Pattern recognition

Assignment #2

Name	ID
مصطفى أحمد كمال أحمد مصباح	18011777
زياد بديع السعيد البنا	18010717
عمرو ابراهيم احمد ابراهيم	18011169

Overview:

Pictures segmentations using K-means clustering and normalized cut clustering techniques

Download dataset:

We used Berkley segmentation benchmark dataset in this assignment, the dataset contains 500 pictures that are splitted into 200 tests, 200 trains and 100 validate, the analysis was done and 50 pictures of the test set.

Visualize images and their ground truth segmentation:

The images and their corresponding ground truth(s) are read and converted into a matrix, each pixel in the image matrix is represented by 3 values (RGB) that represents the color of the pixel, the RGB values are in range (0-->255) each.

- *Reading data and converting them to matrix:*

```
def loadData(imagePath , groundTruthPath):
    images = []
    groundTruths = []
    imageFiles = sorted(os.listdir(imagePath))
    if "Thumbs.db" in imageFiles : imageFiles.remove("Thumbs.db")
    groundTruthFiles = sorted(os.listdir(groundTruthPath))
    numOfImages = len(imageFiles)
    for i in range(numOfImages):
        imgPath = os.path.join(imagePath , imageFiles[i])
        matPath = os.path.join(groundTruthPath , groundTruthFiles[i])
        images.append(image.imread(imgPath))
        #read and extract all ground truths from the .mat file and insert them into the array
        data = scipy.io.loadmat(matPath)
        allGroundTruths = data['groundTruth'][0]
        imageGroundTruths = []
        for gt in range(len(allGroundTruths)):
            imageGroundTruths.append(allGroundTruths[gt][0][0][1])
        groundTruths.append(imageGroundTruths)
    return images , groundTruths
```

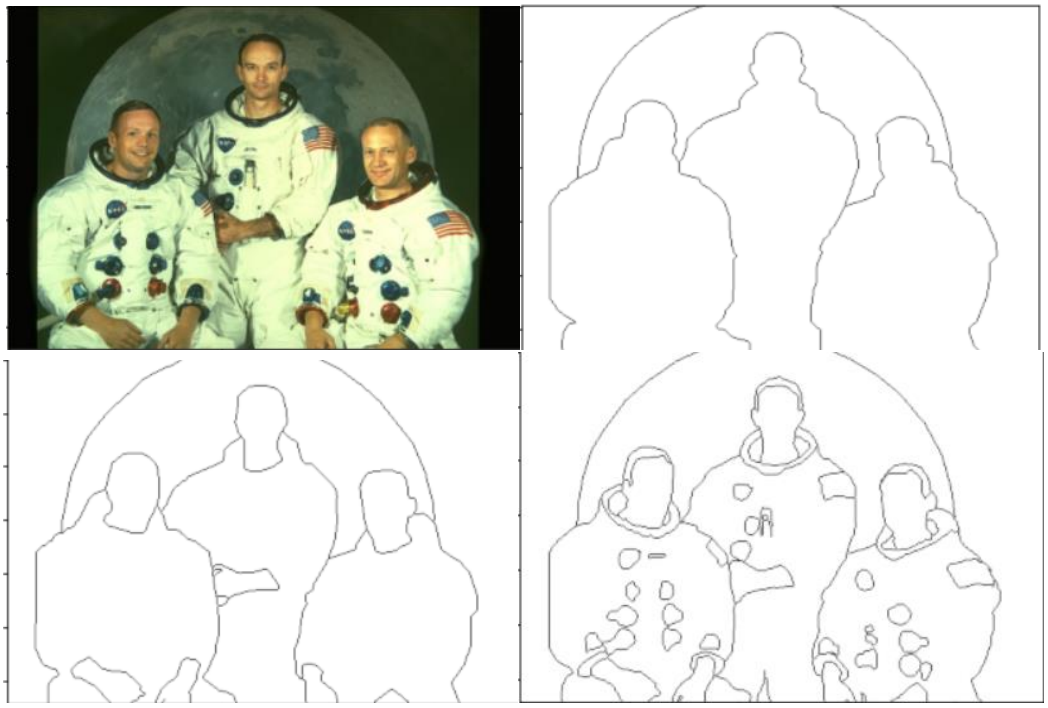
```
TrainImages , TrainGroundTruths = loadData('../input/bsr-images/data-2/data/images/train','../input/bsr-images/data-2/data/g
TestImages , TestGroundTruths = loadData('../input/bsr-images/data-2/data/images/test','../input/bsr-images/data-2/data/grou
```

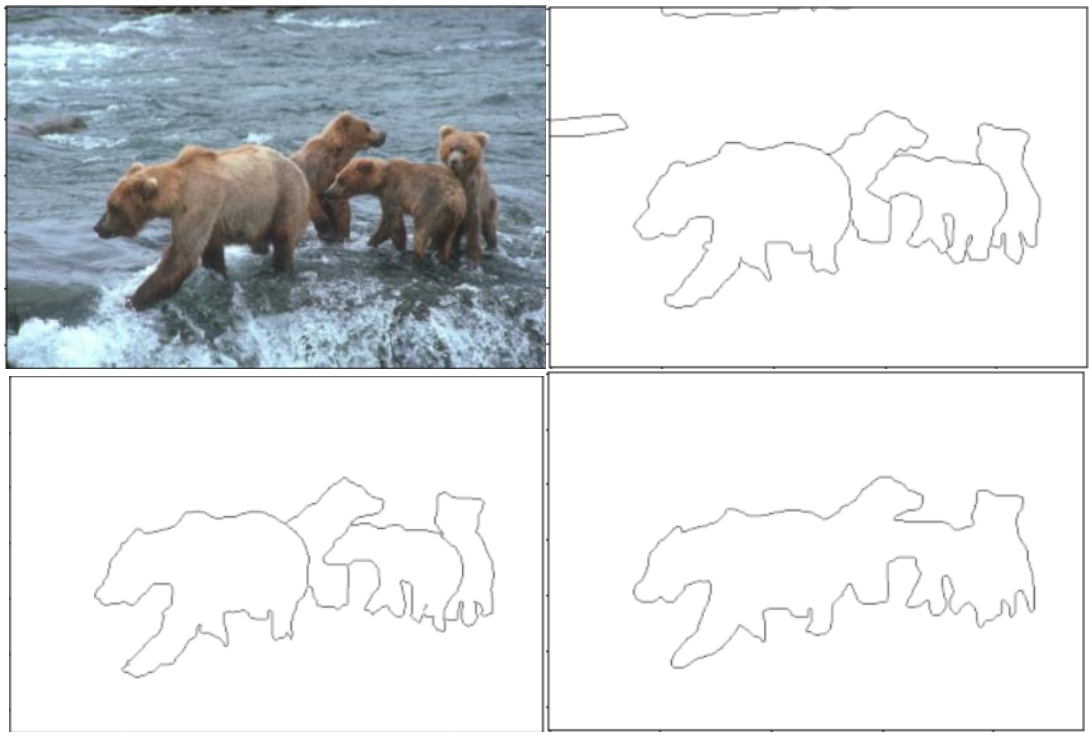
- *Previewing a random sample with its corresponding ground truth(s):*

```
def previewRandomData(imageArray , groundTruthArray):
    a = random.randint(0,199)
    plt.imshow(imageArray[a])
    plt.show()
    for t in range(len(groundTruthArray[a])) :
        plt.imshow(groundTruthArray[a][t] , cmap="Greys")
        plt.show()
```

```
# pyramids photo in the assignment report index = 132
                    #preview random training data
previewRandomData(TrainImages , TrainGroundTruths)

                          #preview random test data
#previewRandomData(TestImages , TestGroundTruths)
```

- *Sample runs:*





Segmentation using K-means:

a-

Implementation of K means:

```
[ ]:
def kmeans(x,k, no_of_iterations):
    idx = np.random.choice(len(x), k, replace=False)
    #Randomly choosing Centroids
    centroids = x[idx, :] #Step 1

    #finding the distance between centroids and all the data points
    distances = cdist(x, centroids ,'euclidean') #Step 2

    #Centroid with the minimum Distance
    points = np.array([np.argmin(i) for i in distances]) #Step 3

    #Repeating the above steps for a defined number of iterations
    #Step 4
    for _ in range(no_of_iterations):
        centroids = []
        for idx in range(k):
            #Updating Centroids by taking mean of Cluster it belongs to
            temp_cent = x[points==idx].mean(axis=0)
            centroids.append(temp_cent)

        centroids = np.vstack(centroids) #Updated Centroids

        distances = cdist(x, centroids ,'euclidean')
        points = np.array([np.argmin(i) for i in distances])

    return points,centroids
```

How The Algorithm Works:

K-means Algorithm:

- Randomly pick k data points as our initial Centroids.
- Find the distance between each data points using Euclidean Distance in our training set with the k centroids.
- assign each data point to the closest centroid according to the distance found.
- Update centroid location by taking the average of the points in each cluster group.
- Repeat till the centroids don't change.

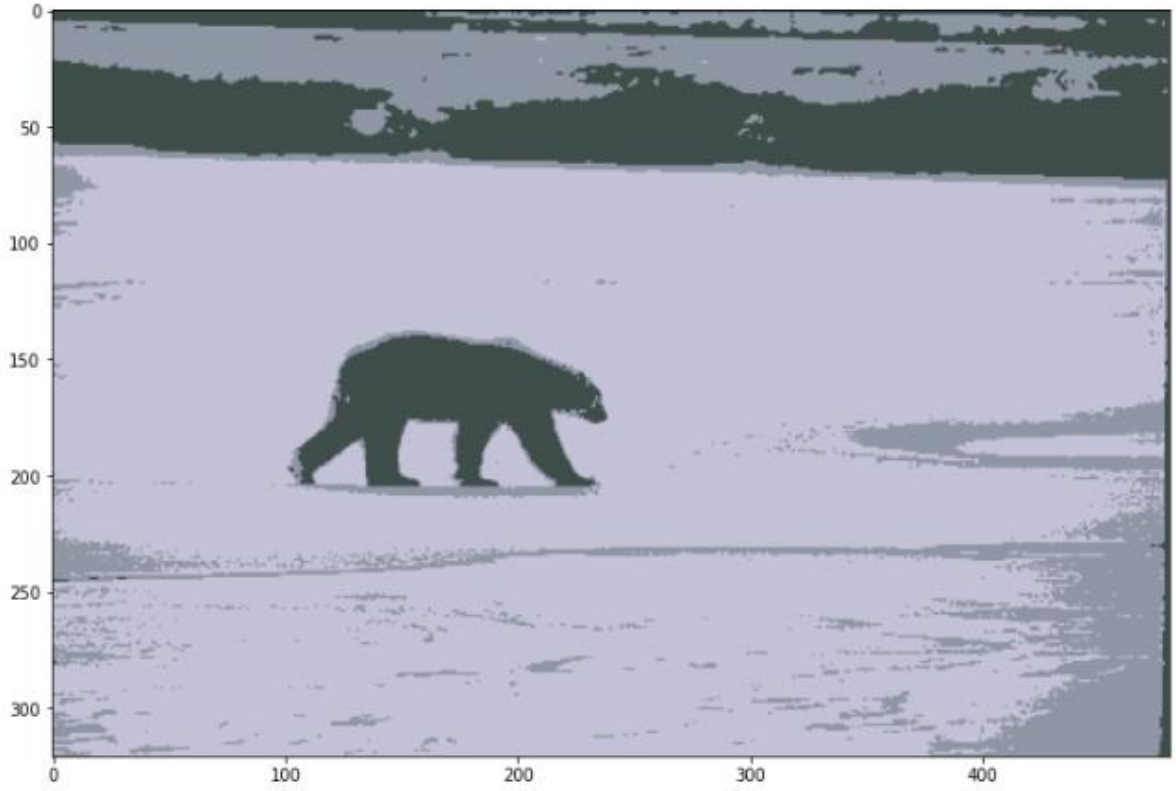
The Parameters:

- x >> the data to be clustered
- k >> number of clusters
- no_of_iterations >> maximum number of allowed iteration to terminate after it

Challenges faced and solutions:

If bad initial centroids were chosen, the algorithm might return an empty cluster as a result of 2 centroids having the exact same coordinates, to solve this a random restart needs to be done

Example for Image segmentation for different K's [3, 5, 7, 9, 11] respectively:


0
50
100
150
200
250
300
0
100
200
300
400


0
50
100
150
200
250
300
0
100
200
300
400


0
50
100
150
200
250
300
0
100
200
300
400


0
50
100
150
200
250
300
0
100
200
300
400


0
50
100
150
200
250
300
0
100
200
300
400


0
50
100
150
200
250
300
0
100
200
300
400


0
50
100
150
200
250
300
0
100
200
300
400


0
50
100
150
200
250
300
0
100
200
300
400


0
50
100
150
200
250
300
0
100
200
300
400


0
50
100
150
200
250
300
0
100
200
300
400

b- Evaluation: -

- **F-measure**

```
def getF_measure(labels,gt):
    gt=np.array(gt).flatten()
    result1=np.unique(gt)
    result2=np.unique(labels)
    pred_cls=[]
    true_cls=[]
    for c in result1:
        pos=np.where(gt== (c))[0]
        true_cls.append(pos)
    for c in result2:
        pos=np.where(labels== (c))[0]
        pred_cls.append(pos)
    prec=[]
    recall=[]
    for i in range(len(pred_cls)):
        maxratio1=- 1
        maxratio2=-1
        for k in range(len(true_cls)):
            pred_true=np.intersect1d(pred_cls[i],true_cls[k] )
            tempPrec=pred_true.size/pred_cls[i].size
            temprecall=pred_true.size/true_cls[k].size
            if(maxratio1<tempPrec):
                maxratio1=tempPrec
            if(maxratio2<temprecall):
                maxratio2=temprecall
        if(maxratio1>0):
            prec.append(maxratio1)
        if(maxratio2>0):
            recall.append(maxratio2)
    f=[]
    for i in range(len(prec)):
        f.append(((2*prec[i]*recall[i])/(prec[i]+recall[i])))
    f=np.array(f)
    s=np.sum(f)/result1.size
    return s
```

- Predicted points clusters by K-means get classified with each other according to cluster with index reference.
- Segmented ground truth images get classified with each other according to cluster with index reference.
- Iterate through predicted cluster and compare it with true clusters and obtain the max recall and prec for each class
- Calculate the sum of F-measure

- **Conditional Entropy**

```
def getC_entropy(labels,gt):
    gt=np.array(gt).flatten()
    result1=np.unique(gt)
    result2=np.unique(labels)
    pred_cls=[]
    true_cls=[]
    for c in result1:
        pos=np.where(gt== (c))[0]
        true_cls.append(pos)
    for c in result2:
        pos=np.where(labels== (c))[0]
        pred_cls.append(pos)
    cod_ent=[]
    for i in range(len(pred_cls)):
        ent_cls=[]
        for k in range(len(true_cls)):
            pred_true=np.intersect1d(pred_cls[i],true_cls[k])
            ratio=pred_true.size/pred_cls[i].size
            if(ratio!=0):
                ent_cls.append(-ratio*np.log10(ratio))
        cod_ent.append((np.sum(ent_cls))*(pred_cls[i].size/labels.size))
    return np.sum(cod_ent)
```

- Predicted points clusters by K-means get classified with each other according to cluster with index reference.
- Segmented ground truth images get classified with each other according to cluster with index reference.
- Iterate through the predicted cluster and compare it with true clusters and compute the ratio.

C-

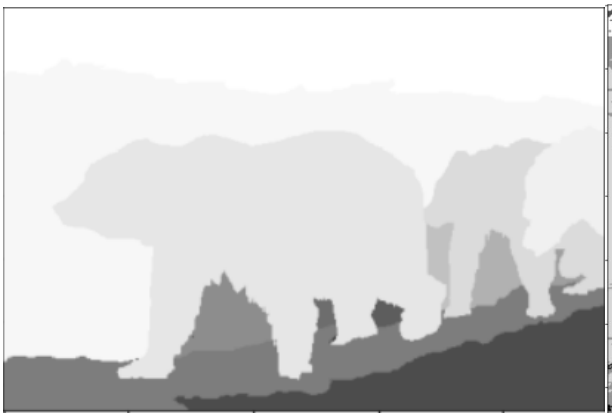
```
Max F_measure:1.3877936275884977
Average F_measure:0.3614330459607477
min Contional entropy:0.1048770606151049
Average Contional entropy:0.3614330459607477
```

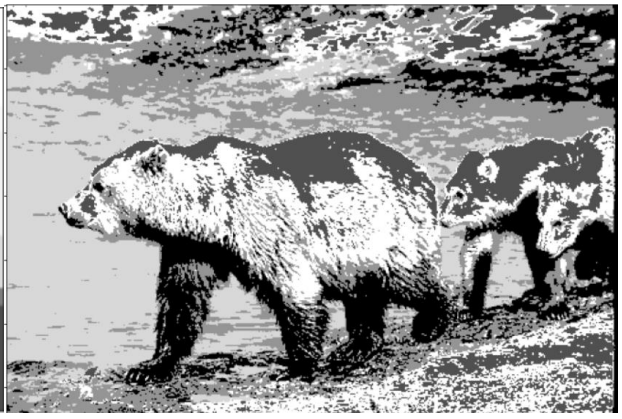
Good photo:-

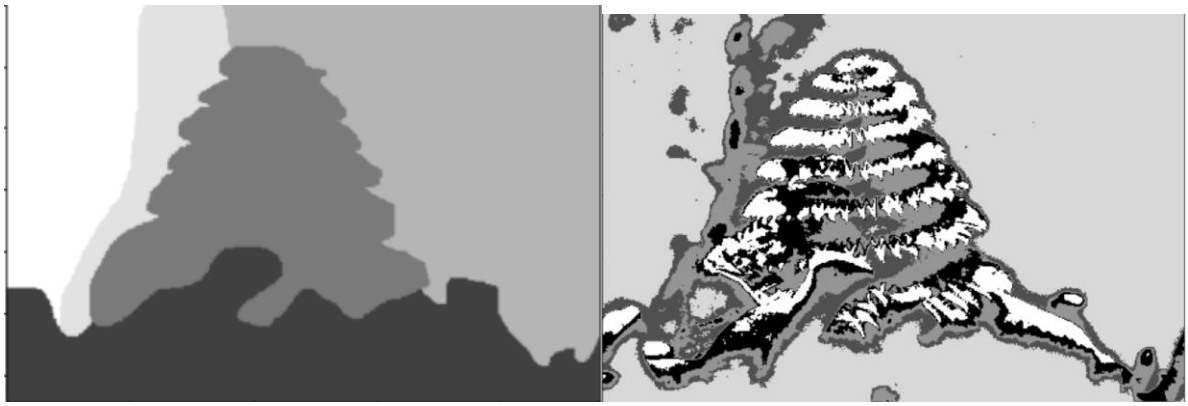
Bad photo:-

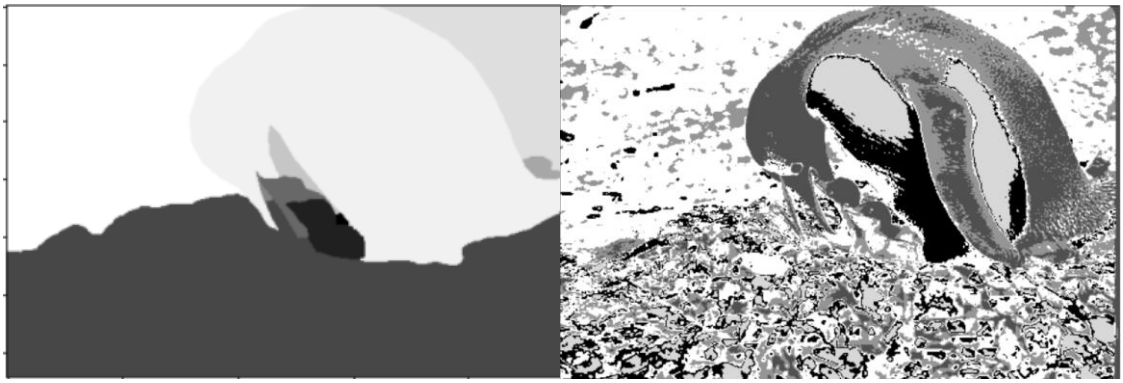
Big picture:

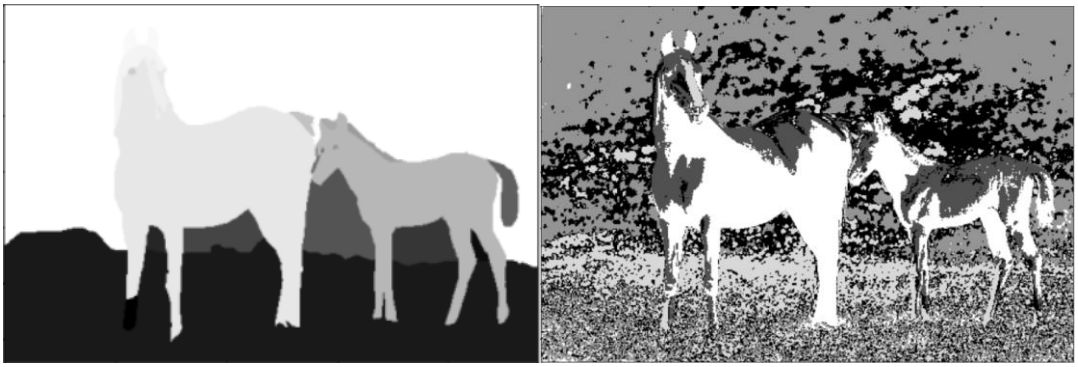
a- Ground truth VS k-means at k=5:

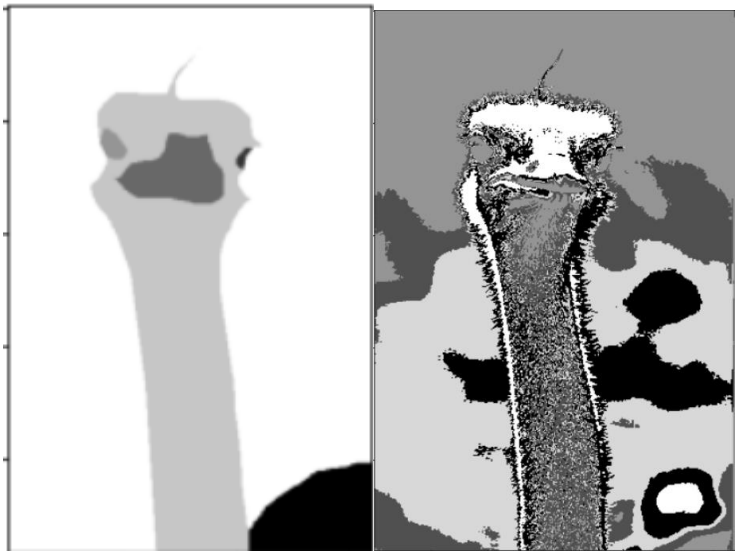












For this last image:

- **F-measure** = 0.39121
- **Entropy** = 0.225

b- Ground truth VS normalized cut:

In this part we implemented the normalized cut algorithm and used 5-nn as our similarity function before applying k-means.

A problem we faced is that the image size is too big (300,400) which would need a (300*400) **2 which means huge storage and exceptionally long running time, to

make things simpler we resized the photos to reduce the amount of time and storage needed

Main function:

```
def segmentationUsingNCut(image , newDimensions):
    image = resizeImage(image , newDimensions)
    image = image.reshape((-1,3))
    image = image.astype('float64')
    n = newDimensions[0]*newDimensions[1]
    similarityMatrixKnn = np.zeros((n,n))
    KnnArray = getKNN(image,5)
    for i in range(n):
        for j in range(n):
            if i==j or (i in KnnArray[j] or j in KnnArray[i]):
                similarityMatrixKnn[i][j] = 1
            else :
                similarityMatrixKnn[i][j] = 0
    Y = spectralClustering(similarityMatrixKnn, 5 , n)
    kmeans = KMeans(5)
    kmeans.fit(Y)
    identified_clusters = kmeans.fit_predict(Y)
    clustered_image = identified_clusters.reshape(newDimensions)
    print('Clustered image:')
    plt.imshow(clustered_image);
    plt.axis('off');
    plt.show()
    return clustered_image
```

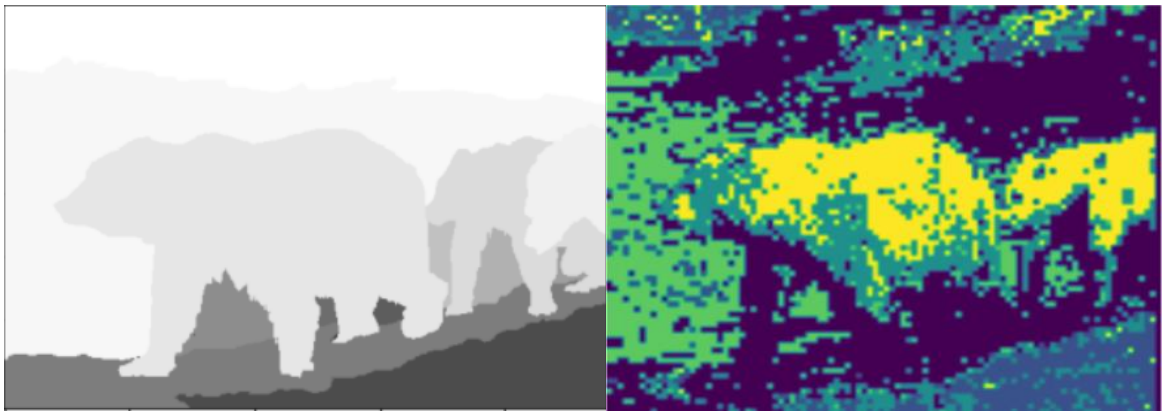
The main function calls the resize to resize the image to its desired size and then gets Y from “spectralClustering” function

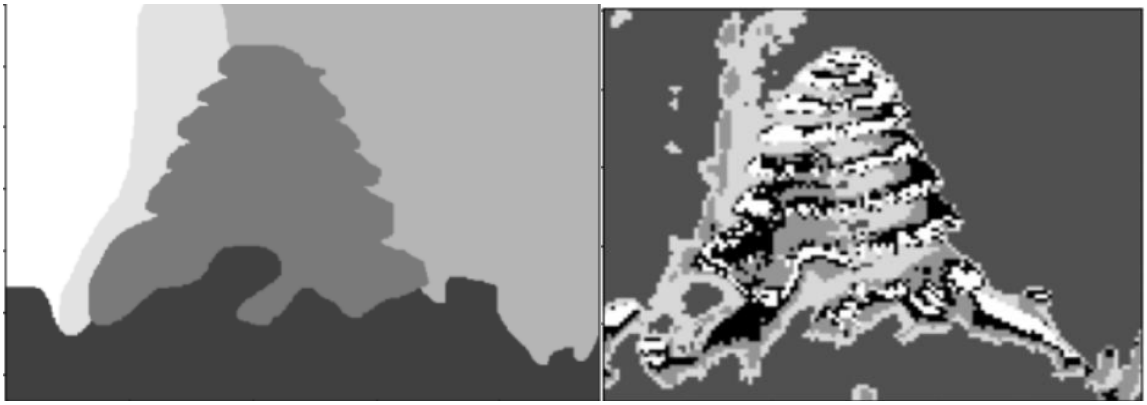
```python
def spectralClustering(adjacencyMatrix , k ,n ):
    # calculate Delta (degree matrix)
    degMatrix = degreeMatrix(adjacencyMatrix)
    # L = delta - A
    L = degMatrix - adjacencyMatrix
    # S-1*B -----> Delata-1 * L
    B = np.dot(np.linalg.inv(degMatrix),L)
    # calculate eigenVectors, sort them and get the least 3
    eigenValues , eigenVectors = np.linalg.eig(B)
    idx = eigenValues.argsort()
    eigenValues = eigenValues[idx]
    U = eigenVectors[:,idx]
    U = U[:,:k]
    # normalize using L2 norm equation
    Y = np.zeros(U.shape)
    norms = np.linalg.norm(U,axis = 1,ord=2)
    for row in range(n):
        Y[row] = U[row] / norms[row]
    return Y

def resizeImage(image , newDimensions):
    print('Original image:')
    plt.imshow(image);
    plt.axis('off');
    plt.show()
    image = cv2.resize(image, (newDimensions[1],newDimensions[0]))
    print('Image after resizing:')
    plt.imshow(image);
    plt.axis('off');
    plt.show()
    return image
```

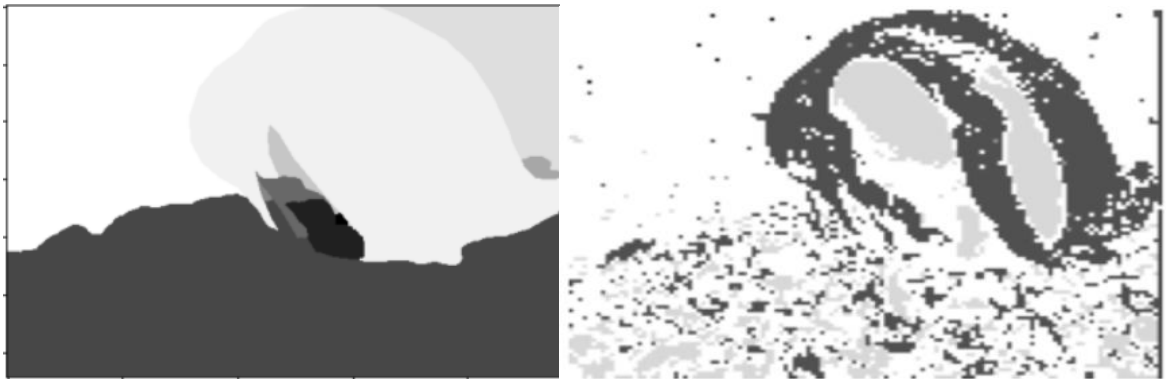
Degree matrix and similarity function:

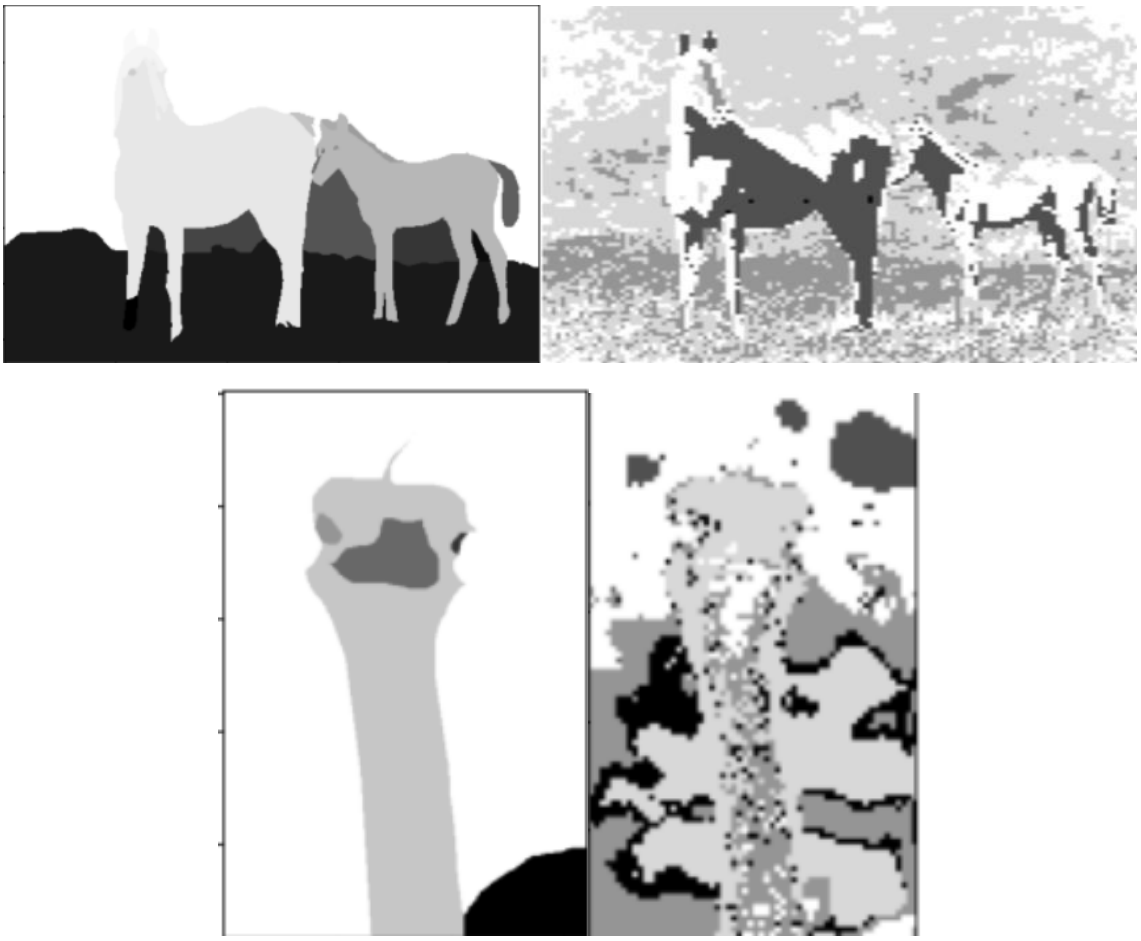
```python
def getKNN(P , k):
    nearestKIndecies = []
    n = len(P)
    for i in range(n):
        distances = []
        for j in range(n):
            if j==i:
                distances.append(float('inf'))
            else:
                distances.append( np.sqrt(np.power((P[i][0]-P[j][0]),2)+np.power((P[i][1]-P[j][1]),2)+np.power((P[i][2]-P[j][2]),2)) )
        idx = np.asarray(distances).argsort()
        nearestKIndecies.append(idx[:k])
    return nearestKIndecies

def degreeMatrix(adjacencyMatrix):
    n = len(adjacencyMatrix)
    degreeMatrix = []
    for i in range(n):
        degree = 0
        for j in range(n):
            if adjacencyMatrix[i][j] != 0 and i != j :
                degree +=1
        degreeMatrix.append(degree)
    return np.diag(degreeMatrix)
```

Results:









For this image:

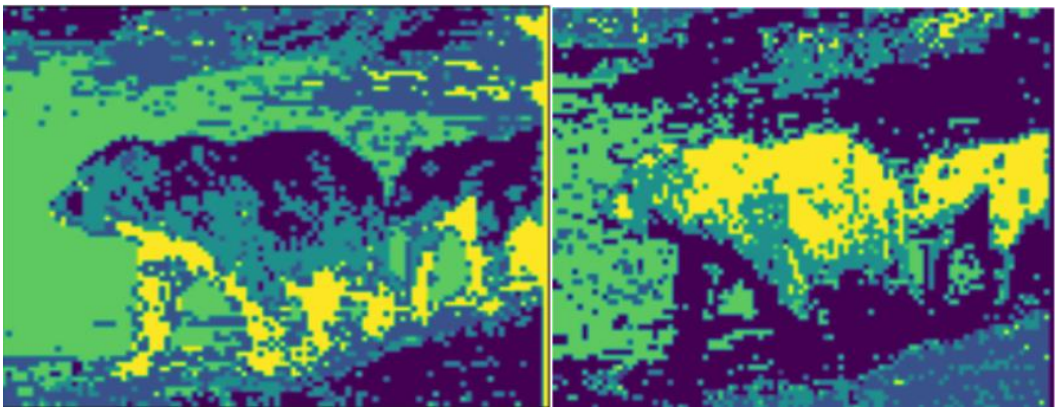
F-measure = 0.4027

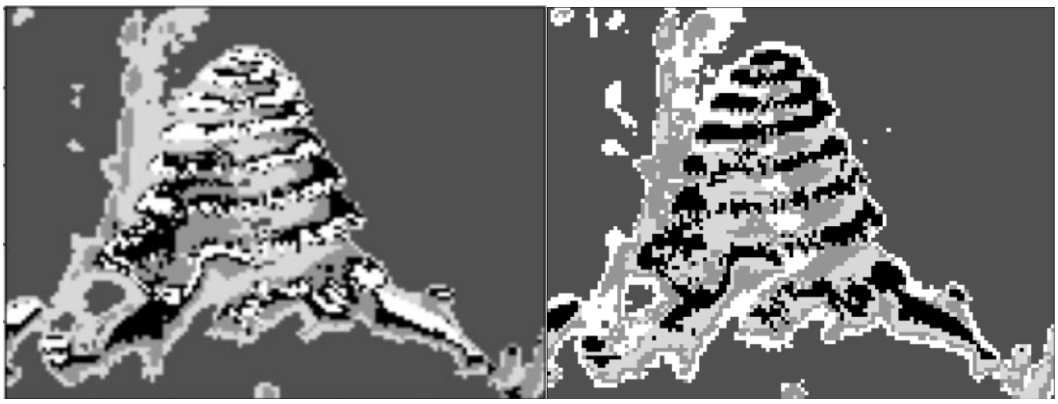
Entropy = 0.2613

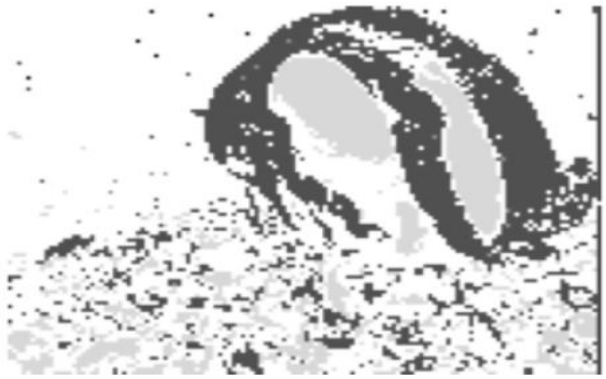
c- K-means VS normalized cut:

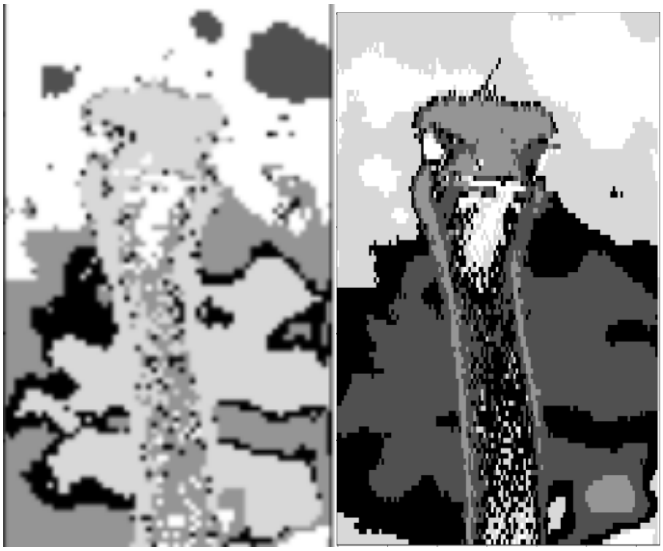
Resizing the image before using k-means to ensure that both algorithms work with the same image quality to ensure better comparison

K-means VS Clustering









For the last image:

F-measure = 0.789

Entropy = 0.24697

Note that results are quite worse and blurry than the normal k-means due to resizing

Comments:

It is noticed that the resulted segmentation from k-means and normalized cut clustering are almost the same, there is high similarity between both resulted segmentations, this is because the RGB measure is good enough to calculate the similarity which makes the needs to normalized cut is low.

However, the F-measure and entropy of both clustering techniques is bad compared to the ground truth segmentation, this is due to using only the pixel color as our measure and not taking into consideration the pixel location.

Also, it is because the ground truth uses higher values ok K than the ones used in our examples.

Extra:

The k-means clustering using only the RGB pixels colors is good at separating distinct color groups together however, it does not separate objects unless they are of distinct color groups, so it does not take into consideration the shadow and 3d figures. To solve this problem, we would add 2 parameters to the point that indicate the points location in the picture. This will help the algorithm cluster points that not only belong to the same color group, but also the ones that are close together.