Computer Vision - CSE344
Assignment 1
Report

Name: Ansh Arora
Roll No: 2019022

Problem 1

Part 1
We have to implement Equation 3 of the paper. For this, we first of all take the leaf image and perform k-means on it to cluster it into the optimal k colours. For this, we make use of OpenCV's inbuilt kmeans implementation. On performing k-means, the leaf's image comes out to be as following –

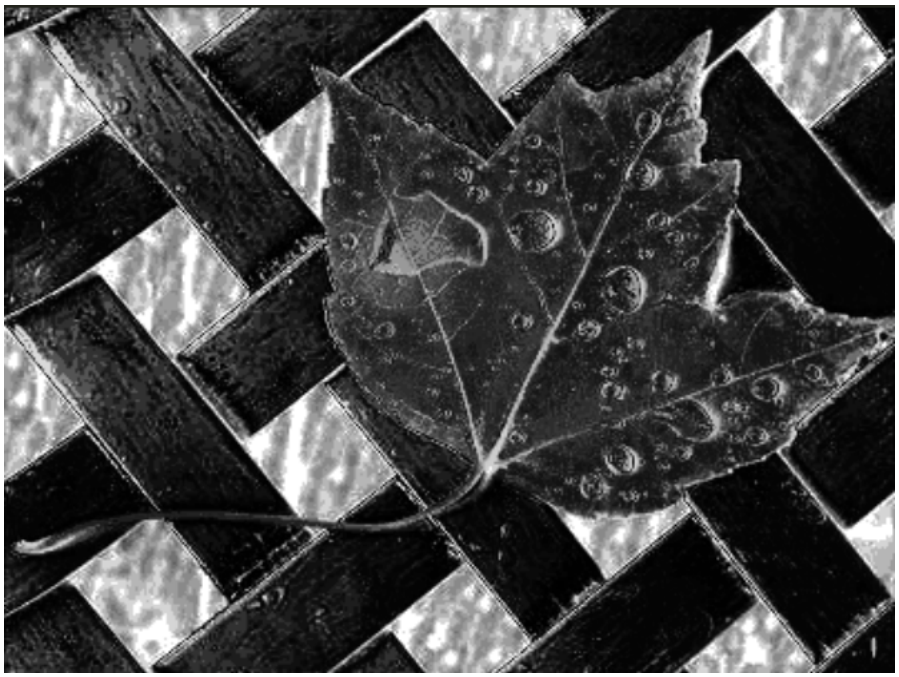
Now that once we have performed k-means, we want to find out the saliency values for each pixel in the image. For this particular equation, since we make use of histogram based contrast, the position of a pixel doesn't play a role in deciding the saliency, just the colour pixel value does. The saliency for a pixel can be calculated by using the following formula –

$$S(I_k) = \sum_{\forall I_i \in I} D(I_k, I_i),$$

which can be reduced into the following form by using the property that each colour combination would contribute equally to the saliency for one colour (and hence the particular pixel exhibiting that colour) –

$$S(I_k) = S(c_l) = \sum_{j=1}^{n} f_j D(c_l, c_j),$$

To implement this, I have first found out the frequency probability (f_j) of each colour in the image and stored it in the color_freq array. The color_freq array is indexed by specific color codes that we have given to each of the 85 colors. The color_freq is simply calculated by counting the instances of each color in the image.
Now for calculating the distance metric for color given by D(cl,cj), I make use of a dictionary where each pair of color distances is stored between each of the 85 colors.
After this, we just simply calculate the saliency for each of the 85 colors by summing over their distances from each colour multiplied by the frequency of that particular colour.
Finally, we allot all the pixels the saliency values of their colors and formulate the image as the saliency map. The final saliency map comes out to be as follows –



Part 2

We have to implement Equation 5 of the paper for this question. For this, we first of all take the leaf image and perform k-means on it to cluster it into the optimal k colours. For this, we make use of OpenCV's inbuilt kmeans implementation. On performing k-means, the leaf's image comes out to be as following –

Now to perform Region Contrast, we need to segment the image into regions. For this we need to segment the image into regions. We use the PEGBIS algortihm for the same and generate the following 5 segment region image –

Now, to perform region contrast on the image, we need to calculate the saliency value of each region. This has to be done using the formula –

$$S(r_k) = \sum_{r_k \neq r_i} w(r_i) D_r(r_k, r_i),$$

Where w(ri) gives the weight of each region (Number of pixels lying in that region divided by the total number of pixels) and Dr gives the Region Distance. This can be calculated using the following formula –

$$D_r(r_1, r_2) = \sum_{i=1}^{n_1} \sum_{j=1}^{n_2} f(c_{1,i}) f(c_{2,j}) D(c_{1,i}, c_{2,j})$$

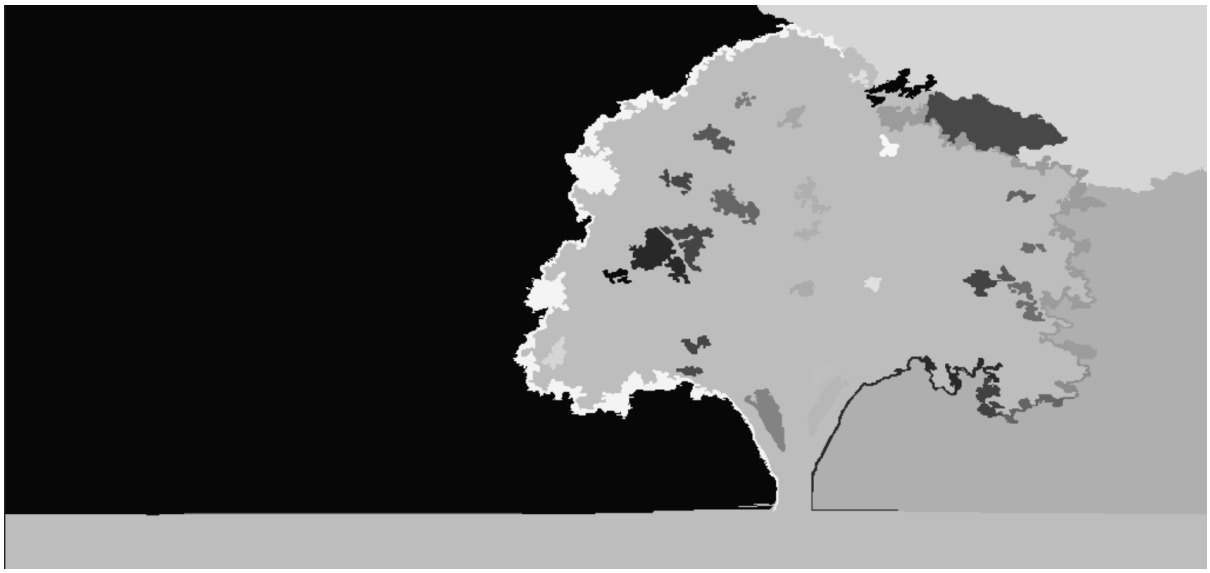
In my implementation, to calculate the region and color's codes I first of all form sets and then index the sets into lists to give each region and each of the 85 colors an index value. Then I calculate the distance metric between each of the 85 colors in the same manner as done in part 1, and store the values into a dictionary. After this, I calculate the frequency color histogram for each region. This is done by first checking the region code from the region image, the color code from the color k-means image and then finally adding the values to the 2-Dimensional Dictionary of the form – regions_freq[region_code][color_code]. The frequency probability for each color in each region is found by dividing the above value by the number of pixels in that particular region. Finally, we create a region distance dictionary by applying a 4-level loop implementing the above given equation as follows –

```
region_dist = {}
for i in tqdm(range(len(region_color_list))):
    for j in range(len(region_color_list)):
        if i==j:
            region_dist[(i,i)] = 0
        else:
            region_dist[(i,j)] = 0
            for k in range(len(color_list)):
                for l in range(len(color_list)):
                    region_dist[(i,j)] += color_dist[(k,l)]*regions_freq[i][k]*regions_freq[j][l]
```

After that, I finally applied equation 5 my adding the product of weights of different regions and their distance from the region in question. In the end, the saliency map is created by alloting each region their saliency values. The result is as follows –

Making use of a 35 segment image, the results were as follows –

Segmented Image –



Saliency Map –

Problem 2

In this problem, we were asked to implement a modification of the Otsu Algorithm by replacing the weight variances with a simple sum of TSS (Total Sum of Squares). To implement this, the image was first of all read in as a grayscale (since Otsu operates on grayscale images only). Now to identify the threshold value, we need to find out the minimum of all the TSS values calculated. For this, we loop over from 0 to 255 (the possible threshold values), dividing the image pixel intensities into two sets – the one from 0 to the threshold value and the one from threshold value+1 to 255.

Now to find out the TSS, we need to first calculate the mean values of the pixels in both of the sets, which is done by a simple mean value calculator function.

```
def mean_calc(X,add=0):
    mean = 0
    count = 0
    for i in range(len(X)):
        mean = mean + X[i]*(i+add)
        count = count + X[i]
    mean = mean/count
    return int(mean)
```

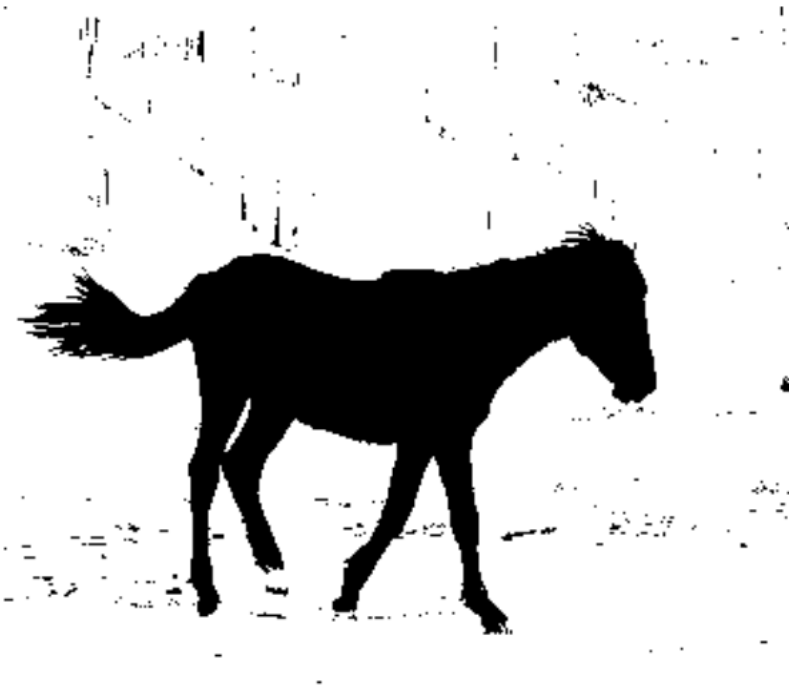
In the implementation I had to take an add variable to add the threshold value in consideration since in my set breaking implementation I make use of array slicing, which subtracts the value each pixel intensity holding index by the threshold value being considered. Similary, the TSS is then caculated for both the classes by using the mean calculated by the following function –

```
def sum_of_sqaures(X,mean,add=0):
    sum = 0
    for i in range(len(X)):
        sum = sum + X[i]*(((i+add)-mean)**2)
    return sum[0]
```

The sum of squares for both the classes are then added and stored in the TSS array.

Finally, the minimum of the TSS array is found, which forms the threshold for our binary mask. All values below the threshold are stored as 0 and all above threshold are stored as 1 (255). The TSS array along with the threshold values are stored in a CSV file.

The final binary mask generated for the horse image is as follows –



```
1.14000000000000000e+02,9.34629280000000000e+07
1.15000000000000000e+02,9.34050240000000000e+07
1.16000000000000000e+02,9.33809600000000000e+07
1.17000000000000000e+02,9.33534960000000000e+07
1.18000000000000000e+02,9.33637760000000000e+07
1.19000000000000000e+02,9.34131440000000000e+07
1.20000000000000000e+02,9.34458800000000000e+07
1.21000000000000000e+02,9.35407520000000000e+07
```

The global minima observed at a pixel intensity value of 117.

Problem 3

In this problem, we had to generate video from the given video such that the new video had a tight bounding circle around the person.

To solve this problem, we first divided the video into frames and stored them using OpenCV's VideoCapture method. Now from these frames, the median image was found (as asked in the question). It came out to be as follows –

Now, we perform the method of background subtraction, where we find the absolute difference of each frame with the above found median image. These images came out to be of this form –

Now we threshold the image to remove any discrepancy values so that we can formulate a tighter mor exact bound. We do this by finding the Otsu Threshold for each of the absolute difference frames generated above using OpenCV's inbuilt threshold function. We the apply the threshold on the above frames and generate binary masks that will now finally be used to find the ROIs in each frame.

The ROI is found by finding the min and max x and y coordinate values which have a 1 value in the above thresholded images. This is done by using a simple loop. Finally, we find the centre of the circle and the radius of the circle from the above identified x and y coordinate values and draw a circle to the original video frames using these values using OpenCV's Circle function. The result comes out to be as follows –

The circled frames are then stitched together and the resultant video is generated.