Computer Vision and Imaging Coursework

Name: Sreekiran Ayiroor Reghunadh
ID: 2306627
Email: sxa1508@student.bham.ac.uk

1. Introduction

Segmentation is one of the most ongoing research areas in the field of Computer Vision. This experiment is trying and comparing different approaches to solving a controlled problem of segmenting different tissues from MRI images. 10 consecutive axial cross-sections of MRI of a single human are given as the input. The objective is to develop different approaches to segmenting these tissue layers from the given images and evaluate their performance based on the ground truth provided. Task 1 is to develop 2d segmentation approaches, Task 2 is to state the evaluation metrics and compare the results of different approaches. Dice/pixelwise_IoU is a good choice to evaluate the segmentation models. Dice coefficient was used in this experiment as it gives the pixel-wise performance of the models. Task3 is to develop a solution based on 3d segmentation, compare its performance with the 2d segmentation, and arrive at an analytical conclusion.

2. 2D image segmentation

2.1 Exploratory Data Analysis:

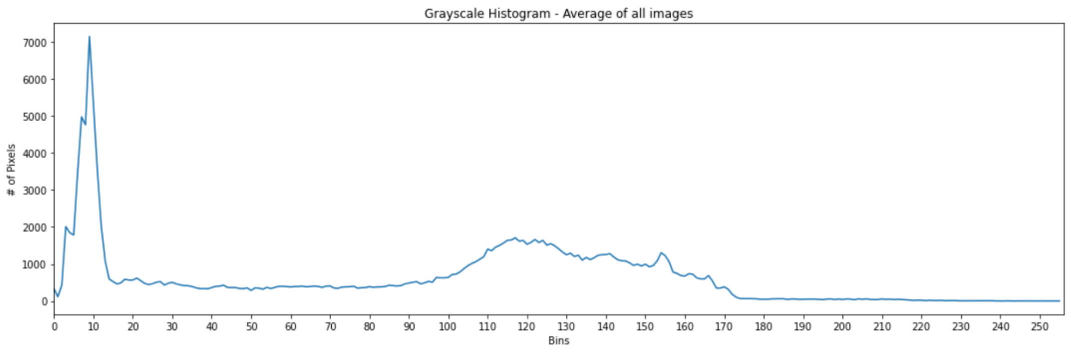

Fig 1: Histogram of the pixel intensities across all images

10 images each of 362x434 resolution are given as input and labels. Images in T1 were in float values and thus the values were standardized between (0-255) and converted to np.unit8 for further processing. Labels to be segmented are {0: air, 1: skin, 2: skull, 3: csf, 4: grey matter, 5: white matter} Plotting the Histogram of a sample image indicated thresholding at multiple levels and combining with the addition of some morphological operations can possibly solve the problem. The thresholded images of threshold range (0,260,10) were analysed for a sample input image and also a Gaussian Blurred image. Gaussian Blurred image provided thresholded images which are closer to the masks required and some suitable threshold values identified. As the data is from the same subject and taken 1mm apart, the threshold values are standard and do not have to spend time on fine-tuning a generic configuration.

2.2 Approach 1: Unsupervised Approach (2D K-Means clustering)

The first approach was using an Unsupervised clustering method called K-means clustering. In this method, the input image is reshaped into an array of one channel (n-channels for n-dimensional K-means) and then the algorithm will initiate random cluster centroids of 6 Labels and It will keep on iterating until the exit condition satisfies or max iterations are reached. Thus the algorithm will return 6 clusters by grouping similar pixels together. This approach performed well in half the classes namely grey matter, white matter and air (more than 80% dice score) and very low in other classes (skin, skull and csf)(less than 20% dice sore). The existence of similar pixels but belonging to different tissues caused this low performance. Hence, a direct unsupervised approach might not work straightforward.

2.3 Approach 2: Class by Class Approach using Thresholding, Morphological Operations & Connected Components

In this approach, a combination of techniques is used to extract each label separately. The approach was labelling from corners to the centre approach. Air was detected first, then skin and so on. This ensured the removal of noises that were detected in regions already marked. The final output is obtained by combining these separate masks. In this algorithm, the threshold values observed during the EDA were utilized to create an initial segmentation. Bitwise operations between other masks, Morphological operations, and Connected components. An interesting logic named Supervised Elliptical Erosion is used to increase the accuracy of the CSF label where the detected mask was accurate in the sides of the brain but needed some erosion in the other 90% of the region. So an elliptical mask is created on the area of interest and then erosion is applied selectively to the region. After some finetuning and experiments, approach 2 yielded a good result of an average 90.0% dice score across all classes. This method can still be finetuned and increased further as there are a lot of hyperparameters involved.

2.4 Approach 3: Combining k-means and Class-by-class

This approach tries to utilise the best out of the first two approaches. Mainly grey matter and white matter were getting detected better using k-means and after the combining, the dice score increased from 0.91 to 0.95 for grey matter and 0.92 to 0.99 for white matter. Some modifications were made to the skin, and skull and overall this achieved the best results among all of the approaches.

3. 3D image segmentation

3.1 Approach 1: k-means 3D segmentation

A modified k-means algorithm takes a 10 channel image and does the clustering and arrives at 6 classes of interest. This method was straightforward, but as experienced in the 2D K-means, the overall performance was low with only one class obtaining more than 80% dice score.

3.2 Approach 2: Grayscaling 10 channel image to 1 channel

In this approach, the 10 channel image is aggregated by mean to form a single channel image and the above 2d thresholding techniques were applied. The ground truth was a mode aggregated single-channel image. The results were similar to the 2d segmentation approach

3.3 Approach 3: Divide and Conquer Method: Channel by Channel approach

Here, we are applying the 2d segmentation techniques to each channel image and then combining them to get the single output. The dice scores were calculated using one more iterative process snippet and the results were identical to the 2d image segmentation. Hence, the 3D image segmentation techniques used above could perform only as good as the 2D image segmentation techniques.

4. Results & Analysis

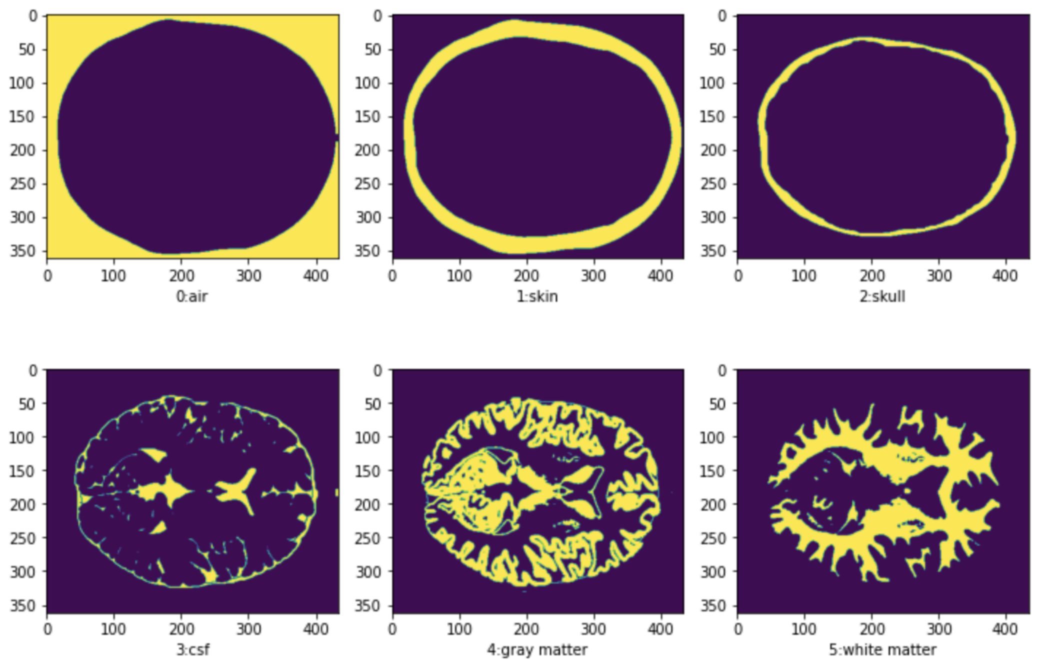

Fig 2: output of the best performing algorithm

method	Avg Dice	air	csf	gray matter	skin	skull	white matter
kmeans2D	53.73%	91.57%	29.26%	82.08%	24.22%	0.00%	95.25%
class_by_class	89.49%	98.54%	81.69%	90.68%	91.45%	82.11%	92.44%
combined	91.45%	98.54%	81.69%	95.45%	91.45%	82.11%	99.43%
kmeans3D	45.51%	79.27%	16.80%	80.65%	72.04%	0.00%	80.65%
Average Channel	88.03%	98.65%	77.11%	90.19%	91.25%	82.28%	88.75%
Channel by Channel	91.45%	98.54%	81.69%	95.45%	91.45%	82.11%	99.43%

Table 1: Tissue wise segmentation Dice scores of different approaches

From Table 1, it is clear that the combined method is the one that outperforms every other one. Dice score was selected as the evaluation metric as it considers both precision and recall. The tissues Gray matter, White Matter and Air were able to get high scores of 90+% because of their unique features. CSF was the most tricky mask and applying a focused erosion with masking a smaller ellipse increased the Dice score from 65% to 81.69%. Skull was a bit tricky with not having a clear boundary on some edges and mixed with skin. Thus, some crossing happened between those class pixels. The 3D segmentation technique could not outperform the 2D solutions. In this experiment with limited data, a 2D solution works well. However, the class by class separation required intensive fine-tuning and hyperparameter tuning. Coming to k-means, 2D k-means was able to contribute to detecting grey matter, white matter, skin and skull whereas k-means 3D performed worse than 2D K-means. As the 2D was performing at a good 91.45% 3D segmentation advanced techniques such as active contours were not implemented.

Overall, In this experiment, a use case that requires the combination of different techniques and finetuning was solved at 91.45% overall accuracy. Rather than using deep learning where the interpretability of the model is limited, using matrix operations, some approaches can be well explained visually to obtain the results. Got the perspective that focusing on one step at a time rather than trying to solve everything can yield better results.

5. Source code:

```
### Importing necessary libraries
import os
import numpy as np
import cv2
import pandas as pd
import matplotlib.pyplot as plt
import time

### Loading matlab files in Python using Scipy
import scipy.io
mat = scipy.io.loadmat('Brain.mat')
T1, label = mat['T1'], mat['label']

### T1 is the array with the input MRI images, 10 samples each of size 362x434 pixels.
T1.shape
images = []
labels = []
for i in range(10):
   img = T1[:,:,i]
   ### standardising the float values between int of range(0-255)
   img = ((img - img.min()) * (1/(img.max() - img.min()) * 255)).astype('uint8')
   images.append(img)
   lab = label[:,:,i]
   labels.append(lab)


input_image = images[5]
label_image = labels[5]

# TASK 2: Evaluating the performance
# Before we move on to different approaches in obtaining the segmentation,
#we will first define the evaluation metrics and get an accuracy measure with the groundtruth and verify
the logics.

def dice_binary(mask1, mask2):
   """function calculate dice scores of two binary images. 2 * (A & B) / (A+B)

   Args:
       mask1 (np.ndarray): mask1 image
       mask2 (np.ndarray): mask2 image

   Returns:
       float : dice score of the given images
   """
   mask1_pos = mask1.astype(np.float32)
   mask2_pos = mask2.astype(np.float32)
   dice = 2 * np.sum(mask1_pos * mask2_pos) / (np.sum(mask1_pos) + np.sum(mask2_pos))
   return dice

def score_image(predict, label):
   """Calculate dice score for each label for the given predict and label image

   Args:
       predict (np.ndarray): predict image
       label (np.ndarray): label image

   Returns:
       DataFrame: data frame with dice scores for each label
   """
   label_index = {0:'air',1:'skin',2:'skull',3:'csf',4:'gray matter',5:'white matter'}
   scores = []
```

```
    for pix_val, category in label_index.items():
        predict_mask = (predict == pix_val)
        label_mask = (label == pix_val)
        dice = dice_binary(predict_mask, label_mask)
        scores.append(dice)
    scores.append(sum(scores)/len(scores))
    df = pd.DataFrame(columns=['air','skin','skull','csf','gray matter','white matter','0_mean'])
    df.loc[0] = scores
    return df

import pandas as pd

def score_images(predicts, labels):
    """Calculate dice for a list of images

    Args:
        predicts (list): list of predicted images
        labels (list): list of label images

    Returns:
        DataFrame: Collated dataframe of dice scores of each catgeory for each image
    """
    df = pd.DataFrame(columns=['air','skin','skull','csf','gray matter','white matter','0_mean'])
    for i, (predict, label) in enumerate(zip(predicts,labels)):
        scores = score_image(predict,label)
        df = pd.concat([df, scores])
    return df
df = score_images(labels,labels)

# TASK 1: MRI Segmentation

## MRI Segementation - Approach 1 : K-means Clustering
def kmeans_segmentation(input_image, no_of_classes = 6):
    """kmeans 2d segmentation

    Args:
        input_image (np.ndarray): input grayscale image
        no_of_classes (int, optional): no of classes to be detected. Defaults to 6.

    Returns:
        np.ndarray: predicted label image
    """
    h, w = input_image.shape
    # reshape to 1D array
    image_2d = input_image.reshape(h*w,1)
    image_2d = np.float32(image_2d)
    ### using cv2 kmeans
    ### In criteria, we first set the algorithm termination criteria, either max no of iterations or desired
result
    ### here max iterations is set to 50, and the max score is 1.0
    criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 50, 1.0)
    K = no_of_classes
    attempts = 100
    ### calling kmeans
    ret, label, center = cv2.kmeans(image_2d,K,None,criteria,attempts,cv2.KMEANS_PP_CENTERS)
    center = np.uint8(center)
    res = center[label.flatten()]
    result_image = res.reshape((input_image.shape))
    ### assigning each cluster to a label based on the range of pixels based on observation
    out_image = np.zeros_like(result_image)
    out_image = np.where(((result_image>45) & (result_image<55)), 1, out_image )
    out_image = np.where(((result_image>75) & (result_image<95)), 3, out_image )
    out_image = np.where(((result_image>115) & (result_image<135)), 4, out_image )
    out_image = np.where(((result_image>145) & (result_image<160)), 5, out_image )
    out_image = np.where(((result_image>200) & (result_image<210)), 2, out_image )
    return out_image

## MRI Segmentation - Approach 2 : Class by class Method: Filtering, Thresholding & Morphological
operations

### function to filter labels with less than 10% area
def filter_noise_label(out_label):
    """filtering small connected components
```

```python
    (To remove noises which has an area<10% of the image)

    Args:
        out_label (np.ndarray): label image

    Returns:
        np.ndarray: noise removed label image
    """
    for i in np.unique(out_label):
        if np.count_nonzero(out_label==i)<(0.1*out_label.shape[0]*out_label.shape[1]):
            out_label = np.where(out_label==i,0,out_label)
    return out_label

### class wise segmentation
def class_wise_segmentation(input_image):
    """class by class segmentation of an mri image. Uses Thresholding, Morphological operations
    connected components and bitwise operations.

    Args:
        input_image (np.ndarray): input image

    Returns:
        list: [out_image,inside_skull, air_mask, inside_skin, skin_mask, skull_mask]
    """
    ### applying a 5x5 Gaussian blur to smoothen the image
    blur_image = cv2.GaussianBlur(input_image.copy(),(5,5),cv2.BORDER_DEFAULT)
    ### creating an empty image with the same shape as input_image
    out_image = np.zeros_like(input_image)
    ### air
    ### thresholding with value of 17 for getting air
    _, air = cv2.threshold(blur_image, 17 , 255, cv2.THRESH_BINARY)
    ### applying a layer of closing and dilation
    kernel = np.ones((5,5),dtype=np.uint8)
    air_mask = ~cv2.morphologyEx(air, cv2.MORPH_CLOSE, kernel,iterations = 1)
    air_mask = cv2.morphologyEx(air_mask, cv2.MORPH_DILATE, kernel, iterations =5)

    ### skin
    ### thresholding with a value of 50
    th, skin = cv2.threshold(blur_image, 50, 255, cv2.THRESH_BINARY)
    ### applying opening
    kernel = np.ones((5,5),dtype=np.uint8)
    dst2_open = cv2.morphologyEx(skin, cv2.MORPH_OPEN, kernel,iterations = 1)
    ### calling connected components function
    retval, out_label, stats, centroids = cv2.connectedComponentsWithStats(dst2_open)
    ### taking only skin
    skin_mask = (out_label==1).astype(np.uint8)
    ### applying dilation
    kernel = np.ones((3,3),dtype=np.uint8)
    skin_mask = cv2.morphologyEx(skin_mask, cv2.MORPH_DILATE, kernel,iterations = 1)
    ### filtering noise labels from the connected components output
    out_label = filter_noise_label(out_label)
    ### masking the non_air parts by 6 to avoid being considered as air
    out_image = np.where(air_mask!=0,0,6)
    ### skin added with value 1 in the out image
    out_image = np.where(skin_mask!=0,1,out_image)

    ## skull
    ### region inside skull is region which is disjoint with the union of the skin and the air mask
    inside_skin = np.zeros_like(out_image)
    inside_skin = np.where(out_image < 5,255,0)
    ### applying dilation
    kernel = np.ones((9,9),dtype=np.uint8)
    inside_skin = cv2.morphologyEx(inside_skin.astype(np.uint8), cv2.MORPH_CLOSE, kernel,iterations = 1)
    ### accessing the brain component with label 2
    brain = np.where(out_label==2,255,0)
    ### applying dilation
    kernel = np.ones((7,7),dtype=np.uint8)
    brain_filled = cv2.morphologyEx(brain.astype(np.uint8), cv2.MORPH_CLOSE, kernel,iterations = 5)
    ### skull is the region between skin and brain
    skull_mask = np.bitwise_xor(~inside_skin, brain_filled)
    ### marking the pixels as skull with 2 in the out_image
    out_image = np.where(skull_mask!=0,2,out_image)
    ### csf
    ### thresholding with 100
    _, csf = cv2.threshold(input_image, 100 , 255, cv2.THRESH_BINARY)
    ### finding region inside skull which is disjoint to the union of the skin,air,and skull masks
    inside_skull = np.zeros_like(out_image)
    inside_skull = np.where(out_image<5, 255, 0).astype(np.uint8)
    ### applying closing
    kernel = np.ones((7,7),dtype=np.uint8)
    inside_skull = cv2.morphologyEx(inside_skull, cv2.MORPH_CLOSE, kernel,iterations = 1)
    ### filtering regions inside skull and the thresholded csf image
    csf = np.bitwise_xor(csf,~inside_skull)
    csf = np.where(out_image>5, csf, 0)

    ### finding the bounding box of the detected brain
    retval, out_label_csf, stats, centroids = cv2.connectedComponentsWithStats(csf)
    x, y, w, h, area = sorted(stats,key = lambda x:x[4],reverse = True)[1]
    ### from the images, we can see that the csf mask obtained need to retain its pixels in the borders
    ### and need to erode the parts inside which is identified as csf
    ### we will use a selective erosion by masking
    # Supervised Eliptical Erosion

    ellipse_mask = np.zeros_like(csf)
    ### finding a large ellipse inside the brain to erode
    cv2.ellipse(ellipse_mask, ((int((x+w+x)/2),int((y+y+h)/2)), (0.95*w,0.95*h), 0.0), (255, 255, 255), -1);
    ### making a mask of the ellipse and copying the contents of csf inside the mask
    csf_centre_thin = np.where(ellipse_mask & csf!=0, csf, 0)
    ### taking the rest of the region outside the ellipse
    csf_rest = np.where(ellipse_mask & csf==0, csf,0)
    ### eroding the content inside the ellipse
    kernel = np.ones((3,3),dtype=np.uint8)
    csf_centre_thin = cv2.morphologyEx(csf_centre_thin, cv2.MORPH_ERODE, kernel,iterations = 1)
    ### merging the newly eroded content inside the ellipse and the original content outside ellipse
    csf = csf_rest + csf_centre_thin

    out_image = np.where(csf!=0, 3, out_image)

    ### gray & white matter
    ### gray matter is directly obtained from thresholding at 140 and removing the non_brain regions
    _, gray_matter = cv2.threshold(blur_image, 140 , 255, cv2.THRESH_BINARY)
    gray_matter = np.where(~inside_skull>0, gray_matter,0)
    out_image = np.where(gray_matter!=0, 5, out_image)
    ### rest of the regions which does not belong to any category is the white matter
    out_image = np.where(out_image>5, 4, out_image)
    return [out_image,inside_skull, air_mask, inside_skin, skin_mask, skull_mask]

### Approach 3: Combining Kmeans and Class_by_Class_segmentation results
def combined_kmeans_and_class_by_class(input_image, masks, kmeans_out_image):
    """combining the best from k-means and class-by-class method.

    Args:
        input_image (np.ndarray): input image
        masks (list): [out_image,inside_skull, air_mask, inside_skin, skin_mask, skull_mask]
        kmeans_out_image (np.ndarray): output from the kmeans

    Returns:
        np.ndarray: combined output image
    """
    ### accesing the intermediate outputs from class_by_class segmentation
    out_image,inside_skull, air_mask, inside_skin, skin_mask, skull_mask = masks
    ### creating new output image, masks for new gray and white matter
    new_image = np.zeros_like(out_image)
    combined_gray_matter = np.zeros_like(out_image)
    combined_white_matter = np.zeros_like(out_image)

    ### taking gray matter from kmeans after filtering the non_brain regions
    gray_matter_kmeans = (kmeans_out_image==4)
    combined_gray_matter = np.where(~inside_skull!=0, gray_matter_kmeans, combined_gray_matter)
    ### taking white matter from kmeans after filtering the non_brain regions
    white_matter_kmeans = (kmeans_out_image==5)
    combined_white_matter = np.where(~inside_skull!=0, white_matter_kmeans, new_image)
    ### labeling gray and white matter
    new_image = np.where(combined_gray_matter!=0, 4, new_image)
    new_image = np.where(combined_white_matter!=0, 5, new_image)
    ### improving skull mask from kmeans
    air_kmeans = (kmeans_out_image==0)
    ### taking the thresholded air_mask
    new_skull_mask= np.bitwise_or(air_kmeans,air_mask)
    new_skull_mask = np.bitwise_and(new_skull_mask,~inside_skin)
    ### applying dilation
    kernel = np.ones((3,3),dtype=np.uint8)
    new_skull_mask = cv2.morphologyEx(new_skull_mask, cv2.MORPH_DILATE, kernel,iterations = 1)
    ### now the skull obtained from supervised approach had no gaps, the gaps are actually part of skin
    skin_gaps = skull_mask - new_skull_mask*255
    ### adding the skin gaps to the detected skin from approach 2
    skin_gaps = np.where(skin_gaps>0, 1,0)
    new_skin_mask = skin_mask | skin_gaps
    ### adding labels to the out image
    new_image = np.where(air_mask!=0,0,new_image)
    new_image = np.where(new_skin_mask!=0, 1, new_image)
    new_image = np.where(new_skull_mask!=0, 2, new_image)
    new_image = np.where(out_image==3, 3, new_image)
    new_image = np.where(((~inside_skin!=0)&(new_image==0)), 4, out_image)
    new_image = np.where(combined_white_matter!=0, 5, new_image)
    return new_image

### Scoring for all 10 images
predicts_class_by_class = []
predicts_kmeans = []
predicts_combined = []
for i, image in enumerate(images):
    masks = class_wise_segmentation(image)
    kmeans_out = kmeans_segmentation(image)
    combined_out = combined_kmeans_and_class_by_class(image, masks, kmeans_out)
    predicts_class_by_class.append(masks[0])
    predicts_kmeans.append(kmeans_out)
    predicts_combined.append(combined_out)

df_class_by_class = score_images(predicts_class_by_class,labels)
df_kmeans = score_images(predicts_kmeans,labels)
df_combined = score_images(predicts_combined,labels)
df_1 = df_class_by_class.mean(axis=0).reset_index(name = 'score')
df_1['method'] = 'class_by_class'
df_2 = df_kmeans.mean(axis=0).reset_index(name = 'score')
df_2['method'] = 'kmeans'
df_3 = df_combined.mean(axis=0).reset_index(name = 'score')
df_3['method'] = 'combined'
df = pd.concat([df_1,df_2,df_3],axis=0)
df.rename(columns = {'index':'label'},inplace=True)
out = pd.pivot_table(df,values='score',index='method',columns='label')

## 3D segmentation
### Approach 1: Kmeans 3D
def kmeans3d_segmentation(input_image):
    """K means segmentation 3D.

    Args:
        input_image (np.ndarray): input image

    Returns:
        np.ndarray: kmeans output image
    """
#     input_image = np.asarray(images).reshape((362,434,10))
    h, w, c = input_image.shape
    ### same as 2d, but adding 10 channels
    # reshape to 1D array of 10 channels
    image_3d = input_image.reshape(h*w,c)

    image_3d = np.float32(image_3d)

    ### using cv2 kmeans
    ### In criteria, we first set the algorithm termination criteria, either max no of iterations or desired
result
    criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 10, 1.0)
    K = 6
    attempts = 10
```

```
   ret,label,center = cv2.kmeans(image_3d,K,None,criteria,attempts,cv2.KMEANS_PP_CENTERS)
   center = np.uint8(center)
   res = center[label.flatten()]
   result_image = res.reshape((input_image.shape))
   ### assigning each cluster to a label based on the range of pixels
   out_image = np.zeros_like(result_image)
   out_image = np.where(((result_image>45) & (result_image<55)), 1, out_image )
   out_image = np.where(((result_image>75) & (result_image<95)), 3, out_image )
   out_image = np.where(((result_image>115) & (result_image<135)), 4, out_image )
   out_image = np.where(((result_image>145) & (result_image<160)), 5, out_image )
   out_image = np.where(((result_image>200) & (result_image<210)), 2, out_image )

   return out_image

predicts = kmeans3d_segmentation(images)

### Grayscaling to a single image and processing
# Considering the 10 images are in 3d, we are converting the image to a
# grayscale image by averaging all channels, and processing through the 2d
# segmentation algorithm. The ground truth will be taken with the pixel_by_pixel mode value.

### taking mean of the 10 images to get the single channel image
images = np.asarray(images)
gray_scale = np.mean(images,axis=0).astype(np.uint8)
### taking the mode across the 10 images for each pixel to arrive at a single label image
mode_label = np.apply_along_axis(lambda x: np.bincount(x).argmax(), axis=0, arr=labels)
### calling three algorithms
masks = class_wise_segmentation(gray_scale)
kmeans_out = kmeans_segmentation(gray_scale)
combined_out = combined_kmeans_and_class_by_class(gray_scale, masks, kmeans_out)
print('Dice scores using combined kmeans + class_by_class method: \n')
out = score_image(combined_out,mode_label)
print(out)
## third approach is essentially processing each channel through the 2d segemntation and combining the
results
images = np.asarray(images)
outs = np.zeros_like(images)
for i,image in enumerate(images):
   masks = class_wise_segmentation(image)
   outs[i,:,:] = masks[0]
### calculating the dice will be essentially calculating the dice of each channel and averaging
df = score_images(outs, labels)
### which gives the same result as 2d segmentation as expected
```