ECE 661: Homework 6

Mridul Gupta
Email: gupta431@purdue.edu

Theory Question

1. Otsu thresholding works really well for images with bimodal histograms. It is very fast method because it just finds one threshold for an entire image but that also makes it very noisy. In iterative Otsu, we will get better result with each iteration only if the foreground image from previous iteration also has bimodal histogram.

2. This method requires good markers. If the markers are not good, the method doesn't work very well. Overall, its a less noisy process than otsu thresholding. Due to oversegmentation, extracted objects might not be good but variations of the algorithm have been proposed to reduce oversegmentation.

Section 3

1. Otsu's Algorithm
   It divides the image into 2 classes , foreground and background. w_0, w_1 are the probabilities of the two classes. Histogram frequencies are normalized and are represented by p_i for gray value i. μ_0 is the mean of class 0 and μ_1 is the mean of class 1. Between-class variance is given by $\sigma_b^2 = w_0 w_1 (\mu_0 - \mu_1)^2$. We have to maximize this between-class variance to get the threshold value. We consider each gray pixel value i as a threshold, n_i represents the number of pixels with gray value i and N is the total number of pixels.

$$w_0 = \Sigma_{k=0}^{i-1} p_i = \Sigma_{k=0}^{k=i-1} \frac{n_i}{N}$$
$$w_1 = 1 - w_0$$
$$\mu_0 = \Sigma_{k=0}^{i-1} \frac{i p_i}{w_0} = \frac{\Sigma_{k=0}^{i-1} i n_i}{\Sigma_{k=0}^{i-1} n_i}$$
$$\mu_1 = \frac{\mu_T - w_0 \mu_0}{w_1}$$

   where μ_T is the mean over the entire image. Over all possible thresholds, we keep track of the threshold which maximizes the between-class variance.

2. Pixel-value based segmentation
   We run Otsu's method on each channel separately. We have the option to run Otsu' method multiple times while keeping track if the foreground class is above or below the computed threshold.

   (a) For cat image, only the Red channel of the image was used for thresholding and the foreground was the class with higher pixel values and only 1 iteration was run.

   (b) For pigeon image, all channels of the image were used for thresholding and the foreground was the class with higher pixel values and only 1 iteration was run for each channel.

   (c) For fox image, only blue channel of the image was used for thresholding and the foreground was the class with higher pixel values and only 2 iterations were run for each channel.

3. Texture-based segmentation
   There are multiple was of calculating texture value for a pixel but we use the sliding window approach. We use three window sizes for each image. For each window size, we calculate the variance of neighbors of a pixel in that window. We treat the variance values at each pixel location as a feature value for that pixel and run Otsu thresholding on this new variance image.

   (a) For cat image, only the Red channel of the image was used for thresholding and the foreground was the class with lower variance values and 6 iterations were run. window sizes were 3,5 and 7 and the final target mask was inverted.

   (b) For pigeon image, all channels (RGB) of the image were used for thresholding and the foreground was the class with lower variance values and 6, 3 and 3 iterations were run respectively. window sizes were 7, 9 and 11 and the final target mask was not inverted.

   (c) For fox image, all channels (RGB) of the image were used for thresholding and the foreground was the class with lower variance values and 1, 1 and 3 iterations were run respectively. window sizes were 3, 5 and 7 and the final target mask was inverted.

4. Contour Extraction
   Contours are basically formed by the pixels that form the edges. Dilation and erosion were performed to connect missing edges and remove small dots from the image respectively. A pixel forms an edge if out of its 9 neighbors, at least 1 pixel belongs to the background i.e. it has the value 0.

   (a) For RGB segmentation based foreground mask:
      i. Cat image - eroded then dilated once with kernel of size 3×3.
      ii. Pigeon image - eroded then dilated once with kernel of size 3×3.
      iii. Fox image - dilated then eroded 5 and 6 times respectively with kernel of size 3×3.

   (b) In cat image, foreground mask was eroded - 3 iterations, with kernel size of 3×3. It was then dilated - 6 iterations with the same kernel size.

   (c) Other images were neither eroded not dilated.

Results

Figure 1: RGB - Cat channel 1 foreground and the final foreground mask

Figure 2: RGB - Cat foreground

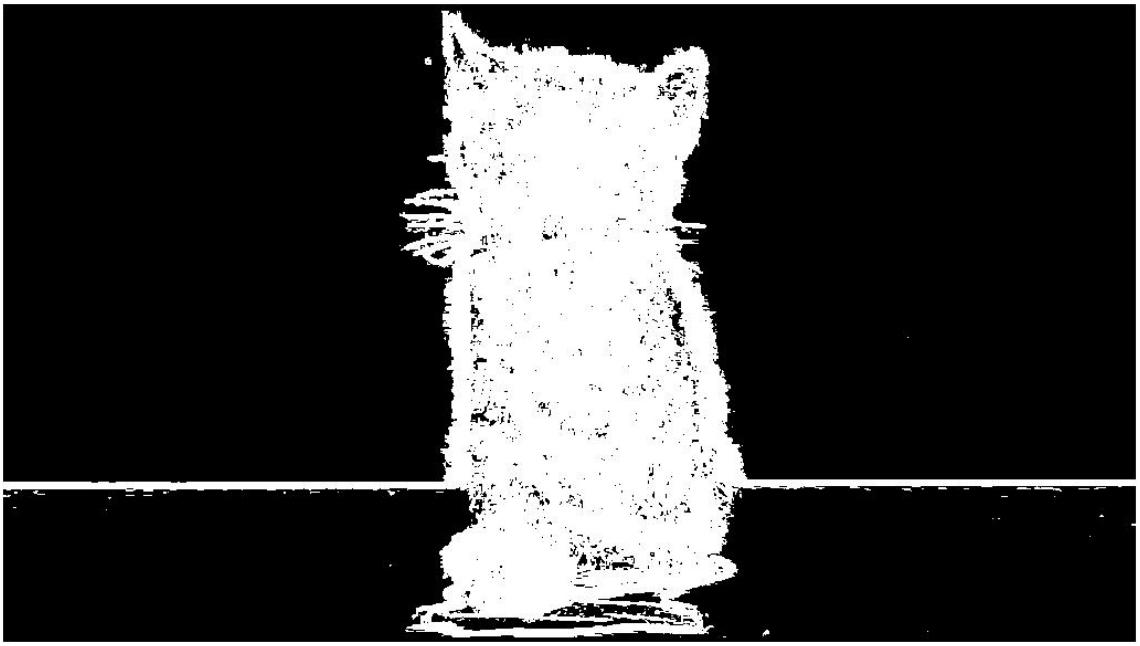

Figure 3: Texture - Cat foreground mask

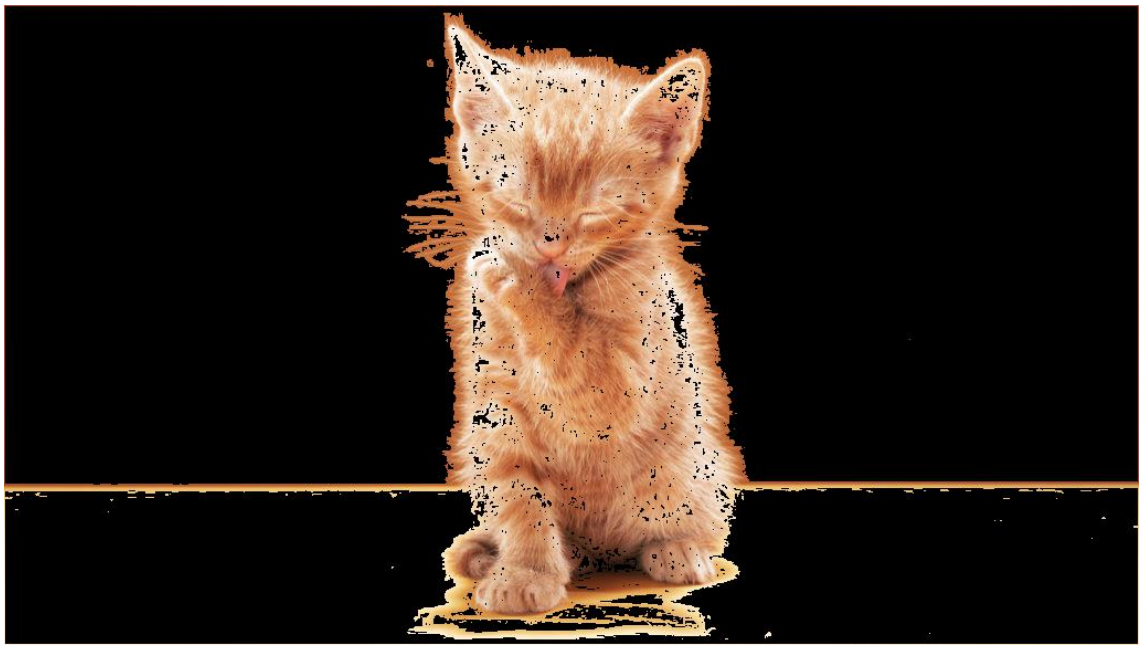

Figure 4: Texture - Cat foreground

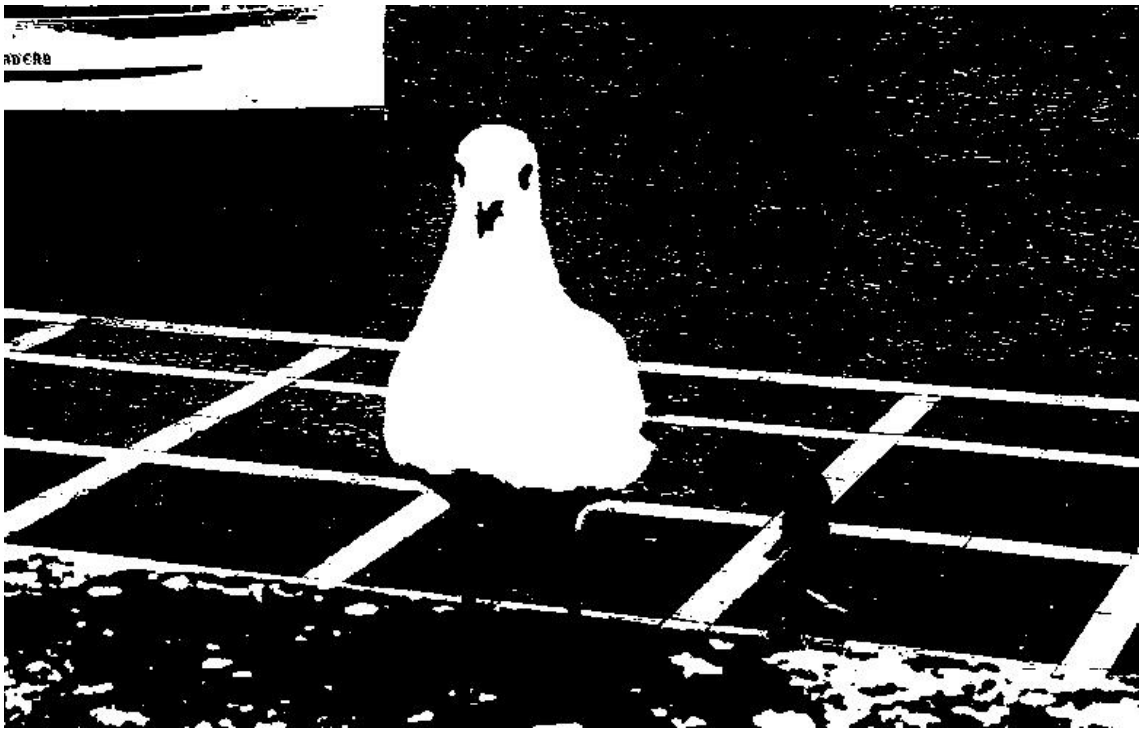

Figure 5: RGB - Pigeon channel 1 foreground mask

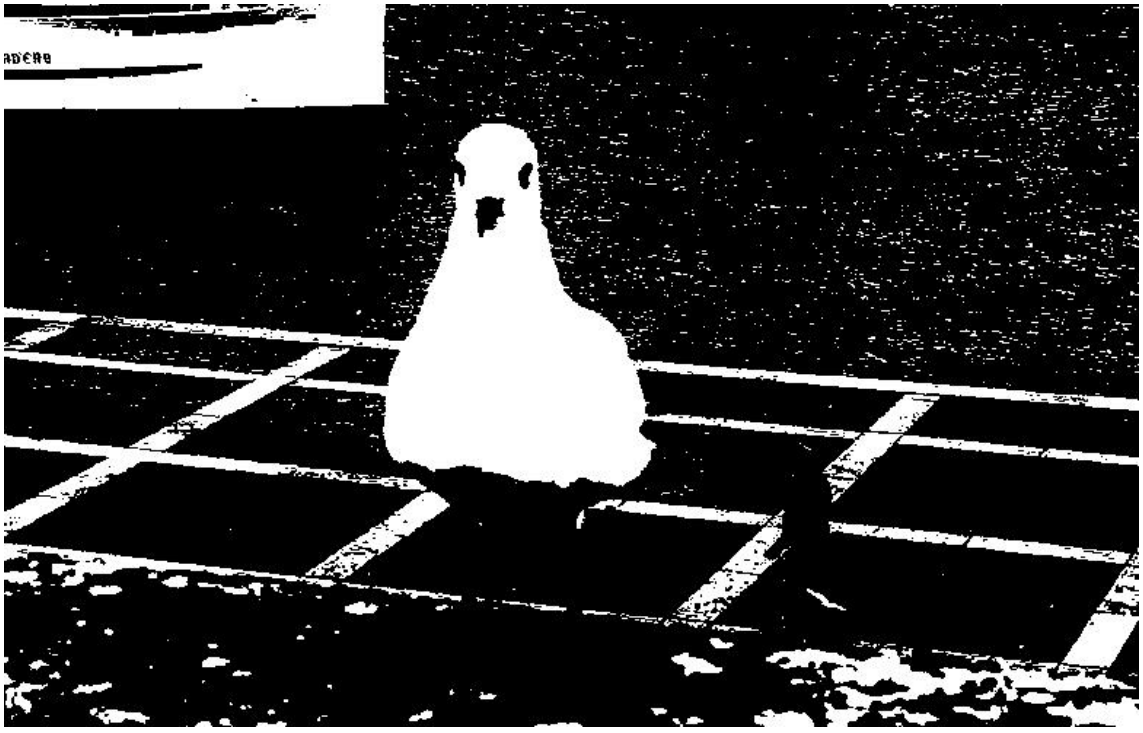

Figure 6: RGB - Pigeon channel 2 foreground mask

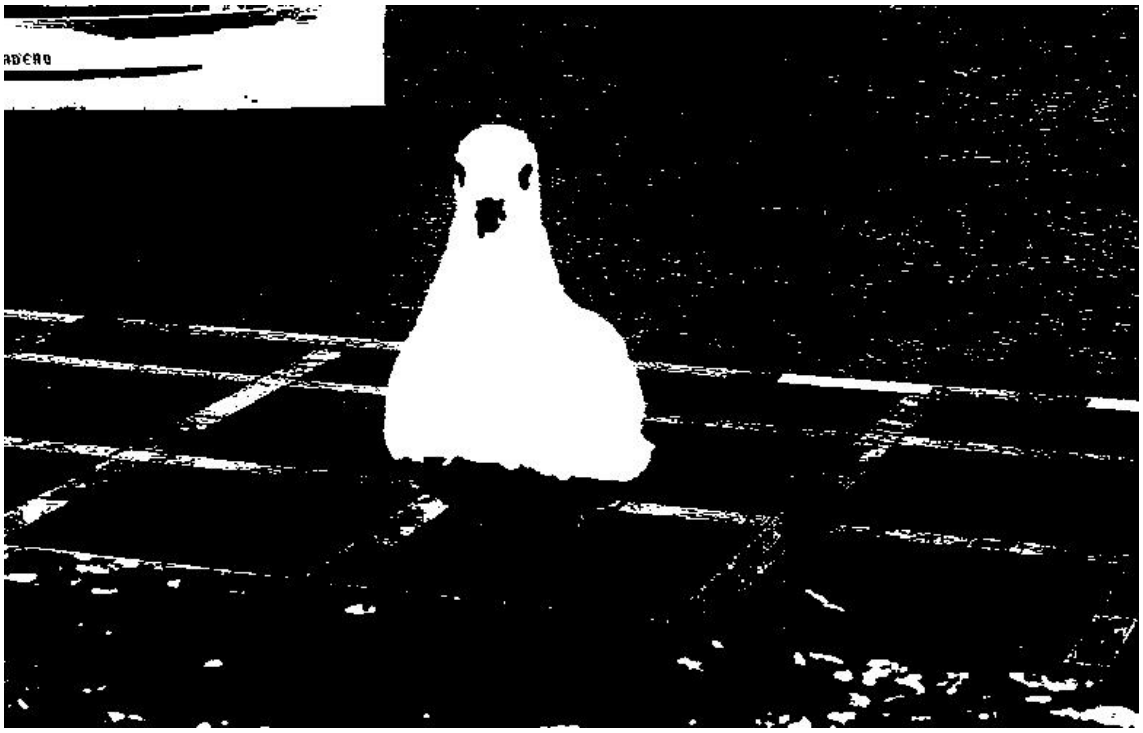

Figure 7: RGB - Pigeon channel 3 foreground mask

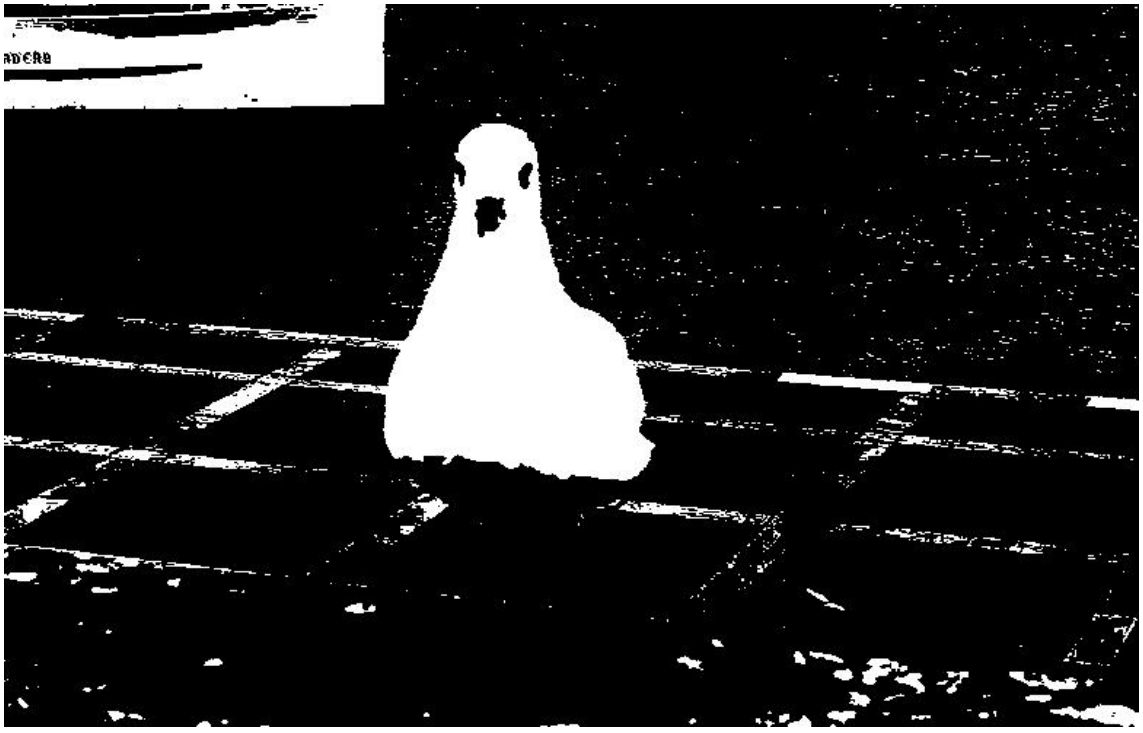

Figure 8: RGB - Pigeon final foreground mask

Figure 9: RGB - Pigeon final foreground

Figure 10: Texture - Pigeon foreground mask

Figure 11: Texture - Pigeon foreground

Figure 12: RGB - Fox channel 3 foreground and the final foreground mask

Figure 13: RGB - Fox foreground

Figure 14: Texture - Fox foreground mask

Figure 15: Texture - Fox foreground

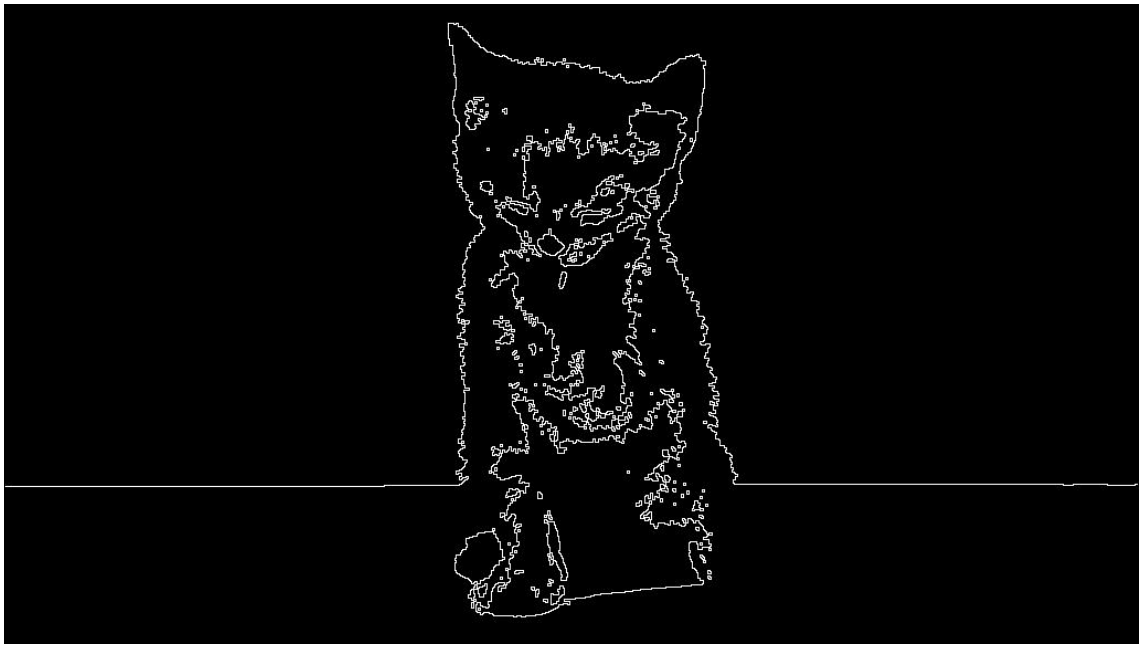

Figure 16: RGB - Cat Contour

Figure 17: Texture - Cat Contour

Figure 18: RGB - Pigeon Contour

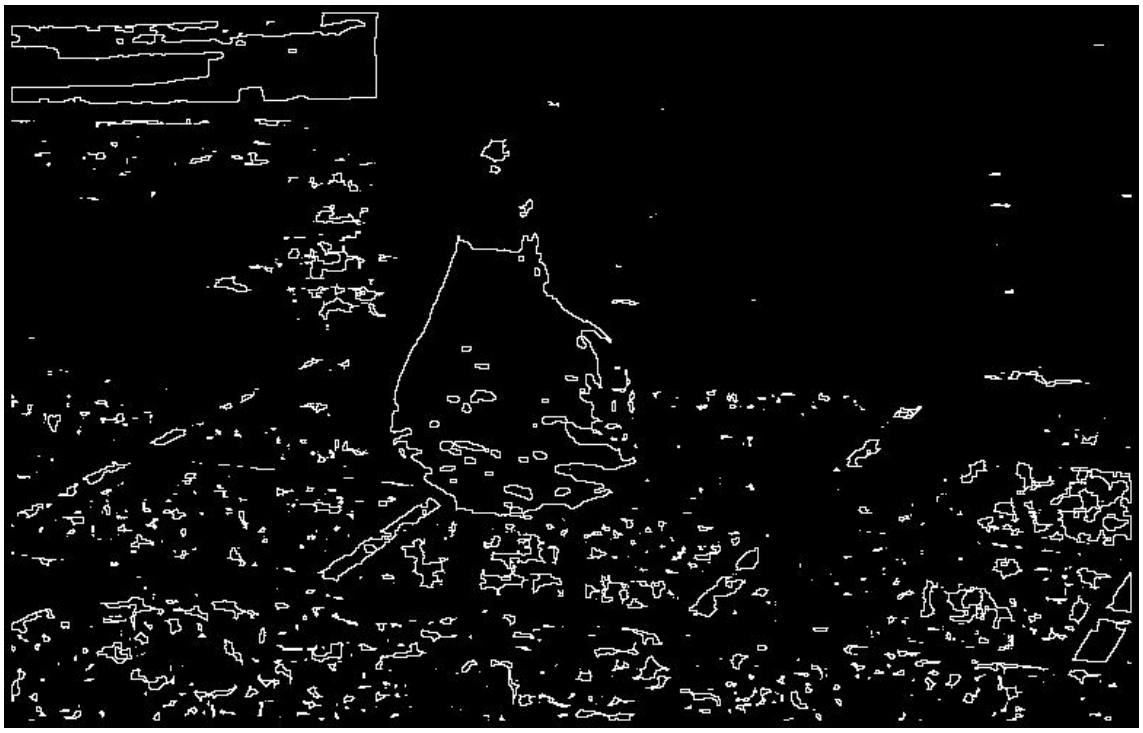

Figure 19: Texture - Pigeon Contour

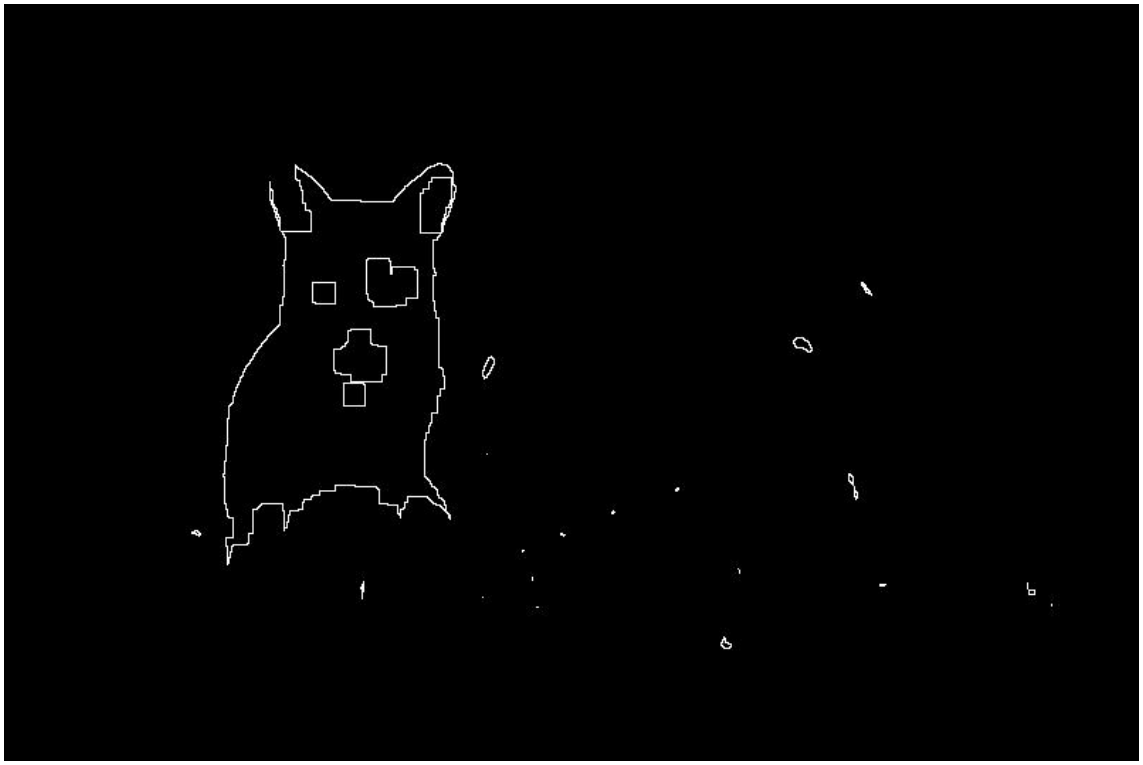

Figure 20: RGB - Fox Contour

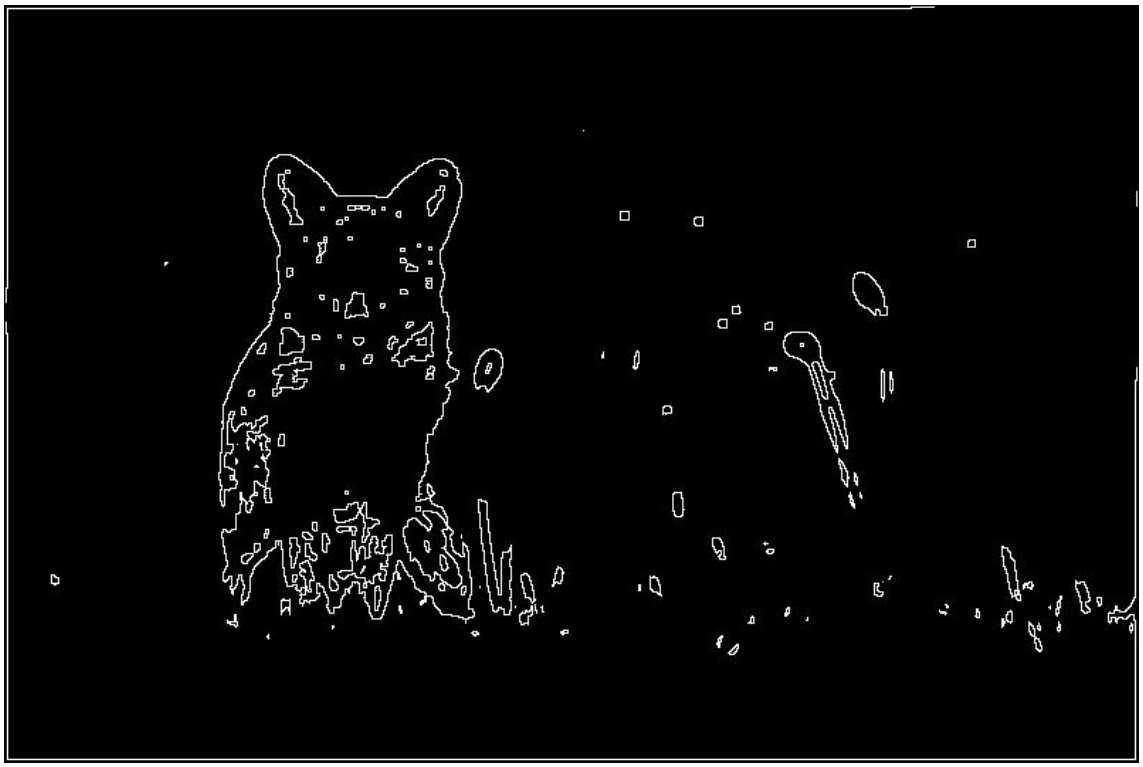

Figure 21: Texture - Fox Contour

Observation

1. For the cat image, table has a color very close to the cat, so RGB-based segmentation does not work very well and texture based segmentation does really well.

2. For pigeon image, RGB based method is not able to separate pigeon from the boat because they have the same color but does much better than texture based method. Texture based method is able to keep pigeon in the foreground but there are quite a lot of false positives especially the ground tiles which have slightly different color than the pigeon.

3. For fox image, both methods did quite well, but RGB based method missed some portion of the legs of the fox whereas texture based method did not. With other post-processing techniques, it might have been possible to make the foreground mask better.

4. I think both the methods perform really well overall but texture based method probably has an edge over RGB based method since it can deal with similar colors comparitively easily.

```
1 ### ECE 661 - HW 6
2 ### Otsu thresholding
3
4 import numpy as np
5 import cv2
6 from matplotlib import pyplot as plt
7 import copy
8
9
```

```
10
11
12  def otsu_threshold(img,iterations = 1,invert=0):
13      ### img is grayscale image, for RGB, send one channel at at time
14      ###returns mask for the image
15      output_mask = np.ones(img.shape)
16      if(len(img.shape)==3):
17          print("Error- " + str(img.shape[2])+" channel image is the
                input to otsu threshold method")
18          return -1
19
20      for i in range(iterations):
21          cur_for = img[output_mask!=0]
22          hist,bins = np.histogram(cur_for,bins= np.unique(cur_for))
23
24          ### normalize histogram
25
26          hist = hist/float(len(cur_for))
27      #       print(np.sum(hist))
28      #       print(bins)
29
30          ##define variables for calculation of threshold
31          w0 =0
32          mu0= 0
33          total_weighted_avg = np.mean(cur_for)
34          otsu_t=-1
35          max_sigmab2 = 0
36          sum1=0
37
38      #       ##sigmaw2 + sigmab2 = constant so we dont need to
            calculate sigmaw2
39          count=-1
40          for th in bins[:-2]:
41              count+=1
42              w0+=hist[count]
43              w1 = 1-w0
44              sum1 += th*hist[count]
45              mu0=sum1/w0
46              if w1==0:
47                  break
48              mu1 = (total_weighted_avg-sum1)/w1
49              sigmab2 = w0*w1*np.square(mu0-mu1)
50              if sigmab2>max_sigmab2:
51                  max_sigmab2 = sigmab2
52                  otsu_t = th
53
54          print(otsu_t)
55
```

```
56          if invert==0:
57              output_mask = img>otsu_t
58          else:
59              output_mask = img<otsu_t
60      return (output_mask!=0)
61
62
63  # In[149]:
64
65
66  def get_variance(img,k):
67      ##avoids nested loops for variance calculation
68      k = int(k/2)
69      img_new = np.zeros((img.shape[0]+2*k,img.shape[1]+2*k))
70      img_new[k:img.shape[0]+k,k:img.shape[1]+k] = img
71      neighbours = []
72      for i in range(k,img.shape[0]+k):
73              for j in range(k,img.shape[1]+k):
74                  neighbours.append(img_new[i-k:i+k+1,j-k:j+k+1].
                        flatten())
75      neighbours = np.array(neighbours)
76      print(neighbours.shape)
77      variances = np.int32(np.var(np.array(neighbours,dtype = np.
            float32),axis=1))
78
79      variances = variances.reshape(img.shape[0],img.shape[1])
80
81      print(variances.shape)
82      return variances
83
84
85  # In[128]:
86
87
88  def texture_otsu(img,N=[3,5,7],iterations=[1,1,1],invert = [0,0,0]):
89      ### function for texture based segemenation
90      gray = cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
91
92      ###place a window of N*N at each pixel location
93      masks = np.zeros(im.shape)
94      for i in range(len(N)):
95          ##this is the new channel values
96          var = get_variance(gray,N[i])
97
98          temp = otsu_threshold(var.flatten(),iterations[i], invert[i
                ])
99          masks[:,:,i] = temp.reshape((img.shape[0],img.shape[1]))
100     ##product of the three masks
```

```
    final_mask = np.multiply(np.multiply(masks[:,:,0],masks[:,:,1]),
       masks[:,:,2])
    final_mask = final_mask!=0
    return final_mask



# In[75]:


def get_contour(mask):
    ## checks if any neighbour is not a part of the foreground to
       compute contours
    contour = np.zeros(mask.shape)
    for i in range(1,mask.shape[0]-1):
        for j in range(1,mask.shape[1]-1):
            if mask[i][j]==0:
                continue
            if np.sum(mask[i-1:i+2,j-1:j+2])<9:
                contour[i][j]=1
    return contour


# In[361]:

###Parameters for the three images

##cat image
# iterations=[1,0,0]
# invert = [0,0,0]

##pigeon image
iterations=[1,1,1]
invert = [0,0,0]

##fox image
# iterations=[0,0,2]
# invert = [0,0,0]

name= 'pigeon'
im = np.uint8(cv2.imread(name+'.jpg')[...,::-1])

plt.imshow(im)
masks = np.zeros(im.shape)
for i in range(3):

    im1 = copy.deepcopy(im[:,:,i])

```

```
    temp = otsu_threshold(im1.flatten(),iterations[i],invert[i])
    masks[:,:,i] = temp.reshape((im.shape[0],im.shape[1]))



# In[362]:


for i in range(1,4):
    # plt.imshow(masks[:,:,2],cmap='gray')
    plt.imsave(name+'foreground_mask_channel'+str(i)+'.jpg',masks
       [:,:,i-1],cmap='gray')


foreground = np.multiply(np.multiply(masks[:,:,0],masks[:,:,1]),
   masks[:,:,2])
plt.imshow(foreground,cmap='gray')
plt.imsave(name+'foreground_mask.jpg',foreground,cmap='gray')

##save masked image
fore_img = np.zeros(im.shape)
for i in range(3):
    fore_img[:,:,i] = np.multiply(im[:,:,i],foreground)
plt.imsave(name+'foreground.jpg',np.uint8(fore_img))


# In[365]:


##so we have a foreground mask at this point
## we will erode and dilate the mask to get rid of single pixel
   peaks and make image smooth

##cat erode - dilate 1
##pigeon erode - dilate 1
## fox dil5  -erode6
kernel = np.ones((3,3))
ero_fore = cv2.erode(np.float32(foreground),kernel,iterations =1)
dil_fore = cv2.dilate(ero_fore,kernel,iterations =1)

plt.imshow(ero_fore,cmap='gray')


# In[366]:


##now we will extract the contours
```

```
cnt = get_contour(ero_fore)
plt.imshow(cnt,cmap='gray')
plt.imsave(name+'contour_RGB.jpg',cnt,cmap='gray')


# In[367]:


##now texture based otsu

###Parameters for the three images

##cat image
# iterations=[6,0,0]
# invert = [1,1,1]
# N=[3,5,7]
# invert_mask=1

##pigeon image
iterations=[6,3,3]
invert = [1,1,1]
N=[7,9,11]
invert_mask=0

##fox image
# iterations=[1,1,3]
# invert = [1,1,1]
# N=[3,5,7]
# invert_mask=1

im1 = copy.deepcopy(im)

temp = texture_otsu(im1,N,iterations,invert)
mask = temp.reshape((im.shape[0],im.shape[1]))
if invert_mask==1:
    mask=(mask==0)
plt.imshow(mask,cmap='gray')


# In[368]:


plt.imsave(name+'texture_foreground_mask.jpg',mask,cmap='gray')
fore_img = np.zeros(im.shape)
for i in range(3):
    fore_img[:,:,i] = np.multiply(im[:,:,i],mask)
plt.imsave(name+'texture_foreground.jpg',np.uint8(fore_img))

```

```
240
241 # In[369]:
242
243
244 ##so we have a foreground mask at this point
245 ## we will erode and dilate the mask to get rid of single pixel
       peaks and make image smooth
246 ##cat dilation  = 5, erosion = 1, kernel = 3*3
247 ##pegion nothing
248 ##fox nothing
249
250 kernel1 = np.ones((5,5))
251 kernel2 = np.ones((3,3))
252
253 dil_fore = cv2.dilate(np.float32(mask),kernel2,iterations =5)
254 ero_fore = cv2.erode(np.float32(dil_fore),kernel1,iterations = 1)
255
256 plt.imshow(dil_fore,cmap='gray')
257
258
259 # In[370]:
260
261
262 ##now we will extract the contours
263 cnt = get_contour(dil_fore)
264 plt.imshow(cnt,cmap='gray')
265 plt.imsave(name+'contour_texture.jpg',cnt,cmap='gray')
```