Source Code:

```
# -*- coding: utf-8 -*-
"""
Created on Thu Mar 30 18:43:17 2017

@author: Kunal
"""


from PIL import Image
import numpy as np
import random as rn
import math as mt

'''To resize the image to replicate border pixels before convolution as per the filter_size'''
def resize_image(ip_im, filter_size):
    r, c = ip_im.shape
    filter_n = int((filter_size-1)/2)
    op_r = r+2*(filter_n)
    op_c = c+2*(filter_n)
    op_im = np.zeros((op_r,op_c))
    for i in range(r):
        for j in range(c):
            op_im[i+filter_n][j+filter_n] = ip_im[i][j]
    for i in range(filter_n):
        for j in range(filter_n):
            op_im[i][j] = op_im[filter_n][filter_n]
    for i in range(filter_n):
        for j in range(op_c-filter_n, op_c):
            op_im[i][j] = op_im[filter_n][op_c-filter_n-1]
    for i in range(op_r-filter_n, op_r):
        for j in range(filter_n):
            op_im[i][j] = op_im[op_r-filter_n-1][filter_n]
    for i in range(op_r-filter_n, op_r):
        for j in range(op_c-filter_n, op_c):
            op_im[i][j] = op_im[op_r-filter_n-1][op_c-filter_n-1]
    for i in range(filter_n):
        for j in range(filter_n, op_c-filter_n):
            op_im[i][j] = op_im[filter_n][j]
    for i in range(op_r-filter_n, op_r):
        for j in range(filter_n, op_c-filter_n):
            op_im[i][j] = op_im[op_r-filter_n-1][j]
    for i in range(filter_n, op_r-filter_n):
        for j in range(filter_n):
            op_im[i][j] = op_im[i][filter_n]
    for i in range(filter_n, op_r-filter_n):
        for j in range(op_c-filter_n, op_c):
            op_im[i][j] = op_im[i][op_c-filter_n-1]
    return op_im
```

```
'''To perform convolution of ip with filter'''
def convolution(ip,filter):
    filter_size = int(mt.sqrt(filter.size))
    filter_n = int((filter_size-1)/2)
    ip_r, ip_c = ip.shape
    r = ip_r - 2*filter_n
    c = ip_c - 2*filter_n
    op_im = np.zeros((r, c))
    for i in range(r):
        for j in range(c):
            for k in range(filter_size):
                for l in range(filter_size):
                    op_im[i][j] = op_im[i][j] + (filter[k][l] * ip[i+k][j+l])
    return op_im

'''To perform K-means Clustering algorithm'''
def k_mean(ip_im, k):
    ip_im = np.asarray(ip_im)
    row, col, x = ip_im.shape
    no_iter = 1
    converge_flag = 0
    centers = []
    while(len(centers) != k):
        ran_i = rn.randint(0, row-1)
        ran_j = rn.randint(0, col-1)
        ran_c = [ran_i, ran_j, ip_im[ran_i][ran_j][0], ip_im[ran_i][ran_j][1], ip_im[ran_i][ran_j][2]]
        if(ran_c not in centers):
            centers.append(ran_c)
    while(converge_flag == 0):
        converge_flag = 1
        print("Iteration "+str(no_iter))
        clusters = [[] for i in range(len(centers))]
        for i in range(row):
            for j in range(col):
                dist = 9999.0
                for k in range(len(centers)):
                    c_dist = dist_color(centers[k][2], centers[k][3], centers[k][4], ip_im[i][j][0], ip_im[i][j][1],
ip_im[i][j][2])
                    if(c_dist < dist):
                        dist = c_dist
                        c_ind = k
                clusters[c_ind].append([i, j, ip_im[i][j][0], ip_im[i][j][1], ip_im[i][j][2]])
        no_new_c = 0
        for i in range(len(clusters)):
            no_pts = len(clusters[i])
            r = 0; g = 0; b = 0
            for pt in clusters[i]:
```

```
                r = r + (int(pt[2])*int(pt[2]))
                g = g + (int(pt[3])*int(pt[3]))
                b = b + (int(pt[4])*int(pt[4]))
            r = r/no_pts; g = g/no_pts; b = b/no_pts
            r = int(mt.sqrt(r)); g = int(mt.sqrt(g)); b = int(mt.sqrt(b))
            new_color = [r, g, b]
            old_color = [centers[i][2], centers[i][3], centers[i][4]]
            if(new_color != old_color):
                no_new_c = no_new_c + 1
                centers[i][2] = r; centers[i][3] = g; centers[i][4] = b
                converge_flag = 0
        print("New centers:", no_new_c)
        print()
        no_iter = no_iter + 1
    im = Image.new('RGB', (col, row))
    op_im = redraw_cluster(im, centers, clusters)
    return op_im

'''To calculate the color distance for K-means'''
def dist_color(r1, g1, b1, r2, g2, b2):
    r_mean = (int(r1) + int(r2)) / 2
    r = int(r1) - int(r2)
    g = int(g1) - int(g2)
    b = int(b1) - int(b2)
    dist = float(mt.sqrt( (((512+r_mean)*r*r)/256) + (4*g*g) + (((767-r_mean)*b*b)/256) ))
    return dist

'''To redraw the image as per the provided clusters and centers'''
def redraw_cluster(ip_im, centers, clusters):
    ip_im = np.asarray(ip_im)
    ip_im.setflags(write = 1)
    for i in range(len(centers)):
        center = centers[i]
        color = [center[2], center[3], center[4]]
        for j in clusters[i]:
            ip_im[j[0]][j[1]] = color
    op_im = Image.fromarray(ip_im)
    return op_im

'''To perform the SLIC algorithm'''
def slic(ip_im, block_n):
    ip_im = np.asarray(ip_im)
    row, col, x = ip_im.shape
    centers = []
    c_i = int(block_n/2)
    for i in range(int(row/block_n)):
        c_j = int(block_n/2)
        for j in range(int(col/block_n)):
```

```
            centers.append([c_i, c_j, ip_im[c_i][c_j][0], ip_im[c_i][c_j][1], ip_im[c_i][c_j][2]])
            c_j = int(c_j + block_n)
        c_i = int(c_i + block_n)
    grad_ip_im = rgb_grad(ip_im)
    for center in centers:
        grad = 9999.0
        c_i = center[0] - 1
        c_j = center[1] - 1
        for i in range(c_i, c_i + 3):
            for j in range(c_j, c_j + 3):
                if(grad_ip_im[i][j] < grad):
                    grad = grad_ip_im[i][j]
                    new_c_i = i
                    new_c_j = j
        center[0] = new_c_i
        center[1] = new_c_j
        center[2] = ip_im[new_c_i][new_c_j][0]
        center[3] = ip_im[new_c_i][new_c_j][1]
        center[4] = ip_im[new_c_i][new_c_j][2]
    converge_flag = 0
    no_iter = 1
    while(converge_flag == 0):
        print("Iteration "+str(no_iter))
        converge_flag = 1
        clusters = [[] for i in range(len(centers))]
        for i in range(row):
            for j in range(col):
                dist = 9999.0
                for k in range(len(centers)):
                    if(((i-(block_n*2))<=centers[k][0]<=(i+(block_n*2))) and ((j-
(block_n*2))<=centers[k][1]<=(j+(block_n*2)))):
                        c_dist = dist_eucl(centers[k][0], centers[k][1], centers[k][2], centers[k][3], centers[k][4], i, j,
ip_im[i][j][0], ip_im[i][j][1], ip_im[i][j][2], block_n)
                        if(c_dist < dist):
                            dist = c_dist
                            c_ind = k
                clusters[c_ind].append([i, j, ip_im[i][j][0], ip_im[i][j][1], ip_im[i][j][2]])
        no_new_c = 0
        for i in range(len(clusters)):
            no_pts = len(clusters[i])
            r = 0; g = 0; b = 0; x = 0; y = 0
            for pt in clusters[i]:
                x = x + pt[0]
                y = y + pt[1]
                r = r + (int(pt[2])*int(pt[2]))
                g = g + (int(pt[3])*int(pt[3]))
                b = b + (int(pt[4])*int(pt[4]))
            if(no_pts != 0):
```

```
            r = r/no_pts; g = g/no_pts; b = b/no_pts; x = int(x/no_pts); y = int(y/no_pts)
        r = int(mt.sqrt(r)); g = int(mt.sqrt(g)); b = int(mt.sqrt(b))
        new_center = [x, y, r, g, b]
        old_center = [centers[i][0], centers[i][1], centers[i][2], centers[i][3], centers[i][4]]
        if(new_center != old_center):
            no_new_c = no_new_c + 1
            centers[i][0] = x; centers[i][1] = y;
            centers[i][2] = r; centers[i][3] = g; centers[i][4] = b
            converge_flag = 0
    print("New centers:", no_new_c)
    print()
    no_iter = no_iter + 1
  im = Image.new('RGB', (col, row))
  op_im = redraw_cluster(im, centers, clusters)
  op_im = redraw_bound(op_im)
  return op_im

'''To calculate the image gradient'''
def rgb_grad(ip_im):
  row, col, x = ip_im.shape
  filter_x = [[-1,0,+1], [-2,0,+2], [-1,0,+1]]
  filter_x = np.array(filter_x)
  filter_y = [[+1,+2,+1], [0,0,0], [-1,-2,-1]]
  filter_y = np.array(filter_y)
  r_chan = np.zeros((row, col))
  g_chan = np.zeros((row, col))
  b_chan = np.zeros((row, col))
  grad = np.zeros((row, col))
  for i in range(row):
    for j in range(col):
      r_chan[i][j] = ip_im[i][j][0]
      g_chan[i][j] = ip_im[i][j][1]
      b_chan[i][j] = ip_im[i][j][2]
  r_chan = resize_image(r_chan, 3)
  r_chan_x = convolution(r_chan,filter_x)
  r_chan_y = convolution(r_chan,filter_y)
  g_chan = resize_image(g_chan, 3)
  g_chan_x = convolution(g_chan,filter_x)
  g_chan_y = convolution(g_chan,filter_y)
  b_chan = resize_image(b_chan, 3)
  b_chan_x = convolution(b_chan,filter_x)
  b_chan_y = convolution(b_chan,filter_y)
  for i in range(row):
    for j in range(col):
      r_comp = mt.sqrt((r_chan_x[i][j]**2) + (r_chan_y[i][j]**2))
      g_comp = mt.sqrt((g_chan_x[i][j]**2) + (g_chan_y[i][j]**2))
      b_comp = mt.sqrt((b_chan_x[i][j]**2) + (b_chan_y[i][j]**2))
      grad[i][j] = mt.sqrt((r_comp**2) + (g_comp**2) + (b_comp**2))
```

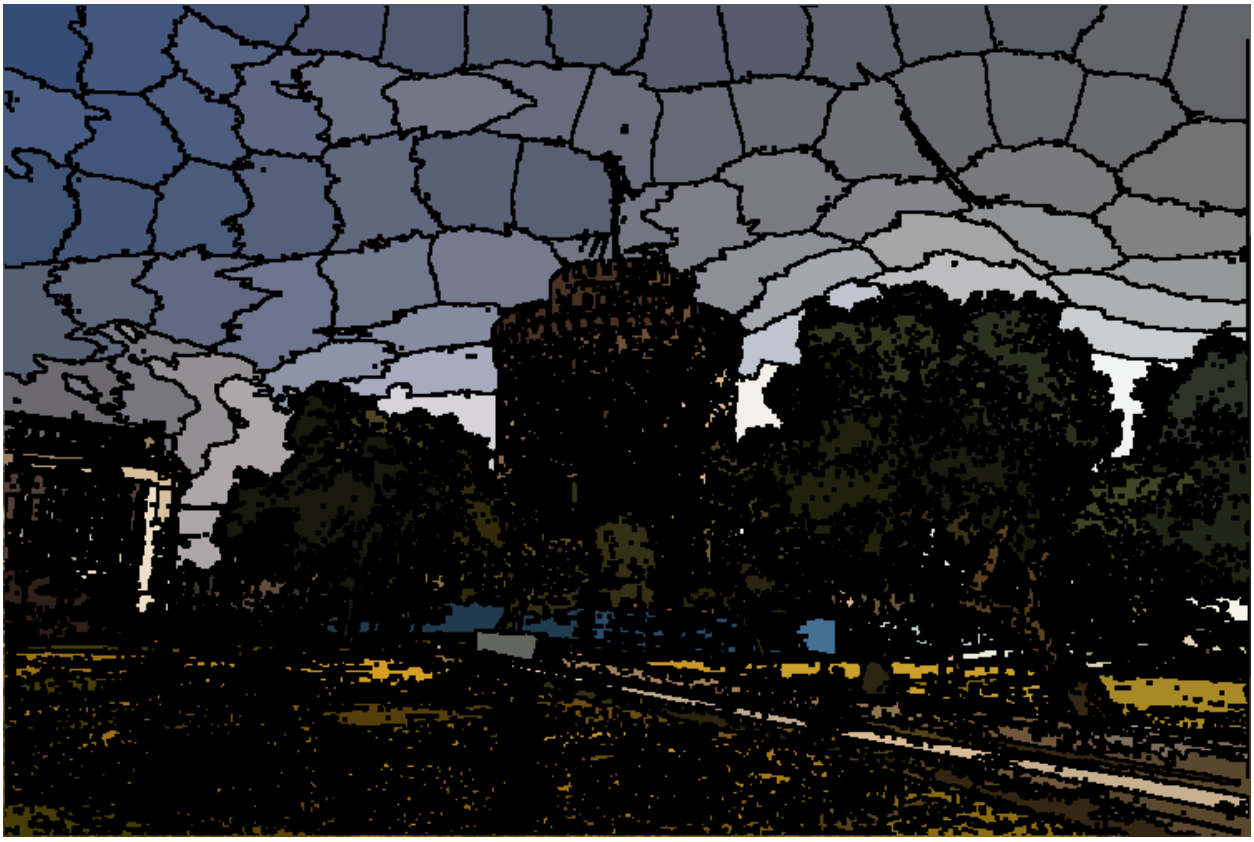
```
    return grad

'''To compute the 5D distance for the SLIC algorithm'''
def dist_eucl(x1, y1, r1, g1, b1, x2, y2, r2, g2, b2, block_n):
    r = int(r2) - int(r1)
    g = int(g2) - int(g1)
    b = int(b2) - int(b1)
    dist = float(mt.sqrt((((x2/2)-(x1/2))**2)+(((y2/2)-(y1/2))**2)+(r**2)+(g**2)+(b**2)))
    return dist

'''To redraw the image with the black boundaries'''
def redraw_bound(ip_im):
    op_im = ip_im.copy()
    op_im = np.asarray(op_im)
    op_im.setflags(write = 1)
    ip_im = np.asarray(ip_im)
    ip_im.setflags(write = 1)
    row, col, x = ip_im.shape
    for i in range(1, row-1):
        for j in range(1, col-1):
            test = []
            test.append(str(ip_im[i-1][j-1][0])+str(ip_im[i-1][j-1][1])+str(ip_im[i-1][j-1][2]))
            test.append(str(ip_im[i-1][j][0])+str(ip_im[i-1][j][1])+str(ip_im[i-1][j][2]))
            test.append(str(ip_im[i-1][j+1][0])+str(ip_im[i-1][j+1][1])+str(ip_im[i-1][j+1][2]))
            test.append(str(ip_im[i][j-1][0])+str(ip_im[i][j-1][1])+str(ip_im[i][j-1][2]))
            test.append(str(ip_im[i][j][0])+str(ip_im[i][j][1])+str(ip_im[i][j][2]))
            test.append(str(ip_im[i][j+1][0])+str(ip_im[i][j+1][1])+str(ip_im[i][j+1][2]))
            test.append(str(ip_im[i+1][j-1][0])+str(ip_im[i+1][j-1][1])+str(ip_im[i+1][j-1][2]))
            test.append(str(ip_im[i+1][j][0])+str(ip_im[i+1][j][1])+str(ip_im[i+1][j][2]))
            test.append(str(ip_im[i+1][j+1][0])+str(ip_im[i+1][j+1][1])+str(ip_im[i+1][j+1][2]))
            if(len(set(test)) != 1):
                op_im[i][j] = [0, 0, 0]
    op_im = Image.fromarray(op_im)
    return op_im

'''The main function'''
def main():
    og_im1 = Image.open("white-tower.png")
    og_im1.show()
    k = 2
    k_im = og_im1.copy()
    k_op_im = k_mean(k_im, k)
    k_op_im.show()
    og_im2 = Image.open("wt_slic.png")
    og_im2.show()
    block_n = 50
    slic_op_im = slic(og_im2, block_n)
    slic_op_im.show()
```

main()

Output Images:

K-means Clustering: The final output of K-means clustering algorithm when the value of k is 10.

SLIC: The output of SLIC algorithm without the black boundaries.



SLIC: The output of SLIC algorithm with the black boundaries.

Notes:

K-means Clustering Algorithm:

For the K-means Clustering algorithm, I am initializing k centers randomly from the given image. The value of k for the given assignment is 10. I compare the color space distances between each pixel and all the centers, and a pixel is assigned to the corresponding cluster for which its color distance is the minimum. For calculating the color distance here, I used the method explained over here: https://www.compuphase.com/cmetric.htm

Once the clusters are formed, I compute the centers of all the clusters again. Here, instead of just averaging, I take squares of values, then average, and then take the square root. The reason for doing this can be understood over here: https://www.youtube.com/watch?v=LKnqECcg6Gw

The newly formed center is then compared with the old center. If it is the same, then that given center has converged. Each of the centers is checked for convergence. Once all the centers have converged, the process is stopped. For each cluster, all the cluster points are then filled with the color of their corresponding cluster center. The output image 'K-means Final.png' was obtained from this process. It can be noted that the final output image has k number of color values (RGB triplets) in it. It took 54 iterations to converge to the attached output image.

SLIC Algorithm:

For SLIC algorithm, I initialize the centers as instructed, such that S is 50. The overall gradient is computed and accordingly, the centers are moved to the position with the smallest image gradient in a 3x3 window placed on the center. Now, K-means is applied in a 5D space of (x, y, R, G, B) as instructed. In this step, x and y are divided by 2 for normalization. Having a 5D space considers not just the color distance, but the spatial distance as well. Also, in this process, I am comparing each pixel only with centers that fall with a region of 2Sx2S around it, i.e., a region of 100x100 around it. As in the K-means algorithm, after the comparisons, clusters are formed, and then from the points in the clusters, new centers are computed by averaging. For this step, I am considering the spatial average as well, because the spatial distance was considered during the process of cluster formation too. However, I tested without considering the spatial average, and it doesn't produce much of a difference in the clusters that are formed, and thus, there is no significant change in the image generated. This process is run till convergence too, and the final image is produced by coloring all the points in a cluster with its average RGB value, which is its center, and coloring the points lying on the boundary between any 2 clusters black.