CV - A2

Q1

Input Image:

Image after applying fuzzy c-means on image
Number of clusters for clustering = 10

Image after merging small regions with their surroundings:
Combined <=100 pixels in a region with its surrounding colours

As can be seen clearly in the above image, **small regions of glasses and plants** have been merged together.

Steps:

- Normalise the RBGxy by dividing by 255,255,255,max_x, max_y respectively.
- Perform fuzzy c-means.
- Merge small regions with surrounding regions.

```
import cv2
import sys

from queue import Queue
import copy
import numpy as np
import itertools
from fcmeans import FCM
import pandas as pd
from matplotlib import pyplot as plt

def perform_fcm_clustering(dataset, number_of_clusters):
    fcm = FCM(n_clusters=number_of_clusters)
    fcm.fit(dataset)
    return fcm

def get_labels(fcm, dataset, height, width):
    labels = fcm.predict(dataset)
    return labels.reshape(height, width).tolist()


def is_valid_move(x, y, rows, cols):
    '''
    check validity of cell whether it is inside matrix or not
    '''
    return x < rows and y < cols and x >= 0 and y >= 0

def color_the_component(row, col, color, matrix, labels, rows, cols):

    # store all coordinates which are at perimeter of current object
    surrounding_labels = {}
    q = Queue()
```

```python
    number_of_pixels = 1
    my_label = labels[row][col]


    q.put((row, col))
     # color current pixel
    matrix[row][col] = color

    # Note: 8 nearby pixels are possible
    # but we are taking only in 4 directions
    # as single diagonal of object will not be possible to see with naked eyes
    moves = [(0, 1), (1, 0), (-1, 0), (0, -1)]

    while q.qsize() != 0:

       x1, y1 = q.get()
       # print(x1,y1)

       # check nearby pixels
       for move in moves:
             x2, y2 = x1 + move[0], y1 + move[1]

             if is_valid_move(x2, y2, rows, cols):
                   # print(x2,y2,"points")
                   # print(labels[x2][y2], my_label, "labels")
                   # print("color", matrix[x2][y2])
                   if (labels[x2][y2] == my_label) :
                         if matrix[x2][y2] == 0:
                               # print("color it")
                               # nearby pixel is valid and is of same label as given

                               q.put((x2, y2)) # keep nearby pixels in the queue for further recursive coloring
                               matrix[x2][y2] = color # color nearby pixel
                               number_of_pixels += 1
                   else:
                         other_label = int(labels[x2][y2])
                         if other_label not in surrounding_labels:
                               surrounding_labels[other_label] = 0

                         surrounding_labels[other_label] += 1

    # find most surrounding labels
    m = max(surrounding_labels.values())
    most_frequent_label_in_surrounding = [i for i in surrounding_labels if surrounding_labels[i] == m][0]
    return (number_of_pixels, most_frequent_label_in_surrounding)

def merge_small(matrix, info_of_colors, labels, rows, cols):

    for row in range(rows):
       for col in range(cols):
             if matrix[row][col] in info_of_colors:
                   labels[row][col] = info_of_colors[matrix[row][col]]



def find_connected_components_and_merge_small(rows, cols, labels, Threshold):
    '''
    find all connected components formed by objects in a binary image
    '''
    matrix = np.zeros((rows, cols), dtype=np.int32)
    matrix = matrix.tolist() # numpy matrix to normal list of lists for fast access
    info_of_colors = {}
    color = 0

    for row in range(rows):
       for col in range(cols):
             if matrix[row][col] == 0:
                   color -= 1
                   number_of_pixels, most_frequent_label_in_surrounding = color_the_component(row, col, color, matrix, labels, rows, 
cols)
```

```python
                    if number_of_pixels < Threshold:
                          info_of_colors[color] = most_frequent_label_in_surrounding

    merge_small(matrix, info_of_colors, labels, rows, cols)


def main(image_path, number_of_clusters, Threshold):

    originalImage = cv2.imread(image_path)
    height, width = originalImage.shape[:2]

    # Normalise the dataset for given image
    dataset = np.array([np.append(np.array([originalImage[i][j][k] / 255 for k in range(3)]), [i / height, j / width], 0) for i in
range(height) for j in range(width)])
    # Perform fuzzy c means
    fcm = perform_fcm_clustering(dataset, number_of_clusters)
    labels = get_labels(fcm, dataset, height, width)
    centers = (fcm.centers).tolist()

    # image after fuzzy c-means
    for row in range(height):
       for col in range(width):
             label = labels[row][col]
             originalImage[row, col] = [int(centers[label][0] * 255), int(255 * centers[label][1]), int(255* centers[label][2])]

    cv2.imshow('Image before merging', originalImage)
    cv2.waitKey()
    cv2.destroyAllWindows()

    # find and merge small components
    find_connected_components_and_merge_small(height, width, labels, Threshold)

    for row in range(height):
       for col in range(width):
             label = labels[row][col]
             originalImage[row, col] = [int(centers[label][0] * 255), int(255 * centers[label][1]), int(255* centers[label][2])]

    cv2.imshow('Image after merging', originalImage)
    cv2.waitKey()
    cv2.destroyAllWindows()

if __name__ == '__main__':
    if len(sys.argv) != 4:
       print("Usage: python3 code.py <path_of_image> <number_of_clusters> <Threshold>")
       exit(1)

    image_path = sys.argv[1]
    number_of_clusters = int(sys.argv[2])
    Threshold = int(sys.argv[3])
    main(image_path, number_of_clusters, Threshold)
```

Q2 and Q3 combined:

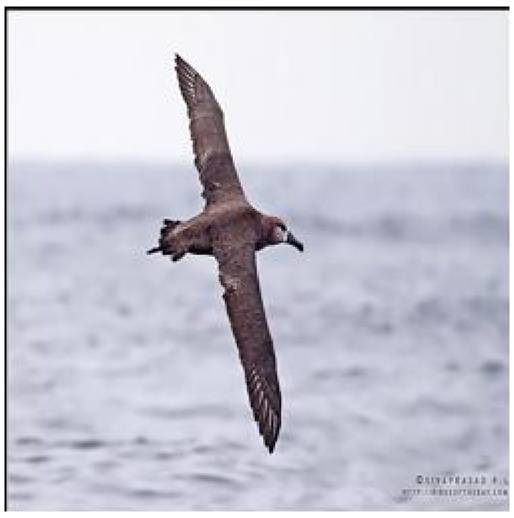
Steps

- Apply SLIC algorithm on the input image to get superpixels
    - ```
      def slic(originalImage):
          slic = cv2.ximgproc.createSuperpixelSLIC(originalImage ,region_size=16, ruler = 12.0)
          slic.iterate(20)      #Number of iterations, the greater the better
          return slic
      ```
- Apply k-means clustering on RGB centres of Superpixels
    - ```
      def get_cluster_of_superpixels(superpixel_centers_rgb, num_clusters):

          Z = np.array(superpixel_centers_rgb)
          Z = np.float32(Z)
          criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 10, 1.0)
          ret, cluster_labels, cluster_centers = cv2.kmeans(Z, num_clusters, None, criteria, 10, cv2.KMEANS_RANDOM_CENTERS
      ```
- Get contrast cue image using RGB data found above
- Display contrast image.
- Find xy centres of Superpixels and calculate spatial cue map using it.
- Display spatial image.
- Find threshold of pixels data of above two images using **Otsu.**
    - ```
      Contrast Otsu Threshold: 0.609458182867613
      Spatial Otsu Threshold: 0.3812414955042821
      ```
- Find Scores of above two images.
    - ```
      def find_score(min_threshold, map):
          fg_x = [j for i in map for j in i if j < min_threshold]
          bg_x = [j for i in map for j in i if j >= min_threshold]
          area_overlap = (NormalDist.from_samples(fg_x)).overlap(NormalDist.from_samples(bg_x))

          score = 1 / (1 + log10(1 + 256 * area_overlap))
          return score
      ```
- Take weighted sum of above two images.
    - ```
      # weighted sum of two cues to get saliency
      final_image = contrast_score * contrast_map + spatial_score * spatial_map
      cv2.imshow('weighted sum of score', final_image)
      cv2.waitKey()
      ```
      ```

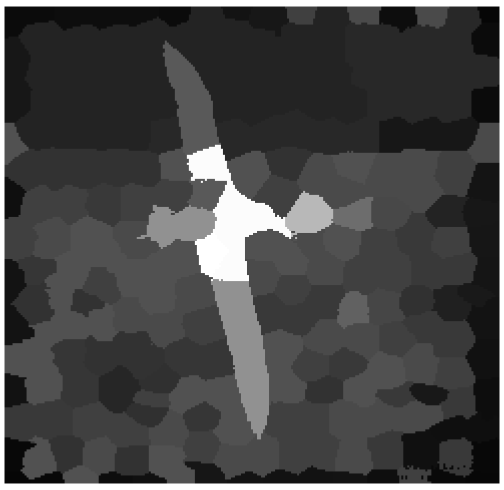
Input image:



Output Image:

Contrast Image:

Spatial Image:



```
Contrast Otsu Threshold: 0.609458182867613
Spatial Otsu Threshold: 0.3812414955042821
Contrast image Score: 0.9884750322276848
Spatial image Score: 0.423457411098437
```

As one can expect from the spatial image, foreground and background are not well separated and so the score is less while score of contrast map is good. Final image is better than both of the above.

Final Image with weighted sum of scores of above two:

```python
import cv2
import sys
import copy
import numpy as np
import itertools, math
import time, scipy
import scipy.stats
import matplotlib.pyplot as plt
import matplotlib
from scipy.stats import norm as NORM
import statistics
from statistics import NormalDist
from math import *

matplotlib.use('TkAgg')

def slic(originalImage):
    slic = cv2.ximgproc.createSuperpixelSLIC(originalImage ,region_size=16, ruler = 12.0)
    slic.iterate(20)    #Number of iterations, the greater the better
    return slic

def find_superpixels_centers_rgb(slic_img, originalImage):
    num_superpixel = slic_img.getNumberOfSuperpixels()
    labels = slic_img.getLabels()
    superpixel_centers_rgb = [[0]*4 for _ in range(num_superpixel)]

    height, width, channel = originalImage.shape

    for i in range(height):
       for j in range(width):
             cluster_label = labels[i][j]
             superpixel_centers_rgb[cluster_label][3] += 1
             for k in range(channel):
                   superpixel_centers_rgb[cluster_label][k] += originalImage[i][j][k]

    for i in range(num_superpixel):
       for j in range(3):
             superpixel_centers_rgb[i][j] /= superpixel_centers_rgb[i][3]
       superpixel_centers_rgb[i] = superpixel_centers_rgb[i][:3]
    return superpixel_centers_rgb

def find_superpixels_centers_xy(slic_img, originalImage):
    num_superpixel = slic_img.getNumberOfSuperpixels()
    labels = slic_img.getLabels()
    superpixel_centers = [[0]*3 for _ in range(num_superpixel)]

    height, width, channel = originalImage.shape

    for i in range(height):
       for j in range(width):
             cluster_label = labels[i][j]
             superpixel_centers[cluster_label][2] += 1
             superpixel_centers[cluster_label][0] += i
             superpixel_centers[cluster_label][1] += j

    for i in range(num_superpixel):
       for j in range(2):
             superpixel_centers[i][j] /= superpixel_centers[i][2]
       superpixel_centers[i] = superpixel_centers[i][:2]
    return superpixel_centers


def get_cluster_of_superpixels(superpixel_centers_rgb, num_clusters):

    Z = np.array(superpixel_centers_rgb)
    Z = np.float32(Z)
    criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 10, 1.0)
    ret, cluster_labels, cluster_centers = cv2.kmeans(Z, num_clusters, None, criteria, 10, cv2.KMEANS_RANDOM_CENTERS)
```

```python
    # Now convert back into uint8, and make original image
    cluster_centers = np.uint8(cluster_centers)

    # cv2.imshow('res2',res2)
    # cv2.waitKey(0)
    # cv2.destroyAllWindows()

    return cluster_centers, cluster_labels

def norm(a, b):
    return sum([(i - j)**2 for i, j in zip(a, b)]) ** 0.5

def contrast_cue(num_clusters, label, cluster_centers):
    cluster_centers = np.int32(cluster_centers)
    N = label.shape[0]
    contrast_cue_list = []
    frequency_pixels = [0] * (num_clusters)

    for i in label:
       frequency_pixels[i[0]] += 1

    for k in range(num_clusters):
       cue_k = 0
       for i in range(num_clusters):
             if i != k:
                   cue_k += frequency_pixels[i] * norm(cluster_centers[k], cluster_centers[i])

       contrast_cue_list.append(cue_k / N)

    return contrast_cue_list


def spatial_cue(num_clusters, label, center, superpixel_centers_xy):
    n = len(superpixel_centers_xy)
    spatial_cue_list = []
    pixel_label = [[] for _ in range(num_clusters)]
    sigma = 40
    frequency_pixels = [0] * (num_clusters)
    center_of_superpixels = [sum(i[0] for i in superpixel_centers_xy) / n,  sum(i[1] for i in superpixel_centers_xy) / n]

    for i in label:
       frequency_pixels[i[0]] += 1

    for i in range(n):
       pixel_label[label[i][0]].append(superpixel_centers_xy[i])

    for k in range(num_clusters):
       cue = 0
       for pixel_loc in pixel_label[k]:
             cue += math.exp(-(norm(center_of_superpixels ,pixel_loc)/sigma))
       spatial_cue_list.append(cue / frequency_pixels[k])

    return spatial_cue_list

def map_cue_to_image(cue, num_clusters, originalImage, superpixel_labels, cluster_labels):
    # normalise cue
    max_cue = max(cue)
    cue = [(i / max_cue) for i in cue]
    superpixels_to_cue = {}
    image = [[0] * originalImage.shape[1] for _ in range(originalImage.shape[0])]

    for i in range(len(cluster_labels)):
       superpixels_to_cue[i] = cue[cluster_labels[i][0]]

    for i in range(originalImage.shape[0]):
       for j in range(originalImage.shape[1]):
             image[i][j] = np.float(superpixels_to_cue[superpixel_labels[i][j]])

    image = np.float64(image)
```

```python
    cv2.imshow('cue image', image)
    cv2.waitKey()
    cv2.destroyAllWindows()
    return image

def get_frequency_distribution(grayImage):
    '''
    returns a mapping of pixel values with their frequency in the grayImage of original Inamge
    '''
    rows, cols = grayImage.shape

    frequency = {} # index denotes the pixel values so frequency[i] denotes the count of pixels of value i in the grayImage

    for row in range(rows):
       for col in range(cols):
             if grayImage[row][col] not in frequency:
                   frequency[grayImage[row][col]] = 0
             frequency[grayImage[row][col]] += 1

    return frequency



def calculate_variance(frequency, start, end):
    '''
    'end' not inclusive in the range
    '''
    mean = 0
    count = 0 # number of pixel in given range
    standard_deviation = 0

    for i in frequency:
       if i >= start and i <= end:
             mean += frequency[i] * i
             count += frequency[i]

    mean /= count
    variance = 0
    for i in frequency:
       if i >= start and i < end:
             variance += ((mean - i) ** 2) * frequency[i]

    if count != 0:
       variance /= count

    standard_deviation = variance ** 0.5
    return (count, variance, standard_deviation, mean)


def otsu(grayImage):
    '''
    Apply Otsu Algorithm on given Image
    '''
    frequency = get_frequency_distribution(grayImage) # a mapping of pixel values with their frequency in the grayImage of original
Inamge
    min_threshold = 0
    min_variance = 10**100


    for i in frequency:
       w0, v0, fg_std, fg_mean = calculate_variance(frequency, -1, i)
       w1, v1, bg_std, bg_mean = calculate_variance(frequency, i, 1.1)
       weighted_variance = w0 * v0 + w1 * v1

       if weighted_variance < min_variance or i == 1:
             min_variance = weighted_variance
             min_threshold = i

    return min_threshold
```

```python
def find_score(min_threshold, map):
    fg_x = [j for i in map for j in i if j < min_threshold]
    bg_x = [j for i in map for j in i if j >= min_threshold]
    area_overlap = (NormalDist.from_samples(fg_x)).overlap(NormalDist.from_samples(bg_x))

    score = 1 / (1 + log10(1 + 256 * area_overlap))
    return score
    # x_values = np.array(sorted([j for i in map for j in i if j < min_threshold]))
    # mean = statistics.mean(x_values)
    # sd = statistics.stdev(x_values)
    # plt.plot(x_values, NORM.pdf(x_values, mean, sd) )

    # x_values = np.array(sorted([j for i in map for j in i if j >= min_threshold]))
    # mean = statistics.mean(x_values)
    # sd = statistics.stdev(x_values)
    # plt.plot(x_values, NORM.pdf(x_values, mean, sd))
    # plt.show()

def main(image_path, num_clusters):

    originalImage = cv2.imread(image_path)
    # Apply Slic
    slic_img = slic(originalImage)
    superpixel_labels = slic_img.getLabels()

    superpixel_centers_rgb = find_superpixels_centers_rgb(slic_img, originalImage)
    # Apply k-means on rbg centers of Superpixels
    cluster_centers, cluster_labels = get_cluster_of_superpixels(superpixel_centers_rgb, num_clusters)

    # Get contrast cue image using rbg data found above
    contrast_cue_list = contrast_cue(num_clusters, cluster_labels, cluster_centers)

    # Find xy centers of Superpixels and calculate spatial cue map using it
    superpixel_centers_xy = find_superpixels_centers_xy(slic_img, originalImage)
    spatial_cue_list = spatial_cue(num_clusters, cluster_labels, cluster_centers, superpixel_centers_xy)

    # Display contrast image
    contrast_map = map_cue_to_image(contrast_cue_list, num_clusters, originalImage, superpixel_labels, cluster_labels)
    # Display spatial image
    spatial_map = map_cue_to_image(spatial_cue_list, num_clusters, originalImage, superpixel_labels, cluster_labels)

    contrast_min_threshold = otsu(contrast_map)
    spatial_min_threshold = otsu(spatial_map)

    print("Contrast Otsu Threshold:", contrast_min_threshold)
    print("Spatial Otsu Threshold:", spatial_min_threshold)

    contrast_score = find_score(contrast_min_threshold, contrast_map)
    print("Contrast image Score:", contrast_score)

    spatial_score = find_score(spatial_min_threshold, spatial_map)
    print("Spatial image Score:", spatial_score)

    # weighted sum of two cues to get saliency
    final_image = contrast_score * contrast_map + spatial_score * spatial_map
    cv2.imshow('weighted sum of score', final_image)
    cv2.waitKey()
    cv2.destroyAllWindows()

if __name__ == '__main__':
    if len(sys.argv) != 3:
       print("Usage: python3 code.py <path_of_image> <num_clusters>")
       exit(1)

    image_path = sys.argv[1]
    num_clusters = int(sys.argv[2])
    main(image_path, num_clusters)
```