Siddhant Barua
Computer Vision Report

K-Means

```
########################################
# value of K for k-means
#########################################


def main():
    input_img = Image.open("white-tower.png")
    input_img.show()
    temp_img = np.asarray(input_img)
    r, c, var = temp_img.shape
    iterations = 1
    convergance = 0
    centres = []
    while (len(centres) != 10):  # this is the value of K i.e. there are 10
centers and clusters
        randomC = [rn.randint(0, (r - 1)), rn.randint(0, (c - 1)),
                   temp_img[rn.randint(0, (r - 1))][rn.randint(0, (c -
1))][0],
                   temp_img[rn.randint(0, (r - 1))][rn.randint(0, (c -
1))][1],
                   temp_img[rn.randint(0, (r - 1))][rn.randint(0, (c -
1))][2]]
        if (randomC not in centres):
            centres.append(randomC)

    while convergance == 0:
        convergance = 1
        clusters = [[] for i in range(len(centres))]  # since there are 10
K means points, there will be 10
        # clusters
        for i in range(r):
            for j in range(c):
                distance = 10211.0
                for k in range(len(centres)):
                    Red = int(centres[k][2]) - int(temp_img[i][j][0])  #
calculating the difference between the
```

```
                    # randomly generated centers Red Value and the Red
Value of the given Image
                    Green = int(centres[k][3]) - int(temp_img[i][j][1])  #
calculating the difference between the
                    # randomly generated centers Green Value and the Green
Value of the given Image
                    Blue = int(centres[k][4]) - int(temp_img[i][j][2])  #
#calculating the difference between the
                    # randomly generated centers RBlue Value and the Blue
Value of the given Image
                    MeanVal = (int(centres[k][2]) + int(temp_img[i][j][0]))
/ 2
                    Color_Distance = float(mt.sqrt((((512 + MeanVal) * Red
* Red) / 256) + (4 * Green * Green) + (
                            ((767 - MeanVal) * Blue * Blue) / 256)))
                    if (Color_Distance < distance):
                        distance = Color_Distance
                        Index = k;  # starts at index 0, then goes on till
go on till k == 10
                clusters[Index].append([i, j, temp_img[i][j][0],
temp_img[i][j][1], temp_img[i][j][2]])
        NewCentCount = 0

        for i in range(len(clusters)):
            point_count = len(clusters[i])
            print(point_count)
            red = 0
            green = 0
            blue = 0
            for points in clusters[i]:
                # this is because clusters = [[], [], []...] and inside
[[x, y, r, g, b], [...], ... ]
                #                                                       0
1  2  3  4
                red = red + (int(points[2]) * int(points[2]))
                green = green + (int(points[3]) * int(points[3]))
                blue = blue + (int(points[4]) * int(points[4]))

            # if(point_count == 0):
            #     print("some issue with Random Numbers")
            #     exit()
```

```
            if (point_count != 0):
                red = int(mt.sqrt(red / point_count))
                green = int(mt.sqrt(green / point_count))
                blue = int(mt.sqrt(blue / point_count))

            # here we notice that the length of some clusters is 0 , in
which case K-Means takes longer to execute
            new_bin = [red, green, blue]  # here we assign the new colours
to the new color space
            old_bin = [centres[i][2], centres[i][3], centres[i][4]]  # this
is the old color- space
            print("this is  BIN")
            print(new_bin)
            print(old_bin)

            # if(abs(new_bin[0]-old_bin[0])<=0.000025 and
abs(new_bin[1]-old_bin[1])<=0.000025 and
abs(new_bin[2]-old_bin[2])<=0.000025 ):
            if new_bin != old_bin:
                NewCentCount += 1
                centres[i][
                    2] = red  # in this case if the new color space is not
equal to the older color space , then we iterate over the loop again
                centres[i][3] = green  # assign the new color centers
                centres[i][4] = blue
                convergance = 0  # this is what makes the flag condition
fail

        print("Number of centers: ", NewCentCount)
        print()

    Imgae = np.asarray(
        Image.new('RGB', (c, r)))  # creating a new RGB image with the same
dimentions as the original image.
    Imgae.setflags(write=1)
    for i in range(len(centres)):
        cent = centres[i]
        RGBval = [cent[2], cent[3], cent[4]]
        for j in clusters[i]:
            Imgae[j[0]][j[1]] = RGBval
    # here we write the new RGB value to the image, for K-means
    # this basically is used to reduce the size of the image, or compress
```

```
the image. This is done by taking the major color centers k and relegating
everything else
    FINAL_IMAGE = Image.fromarray(Imgae)
    FINAL_IMAGE.show()


main()
```

SLIC Algorithm

```
# From last week's lab

def add_pad(Test_img, Block):
    Row, Column = Test_img.shape
    SizeF = int((Block - 1) / 2)
    Image_row = Row + 2 * (SizeF)
    Image_col = Column + 2 * (SizeF)
    Final_img = np.zeros((Image_row, Image_col))

    # this is where padding takes place
    # this block adds padding to the original image , the borders of the
image
    # i.e. the rows and column values are copied to the added border. -
this is called replication of border pixels

    '''
    consider a matrix as follows [[1,2],
                                  [1,2,]]

    '''

    for i in range(Row):
        for j in range(Column):
            Final_img[i + SizeF][j + SizeF] = Test_img[i][j]

            # this add's zeroes to the borders of the image
            # [[0, 0, 0, 0],
            # [0, 1, 2, 0],
            # [0, 1, 2, 0],
```

```
            # [0, 0, 0, 0]]
    for i in range(SizeF):
        for j in range(SizeF):
            Final_img[i][j] = Final_img[SizeF][SizeF]

            # this copies the pixel value from location img[1,2] to the
borders of the image i.e [0,4]
            # [[1, 0, 0, 2],
            # [0, 1, 2, 0],
            # [0, 1, 2, 0],
            # [0, 0, 0, 0]]
    for i in range(SizeF):
        for j in range(Image_col - SizeF, Image_col):
            Final_img[i][j] = Final_img[SizeF][Image_col - SizeF - 1]

    # this copies the pixel value from location img[2,1] to the borders of
the image i.e [3,0]
    # [[1, 0, 0, 2],
    # [0, 1, 2, 0],
    # [0, 1, 2, 0],
    # [1, 0, 0, 0]]
    for i in range(Image_row - SizeF, Image_row):
        for j in range(SizeF):
            Final_img[i][j] = Final_img[Image_row - SizeF - 1][SizeF]
    # this copies the pixel value from location img[2,2] to the borders of
the image i.e [3,3]
    # [[1, 0, 0, 2],
    # [0, 1 2, 0],
    # [0, 1 2, 0],
    # [1, 0, 0, 2]]
    for i in range(Image_row - SizeF, Image_row):
        for j in range(Image_col - SizeF, Image_col):
            Final_img[i][j] = Final_img[Image_row - SizeF - 1][Image_col -
SizeF - 1]

    # this block makes sure the top border of the padded image has the same
pixel
    # values as the original image at the border , this is done for
dynamically changing
    # sizes of the image and it's applied gauss filter
    # [[1, 1, 2, 2],
    # [0, 1,  2, 0],
```

```
    # [0, 1,  2, 0],
    # [1, 0, 0, 2]]

    for i in range(SizeF):
        for j in range(SizeF, Image_col - SizeF):
            Final_img[i][j] = Final_img[SizeF][j]

    # this block makes sure the bottom border of the padded image has the
same pixel
    # values as the original image at the border , this is done for
dynamically changing
    # sizes of the image and it's applied gauss filter
    # [[1, 1, 2, 2],
    # [0, 1,  2, 0],
    # [0, 1,  2, 0],
    # [1, 1, 2, 2]]

    for i in range(Image_row - SizeF, Image_row):
        for j in range(SizeF, Image_col - SizeF):
            Final_img[i][j] = Final_img[Image_row - SizeF - 1][j]

    # this block makes sure the left border of the padded image has the
same pixel
    # values as the original image at the border , this is done for
dynamically changing
    # sizes of the image and it's applied gauss filter i.e the entire left
column
    # is changed to the same pixel values as the original image at the left
border / 1st column
    # [[1, 1, 2, 2],
    # [1, 1,  2, 0],
    # [1, 1,  2, 0],
    # [1, 1, 2, 2]]

    for i in range(SizeF, Image_row - SizeF):
        for j in range(SizeF):
            Final_img[i][j] = Final_img[i][SizeF]
    # this block makes sure the right border of the padded image has the
same pixel
    # values as the original image at the border , this is done for
dynamically changing
    # sizes of the image and it's applied gauss filter i.e the right left
```

```
column/ i =3 column/ 4th column
    # is changed to the same pixel values as the original image at the
right border / i = 3rd column /4th column
    # [[1, 1, 2, 2],
    # [1, 1,  2, 0],
    # [1, 1,  2, 0],
    # [1, 1, 2, 2]]

    for i in range(SizeF, Image_row - SizeF):
        for j in range(Image_col - SizeF, Image_col):
            Final_img[i][j] = Final_img[i][Image_col - SizeF - 1]
    return Final_img


'''To perform ConvoleMatrix of Test_img2 with filter'''


def ConvoleMatrix(Test_img2, filter):
    # this is Convolution ideas from the following source :
    # https: // matthew - brett.github.io / teaching / smoothing_intro.html
    #
https://stackoverflow.com/questions/2448015/2d-ConvoleMatrix-using-python-a
nd-numpy/42579291#42579291
    #
Ghttps://stackoverflow.com/questions/43086557/convolve2d-just-by-using-nump
y

    # formula h[m,n] =  SUMMATION   (g[k,l] + f[m-k, n-l])
    #                            k,l

    Block = int(mt.sqrt(filter.size))
    SizeF = int((Block - 1) / 2)
    Image_row, Image_col = Test_img2.shape
    Row = Image_row - 2 * SizeF
    Column = Image_col - 2 * SizeF
    Final_img = np.zeros((Row, Column))
    for i in range(Row):
        for j in range(Column):
            for k in range(Block):
                for l in range(Block):
                    Final_img[i][j] = Final_img[i][j] + (filter[k][l] *
Test_img2[i + k][j + l])
```

```
    return Final_img


# we first break the image into blocks and initialize a centroid value in
each block

def main():
    First_Imge = Image.open("wt_slic.png")
    First_Imge.show()
    block_n = 50

    Test_img = np.asarray(First_Imge)
    row, col, someVal = Test_img.shape
    centers = []
    Center_Row = int(block_n / 2)  # we are breaking the block into half
    for i in range(int(row / block_n)):
        Center_Col = int(block_n / 2)
        for j in range(int(col / block_n)):
            centers.append(
                [Center_Row, Center_Col,
Test_img[Center_Row][Center_Col][0], Test_img[Center_Row][Center_Col][1],
                 Test_img[Center_Row][Center_Col][2]])  # here the centers
value is [[x,y,R,G,B], [], ... ]
            Center_Col = int(Center_Col + block_n)
        Center_Row = int(Center_Row + block_n)
        # here we take the block and break it down into 50 X  50 parts and
then iterate through them.

        ROW, COL, x = Test_img.shape
        FiltX = [[-1, 0, +1], [-2, 0, +2], [-1, 0, +1]]
        # we are using a basic filter and then using ConvoleMatrix on it to
break it up into its gradiant coordinates
        FiltX = np.array(FiltX)
        FiltY = [[+1, +2, +1], [0, 0, 0], [-1, -2, -1]]
        FiltY = np.array(FiltY)
        Red_Channel = np.zeros((ROW, COL))  # creating a Red Color Channel
        Green_Channel = np.zeros((ROW, COL))  # creating a Green Color
Channel
        Blue_Channel = np.zeros((ROW, COL))  # creating a Blue Color
Channel
        grad = np.zeros((ROW, COL))  # gradiant coordinates
        for i in range(ROW):
```

```
            for j in range(COL):
                Red_Channel[i][j] = Test_img[i][j][0]  # assigning red 
color to the channel
                Green_Channel[i][j] = Test_img[i][j][1]  # assigning Green 
color to the channel
                Blue_Channel[i][j] = Test_img[i][j][2]  # assigning Blue 
color to the channel
        Red_Channel = add_pad(Red_Channel, 3)
        RX = ConvoleMatrix(Red_Channel, FiltX)  # performing convolution 
with the defined X filter
        RY = ConvoleMatrix(Red_Channel, FiltY)  # performing convolution 
with the defined Y filter
        Green_Channel = add_pad(Green_Channel, 3)
        GX = ConvoleMatrix(Green_Channel, FiltX)  # performing convolution 
with the defined X filter
        GY = ConvoleMatrix(Green_Channel, FiltY)  # performing convolution 
with the defined Y filter
        Blue_Channel = add_pad(Blue_Channel, 3)
        BX = ConvoleMatrix(Blue_Channel, FiltX)  # performing convolution 
with the defined X filter
        BY = ConvoleMatrix(Blue_Channel, FiltY)  # performing convolution 
with the defined Y filter
        for i in range(ROW):
            for j in range(COL):
                r_comp = mt.sqrt((RX[i][j] ** 2) + (RY[i][j] ** 2))  # 
sqaring the channel value
                g_comp = mt.sqrt((GX[i][j] ** 2) + (GY[i][j] ** 2))
                b_comp = mt.sqrt((BX[i][j] ** 2) + (BY[i][j] ** 2))
                grad[i][j] = mt.sqrt((r_comp ** 2) + (g_comp ** 2) + 
(b_comp ** 2))

    Gradiant_of_Image = grad

    for ColorCenter in centers:
        grad = 10231.0
        Center_Row = ColorCenter[0] - 1
        Center_Col = ColorCenter[1] - 1
        for i in range(Center_Row, Center_Row + 3):
            for j in range(Center_Col, Center_Col + 3):
                if (Gradiant_of_Image[i][j] < grad):
                    grad = Gradiant_of_Image[i][j]
                    New_Color_Row = i
```

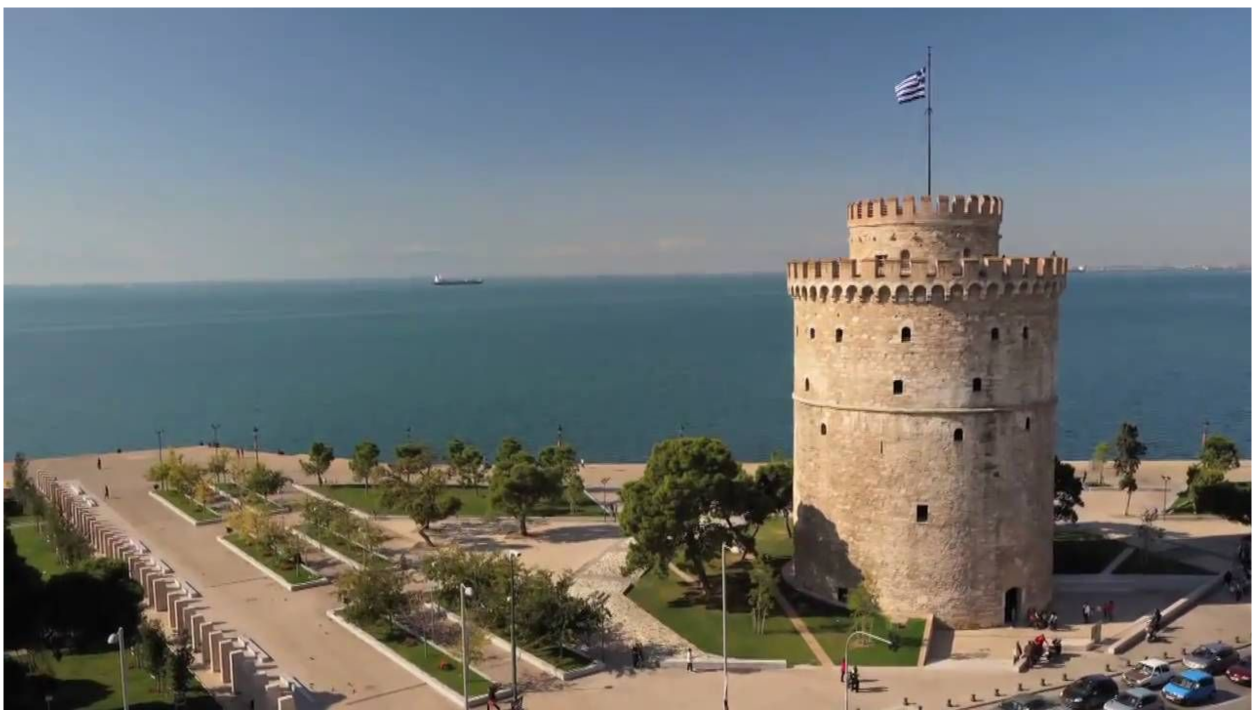
```
                    New_Color_Col = j
        ColorCenter[0] = New_Color_Row  # assigning the x coordinates
        ColorCenter[1] = New_Color_Col  # assigning the y coordinates
        ColorCenter[2] = Test_img[New_Color_Row][New_Color_Col][0]  #
assigning the corresponding RGB coordinates
        ColorCenter[3] = Test_img[New_Color_Row][New_Color_Col][1]
        ColorCenter[4] = Test_img[New_Color_Row][New_Color_Col][2]
    Convergance = 0
    Iterations = 1
    while (Convergance == 0):
        Convergance = 1
        clusters = [[] for i in range(len(centers))]
        for i in range(row):
            for j in range(col):
                dist = 10231.0
                # here we calculate the euclidian distance src:
https://en.wikipedia.org/wiki/Euclidean_distance
                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)))):
                        RED = int(Test_img[i][j][0]) - int(centers[k][0])
                        GREEN = int(Test_img[i][j][1]) - int(centers[k][3])
                        BLUE = int(Test_img[i][j][2]) - int(centers[k][4])
                        distance = float(mt.sqrt(
                            (((i / 2) - (centers[k][0] / 2)) ** 2) + (((j /
2) - (centers[k][1] / 2)) ** 2) + (
                                    RED ** 2) + (GREEN ** 2) + (BLUE **
2)))
                        CentralDistance = distance
                        if (CentralDistance < dist):
                            dist = CentralDistance
                            Index = k
                clusters[Index].append([i, j, Test_img[i][j][0],
Test_img[i][j][1], Test_img[i][j][2]])
        New_Centers = 0
        for i in range(len(clusters)):
            PointNumber = 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 (PointNumber != 0):
                r = r / PointNumber
                g = g / PointNumber
                b = b / PointNumber
                x = int(x / PointNumber)
                y = int(y / PointNumber)
            r = int(mt.sqrt(r))
            g = int(mt.sqrt(g))
            b = int(mt.sqrt(b))
            new_center = [x, y, r, g, b]  # assigning values for the new
central value
            old_center = [centers[i][0], centers[i][1], centers[i][2],
centers[i][3],
                          centers[i][4]]  # retreiving old central values

            # checking the difference between the old centers and the new
one
            if (abs(new_center[0] - old_center[0]) <= 0.000025 and abs(
                    new_center[1] - old_center[1]) <= 0.000025 and abs(
                new_center[2] - old_center[2]) <= 0.000025 and
abs(new_center[3] - old_center[3]) <= 0.000025 and abs(
                new_center[4] - old_center[4]) <= 0.000025):
                # if(new_center != old_center):
                New_Centers = New_Centers + 1
                centers[i][0] = x;
                centers[i][1] = y;
                centers[i][2] = r;
                centers[i][3] = g;
                centers[i][4] = b
                Convergance = 0
        print("DONE")
        print()
        Iterations = Iterations + 1
```

```
    Imgae = np.asarray(
        Image.new('RGB', (col, row)))  # creating a new RGB image with the same dimentions as the original image.
    #assigning gradiant image
    Imgae.setflags(write=1)
    for i in range(len(centers)):
        cent = centers[i]
        RGBval = [cent[2], cent[3], cent[4]]
        for j in clusters[i]:
            Imgae[j[0]][j[1]] = RGBval
    # here we write the new RGB value to the image, for K-means
    # this basically is used to reduce the size of the image, or compress the image. This is done by taking the major color centers k and relegating everything else

    Final_img = Imgae.copy()
    Final_img = np.asarray(Final_img)
    Final_img.setflags(write=1)
    Test_img = Imgae.copy()
    Test_img = np.asarray(Test_img)
    Test_img.setflags(write=1)
    row, col, x = Test_img.shape  # redrawing the separations in the image obtained from slic
    for i in range(1, row - 1):
        for j in range(1, col - 1):
            test = []
            test.append(
                str(Test_img[i - 1][j - 1][0]) + str(Test_img[i - 1][j - 1][1]) + str(Test_img[i - 1][j - 1][2]))
            test.append(str(Test_img[i - 1][j][0]) + str(Test_img[i - 1][j][1]) + str(Test_img[i - 1][j][2]))
            test.append(
                str(Test_img[i - 1][j + 1][0]) + str(Test_img[i - 1][j + 1][1]) + str(Test_img[i - 1][j + 1][2]))
            test.append(str(Test_img[i][j - 1][0]) + str(Test_img[i][j - 1][1]) + str(Test_img[i][j - 1][2]))
            test.append(str(Test_img[i][j][0]) + str(Test_img[i][j][1]) + str(Test_img[i][j][2]))
            test.append(str(Test_img[i][j + 1][0]) + str(Test_img[i][j + 1][1]) + str(Test_img[i][j + 1][2]))
            test.append(
                str(Test_img[i + 1][j - 1][0]) + str(Test_img[i + 1][j -
```

```
1][1]) + str(Test_img[i + 1][j - 1][2]))
            test.append(str(Test_img[i + 1][j][0]) + str(Test_img[i +
1][j][1]) + str(Test_img[i + 1][j][2]))
            test.append(
                str(Test_img[i + 1][j + 1][0]) + str(Test_img[i + 1][j +
1][1]) + str(Test_img[i + 1][j + 1][2]))

    Final_img = Image.fromarray(Final_img)
    Final_img.show()


main()
```

Original Image:

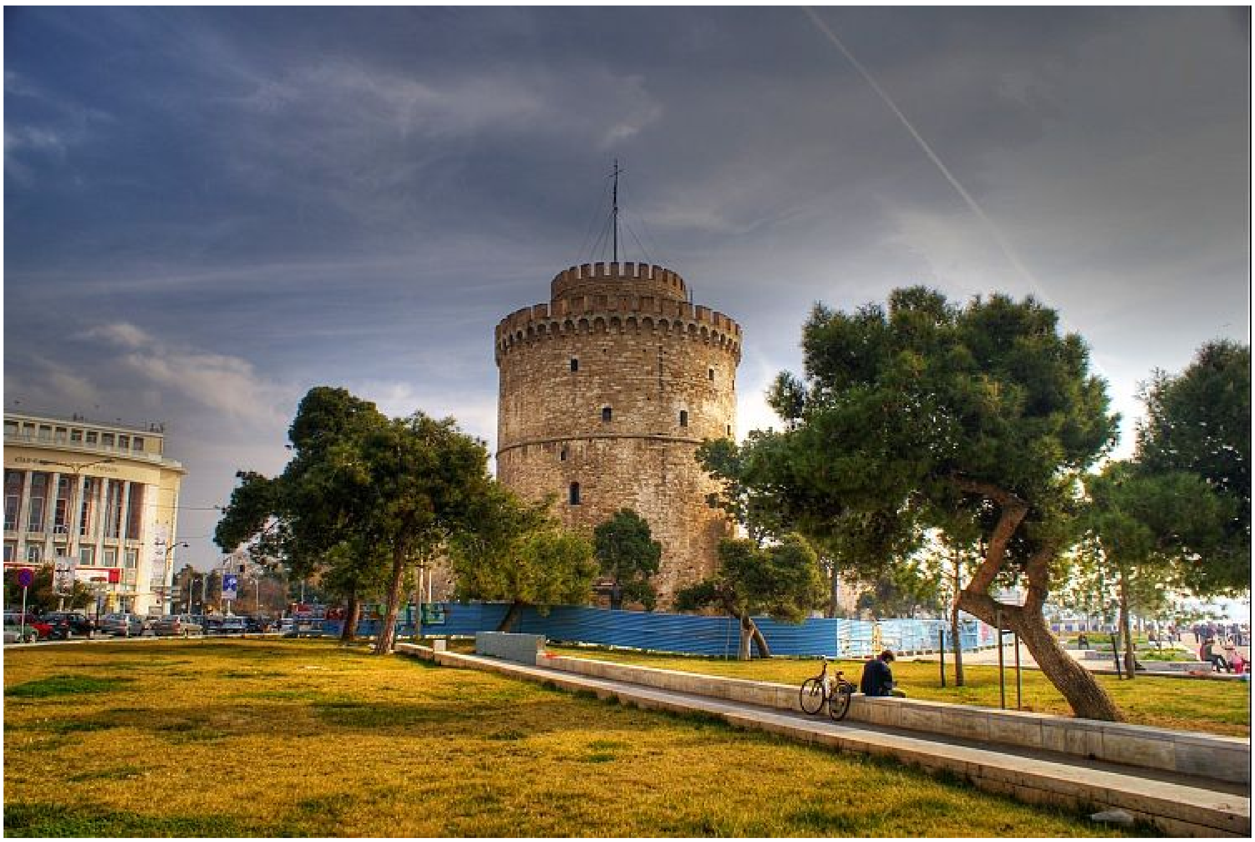


In K means sometimes the random numbers may generate empy clusters , in which case the algorithm will take a lot longer to run,
If there are all non zero clusters, generated then it takes shorter to run .
Eitherway the estimated runtime is 15 mins or more .
THe k Means algorithm is used to compress the images as they break the image down into its constituent colors. In this case 10 majour colors.

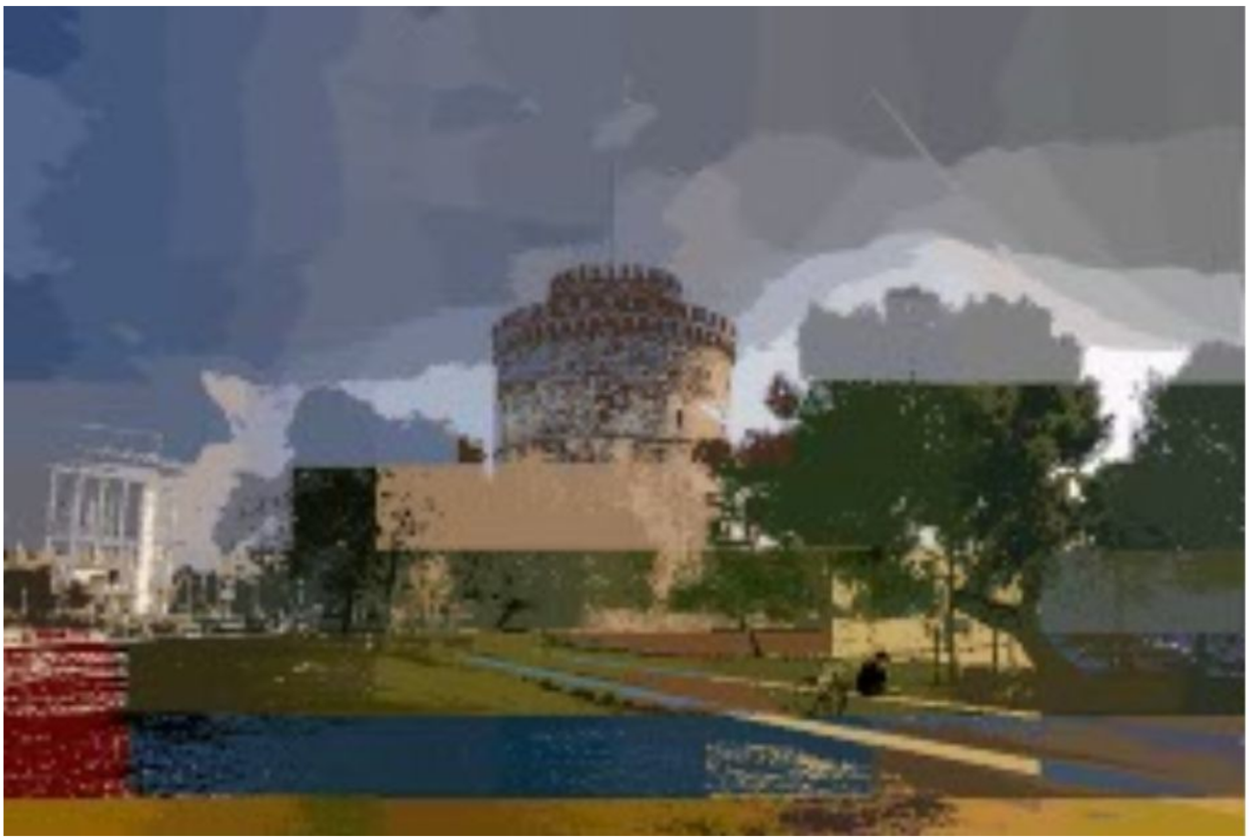
The Output for K-Means :

Now For Slic Algorithm

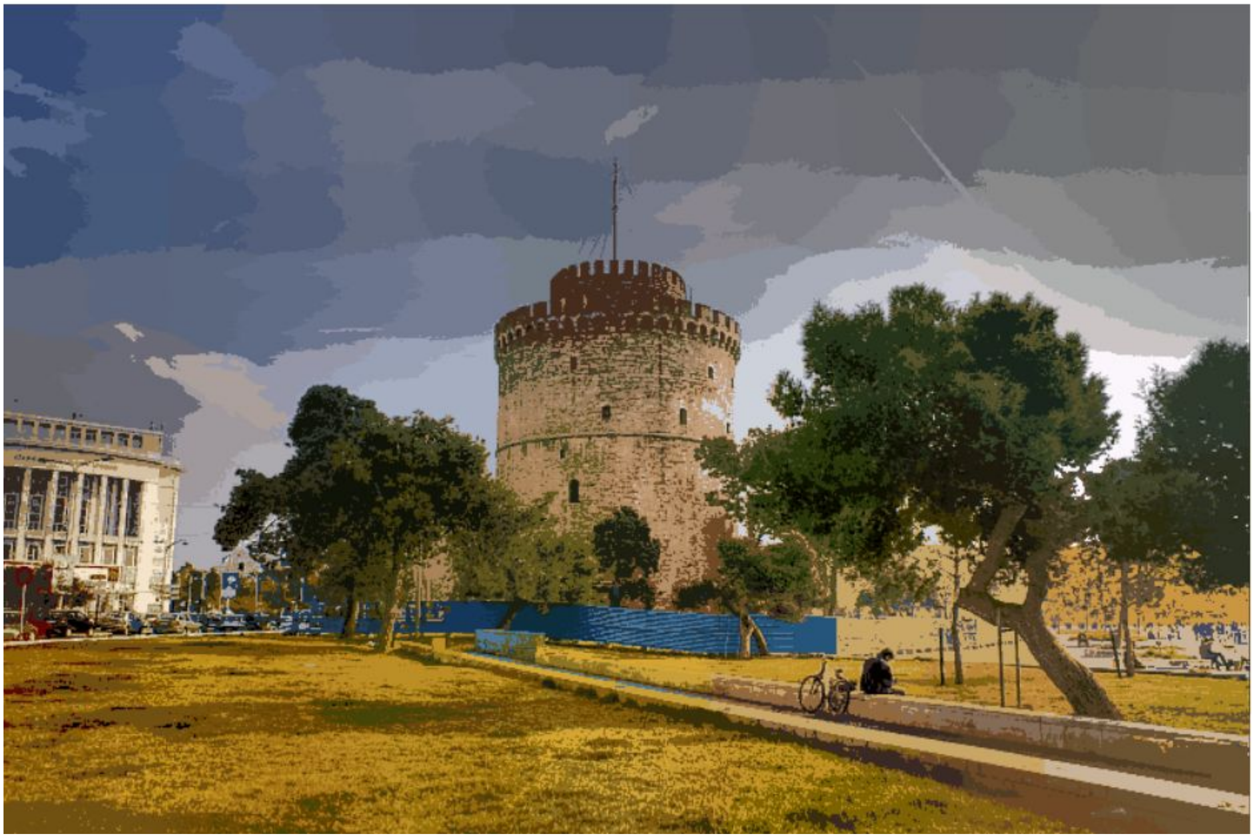
Original Image



Output Image



If we play arround with the variables we get a more defined image for



Now here in slic we dont check if Old Centeres are exactly Different from the new centers , because this can take upto 4 hours to run , this way hence we take the absolute difference between the [x,y,r,g,b] values and check if they are infintecimally small so we can actually improve the run time of this program .
This program doesnt check if Old Centers == New Centers hence it takes almost as long if not as long as the aforementioned K-Means Algorithm