PATTERN RECOGNITION

IMAGE SEGMENTATION

PROBLEM STATEMENT

We intend to perform image segmentation. Image segmentation means that we can group similar pixels together and give these grouped pixels the same label. The grouping problem is a clustering problem. We want to study the use of K-means on the Berkeley Segmentation Benchmark. Below we will show the needed steps to achieve the goal of the assignment. This task is achieved by following the steps above…

DOWNLOAD DATASET & UNDERSTAND THE FORMAT

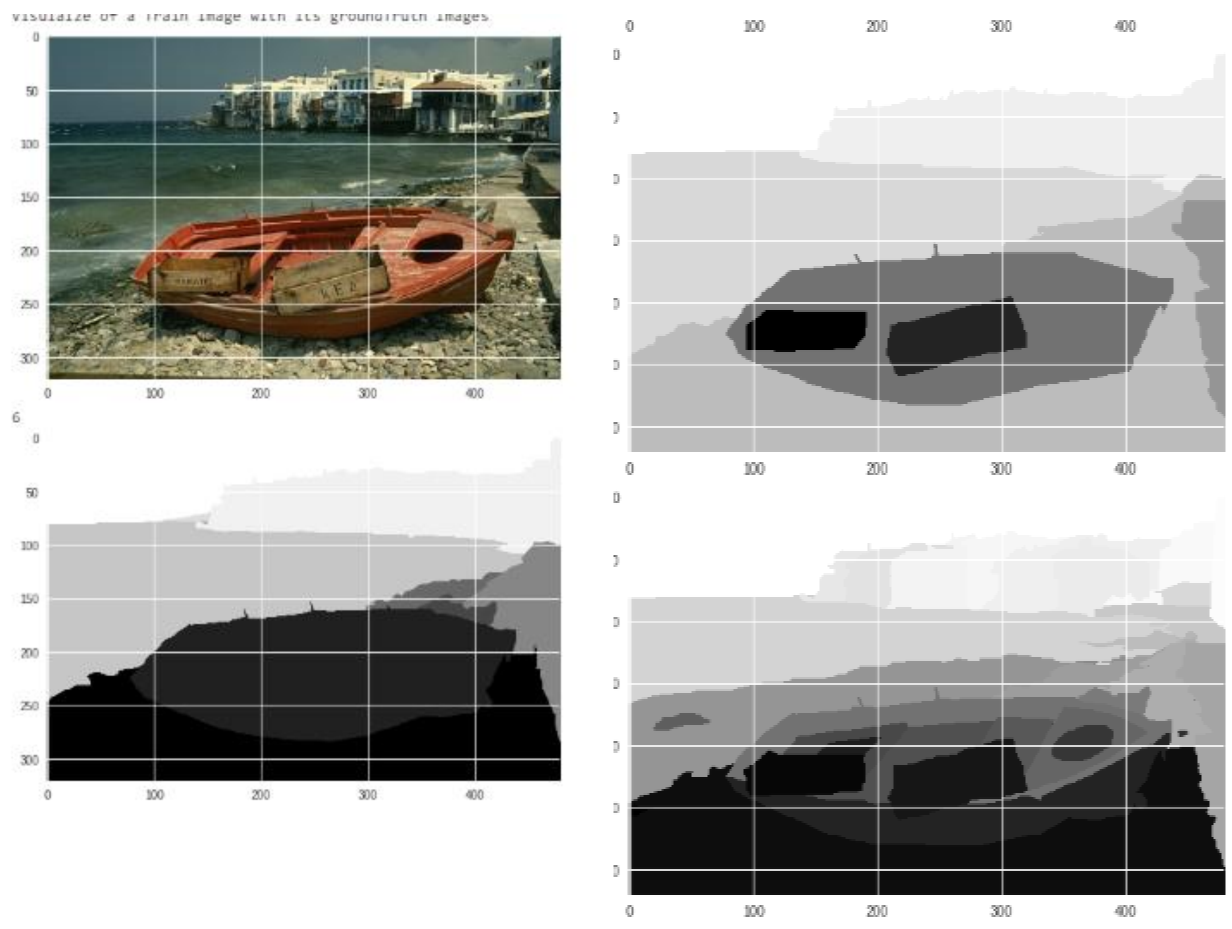
Our database of subject is very simple. We used Berkeley Segmentation Benchmark. The dataset has 500 images. The test set is 200 images only . We will report our results on the first 50 images of the test set only.
Here is an examples of download training (JPG images) dataset…

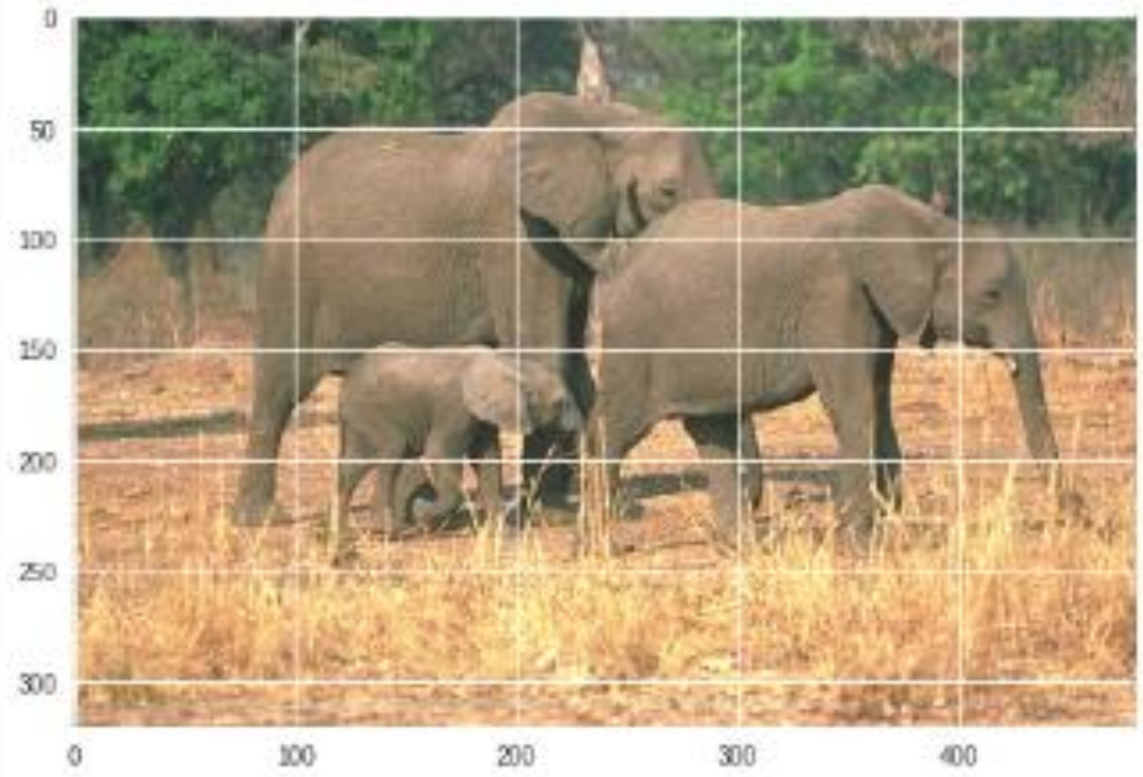
1. VISUALIZE THE IMAGE AND THE GROUND TRUTH SEGMENTATION.

The following steps must be done:

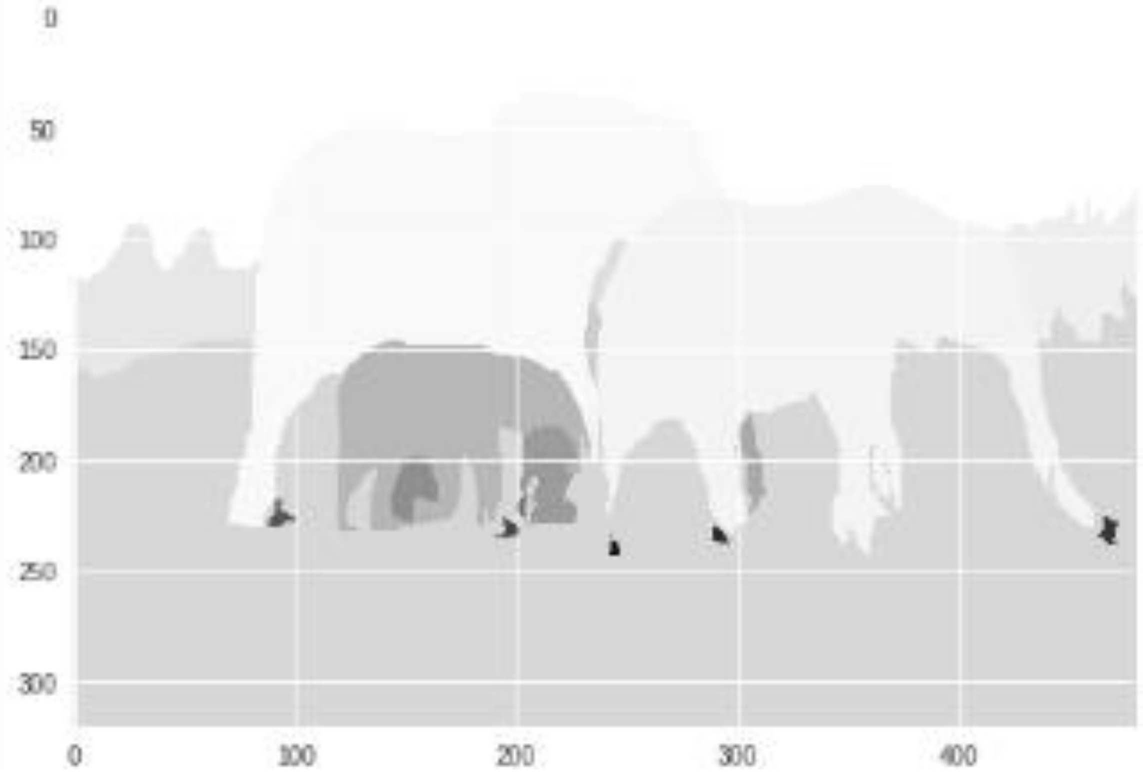
- Function that reads an image
- Display an image with its associated ground truth segmentation(s) (mat file)
- Here is the implementation code …

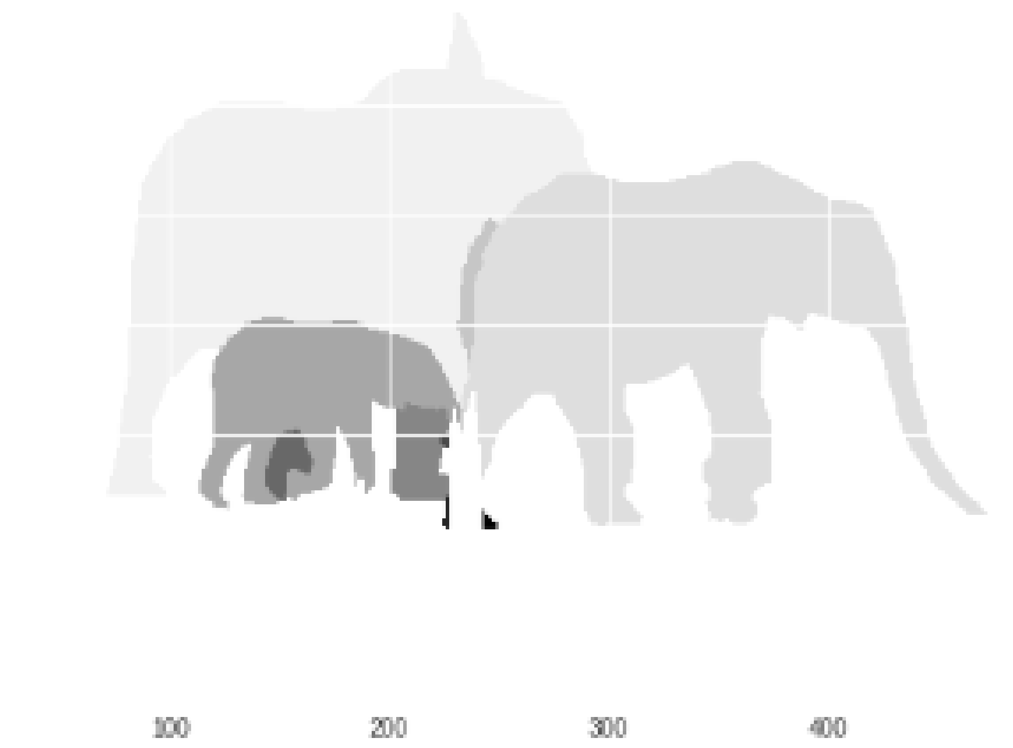
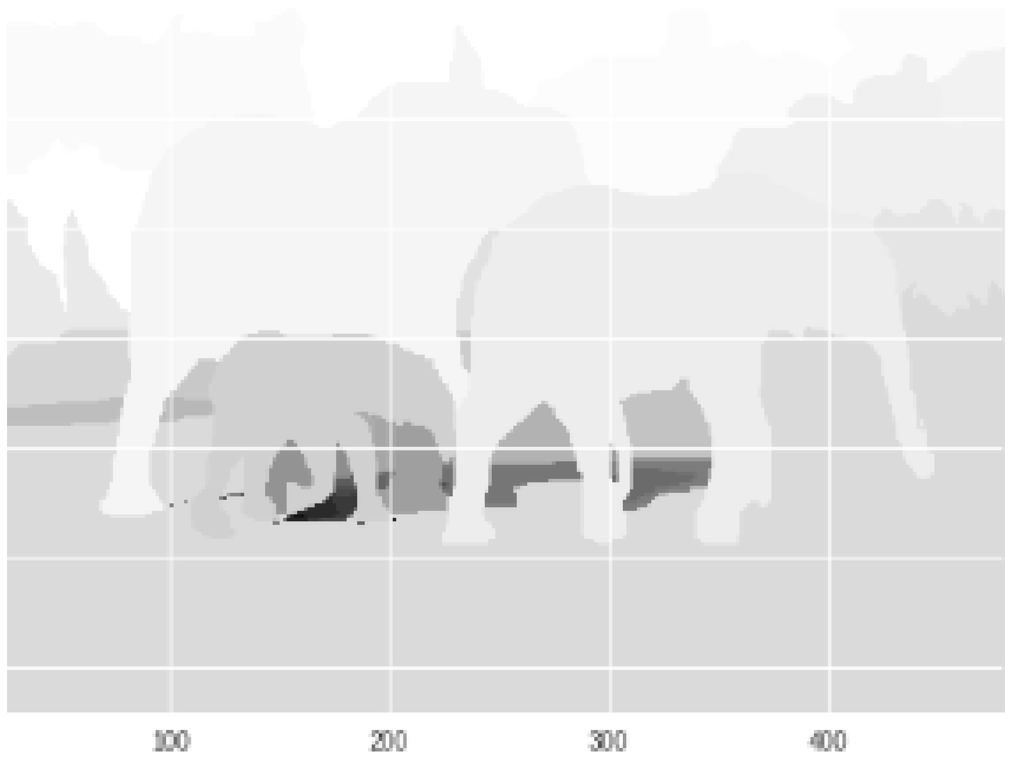
Reading .JPG images and corresponding GroundTruth images from Train Data




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

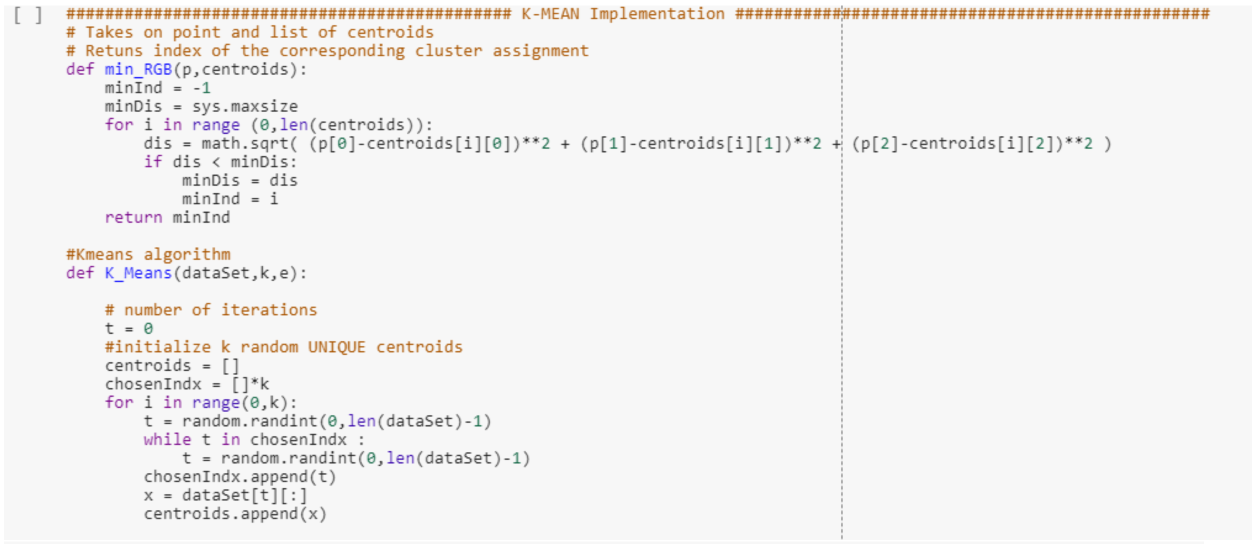
7


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

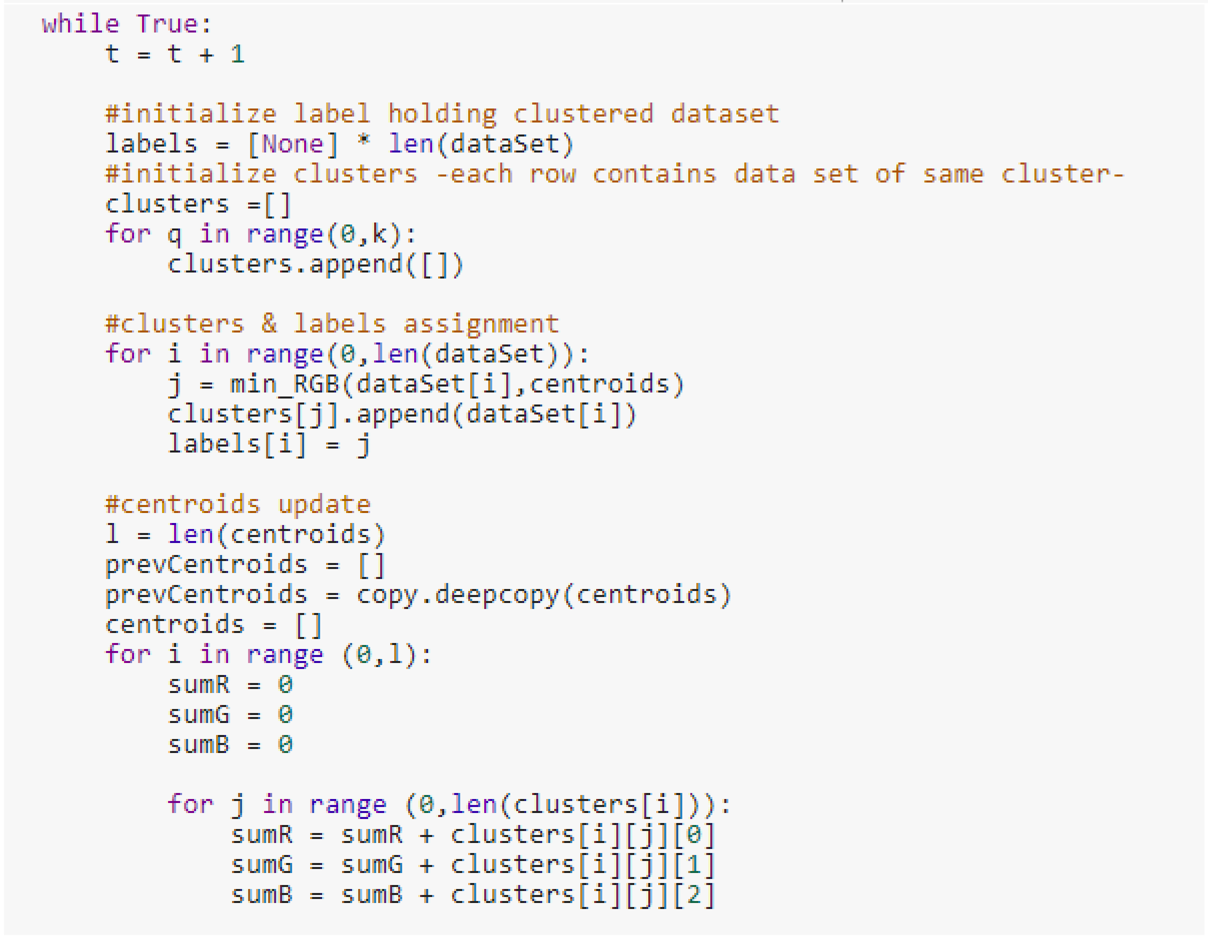

100
200
300
400

100
200
300
400

SEGMENTATION USING K-MEANS

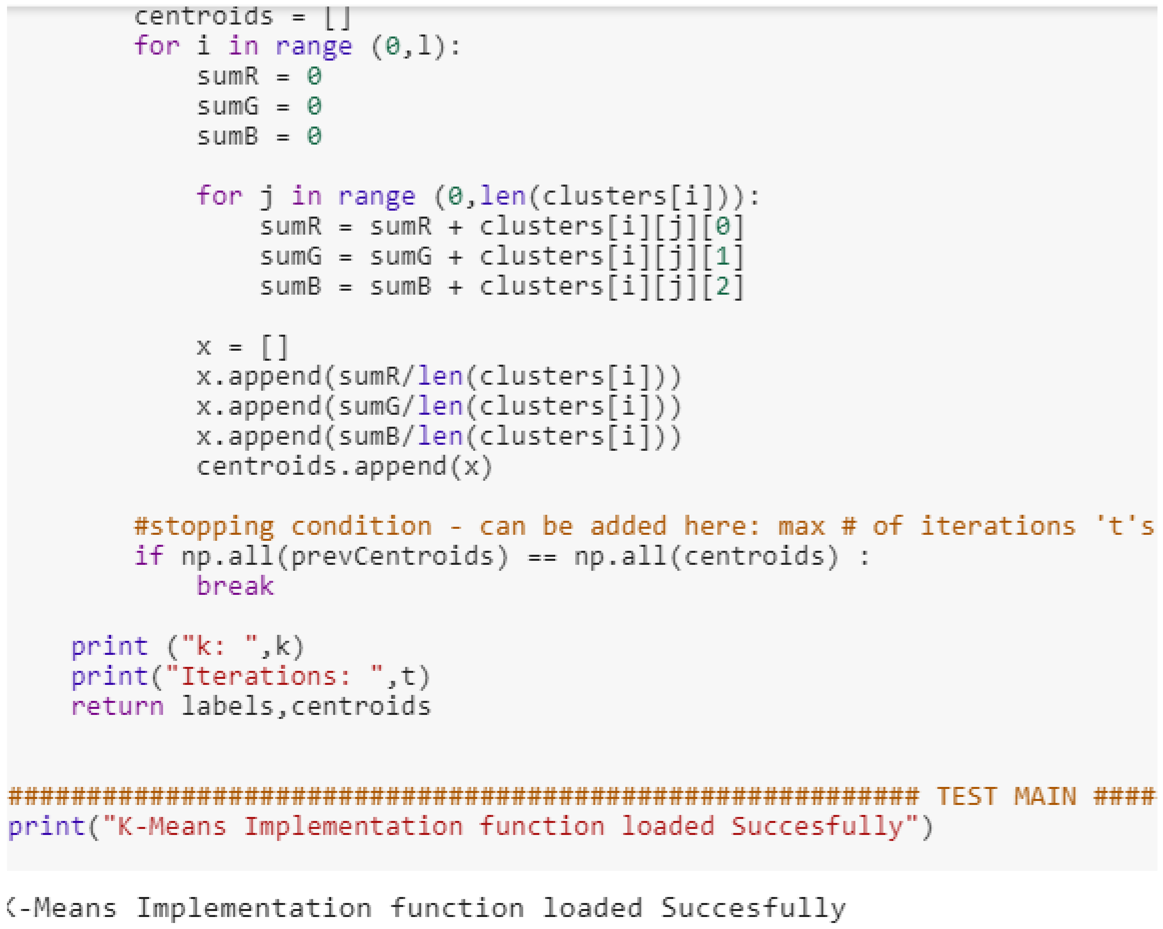
each pixel of an image is an instanace in data set and each pixel has 3 features : RGB .



```
[ ]  ############################################### K-MEAN Implementation ##################################################
     # Takes on point and list of centroids
     # Retuns index of the corresponding cluster assignment
     def min_RGB(p,centroids):
         minInd = -1
         minDis = sys.maxsize
         for i in range (0,len(centroids)):
             dis = math.sqrt( (p[0]-centroids[i][0])**2 + (p[1]-centroids[i][1])**2 + (p[2]-centroids[i][2])**2 )
             if dis < minDis:
                 minDis = dis
                 minInd = i
         return minInd

     #Kmeans algorithm
     def K_Means(dataSet,k,e):

         # number of iterations
         t = 0
         #initialize k random UNIQUE centroids
         centroids = []
         chosenIndx = []*k
         for i in range(0,k):
             t = random.randint(0,len(dataSet)-1)
             while t in chosenIndx :
                 t = random.randint(0,len(dataSet)-1)
             chosenIndx.append(t)
             x = dataSet[t][:]
             centroids.append(x)
```



```
while True:
    t = t + 1

    #initialize label holding clustered dataset
    labels = [None] * len(dataSet)
    #initialize clusters -each row contains data set of same cluster-
    clusters =[]
    for q in range(0,k):
        clusters.append([])

    #clusters & labels assignment
    for i in range(0,len(dataSet)):
        j = min_RGB(dataSet[i],centroids)
        clusters[j].append(dataSet[i])
        labels[i] = j

    #centroids update
    l = len(centroids)
    prevCentroids = []
    prevCentroids = copy.deepcopy(centroids)
    centroids = []
    for i in range (0,l):
        sumR = 0
        sumG = 0
        sumB = 0

        for j in range (0,len(clusters[i])):
            sumR = sumR + clusters[i][j][0]
            sumG = sumG + clusters[i][j][1]
            sumB = sumB + clusters[i][j][2]
```



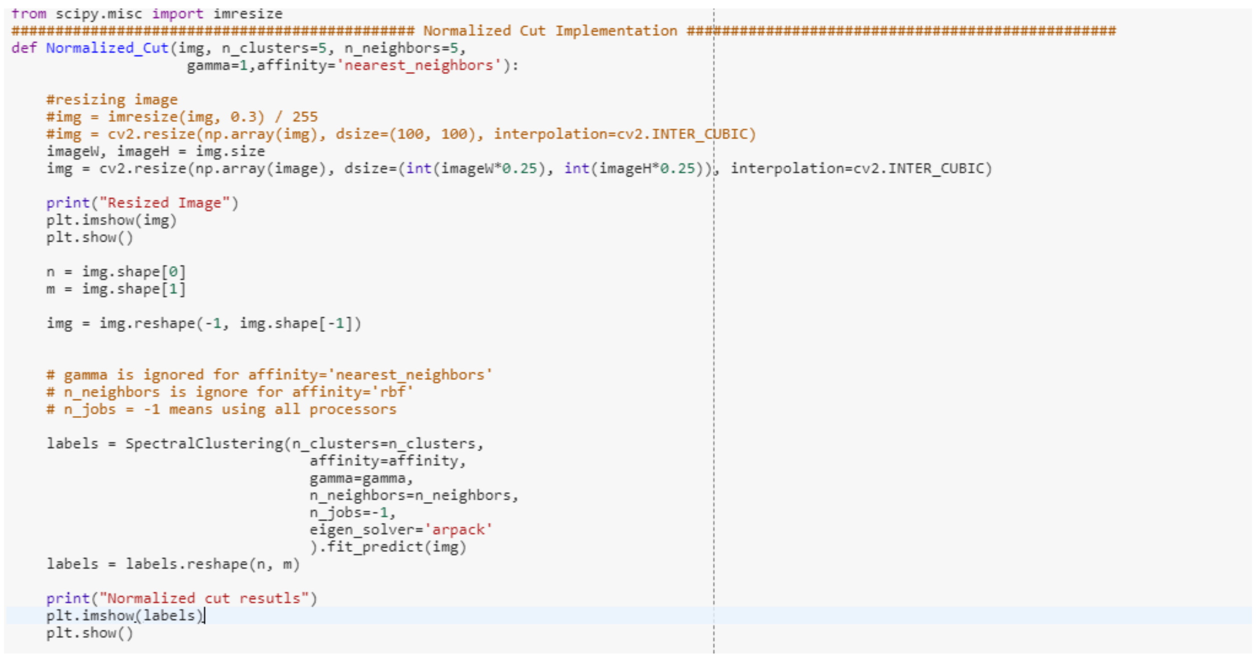
```
        centroids = []
        for i in range (0,l):
            sumR = 0
            sumG = 0
            sumB = 0

            for j in range (0,len(clusters[i])):
                sumR = sumR + clusters[i][j][0]
                sumG = sumG + clusters[i][j][1]
                sumB = sumB + clusters[i][j][2]

            x = []
            x.append(sumR/len(clusters[i]))
            x.append(sumG/len(clusters[i]))
            x.append(sumB/len(clusters[i]))
            centroids.append(x)

        #stopping condition - can be added here: max # of iterations 't's
        if np.all(prevCentroids) == np.all(centroids) :
            break

    print ("k: ",k)
    print("Iterations: ",t)
    return labels,centroids


########################################################## TEST MAIN ####
print("K-Means Implementation function loaded Succesfully")
```

```
(-Means Implementation function loaded Succesfully
```

NORMALIZED-CUT

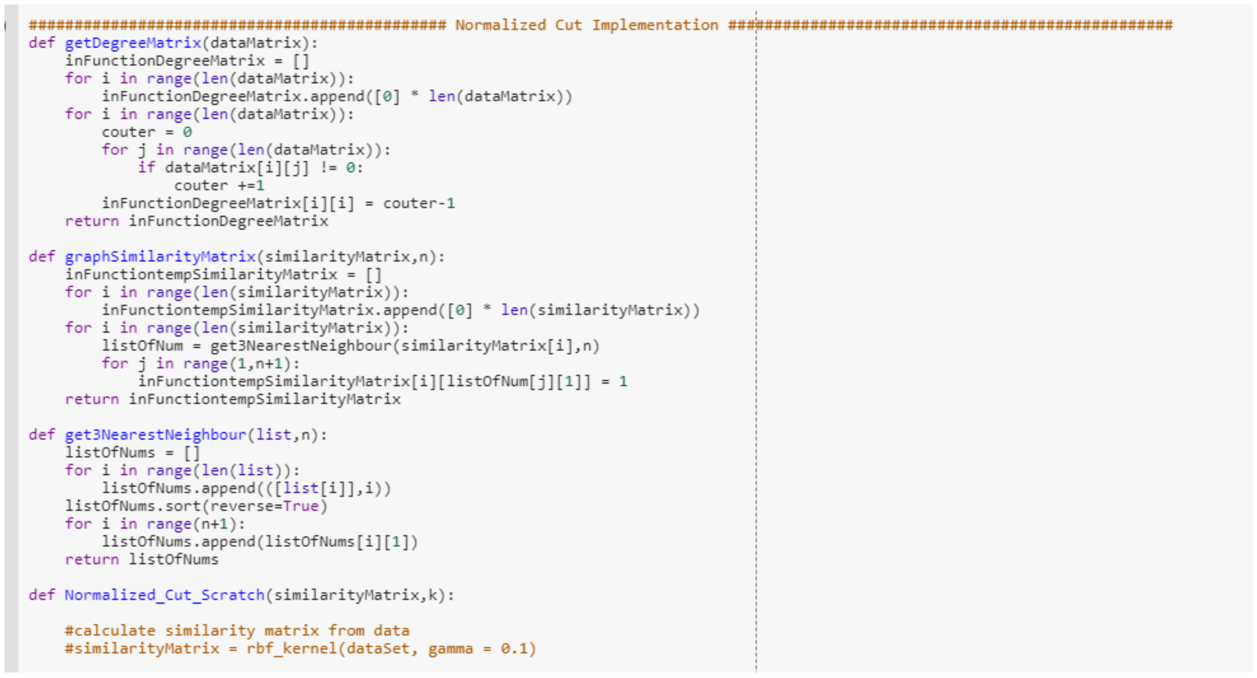
We did our own normalized cut and we also try the built in function

- Normalized Cut Implementation:

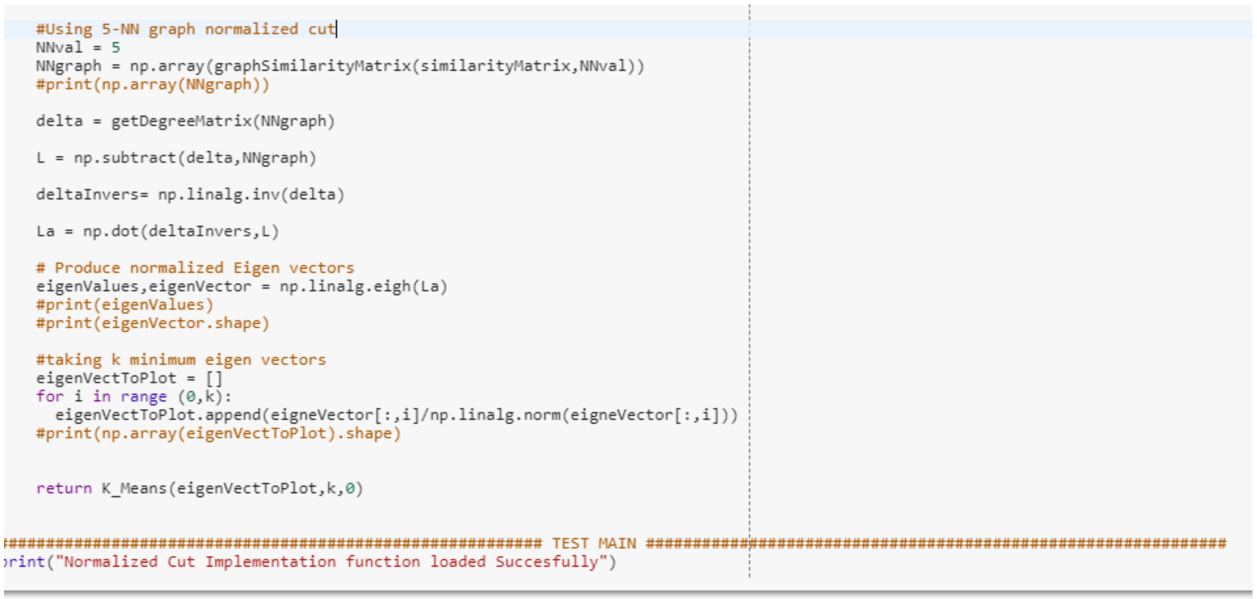


```python
from scipy.misc import imresize
############################################## Normalized Cut Implementation ################################################
def Normalized_Cut(img, n_clusters=5, n_neighbors=5,
                    gamma=1,affinity='nearest_neighbors'):

    #resizing image
    #img = imresize(img, 0.3) / 255
    #img = cv2.resize(np.array(img), dsize=(100, 100), interpolation=cv2.INTER_CUBIC)
    imageW, imageH = img.size
    img = cv2.resize(np.array(image), dsize=(int(imageW*0.25), int(imageH*0.25)), interpolation=cv2.INTER_CUBIC)

    print("Resized Image")
    plt.imshow(img)
    plt.show()

    n = img.shape[0]
    m = img.shape[1]

    img = img.reshape(-1, img.shape[-1])


    # gamma is ignored for affinity='nearest_neighbors'
    # n_neighbors is ignore for affinity='rbf'
    # n_jobs = -1 means using all processors

    labels = SpectralClustering(n_clusters=n_clusters,
                                affinity=affinity,
                                gamma=gamma,
                                n_neighbors=n_neighbors,
                                n_jobs=-1,
                                eigen_solver='arpack'
                                ).fit_predict(img)
    labels = labels.reshape(n, m)

    print("Normalized cut resutls")
    plt.imshow(labels)
    plt.show()
```

- Normalized Cut Implementation (From Scratch)

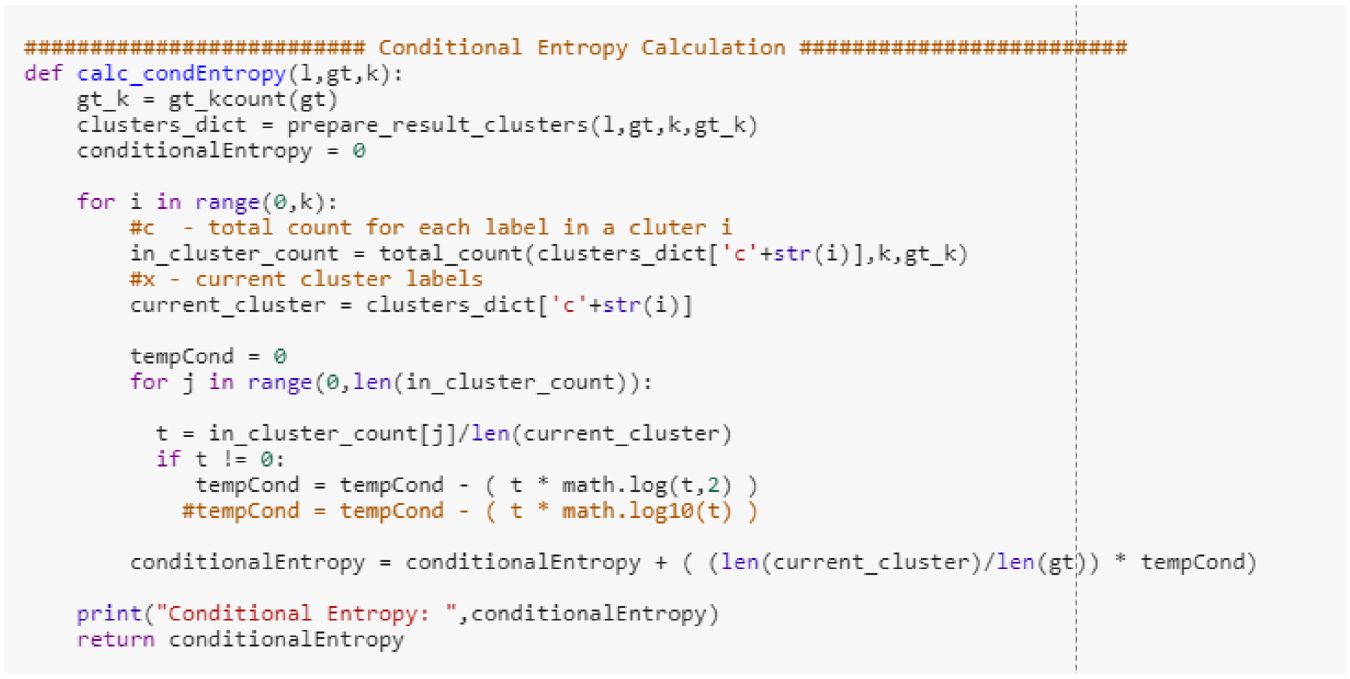


```python
############################################### Normalized Cut Implementation ##################################################
def getDegreeMatrix(dataMatrix):
    inFunctionDegreeMatrix = []
    for i in range(len(dataMatrix)):
        inFunctionDegreeMatrix.append([0] * len(dataMatrix))
    for i in range(len(dataMatrix)):
        couter = 0
        for j in range(len(dataMatrix)):
            if dataMatrix[i][j] != 0:
                couter +=1
        inFunctionDegreeMatrix[i][i] = couter-1
    return inFunctionDegreeMatrix

def graphSimilarityMatrix(similarityMatrix,n):
    inFunctiontempSimilarityMatrix = []
    for i in range(len(similarityMatrix)):
        inFunctiontempSimilarityMatrix.append([0] * len(similarityMatrix))
    for i in range(len(similarityMatrix)):
        listOfNum = get3NearestNeighbour(similarityMatrix[i],n)
        for j in range(1,n+1):
            inFunctiontempSimilarityMatrix[i][listOfNum[j][1]] = 1
    return inFunctiontempSimilarityMatrix

def get3NearestNeighbour(list,n):
    listOfNums = []
    for i in range(len(list)):
        listOfNums.append(([list[i]],i))
    listOfNums.sort(reverse=True)
    for i in range(n+1):
        listOfNums.append(listOfNums[i][1])
    return listOfNums

def Normalized_Cut_Scratch(similarityMatrix,k):

    #calculate similarity matrix from data
    #similarityMatrix = rbf_kernel(dataSet, gamma = 0.1)
```



```
    #Using 5-NN graph normalized cut
    NNval = 5
    NNgraph = np.array(graphSimilarityMatrix(similarityMatrix,NNval))
    #print(np.array(NNgraph))

    delta = getDegreeMatrix(NNgraph)

    L = np.subtract(delta,NNgraph)

    deltaInvers= np.linalg.inv(delta)

    La = np.dot(deltaInvers,L)

    # Produce normalized Eigen vectors
    eigenValues,eigenVector = np.linalg.eigh(La)
    #print(eigenValues)
    #print(eigenVector.shape)

    #taking k minimum eigen vectors
    eigenVectToPlot = []
    for i in range (0,k):
      eigenVectToPlot.append(eigneVector[:,i]/np.linalg.norm(eigneVector[:,i]))
    #print(np.array(eigenVectToPlot).shape)


    return K_Means(eigenVectToPlot,k,0)

############################################################ TEST MAIN #################################################################
rint("Normalized Cut Implementation function loaded Succesfully")
```

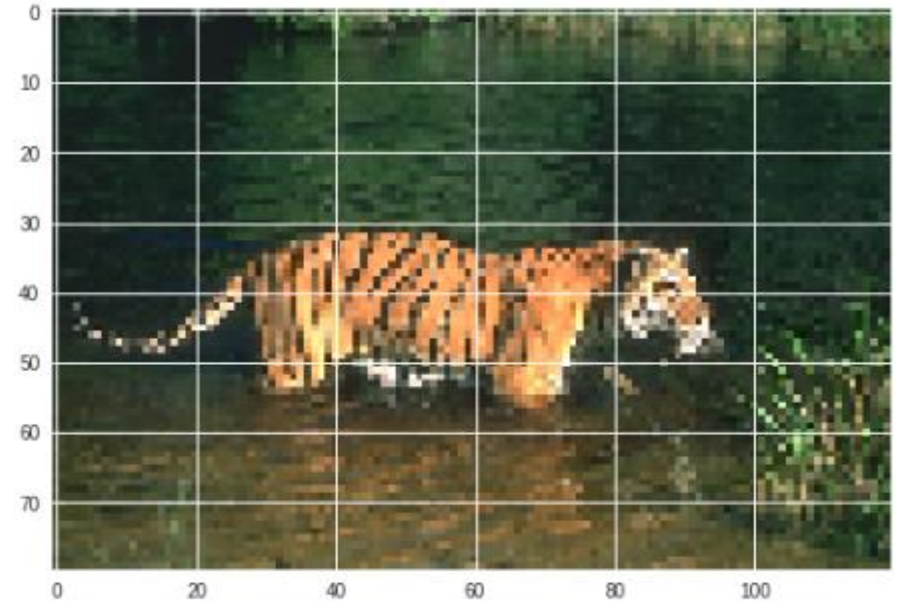
External Measures Evaluation Implementation (From Scratch):



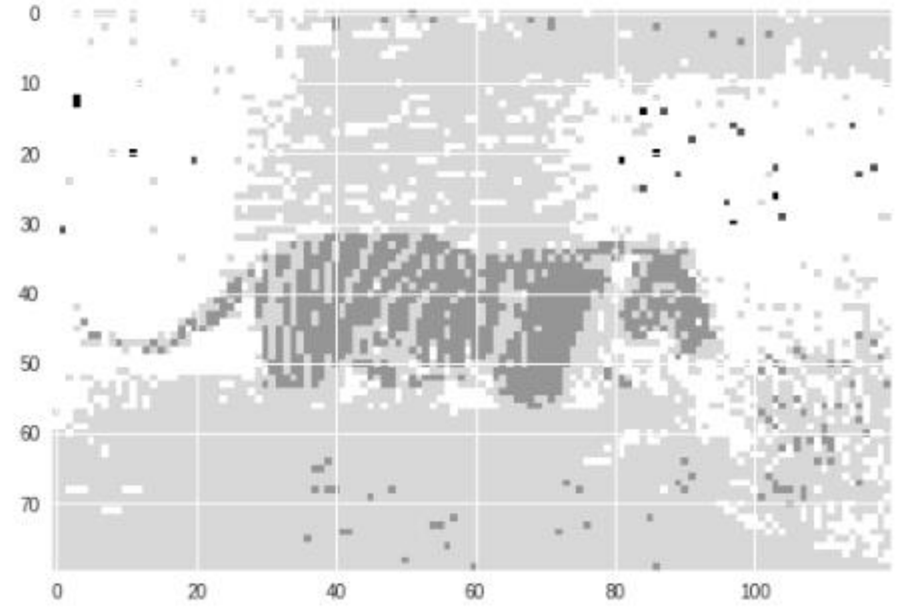
```
########################### Conditional Entropy Calculation ##########################
def calc_condEntropy(l,gt,k):
    gt_k = gt_kcount(gt)
    clusters_dict = prepare_result_clusters(l,gt,k,gt_k)
    conditionalEntropy = 0

    for i in range(0,k):
        #c  - total count for each label in a cluter i
        in_cluster_count = total_count(clusters_dict['c'+str(i)],k,gt_k)
        #x - current cluster labels
        current_cluster = clusters_dict['c'+str(i)]

        tempCond = 0
        for j in range(0,len(in_cluster_count)):

          t = in_cluster_count[j]/len(current_cluster)
          if t != 0:
             tempCond = tempCond - ( t * math.log(t,2) )
            #tempCond = tempCond - ( t * math.log10(t) )

        conditionalEntropy = conditionalEntropy + ( (len(current_cluster)/len(gt)) * tempCond)

    print("Conditional Entropy: ",conditionalEntropy)
    return conditionalEntropy
```

Test Segmentation (K-means & Normalized cut) on .JPG Image & Test Evaluation Measures

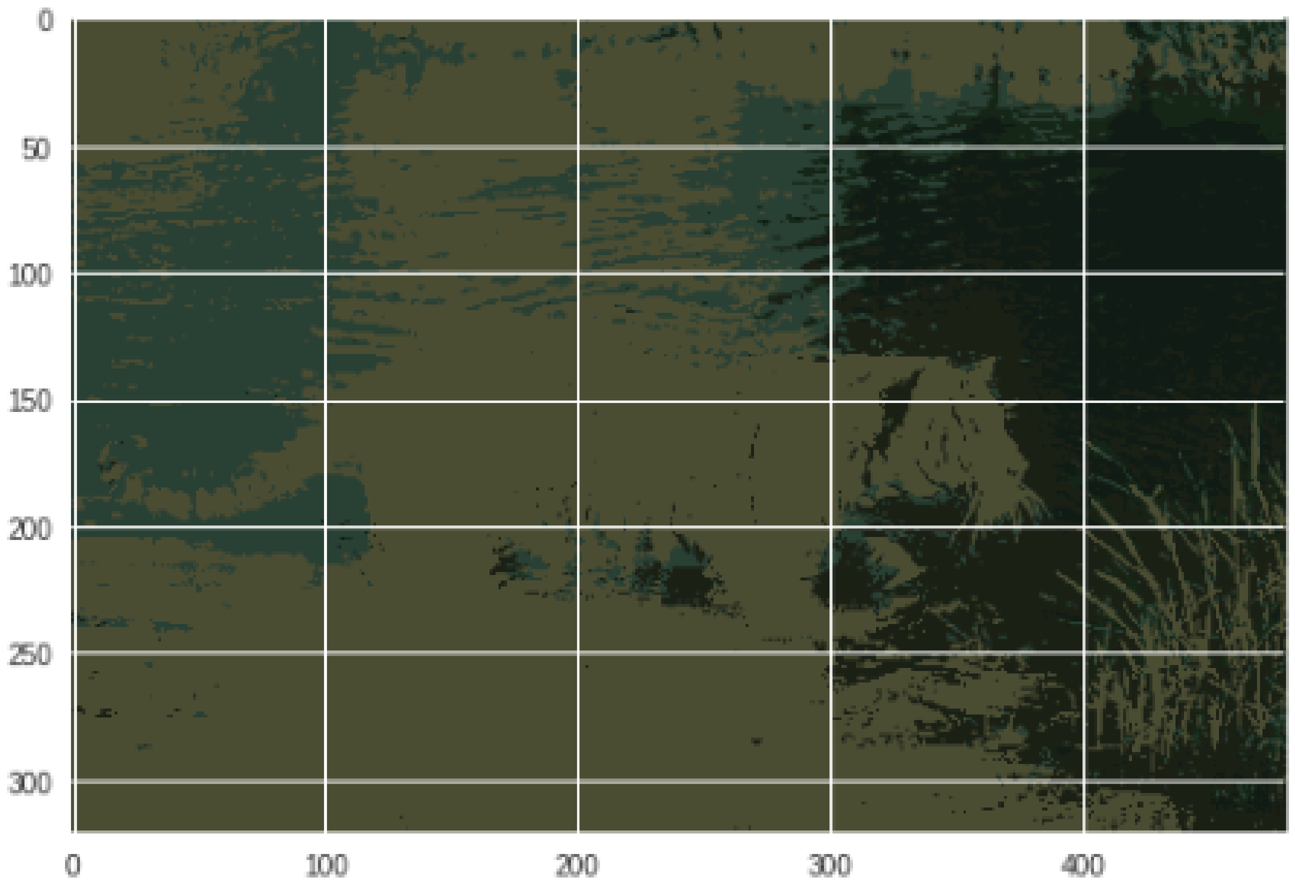
```
Normalized Cut Test
Resized Image
```



```
/usr/local/lib/python3.6/dist-packages/sklearn/manifold/spectral_embedd
  warnings.warn("Graph is not fully connected, spectral embedding"
Normalized cut resutls
```

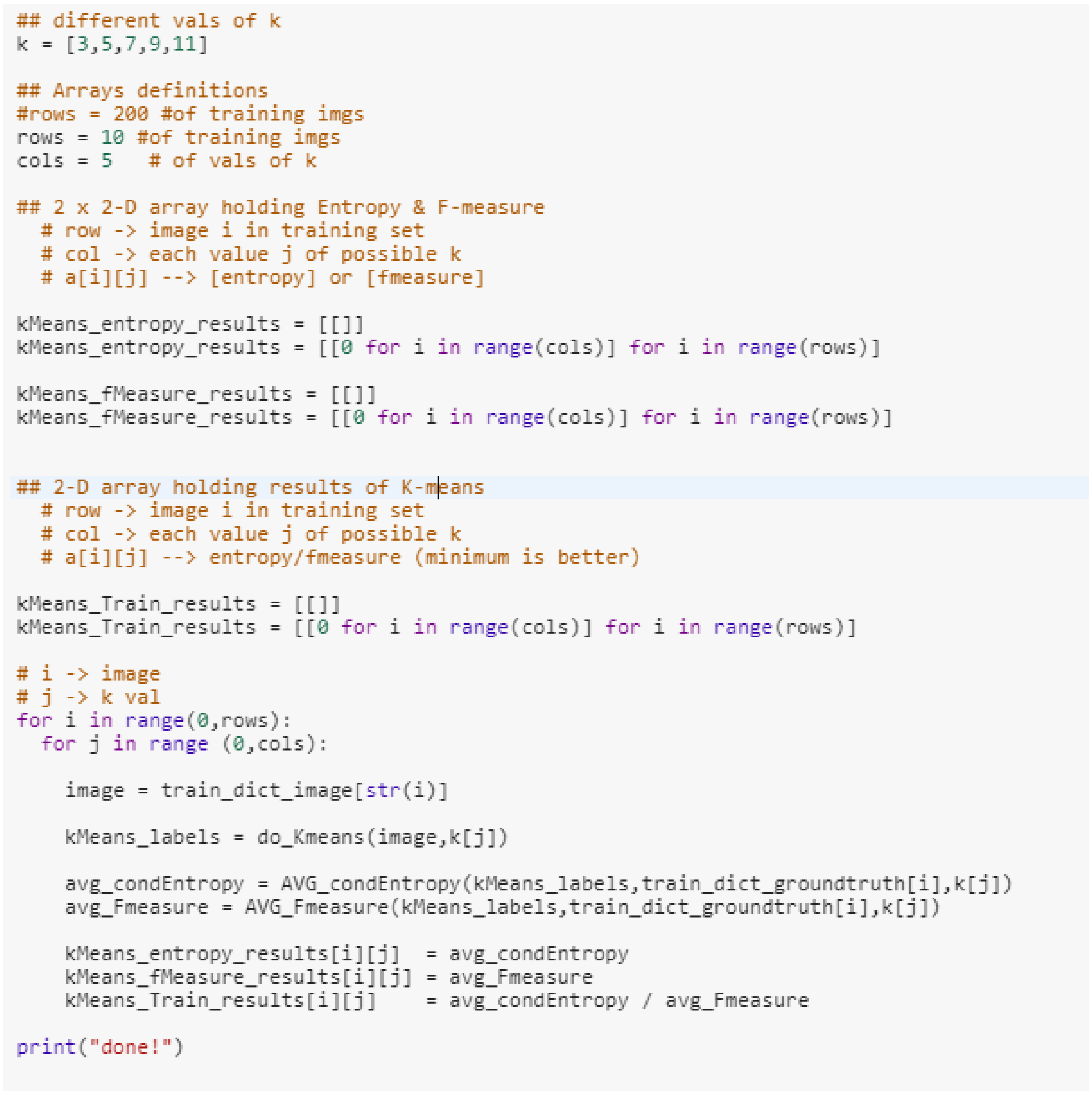


```
K means Test
rgb:  (79, 85, 49)
k:  5
Iterations:  70737
154401
```

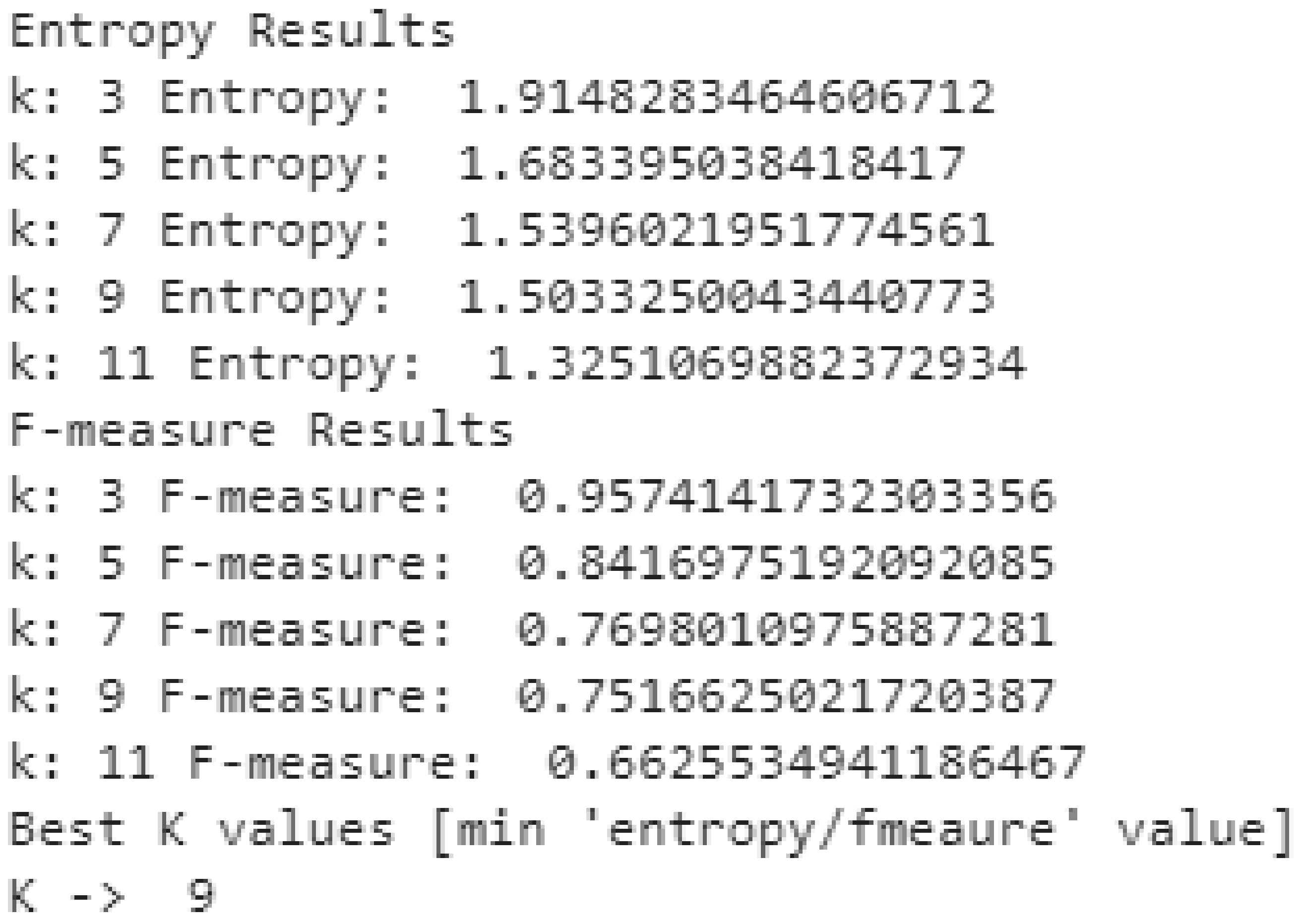


```
Conditional Entropy:  0.5172830453188827
F-Measure:  0.29289933092743503
```

Apply K-means on k = [3,5,7,9,11] on Training set images - Comparing measures for each k - Decide best to be run on Test - Showing measure results on Test set

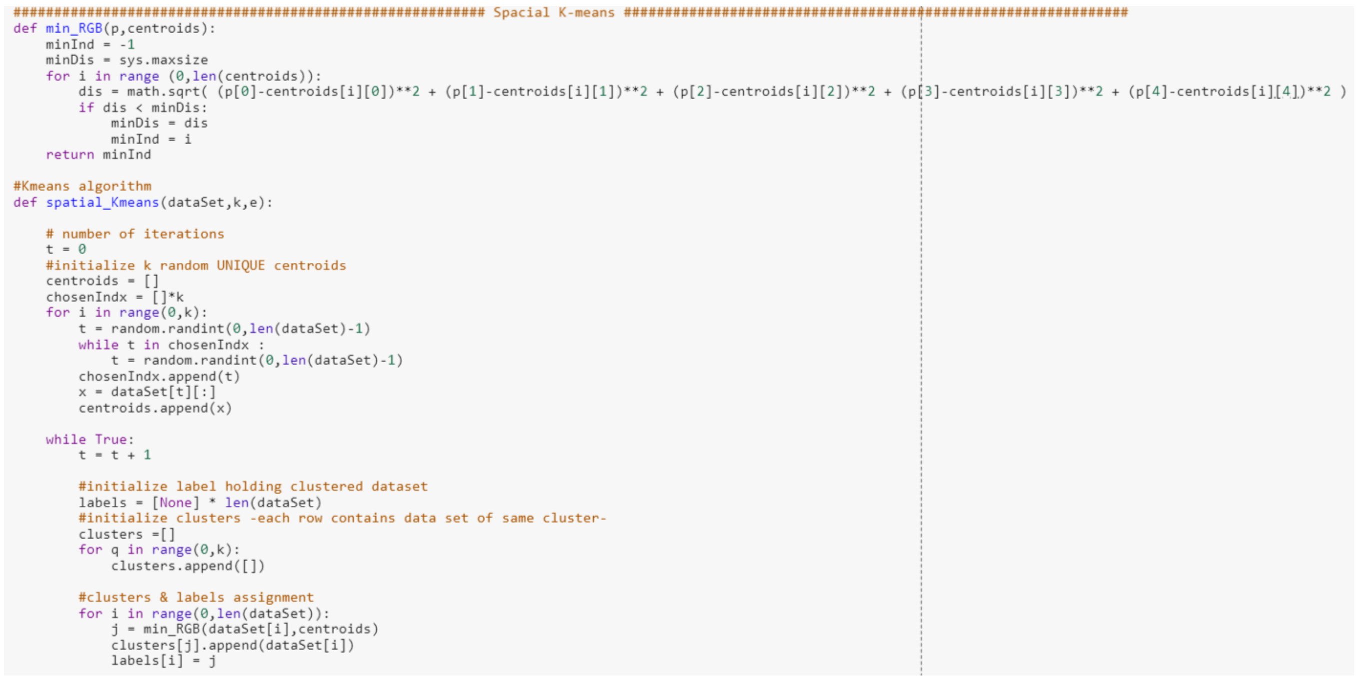


```python
## different vals of k
k = [3,5,7,9,11]

## Arrays definitions
#rows = 200 #of training imgs
rows = 10 #of training imgs
cols = 5   # of vals of k

## 2 x 2-D array holding Entropy & F-measure
  # row -> image i in training set
  # col -> each value j of possible k
  # a[i][j] --> [entropy] or [fmeasure]

kMeans_entropy_results = [[]]
kMeans_entropy_results = [[0 for i in range(cols)] for i in range(rows)]

kMeans_fMeasure_results = [[]]
kMeans_fMeasure_results = [[0 for i in range(cols)] for i in range(rows)]


## 2-D array holding results of K-means
  # row -> image i in training set
  # col -> each value j of possible k
  # a[i][j] --> entropy/fmeasure (minimum is better)

kMeans_Train_results = [[]]
kMeans_Train_results = [[0 for i in range(cols)] for i in range(rows)]

# i -> image
# j -> k val
for i in range(0,rows):
  for j in range (0,cols):

    image = train_dict_image[str(i)]

    kMeans_labels = do_Kmeans(image,k[j])

    avg_condEntropy = AVG_condEntropy(kMeans_labels,train_dict_groundtruth[i],k[j])
    avg_Fmeasure = AVG_Fmeasure(kMeans_labels,train_dict_groundtruth[i],k[j])

    kMeans_entropy_results[i][j]  = avg_condEntropy
    kMeans_fMeasure_results[i][j] = avg_Fmeasure
    kMeans_Train_results[i][j]    = avg_condEntropy / avg_Fmeasure

print("done!")
```

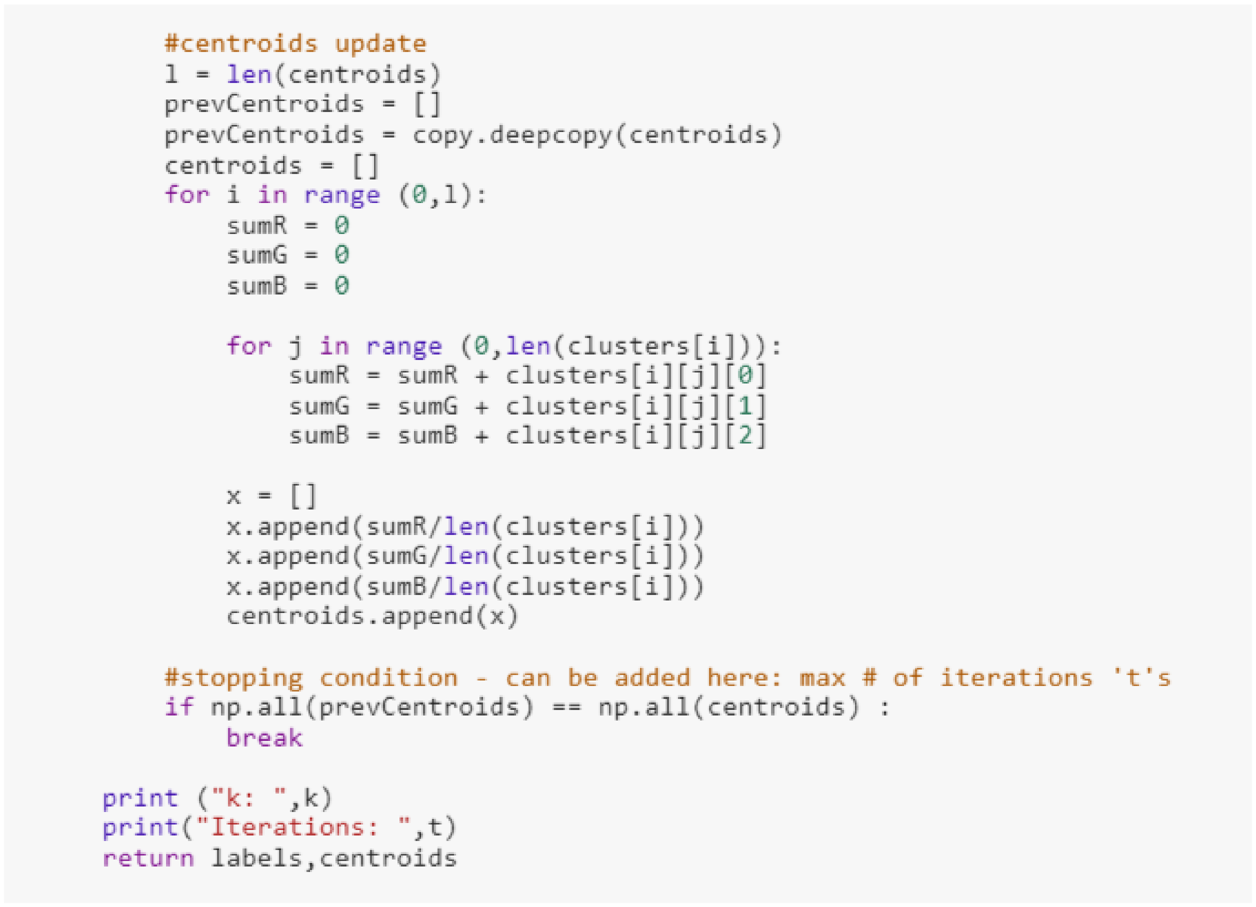


```
Entropy Results
k: 3 Entropy:  1.9148283464606712
k: 5 Entropy:  1.683395038418417
k: 7 Entropy:  1.5396021951774561
k: 9 Entropy:  1.5033250043440773
k: 11 Entropy:  1.3251069882372934
F-measure Results
k: 3 F-measure:  0.9574141732303356
k: 5 F-measure:  0.8416975192092085
k: 7 F-measure:  0.7698010975887281
k: 9 F-measure:  0.7516625021720387
k: 11 F-measure:  0.6625534941186467
Best K values [min 'entropy/fmeaure' value]
K ->  9
```

Bonus - Spatial K-means (Implementation from Scratch)

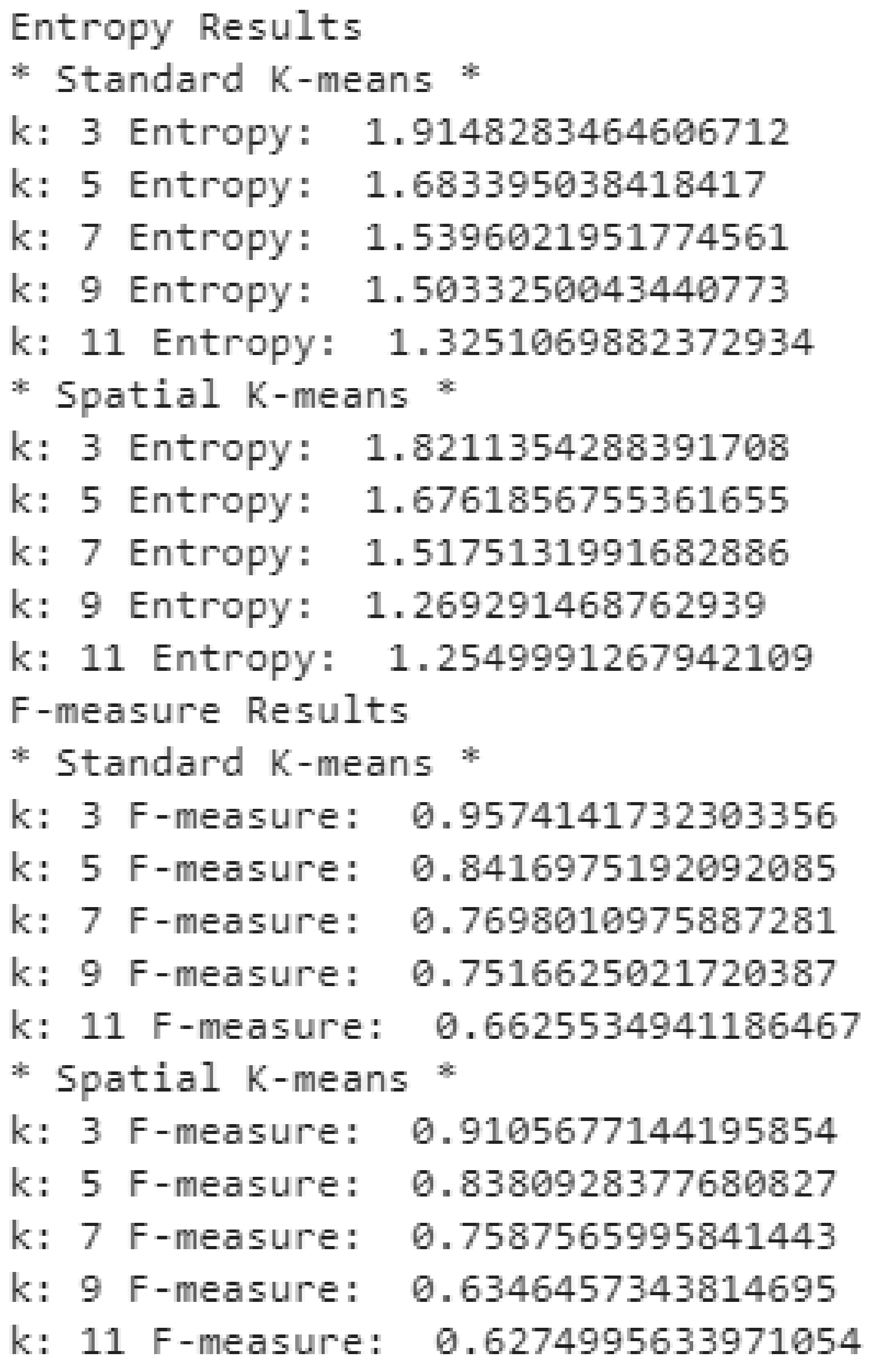


```
########################################################## Spacial K-means ##############################################################
def min_RGB(p,centroids):
    minInd = -1
    minDis = sys.maxsize
    for i in range (0,len(centroids)):
        dis = math.sqrt( (p[0]-centroids[i][0])**2 + (p[1]-centroids[i][1])**2 + (p[2]-centroids[i][2])**2 + (p[3]-centroids[i][3])**2 + (p[4]-centroids[i][4])**2 )
        if dis < minDis:
            minDis = dis
            minInd = i
    return minInd

#Kmeans algorithm
def spatial_Kmeans(dataSet,k,e):

    # number of iterations
    t = 0
    #initialize k random UNIQUE centroids
    centroids = []
    chosenIndx = []*k
    for i in range(0,k):
        t = random.randint(0,len(dataSet)-1)
        while t in chosenIndx :
            t = random.randint(0,len(dataSet)-1)
        chosenIndx.append(t)
        x = dataSet[t][:]
        centroids.append(x)

    while True:
        t = t + 1

        #initialize label holding clustered dataset
        labels = [None] * len(dataSet)
        #initialize clusters -each row contains data set of same cluster-
        clusters =[]
        for q in range(0,k):
            clusters.append([])

        #clusters & labels assignment
        for i in range(0,len(dataSet)):
            j = min_RGB(dataSet[i],centroids)
            clusters[j].append(dataSet[i])
            labels[i] = j
```



```
        #centroids update
        l = len(centroids)
        prevCentroids = []
        prevCentroids = copy.deepcopy(centroids)
        centroids = []
        for i in range (0,l):
            sumR = 0
            sumG = 0
            sumB = 0

            for j in range (0,len(clusters[i])):
                sumR = sumR + clusters[i][j][0]
                sumG = sumG + clusters[i][j][1]
                sumB = sumB + clusters[i][j][2]

            x = []
            x.append(sumR/len(clusters[i]))
            x.append(sumG/len(clusters[i]))
            x.append(sumB/len(clusters[i]))
            centroids.append(x)

        #stopping condition - can be added here: max # of iterations 't's
        if np.all(prevCentroids) == np.all(centroids) :
            break

    print ("k: ",k)
    print("Iterations: ",t)
    return labels,centroids
```

Bonus - Spatial K-means VS K-means



```
Entropy Results
* Standard K-means *
k: 3 Entropy:  1.9148283464606712
k: 5 Entropy:  1.683395038418417
k: 7 Entropy:  1.5396021951774561
k: 9 Entropy:  1.5033250043440773
k: 11 Entropy:  1.3251069882372934
* Spatial K-means *
k: 3 Entropy:  1.8211354288391708
k: 5 Entropy:  1.6761856755361655
k: 7 Entropy:  1.5175131991682886
k: 9 Entropy:  1.269291468762939
k: 11 Entropy:  1.2549991267942109
F-measure Results
* Standard K-means *
k: 3 F-measure:  0.9574141732303356
k: 5 F-measure:  0.8416975192092085
k: 7 F-measure:  0.7698010975887281
k: 9 F-measure:  0.7516625021720387
k: 11 F-measure:  0.6625534941186467
* Spatial K-means *
k: 3 F-measure:  0.9105677144195854
k: 5 F-measure:  0.8380928377680827
k: 7 F-measure:  0.7587565995841443
k: 9 F-measure:  0.6346457343814695
k: 11 F-measure:  0.6274995633971054
```

SOURCE CODE

https://colab.research.google.com/drive/1eIaFmnEzunF5w99a
b6YOlyCNlK6SBCpr#scrollTo=vxQk6s3EH2lh&uniqifier=4

THANK YOU