Julia Nelson
CS 558
Homework 2
"I pledge my honor that I have abided by the Stevens Honor System."

Source Code 1: **kmeans.py**

```
import os
import numpy as np
import matplotlib.pyplot as plt
import cv2

def kmeans_seg(filename, k, iter):
    # Read in the image
    image = cv2.imread('white-tower.png')
    # Change color to RGB (from BGR)
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

    # plt.imshow(image)
    # plt.show()
    print(image.shape)
    # Reshaping the image into a 2D array of pixels and 3 color values (RGB)
    pixel_vals = image.reshape((-1, 3))
    print(pixel_vals.shape)

    # Convert to float type
    pixel_vals = np.float32(pixel_vals)

    stop_criteria = (
        cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER,
        iter,
        0.25)

    retval, labels, centers = cv2.kmeans(
        pixel_vals,
        k,
        None,
        stop_criteria,
        10,
        cv2.KMEANS_RANDOM_CENTERS)

    # convert data into 8-bit values
    centers = np.uint8(centers)
    segmented_data = centers[labels.flatten()]

    # reshape data into the original image dimensions
    segmented_image = segmented_data.reshape((image.shape))

    return segmented_image

if __name__ == '__main__':
    filename = "white-tower.png"
    k = 10
    iter = 100
    segmented_image = kmeans_seg(filename, k, iter)
    # plt.imshow(segmented_image)
    filename = os.path.splitext(filename)[0]
    new_filename = '{}_k{}_i{}.jpg'.format(filename, k, iter)

    bgr_image = cv2.cvtColor(segmented_image, cv2.COLOR_BGR2RGB)
    cv2.imwrite(new_filename, bgr_image)
```

1. In the first problem to implement k-means segmentation on the white-tower.png with k=10, I first read the image with cv2 and change the color to only consider the RGB color channel. Next, I reshape the image into a 2D array of pixels and 3 color values (RGB) and convert the pixel values to type float to provide higher precision in calculation, otherwise values will be rounded. The stop criteria is the condition that should be satisfied for the K-means algorithm to stop, either by reaching the max number of iterations, or reaching the threshold. The output of the K- means algorithm returns names, and centers coordinates for each cluster and the image is reshaped back to its original dimensions to be returned. I used "import os"'s function "splittext" in order to change the filename of the output.

Resulting Image:

Source Code 2: **slic.py**

```
import math
import numpy as np
from skimage import io, color

class Cluster(object):
    """  Data structure for cluster (representing superpixel)  """
    # cluster index counter
    clusterIdx = 0

    def __init__(self, x, y, r, g, b):
        self.x = x
        self.y = y
        self.r = r
        self.g = g
        self.b = b
        self.pixelsOfCluster = set()
        self.idx = Cluster.clusterIdx
        Cluster.clusterIdx += 1

    def setCluster(self, x, y, r, g, b):
        self.x = x
        self.y = y
        self.r = r
        self.g = g
        self.b = b

class Slic(object):
    def __init__(self, filepath, S=50, M=10):
        """ Input:
            Filepath: Name of file, or path to file
            S: (int) Size of superpixels
            M: (Int) Compactness (scaling of distance)
        """
        self.S = S
        # Initialize number of superpixels (K), and compactness (M)
        # self.K = K
        self.M = M

        # Read in image from filepath as CIELAB color space ndarray
        self.colorArr = color.rgb2lab(io.imread(filepath))
        # self.colorArr = io.imread(filepath)
        self.imageArr = np.copy(self.colorArr)

        # Set dimensions
        self.height = self.colorArr.shape[0]
        self.width = self.colorArr.shape[1]

        # Set number of pixels (N), and superpixel interval size (S)
        self.N = self.height * self.width
        # self.S = int(math.sqrt(self.N / K))
        self.K = int(self.N // math.pow(self.S, 2))

        # Track clusters, and labels for each pixel
        self.clusters = []
        self.labels = {}

        # Tracks distances to nearest cluster center (Initialized as largest possible value)
        self.distanceArr = np.full((self.height, self.width), np.inf)

    def run(self, iter, weight=0.5, bordered=False):
        """ Runs the Slic operation on the given image
        :param iter: number of iterations
        :param weight: label weight weight
        :param bordered: True if the clusters are bordered
        :return: None
        """
        self.initClusters()
        self.updateClusters()

        for i in range(iter):
            self.updatePixels(weight)
            self.updateCenters()
            name = 'slic_out/imseg_M{m}_K{k}_iter{loop}.png'.format(loop = i, m = self.M, k = self.K)
            self.save_image(name, bordered)
            print("clusters: {}".format(len(self.clusters)))

def initClusters(self):
        """  Initialize clusters based on S
        :return:
        """
        # (x, y) serves as current coordinates for setting cluster centers
        x = self.S // 2
        y = self.S // 2

        # Run across image and set cluster centers
        while y < self.height:

            while x < self.width:
                # Add new cluster centered at (x, y)
                r, g, b = self.colorArr[y][x]
                cluster = Cluster(x, y, r, g, b)
                self.clusters.append(cluster)

                # Iterate horizontally by the cluster iteration size S
                x += self.S

            # Reset horizontal coordinate, and iterate vertically by S
            x = self.S // 2
            y += self.S
```

```
def updateClusters(self):
    """Execute update if gradient of neighbor is smaller than current gradient"""

    def calculateGradient(x, y):
        """Compute the gradient for the pixel with coordinates (x, y) using L2 norm
        Return: Gradient from L2 norm of lab-vector
        Input:
            x - (Int) Horizontal Coordinate
            y - (Int) Vertical Coordinate
        """
        # Handle coordinates on edge
        if not (x + 1 < self.width):
            x = self.width - 2

        if not (y + 1 < self.height):
            y = self.height - 2
        # Computes the gradient using L2 norm
        Gx = np.linalg.norm(
            self.colorArr[y][x + 1] - self.colorArr[y][x - 1],
            ord=2) ** 2
        Gy = np.linalg.norm(
            self.colorArr[y + 1][x] - self.colorArr[y - 1][x],
            ord=2) ** 2
        return Gx + Gy

    for cluster in self.clusters:
        currGradient = calculateGradient(cluster.x, cluster.y)
        changeMade = True

        # Continue while gradient is not minimal
        while (changeMade):
            changeMade = False

            # Check gradients on each adjacent pixel and adjust accordingly
            for (dx, dy) in ((-1, -1), (-1, 0), (-1, 1), (0, -1), (0, 1), (1, -1), (1, 0), (1, 1)):
                _x = cluster.x + dx
                _y = cluster.y + dy

                if _x > 0 and _x < self.width - 1 and _y > 0 and _y < self.height - 1:
                    newGradient = calculateGradient(_x, _y)
                    if newGradient < currGradient:
                        changeMade = True
                        _l, _a, _b = self.colorArr[_y][_x]
                        cluster.setCluster(_x, _y, _l, _a, _b)
                        currGradient = newGradient

        cluster.pixelsOfCluster.add((cluster.y, cluster.x))

def updateCenters(self):
    """ Update centers for each clusters by using mean of (x, y) coordinates  """

    for cluster in self.clusters:
        widthTotal = heightTotal = count = 0
        for p in cluster.pixelsOfCluster:
            heightTotal += p[0]
            widthTotal += p[1]
            count += 1
        if count == 0:
            count = 1
        _x = widthTotal // count
        _y = heightTotal // count
        _r = self.colorArr[_y][_x][0]
        _g = self.colorArr[_y][_x][1]
        _b = self.colorArr[_y][_x][2]
        cluster.setCluster(_x, _y, _r, _g, _b)

def updatePixels(self, weight):
    """ Update each pixel to the closest cluster"""
    it = self.S * 2
    for cluster in self.clusters:
        for y in range(max(cluster.y - it, 0), min(cluster.y + it, self.height)):
            for x in range(max(cluster.x - it, 0), min(cluster.x + it, self.width)):
                r, g, b = self.colorArr[y][x]

                labDistance = math.sqrt(
                    (r - cluster.r) ** 2
                    + (g - cluster.g) ** 2
                    + (b - cluster.b) ** 2)

                xyDistance = math.sqrt(
                    (x - cluster.x) ** 2
                    + (y - cluster.y) ** 2)

                # Avoiding scaling xyDistance by 1 / S for prettier superpixels
                D = weight * labDistance + (1 - weight) * (self.M) * xyDistance

                # Update if scaled distance is better than previous minimal distance
                if D < self.distanceArr[y][x]:
                    pixel = (y, x)

                    # Update label for this pixel
                    if pixel not in self.labels:
                        self.labels[pixel] = cluster
                        cluster.pixelsOfCluster.add(pixel)
                    else:
                        self.labels[pixel].pixelsOfCluster.remove(pixel)
                        self.labels[pixel] = cluster
                        cluster.pixelsOfCluster.add(pixel)
                    self.distanceArr[y][x] = D
```

```
    def save_image(self, path, isBordered):
        """
        Saves segmented image, along with borders and center indications
        """
        def isBlack(px, py):
            return (
                    self.imageArr[py][px][0] == 0
                    and self.imageArr[py][px][1] == 0
                    and self.imageArr[py][px][2] == 0)

        def willBeBorder(px, py):
            """
            Return:
                True, if any pixel adjacent to the passed pixel is of a different cluster and not black
                False, otherwise
            Input:
                px - (Int) Horizontal component of pixel
                py - (Int) Vertical component of pixel
            """

            L = self.labels[(py, px)]

            return (((px == 0) or ((self.labels[(py, px - 1)] != L) and not isBlack(px - 1, py))) \
                    or ((px == self.width - 1) or ((self.labels[(py, px + 1)] != L) and not isBlack(px + 1, py))) \
                    or ((py == 0) or ((self.labels[(py - 1, px)] != L) and not isBlack(px, py - 1))) \
                    or ((py == self.height - 1) or ((self.labels[(py + 1, px)] != L) and not isBlack(px, py + 1))))

        def indicateClusterCenter(cx, cy):
            self.imageArr[cy][cx][0] = 0
            self.imageArr[cy][cx][1] = 0
            self.imageArr[cy][cx][2] = 0

        # Starting
        self.imageArr = np.copy(self.colorArr)

        if isBordered:
            for cluster in self.clusters:
                for p in cluster.pixelsOfCluster:
                    px = p[1]
                    py = p[0]

                    # If not completed surrounded by pixels of same label, change to black to indicate border
                    if (willBeBorder(px, py)):
                        self.imageArr[py][px][0] = 0
                        self.imageArr[py][px][1] = 0
                        self.imageArr[py][px][2] = 0

                    # Indicate pixel labels if it is not a border of the cluster
                    else:
                        self.imageArr[py][px][0] = cluster.r
                        self.imageArr[py][px][1] = cluster.g
                        self.imageArr[py][px][2] = cluster.b

                indicateClusterCenter(cluster.x, cluster.y)
        else:

            for cluster in self.clusters:
                for p in cluster.pixelsOfCluster:
                    px = p[1]
                    py = p[0]

                    self.imageArr[py][px][0] = cluster.r
                    self.imageArr[py][px][1] = cluster.g
                    self.imageArr[py][px][2] = cluster.b

                indicateClusterCenter(cluster.x, cluster.y)
        # img_uint8 = self.imageArr.astype(np.uint8)
        io.imsave(path, color.lab2rgb(self.imageArr))
        # io.imsave(path, self.imageArr)

if __name__ == '__main__':

    processor = Slic('wt_slic.png', 50, 10)
    processor.run(15, 0.2, True)
```

2. In the second problem to implement SLIC, the method of applying a variant of the SLIC algorithm to wt_slic.png tries to represent the image by superpixels instead of original pixels, the superpixel is formed by a group of similar pixels, it's first initialized by a square (ie. 50x50), then it will be updated through iterations by updating the membership of each pixel in the appropriate superpixel. This is through calculating the combined gradient magnitude. Then apply kmeans and display output.

Resulting Images: