EE588 Advanced Image Processing
Project #1 Image Segmentation
(due Monday, Sep. 10, 2018, 10:30:00 am)

Matlab commands are specified in the text and in `blue Courier font` and python commands in parenthesis and in `red Courier font`; variables and filenames (or commands that are consistent between Matlab and python) are specified in `black Courier font`. In the following python syntax, I have used `import matplotlib.pyplot as plt`, `import numpy as np`, `import os`, `import glob`, `import imageio`, `import skimage.color`, `import skimage.filters`, and `import skimage.measure`, `import skimage.morphology`, and `import skimage.feature`; additionally, you probably want to use `%matplotlib inline` to include figures inline in your jupyter notebook. Additionally, for python users, I recommend using the image I/O functions in `imageio` as they have caused me the least heartburn of other options.

We will be using the Berkeley Segmentation Dataset (BSDS). Familiarize yourself with the overview of the dataset at https://www2.eecs.berkeley.edu/Research/Projects/CS/vision/bsds/. Of particular note is the original conference paper describing the dataset and the segmentation metrics that we will use in this project https://www2.eecs.berkeley.edu/Research/Projects/CS/vision/grouping/papers/mftm-iccv01.pdf (and `martin2001_BSDS.pdf` on canvas), the description of the segmentation (ground truth) file format at https://www2.eecs.berkeley.edu/Research/Projects/CS/vision/bsds/seg-format.txt, and the links to download the images and the human segmentations (ground truths) at https://www2.eecs.berkeley.edu/Research/Projects/CS/vision/bsds/BSDS300-images.tgz and https://www2.eecs.berkeley.edu/Research/Projects/CS/vision/bsds/BSDS300-human.tgz, respectively. Download both the images and human segmentations and extract those files on your computer. You notice that the images are included in two basic directories `BSDS300/images/train` and `BSDS300/images/test` while the ground truths in `BSDS300/human` are separated by a numeric identifier associated with which human generated that segmentation. Dr. Boucheron has kindly done some leg work for you and sorted out one human generated segmentation for each dataset image so that you can have an easy correspondence. You can download this image sorted ground truth from canvas (`BSDS_GT_image_sorted.tgz`) and extract it from the same directory where you extracted the other `.tgz` files and it will extract to a directory `BSDS300/ground_truth`. Underneath that directory are `gray/` and `color/` for the human segmentations done on gray and color images, respectively. Underneath those directories are `train/` and `test/`.

(a) **Generate GT image masks.**

- After carefully reading the description of the segmentation file format (https://www2.eecs.berkeley.edu/Research/Projects/CS/vision/bsds/seg-format.txt), write a function `make_mask(seg_filename,mask_filename)` that creates image masks of the ground truth. The function `make_mask` should write out the image masks to filename `mask_filename` to be read in later; as a suggestion, you may want to create folders `masks/color/test`, `masks/color/train`, `masks/gray/test`, and `masks/gray/train` directly underneath the directory where you extracted the BSDS and same masks with the same base filename ID as the original image. Note—you will want to save to an image format that does not compress, otherwise, you will introduce compression artifacts. I found `.png` to work well for me. The mask created should be the same size as the image (481x321 or 321x481) wherein each pixel is assigned a number corresponding to the “segment number” from the segmentation file. If you were to view your mask image using `imagesc` (`plt.imshow`), you should see regions with different intensities.
- Loop over the 200 training images and 100 test images and create masks for each. You can loop over filenames in a directory with the following code

```
image_directory = “path_to_BSDS_images”; % with trailing /
seg_directory = “path_to_BSDS_segs”; # with trailing /
image_filenames = dir([image_directory,'*.jpg']);
seg_filenames = dir([seg_directory,'*.seg']);
for f=1:length(image_filenames)
    image_filename = image_filenames(f).name;
    seg_filename = seg_filenames(f).name;
```

```
end
```
(
```
image_directory = 'path_to_BSDS_images' # with trailing /
seg_directory = 'path_to_BSDS_segs' # with trailing /
image_filenames = sorted(glob.glob(image_directory+'*.jpg'))
seg_filenames = sorted(glob.glob(seg_directory+'*.seg'))
for f,image_filename in enumerate(image_filenames):
    seg_filename = seg_filenames[f]
```
)

at which point `f` will be a handy index and `image_filename` and `seg_filename` contain a string with the image filename or segmentation filename, respectively. Notes: 1) For Matlab, you will need to prepend the path to filename in question, e.g., `[image_directory,image_filename]` (for python, the filenames will have the paths included and can be removed with `os.path.basename(image_filename)` if desired). 2) The above code is not robust—it assumes the image and segmentation filenames are in the same order (so that the `f`-th image filename corresponds to the `f`-th segmentation filename); more robust code can be defined, but is out of the scope of this project.

- Display the image `100075.jpg`, the segmentation mask generated from the human color segmentation, and the segmentation mask generated from the human grayscale segmentation.

(b) **Create GCE/LCE function.**

- After carefully reading the definition of the global consistency error and local consistency error in `martin2001_BSDS.pdf`, write a function `[GCE,LCE]=compute_GCE_LCE_loopy(Seg,GT)` (`[GCE,LCE]=compute_GCE_LCE_loopy(Seg,GT)`) where `GCE` and `LCE` are the global consistency error and the local consistency error, respectively for the given segmentation `Seg` and ground truth `GT`. The way the equation is defined in `martin2001_BSDS.pdf`, you will loop over all pixels in the image. This will not be an efficient implementation, but is often the best place to start to make sure you have the correct logic. Vectorization can be done later.
- Using image `100075.jpg`, read in the image `I=double(imread('100075.jpg'))` (`I=imageio.imread('100075.jpg')`). Convert the image `I` to grayscale using the equation `I_gray=(0.2125*I(:,:,1)+0.7154*I(:,:,2)+0.0721*I(:,:,3))/255` (`I_gray=skimage.color.rgb2gray(I)`). Threshold the image `I_gray` at a value of `127/255` (`127/255.`), setting the lighter pixels to a value of `1`. Display this thresholded image.
- You will note that this thresholding has resulted in many disconnected regions of white. We will treat each of the connected components in this thresholded image as a separate region (or "segment" in the terminology of the BSDS). Use the command `bwlabel` (`skimage.measure.label`) to assign a different integer number to each of the connected components in your thresholded image, and call this labeled image `Seg`. Note—while both the GT mask `GT` and your labeled image `Seg` have integers assigned to different segments, there is not necessarily a correspondence (i.e., segment '3' is not necessarily the same entity in your GT mask and segmentation). Display this labeled image `Seg` using `imagesc` (`plt.imshow`).
- Use your function `compute_GCE_LCE_loopy` to compute the `GCE` and `LCE` for your labeled image `Seg` and the GT mask from `masks/gray/` and print out your values for `GCE` and `LCE`. You should get values of (`0.2864`, `0.1561`) for (`GCE`, `LCE`), respectively. My `compute_GCT_LCE_loopy` (python) code took a few minutes to compute these values for the `100075.jpg` image.
- You can avoid looping over all pixels and speed up the computation significantly. I have uploaded obfuscated code in `compute_GCE_LCE.m` (`compute_GCE_LCE.py`) which runs in a few seconds for the same `100075.jpg` image. Include `compute_GCE_LCE.m` in your working directory (include `compute_GCE_LCE.py` in your working directory and import with `from compute_GCE_LCE import compute_GCE_LCE`) and call with the basic syntax `[GCE,LCE]=compute_GCE_LCE(Seg,GT)` (`GCE,LCE=compute_GCE_LCE(Seg,GT)`). You should get the same values (`0.2864`, `0.1561`) for (`GCE`, `LCE`), respectively, using this code.
- I will give 10% extra credit if you write your own computationally efficient code to compute the GCE and LCE values without looping over each pixel. Please do not work on this at the expense of the rest of the project.

(c) **Evaluate Otsu thresholding.**

- In this part, you will loop over the 100 test images in `BSDS300/images/test` and compute the performance of the Otsu threshold for image segmentation. There are slight differences in implementation of the commands for converting images to grayscale using `rgb2gray` (`skimage.color.rgb2gray`) and in computing the Otsu threshold using `graythresh` (`skimage.filters.threshold_otsu`). In light of keeping you familiar with the most common functions in each language, the numerical results for Matlab versus python will be slightly different from here on out.

- As a check, using our now favorite image `100075.jpg`, read in the image `I=imread('100075.jpg')` (`I=imageio.imread('100075.jpg')`). Convert the image `I` to grayscale using `I_gray=rgb2gray(I)` (`I_gray=skimage.color.rgb2gray(I)`). Calculate the Otsu threshold for this image `T=graythresh(I_gray)` (`T=skimage.filters.threshold_otsu(I_gray)`). You should get `0.4196` (`0.4133`). Note that Matlab returns a threshold scaled to the range [0,1] even though your image I and I_gray are in the range [0,255]; this will be important when you apply this threshold to your image.
- Since there are no parameters in the Otsu method available to tune, we don't need to use the `BSDS300/images/train` images. You will compute the GCE and LCE value for each of the 100 images in `BSDS/images/test`, comparing to the masks in `masks/gray/test` and `masks/color/test`. You will thus create four length-100 vectors `GCE_gray`, `LCE_gray`, `GCE_color`, and `LCE_color`. Within your loop over the 100 images, your code will likely be structured something like the following pseudocode:
    - Read in each image `I`
    - Convert the image `I` to grayscale as described above compute, yielding `I_gray`
    - Compute the Otsu threshold `T`
    - Threshold the image using the Otsu threshold, yielding `I_thresh`
    - Assign each connected component in `I_thresh` to a segment using `bwlabel` (`skimage.measure.label`) yielding `Seg`
    - Read in the corresponding `masks/gray/` mask to `GT_gray`
    - Read in the corresponding `masks/color/` mask to `GT_color`
    - Call `compute_GCE_LCE(Seg,GT_gray)` and store the outputs to the `GCE_gray` and `LCE_gray` vectors
    - Call `compute_GCE_LCE(Seg,GT_color)` and store the outputs to the `GCE_color` and `LCE_color` vectors
- Display the average and standard deviation of the `GCE_gray`, `LCE_gray`, `GCE_color`, and `LCE_color`. What do these numbers tell you about the performance of the Otsu algorithm compared to the human segmentations?
- Display the image and segmented image that had the worst segmentation performance (according to any of the four measures `GCE_gray`, `LCE_gray`, `GCE_color`, and `LCE_color`).
- Display the image and segmented image that had the best segmentation performance (again according to any of the four measures `GCE_gray, LCE_gray, GCE_color, LCE_color).`

(d) **Evaluate Canny edge detection.**

- In this part, you will loop over the 100 test images in `BSDS300/images/test` and compute the performance of the Canny edge detection for image segmentation.
- We have an interesting problem in interpreting the Canny edge detector since it returns edges rather than regions. We have two choices: 1) we can try to wrangle the output of the Canny edge detector to result in closed contours which we can convert to edges or 2) we can wrangle the GT to output edges. We'll take the approach of the latter. We will then use the GCE and LCE to compare those two edgy outputs. Code to compute the labeled boundaries of the ground truth masks:

```
GT_b = zeros(size(GT));
for s = 0:max(max(GT))
    current_b = (GT==s) – imerode((GT==s),strel('disk',1));
    GT_b = GT_b | current_b;
end
GT_b = bwlabel(GT_b);
```

(
```
GT_b = np.zeros(GT.shape,'uint8')
for s in range(0,GT.max()):
```

```
    current_b = np.asarray((GT==s),'uint8') -\
                skimage.morphology.binary_erosion(GT==s,\
                skimage.morphology.disk(1))
    GT_b = GT_b | current_b
GT_b[0,:] = 0; GT_b[-1,:] = 0; GT_b[:,0] = 0; GT_b[:,-1] = 0
GT_b = skimage.measure.label(GT_b))
```

- We will use the default parameters (the upper and lower thresholds for the Canny edge detector) and thus don't need to use the `BSDS300/images/train` images. You will compute the GCE and LCE value for each of the 100 images in `BSDS/images/test`, comparing to the masks in `masks/gray/test` and `masks/color/test`. You will again create four length-100 vectors `GCE_gray`, `LCE_gray`, `GCE_color`, and `LCE_color`. Within your loop over the 100 images, your code will likely be structured something like the following pseudocode:
  - Read in each image `I`
  - Convert the image `I` to grayscale as described above compute, yielding `I_gray`
  - Compute the image edges using `I_canny=edge(I_gray,'canny')` (`I_canny=skimage.feature.canny(I_gray)`)
  - Assign each connected component in `I_canny` to a segment using `bwlabel` (`skimage.measure.label`) yielding `Seg`
  - Read in the corresponding `masks/gray/` mask to `GT_gray`
  - Read in the corresponding `masks/color/` mask to `GT_color`
  - Call `compute_GCE_LCE(Seg,GT_gray)` and store the outputs to the `GCE_gray` and `LCE_gray` vectors
  - Call `compute_GCE_LCE(Seg,GT_color)` and store the outputs to the `GCE_color` and `LCE_color` vectors
- Display the average and standard deviation of the `GCE_gray`, `LCE_gray`, `GCE_color`, and `LCE_color`. What do these numbers tell you about the performance of the Canny algorithm compared to the human segmentations?
- Display the image and segmented image that had the worst segmentation performance (according to any of the four measures `GCE_gray`, `LCE_gray`, `GCE_color`, and `LCE_color`).
- Display the image and segmented image that had the best segmentation performance (again according to any of the four measures `GCE_gray`, `LCE_gray`, `GCE_color`, `LCE_color`).
- Briefly discuss the implications of comparing only edges for quantifying segmentation performance in this case as compared to using regions in the Otsu case. Do you think the GCE and LCE metrics are valid and/or accurate for this scenario?

(e) **Evaluate active contours.**

- In this part, you will use the 200 training images in `BSDS300/images/train` to tune performance of the active contours without edges (ACWE) algorithm and the 100 test images in `BSDS300/images/test` to compute the performance.
- We need to decide how to specify the initial contour for the active contours algorithm. It is common practice to use a threshold to initialize the algorithm so as to put the initial contour relatively close to the final location. In this project, we will use the Otsu threshold to intialize the active contours algorithm. We also need to determine a reasonable number of iterations for the method to properly converge.
- Using our now favorite image `100075.jpg`, read in the image `I`, convert to grayscale `I_gray`, and threshold at the Otsu threshold `T`, yielding `I_init`. Compute the ACWE segmentation using `I_acwe=activecontour(I_gray,I_init,NumIts,'Chan-Vese','SmoothFactor',0.25)` (`I_acwe=skimage.segmentation.chan_vese(I_gray,mu=smoothval,max_iter=NumIts,init_level_set=I_init)`) for `NumIts=1:250:1000` (`NumIts=np.arange(1,1100,250)`). This will give you an idea of how the contour is evolving over iterations. Display the resultant segmentation for each of these four segmentations.
- We will next look at the effect of the smoothing parameter `'SmoothFactor'` (`mu`) on the performance of the resultant segmentations.
- You will loop over the range of smoothing parameters `0:0.1:1`, (`np.arange(0,0.055,0.005`), compute the ACWE segmentation and GCE, LCE value for each of the 200 images in `BSDS/images/`

`train`, comparing to the masks in `masks/gray/train` and `masks/color/train`. You will create four 200x10 arrays `GCE_gray_train`, `LCE_gray_train`, `GCE_color_train`, and `LCE_color_train` to store the results for each image and each smoothing parameter value. Within your loop over the 200 images, your code will likely be structured something like the following pseudocode (note that you will also have an outer loop over the range of smoothing parameters):

  - Read in each image `I`
  - Convert the image `I` to grayscale as described above compute, yielding `I_gray`
  - Compute the contour initialization `I_init=I_gray>T` by thresholding with the Otsu threshold `T`
  - Compute the ACWE segmentation using `I_acwe=activecontour(I_gray,I_init,NumIts,'Chan-Vese','SmoothFactor',smoothval)` (`I_acwe=skimage.segmentation.chan_vese(I_gray,mu=smoothval,max_iter=NumIts,init_level_set=I_init)`)
  - Assign each connected component in `I_acwe` to a segment using `bwlabel` (`skimage.measure.label`) yielding `Seg`
  - Read in the corresponding `masks/gray/` mask to `GT_gray`
  - Read in the corresponding `masks/color/` mask to `GT_color`
  - Call `compute_GCE_LCE(Seg,GT_gray)` and store the outputs to the `GCE_gray_train` and `LCE_gray_train` arrays
  - Call `compute_GCE_LCE(Seg,GT_color)` and store the outputs to the `GCE_color_train` and `LCE_color_train` arrays

- Display the average and standard deviation of the `GCE_gray_train`, `LCE_gray_train`, `GCE_color_train`, and `LCE_color_train` for each of the smoothing parameters studied. What value for the smoothing parameter `'SmoothFactor'` (`mu`) would you choose as the "best"? Why?
- For the value of `'SmoothFactor'` (`mu`) that you determined to be the best, compute the segmentation for the 100 test images in `BSDS/images/test` and their corresponding `GCE_gray`, `LCE_gray`, `GCE_color`, and `LCE_color` metrics. Display the average and standard deviation of these four measures. What do these numbers tell you about the performance of the ACWE algorithm compared to the human segmentations?
- Display the image and segmented image that had the worst segmentation performance (according to any of the four measures `GCE_gray`, `LCE_gray`, `GCE_color`, and `LCE_color`).
- Display the image and segmented image that had the best segmentation performance (again according to any of the four measures `GCE_gray`, `LCE_gray`, `GCE_color`, `LCE_color`).

(f) **Summary.**

- Provide a summary of what you learned about image segmentation and the process of quantifying image segmentation.
- Provide a summary of the highlights of the results obtained for the various image segmentations. What algorithms would you claim is "best"? "Worst"? Looking qualitatively at the best segmentations, do you feel they are doing a particularly good job?
- What do you think is the biggest issue affecting the segmentation performance? How might you address that issue?
- This summary should be lengthy enough to get your point across (probably 1-2 paragraphs), but not long for the sake of length. Quantity does not equal quality here...

EE588 Advanced Image Processing
Project #1 Image Segmentation
(due Monday, Sep. 10, 2018, 10:30:00 am)

Confidence (number 0 to 100% about your confidence in your performance on this project and optional statement about areas where you are particularly confidence or not):

Difficulty (number 0 to 100%):

Time Spent (hours):

Problem	**Points**
(a) GT Masks	/20
(b) GCE/LCE function	/20
(c) Otsu	/15
(d) Canny	/15
(e) ACWE	/20
(f) Summary	/10
TOTAL	/100