CMSC 733, Computer Processing of Pictorial Information
Homework 0: Alohomora!
Due on: 11:59:59PM on Wednesday, Jan 29 2016

Prof. Yiannis Aloimonos,
Nitin J. Sanket

December 25, 2016

First of all, welcome to CMSC 733. The course website is up and running and can be found here: https://www.cs.umd.edu/class/spring2017/cmsc733/. The course announcements will be made through Piazza and the link can be found here: https://www.piazza.com/umd/spring2017/cmsc733/home. We sent out Piazza invitations to everyone enrolled in the course. In case you did not get it please add yourself.

For those who are not familiar with Harry Potter, *Alohomora* is the spell used to open doors. This homework is aimed at giving you an opportunity to judge the amount of workload of this course. This homework is designed as a combination of CMSC426's first homework and first project. **We would recommend dropping this course if you are not able to complete this homework within the deadline.** This homework will make sure that you come equiped with the right skillset, both in terms of theory and coding expertise to plough through this course comfortably.

This homework is broken into 2 parts, the first part (section) teaches you basic color segmentation and simple image handling functionality in MATLAB. The second part (section) teaches you about boundary detection.

1 Pin It!

You are given an image of colored objects a white background (Check file named `TestImgResized.jpg`). Your task is to segment out the objects, count the number of colored objects and also count the objects of the same color, i.e., green, blue, yellow and red. To have some fun, we also threw in a white object and a transparent object (we really want you to try to get these as well to make your fundamentals stronger).

1.1 Functions you are allowed to use for this part

Any built-in Matlab function except the colorThresholder App (http://www.mathworks.com/help/images/ref/colorthresholder-app.html). If you have a doubt whether a function can be used or not e-mail Nitin at nitinsan@terpmail.umd.edu.

1.2 Various Steps Involved

1.2.1 Denoise Images - 10Pts

You can use any denoising filter like a gaussian or a median filter to 'smooth out' the image to reduce noise.

1.2.2 Find total number of colored objects (excluding white and transparent pin) - 70Pts

You can use a combination of morphological operations and blob based properties (regionprops) to do this.

1.2.3 Find individual colored objects - Red, Green, Blue and Yellow - 20pts

Count and find individually red, green, blue and yellow objects. Use color information in any color space you want in conjunction with the previous step output to do this.

1.2.4 EXTRA CREDIT: Detect the white and transparent colored pins - 20pts

Do anything you want to find this. (If possible, avoid hard-coding the thresholds).

1.2.5 EXTRA CREDIT: Implement a simple 1D Gaussian to detect colored pins - 20pts

Follow points from the gaussian tutorial. You'll need a 1D gaussian for each color.

1.2.6 EXTRA CREDIT: Implement a 3D Gaussian to detect colored pins - 20pts

Follow points from the gaussian tutorial. You'll need a 3D gaussian for each color.

1.2.7 EXTRA CREDIT: Implement a Gaussian Mixture model to detect any one of the colored pin - 100pts

Follow points from the gaussian tutorial. You'll need multiple 1D/3D gaussians for each color.

2 Shake my Boundary!

Boundary detection is an important, well-studied computer vision problem. Clearly it would be nice to have algorithms which know where one object stops and another starts. But boundary detection from a single image is fundamentally difficult. Determining boundaries could require object-specific reasoning, arguably making the task "vision hard".

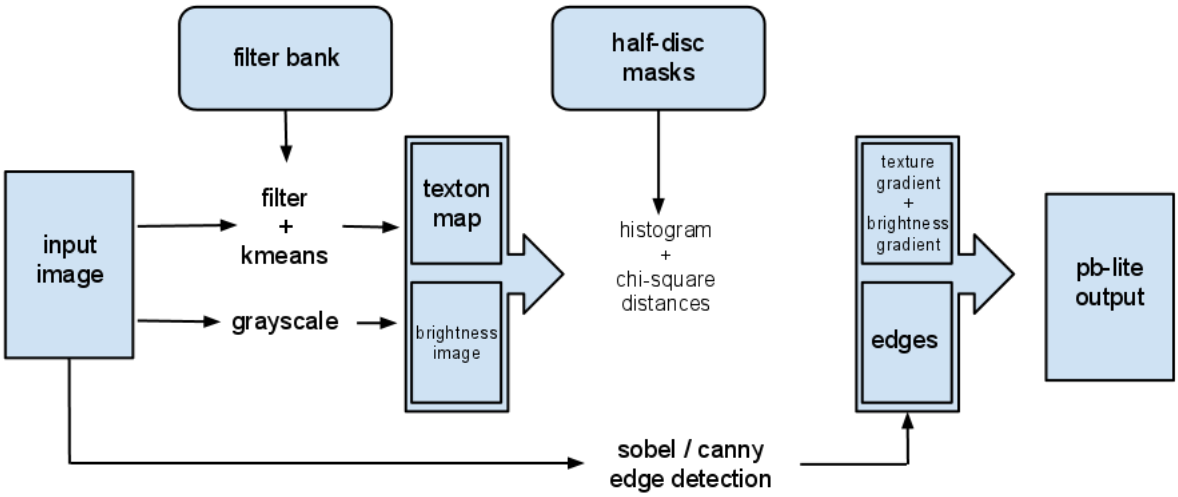
Classical edge detection algorithms, including the Canny and Sobel baselines we will compare against, look for intensity discontinuities. The more recent `pb` (probability of boundary) boundary detectors significantly outperform these classical methods by considering texture and color gradients in addition to intensity. Qualitatively, much of this performance jump comes from the ability of the `pb` algorithm to suppress false positives that the classical methods produce in textured regions.

In this homework, you will develop a simplified version of `pb`, which finds boundaries by examining brightness, color, and texture information across multiple scales. The output of your algorithm will be a per-pixel probability of boundary. Several papers from Berkeley describe their algorithms and how their methods evolved over time. Their source code is also available for reference (don't use it). Here we investigate a simplified version of the recent work from Ref. [1]. Our simplified boundary detector will still significantly outperform the well regarded Canny edge detector. Evaluation is carried out against human annotations (ground truth) from a subset of the Berkeley Segmentation Data Set 500 (BSDS500).

2.1 Overview

The main steps in Ref. [1] are:

- Low-level feature extraction: (1) brightness, (2) color, and (3) textons
- Multiscale cue combination with non-maximum suppression
- Spectral clustering



The focus of `pb-lite` will be on the representation of brightness, color and texture gradients (bullet point 1). This is covered (briefly) in Section 3.1 of Ref. [1]. We will trivialize the multi-scale cue combination. We will get a simple form of non-maximum suppression by combining our `pb-lite` estimates with a classical edge detector. We will skip spectral clustering. You will implement the following pipeline:

The major steps are to pre-define:

- a filter bank of multiple scales and orientations.
- half-disc masks of multiple scales and orientations.

And then for every image:

- create a texton map by filtering and clustering the responses with kmeans.
- compute per pixel texture gradient (`tg`) and brightness gradient (`bg`) by comparing local half-disc distributions.
- output a per-pixel boundary score based on the magnitude of these gradients combined with a baseline edge detector (Canny or Sobel).

Finally the output for all of the evaluate using the the Berkeley Segmentation Data Set 500 (BSDS500) (code will be given soon).

2.2 Texture Representation

The key distinguishing element between `pb-lite` and classical edge detection is the ability to measure texture gradients in addition to intensity gradients. The texture gradient at any pixel should summarize how quickly the texture is changing at that point. The key technical challenges are how to represent local texture distributions and how to measure distances

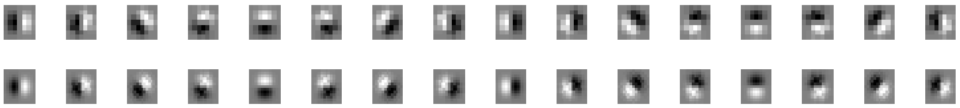


Figure 1: Oriented First Derivative Gaussian Filter Bank

between them. As in `pb`, we will represent texture as a local distribution of textons, where textons are discrete texture elements generated by clustering filter bank responses. Texture and brightness gradients will be measured by comparing the distributions of textons or brightnesses within half-discs centered on a pixel of interest.

2.2.1 Filter Bank - 25Pts

Filtering is at the heart of building the low level features we are interested in. We will use filtering both to measure texture properties and to aggregate regional texture and brightness distributions. A simple but effective filter bank is a collection of oriented derivative of Gaussian filters. These filters can be created by convolving a simple Sobel filter and a Gaussian kernel and then rotating the result. Suppose we want o orientations(from 0 to 360°) and s scales, we should end up with a total of $s \times o$ filters. A sample filter bank of size 2×16 is shown in Fig. 1.

2.2.2 Half-disc Masks - 25Pts

The half-disc masks are simply (pairs of) binary images of half-discs. This is very important because it will allow us to compute the chi-square distances using a filtering operation, which is much faster than looping over each pixel neighborhood and aggregating counts for histograms. Once you get the above filter bank right, forming the masks will be trivial. A sample set of masks (8 orientations, 3 scales) is shown in Fig. 2.

The filter banks and masks only need to be defined once and then they will be used on all images. You have some discretion to experiment with different filter banks and masks.

Hint: some useful MATLAB functions include: imrotate, conv2, imfilter, cell, reshape.

2.2.3 Generating a Texton Map

Filtering an input image with each element of your filter bank results in a vector of filter responses centered on each pixel. For instance, if your filter bank has 16 orientations and 2 scales, you would have 32 filter responses at each pixel. A distribution of these 32-dimensional filter responses could be thought of as encoding texture properties. We will simplify this representation by replacing each 32-dimensional vector with a discrete texton id. We will do

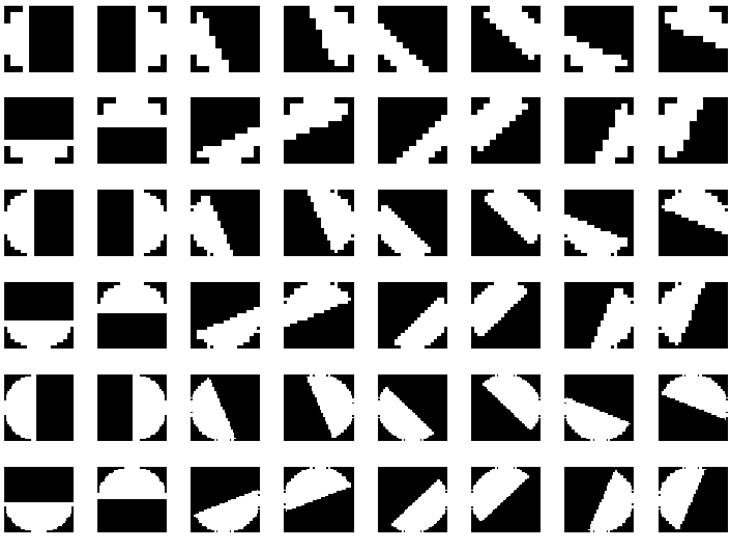


Figure 2: Half disk masks

this by clustering the filter responses at all pixels in the image in to K textons using kmeans (use MATLAB's `kmeans` function). Each pixel is then represented by a one dimensional, discrete cluster id instead of a vector of high-dimensional, real-valued filter responses (this process of dimensionality reduction from 32 to 1 is called "Vector Quantization"). This can be represented with a single channel image with values in the range of [1, 2, 3, ... , K]. K = 64 seems to work well but feel free to experiment. To visualize the a texton map, you can try `imagesc(tmap); colormap(jet);`

2.2.4 Local Texton Distributions

We will represent local texture distributions by building K-dimensional texton histograms over regions of interest. These histograms count how often each texton is observed. The regions of interest we will use are the half-disc masks previously discussed.

Analogous local histograms could be built for brightness or color, after brightness and color have been quantized in to some number of clusters.

2.2.5 Computing Texture Gradient (`tg`) and Brightness Gradient (`bg`) - 10Pts (5Pts each)

The local texton gradient (`tg`) and brightness gradient (`bg`) encode how much the texture and brightness distributions are changing at a pixel. We compute `tg` and `bg` by comparing the distributions in left/right half-disc pairs centered at a pixel. If the distributions are the similar, the gradient should be small. If the distributions are dissimilar, the gradient

should be large. Because our half-discs span multiple scales and orientations, we will end up with a series of local gradient measurements encoding how quickly the texture or brightness distributions are changing at different scales and angles.

We will compare texton and brightness distributions with the chi-square measure. The chi-square distance is a frequently used metric for comparing two histograms. It is defined as follows:

`chi_sqr(g,h) = .5 * sum[ i=1:K ((g_i - h_i)^2 / (g_i + h_i)) ]`, where g and h are histograms with the same binning scheme, and K indices through these bins. Note that the numerator of this expression is simply the sum of squared difference between histogram elements. The denominator adds a "soft" normalization to each bin so that less frequent elements still contribute to the overall distance.

To efficiently compute `tg` and `bg`, filtering can used to avoid nested loops over pixels. In addition, the linear nature of the formula above can be exploited. At a single orientation and scale, we can use a particular pair of masks to aggregate the counts in a histogram via a filtering operation, and compute the chi-square distance (gradient) in one loop over the bins according to the following outline:

```
   chi_sqr_dist=img*0
for i=1:num_bins
      tmp = 1 where img is in bin i and 0 elsewhere
      g_i = convolve tmp with left_mask
      h_i = convolve tmp with right_mask
      update chi_sqr_dist
end
```

The above procedure should generate a 2D matrix of gradient values. Simply repeat this for all orientations and scales, you should end up with a 3D matrix of size $n \times m \times (o \times s)$, where n , m are dimensions of the image.

Hint: you might want less than 256 bins when computing `bg`.

2.2.6 Rich Filter Banks - LM, S and MR - 30Pts (10Pts each)

Instead of using filter bank of Oriented First Derivative Gaussian filters, we can use richer (more complex and hopefully better) filter banks like Leung-Malik, Schmid and Maximum Response for rich feature descriptors. For detailed explanation refer to http://www.robots.ox.ac.uk/~vgg/research/texclass/filters.html.

Note that, the equations are not given to you for this part and hence we want you guys to figure it out on your own. However, feel free to discuss with upto 2 other students and/or come to office hours for some tips.
It's interesting to see how each filter bank affects the response on `tg`. One can use all the filter banks at once to generate a single `tg` or you can try to generate 3 different `tg`s, one for each of the filter banks, i.e., $\texttt{tg}_1$ for LM, $\texttt{tg}_2$ for S and $\texttt{tg}_3$ for MR (you will be running K-means three times here). See which performs better. You can also come up with your own way of combining the filter banks. The end goal is to get your precision-recall curve above the canny response curve.

2.2.7 Color Gradient - 10Pts

You need to use color information in L*a*b color space to use Color Gradient in addition to Texture Gradient and Brightness Gradient as used in Ref. You can bin the values of `L` and then compute chi_square distance to get brightness gradient `bg`. Similarly we use $\texttt{a}^*$ and $\texttt{b}^*$ channels to compute color gradients $\texttt{cg(a}^*\texttt{)}$ and $\texttt{cg(b}^*\texttt{)}$. Now you can combine these two to get one unified color gradient $\texttt{cg(ab}^*\texttt{) = cg(a}^*\texttt{) + cg(b}^*\texttt{)}$ check Ref. [1]

2.2.8 Sobel and Canny baseline

You need to run `canny_pb` and `sobel_pb` functions to generate canny and sobel baseline edges which we further use for `pb` edges

2.2.9 Output pb-lite

The final step is to combine information from the features with a baseline method (based on Sobel or Canny edge detection) using a simple equation

$$\texttt{PbEdges = (tg + bg + cg).*(w}_1\texttt{*cannyPb + w}_2\texttt{*sobelPb)}$$

A simple choice for $\texttt{w}_1$ and $\texttt{w}_2$ would be 0.5. However, one could make these wights dynamic (as you would do it in the extra credit).

The magnitude of the features represents the strength of boundaries, hence, a simple mean of the feature vector at location i should be somewhat proportional to `pb`. Of course, fancier ways to combine the features can be explored for better performance and extra credit. As a starting point, you can simply use an element-wise product of the baseline output and the mean feature strength to form the final `pb` value, this should work reasonably well.

2.2.10 Evaluating Boundary Detection

The goal of this project is to have a boundary detector that beats the baseline method provided, which is based on Sobel/Canny edge detection. The evaluation is based human annotation (groundtruth), collected as part of the BSDS500 dataset. A detailed description of the evaluation scheme is presented in Ref. [1]. A good performance measure is the F score of the precision-recall curve. The stencil code does automatic evaluation for you. A sample precision-recall curve is shown in Fig. 3.

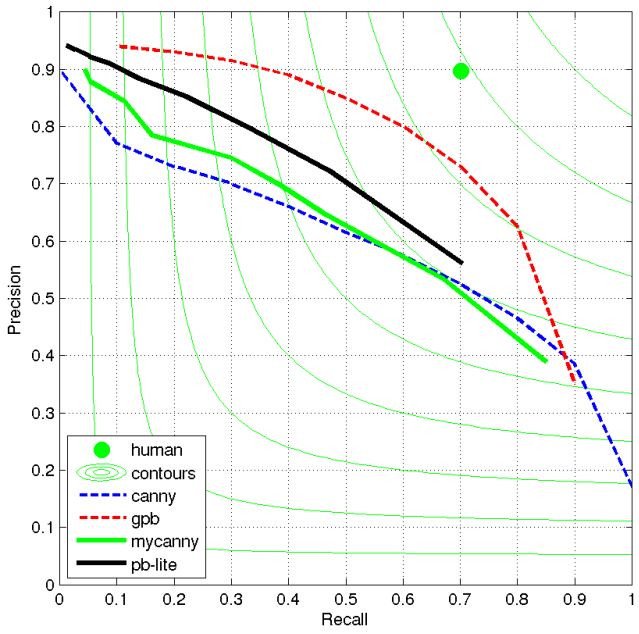


Figure 3: Precession Recall curve from Ref. [1]

(dotted lines are copied from figure 17 in Ref. [1], solid lines come from actual implementations in MATLAB)

The above diagram is generated using all of the 200 test images in the dataset. If you are using a subset of the testset, you might get better/worse performance because some images are "easier" or "harder". But if you do better than gpb (dotted red line), something is either wrong or you can publish a top paper.

2.3 Extra Credit

2.3.1 Dynamic weights - 20Pts

In this boundary detection algorithm we fuse 'Pb-lite output' with Canny or Sobel baseline edges. Instead of weighing them equally i.e.,

```
PbEdges = (tg + bg).*(cannyPb + sobelPb)
```

You have to come up with different weights for Canny and Edge depending upon image.

2.3.2 Color Segmentation - 40Pts

Come with some approach to use Color prior to detect Object boundaries which can help further boost the quality of boundary detection.

2.3.3 Do Something Cool! - upto 20Pts

Add any other heuristic which will help improve the over all boundary detection.

2.4 Starter Code

The full pipeline has been given in the `Starter.m` and you need to fill in the code in each section. Use the images from `../TestImages` folder to test your algorithm. At the end of each section save the images as mentioned in the comments.

- Texton Maps are saved to `../Images/TextonMap/TextonMap_ImageName.png` (`ImageName` is the file name of the image)
- Texture Gradients to `../Images/tg/tg_ImageName.png`
- Brightness Maps to `../Images/BrightnessMap/BrightnessMap_ImageName.png`
- Brightness Gradients to `../Images/bg/bg_ImageName.png`
- Sobel Pb outputs to `../Images/SobelPb/SobelPb_ImageName.png`
- Canny Pb outputs to `../Images/CannyPb/CannyPb_ImageName.png`
- Pb Lite outputs to `../Images/PbLite/PbLite_ImageName.png`
- Plot PR curves and save it as `../PR_Curve.png` (code and the accompanying Readme are in the `Benchmark` folder).

You can refer to some sample reports (which might help you understand the pipeline better) at http://cs.brown.edu/courses/cs143/2011/results/proj2/.

2.5 Submission Guidelines

Submit your codes (`.m` files) with the naming convention `YourDirectoryName_hw0.zip` onto ELMS/Canvas (**Please compress it to .zip and no other format**). Your `DirectoryName` is the username to your UMD e-mail ID. If your email ID is `nitinsan@terpmail.umd.edu`, your `DirectoryName` is `nitin`. Your zip file should have the following things:

- Folder named `Code` with all your code. Please make subfolders `Part1` and `Part2` for code from both parts.
- Folder named `Images` with sub-directories mentioned in the Starter Code section. (The folders are already given to you, do not change them).
- Typeset a report in LaTeXusing the IEEETran format given to you in `Draft` folder. The output file should be (**pdf and pdf ONLY**). Describe the pipeline with a lot of images, intermediate outputs (include filter visualization), your implementation details and any observations in detail with appropriate references for both sections.

- A `Readme.txt` file on how to run your code for both parts.

If your code does not comply with the above guidelines, you'll be given **ZERO** credit.

3 Allowed Matlab functions

`imfilter, conv2, imrotate, im2double, rgb2gray, rgb2lab, rgb2ycbcr, rgb2hsv, kmeans` and all other plotting and matrix operation/manipulation functions are allowed.
`fspecial, imgaussfilt` are NOT ALLOWED!

4 Collaboration Policy

You are restricted to discuss the ideas with at most two other people. But the code you turn-in should be your own and if you **DO USE** (try not it and it is not permitted) other external codes/codes from other students - do cite them. For other honor code refer to the CMSC733 Spring 2017 website here https://www.cs.umd.edu/class/spring2017/cmsc733/.

Acknowledgements

This fun project was inspired from 'Introduction to Computer Vision' (CS 143) course of Brown University (http://cs.brown.edu/courses/cs143/2011/proj2/).

DON'T FORGET TO HAVE FUN AND PLAY AROUND WITH IMAGES!.

References

[1] Pablo Arbelaez, Michael Maire, Charless Fowlkes, and Jitendra Malik. Contour detection and hierarchical image segmentation. *IEEE transactions on pattern analysis and machine intelligence*, 33(5):898–916, 2011.