2. Initialize the cluster centers: The algorithm randomly selects K data points from the image as the initial cluster centers.

3. Assign each pixel to a cluster: Each pixel in the image is assigned to the cluster whose center is closest to it. The distance between a pixel and a cluster center can be measured using a variety of distance metrics, such as the Euclidean distance or the Manhattan distance.

4. Recalculate the cluster centers: Once all pixels have been assigned to a cluster, the algorithm calculates new cluster centers by taking the mean of the pixel values in each cluster.

5. Repeat steps 3 and 4 until convergence: Steps 3 and 4 are repeated until convergence, which is achieved when the cluster centers no longer change

significantly between iterations. Convergence can be detected by monitoring the change in the cluster centers or the sum of squared distances between the pixels and their assigned cluster centers.

6. Label each pixel: Once convergence is reached, each pixel is labeled with the index of the cluster to which it belongs. This creates a segmented image, where each segment corresponds to a cluster.

7. Post-processing: Post-processing steps can be applied to refine the segmentation further. For example, small clusters or regions with a low pixel count can be merged with adjacent clusters or regions. Additionally, post-processing techniques can be used to smooth the boundaries between segments.

```python
def kmeans_segmentation(path, k, max_iterations=100):
    # Reshape image to a 2D array of pixels
    image = im.readImg(path,(400,400),'gray')
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    pixels = np.float32(image.reshape(-1, 3)) # type: ignore
    # Initialize centroids randomly
    centroids = pixels[np.random.choice(pixels.shape[0], k, replace=False)] # type: ignore

    # Run K-means algorithm for max_iterations
    for i in range(max_iterations):
        # Calculate distances between each pixel and each centroid
        distances = np.sqrt(np.sum((pixels - centroids[:, np.newaxis])**2, axis=2))

        # Assign each pixel to the closest centroid
        labels = np.argmin(distances, axis=0)

        # Update centroids to be the mean of all pixels assigned to them
        for j in range(k):
            centroids[j] = np.mean(pixels[labels == j], axis=0)

    # Assign each pixel to its final centroid
    final_labels = np.argmin(np.sqrt(np.sum((pixels - centroids[:, np.newaxis])**2, axis=2)), axis=0)

    # Reshape labels back to the shape of the original image
    kmean_segmented_image = final_labels.reshape(image.shape[:2])

    os.remove("static/images/output/kmean_segmented_image.jpg")
    pathOFResult= f"static/images/output/kmean_segmented_image.jpg"
    cv2.imwrite(pathOFResult,kmean_segmented_image)
```

Results could have been better but it takes computational time

- Region Growing Segmentation:

    1. Selection of the initial seeds
    2. Seed growing criteria
    3. Termination of the segmentation process(condition)

We choosed 3 random points and getting 8 neighbour points to the current seed then substract pixels values from the seed point then verify wit hthe threshold.

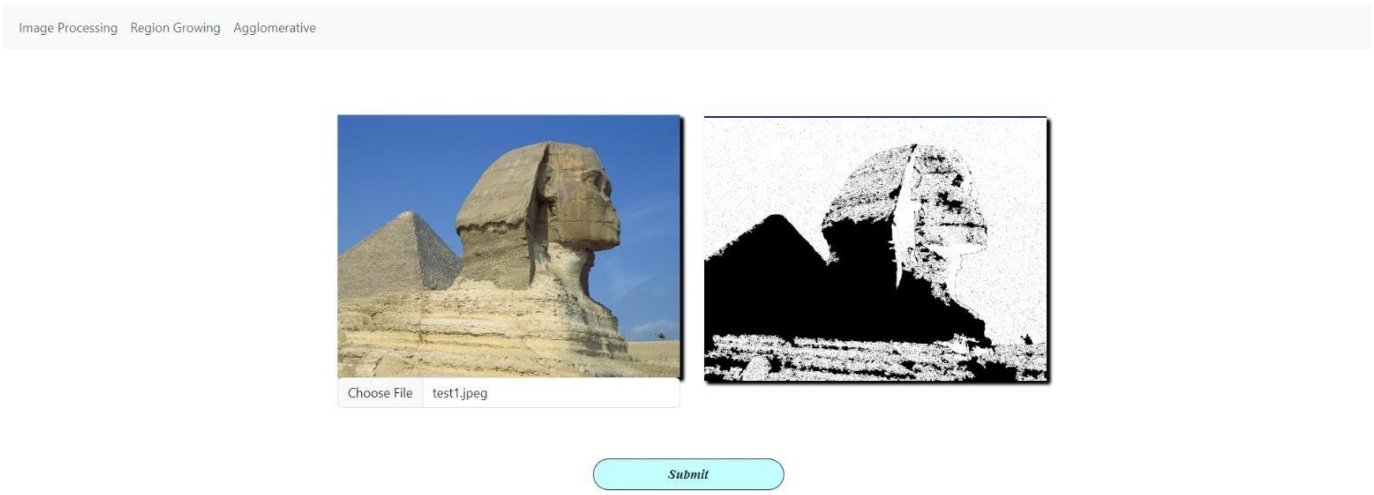
The seed pixel and the pixels in the surrounding neighborhood that have the same or similar properties as the seed pixel are merged into the region where the seed pixel is located.

These new pixels are treated as new seeds to continue the above process until pixels that do not meet the conditions can be included

```python
class Point(object):

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def getX(self):
        return self.x

    def getY(self):
        return self.y


def getGrayDiff(img, currentPoint, tmpPoint):
    return abs(int(img[currentPoint.x, currentPoint.y]) - int(img[tmpPoint.x, tmpPoint.y]))


def selectConnects(p):
    if p != 0:
        connects = [Point(-1, -1), Point(0, -1), Point(1, -1),
                    Point(1, 0), Point(1, 1), Point(0, 1),
                    Point(-1, 1), Point(-1, 0)]
    else:
        connects = [Point(0, -1), Point(1, 0), Point(0, 1), Point(-1, 0)]

    return connects
```

```python
def Region_growing(img, seeds, thresh, p=1):

    height, weight = img.shape
    seedMark = np.zeros(img.shape)
    seedList = []

    for seed in seeds:
        seedList.append(seed)
    label = 1
    connects = selectConnects(p)

    while (len(seedList) > 0):
        currentPoint = seedList.pop(0)

        seedMark[currentPoint.x, currentPoint.y] = label

        for i in range(8):
            tmpX = currentPoint.x + connects[i].x
            tmpY = currentPoint.y + connects[i].y

            if tmpX < 0 or tmpY < 0 or tmpX >= height or tmpY >= weight:
                continue

            grayDiff = getGrayDiff(img, currentPoint, Point(tmpX, tmpY))

            if grayDiff < thresh and seedMark[tmpX, tmpY] == 0:
                seedMark[tmpX, tmpY] = label
                seedList.append(Point(tmpX, tmpY))

    return seedMark
```

(variable) connects: list[Point]



- Agglomerative Segmentation:

Feature space is given by the luv pixel values. Steps:

1. Each data point is assigned as a single cluster.
2. Determine the distance measurement and calculate the distance matrix
3. find the closest (most similar) pair of clusters and merge them
4. Repeat until specified number of clusters is obtained .

```
def euclidean_distance(point1, point2):
    return np.linalg.norm(np.array(point1) - np.array(point2))


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


def clusters_distance_2(cluster1, cluster2):

    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):
        groups = {}
        d = int(256 / (self.initial_k))
        for i in range(self.initial_k):
```

```
def fit(self, points):

    self.clusters_list = self.initial_clusters(points)
    while len(self.clusters_list) > self.k:
        cluster1, cluster2 = min([(c1, c2) for i, c1 in enumerate(self.clusters_list) for c2 in self.clusters_list[:
                                  key=lambda c: clusters_distance_2(c[0], c[1]))

        self.clusters_list = [c for c in self.clusters_list if np.all(np.array(
            c) != np.array(cluster1)) and np.all(np.array(c) != np.array(cluster2))]

        merged_cluster = cluster1 + cluster2
        self.clusters_list.append(merged_cluster)

    self.cluster = {}
    for cl_num, cl in enumerate(self.clusters_list):
        for point in cl:
            self.cluster[tuple(point)] = cl_num

    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):

    return self.cluster[tuple(point)]
```

```
        merged_cluster = cluster1 + cluster2
        self.clusters_list.append(merged_cluster)

    self.cluster = {}
    for cl_num, cl in enumerate(self.clusters_list):
        for point in cl:
            self.cluster[tuple(point)] = cl_num

    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):

    return self.cluster[tuple(point)]

def predict_center(self, point):

    point_cluster_num = self.predict_cluster(point)
    center = self.centers[point_cluster_num]
    return center
```

Image Processing Region Growing Agglomerative

Choose File dog.jpg

Submit