Sam Willenson (shw58)
ECE 5242

Project 1 Write-up

Introduction

For this assignment, I was tasked with creating and training a model to detect orange traffic cones in real world images, noting the relative center point and distance to camera for these cones. I was given 25 training images of cones placed at varying distances from the camera, as well as labels for the distance of the cone to the camera for each of these images. After hand labeling the data, I used color segmentation and created a single gaussian model for my cone detector with promising results.

Problem Statement

How do you create a model to detect and classify the center coordinates as well as distance to camera for orange traffic cones?

Approach

My model was built in three steps using three python scripts I wrote which are named label.py, createLookupTable.py, and predict.py.

First I needed to analyze each pixel value of all my training images to calculate the amount of pixels contained in the cone, amount of pixels outside the cone, and what those pixels values actually are; I performed this feat with my script label.py. To actually start building this script, the very first thing I had to do was organize all my files into folders so that I could systematically go through all my training files in one run. After these logistics were sorted out, I needed to start pulling data from my training set. I used the python module RoiPoly to manually hand label all my regions of interest on the training dataset. As each training image popped up in a new window, I carefully drew a boundary around each cone in my training set, which ultimately created a mask for each image. I saved these masks to a previously empty array, which gave me a True/False label for each pixel in each of my images. Because each of these images is essentially a 600x800x3 (width x height x RGB values) list, I was able to use each mask as an index for the list of pixel values. I then accumulated all the pixel values for pixels inside and outside this mask into two separate arrays, which contain all these sorted pixel values for each image in my training set. This was how I performed my color segmentation for two classes, orange cone and not cone. Now that I have all my pixel values sorted appropriately, I save these arrays as .npy files in folders that are named for each file.

Next I needed to use this previously accumulated data (from label.py) to try to identify cones on previously unseen test images. I do this by taking in values for each pixel in my test images, and

calculating the probability that this RGB value is either part of a cone or not. In my first iteration of my model, I ran my predictions within a for loop that looked at each test pixel. However this meant that I had to run my probability function many more times than necessary, which resulted in my final analysis of each test image taking around 2 minutes. My final implementation of my model instead uses a more clever method which takes my run time for each test image down to about 10 seconds. Instead of running the probability function each time a new pixel value is loaded from my test image, I generate a list of all possible RGB values and calculate all possible probability values based on my training data before any test image is even seen.

This is achieved in my script createLookupTable.py. In this script I first load all my previously saved .npys from the label.py step, getting the information for each pixel value within and outside of my regions of interest. Using numpy, I can quickly get a calculated mean and covariance for both of these arrays. Next I defined my functions for calculating the pdf of each pixel value, using a single gaussian and Bayes Rule. These are the 3 formulas that I implemented:

$$p(\vec{x}|Y = y_\alpha) = \frac{1}{\sqrt{2\pi|\Sigma_\alpha|}} \exp\left[-(\vec{x} - \vec{\mu}_\alpha)^T \Sigma_\alpha^{-1} (\vec{x} - \vec{\mu}_\alpha)\right]$$

$$Pr(Y|X) = \frac{Pr(X|Y)Pr(Y)}{Pr(X)}$$

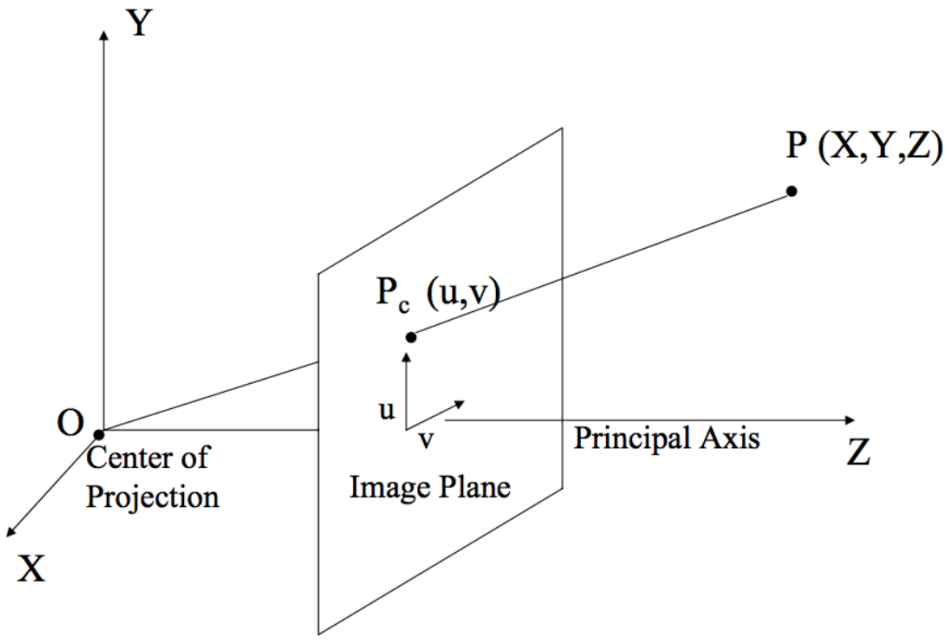
$$Pr(X) = \sum_Y Pr(X|Y)Pr(Y)$$

After turning these equations into functions I could use for each pixel value, and using my mean and covariance that was just calculated, I can find the probability of every possible pixel value in the RGB color space to be classified as a cone. However, instead of implementing these probabilities to every possible RGB value from 0-255 for each color, I used a bucketing system to cut down on my run time. By dividing by 4 for each 0-255 range, I can have my 3 for loops (one loop for each color) going from 0-64 instead of 0-255. This way every 4 pixels get assigned the same probability, which essentially cuts off my last 4 bits. While technically this is less accurate, in theory it will even out as some pixels will be higher than this probability value and some will be lower. After I have all my possible probability values calculated, I can save this result in a 64x64x64 array as my lookup table to be used in the next step.

Now that I have all my probability values calculated, my final step is actually predicting the location and centroids for new test images. This is accomplished in my script predict.py. First I load my RGB lookup table that was calculated in the previous step. Next I pick a threshold value to compare these probabilities against. If my probability is above this threshold, it will be classified as part of the cone, if it is below this value then it will be classified as not part of the cone. To start off the process I picked 0.8 as my threshold. Then I load and start iterating through all of my test images. For each test image, I iterate through each pixel and use that pixel value as the index for my RGB lookup table to get a probability value for that specific pixel. Because I shaved off the last 4 bits when generating my lookup table and because my RGB value arrays from my test images are saved as 0-1 instead of 0-255, I multiply each color value for that pixel by 255 and divide by 4, which gives me an appropriate index for my lookup table. I then compare this probability value against my threshold, and save the True/False value into an array for my full list of classifications of each test image.

After this I need to draw a bounding box around my predictions. To do this I used regionprops and label from the skimage.measure module. I use my array of True/False probability classifications to generate the labels, and run this through regionprops to generate my bounding boxes. I can then use this module to get my centroids, as well as the size of each bounding box using simple geometry from the local max and min points of my classification array's True values.

This gave me my first implementation of my model and I was able to see results of my predictions. At first I used cross validation, holding out some of my training examples to be used for my test set. Quickly I saw that my model was predicting too many non cone pixels as part of the cone class, so I upped my threshold to 0.9. After performing many more runs I continued to fine tune my threshold until I settled on a threshold of 0.99999999999; this seemed to work best to cut down on misclassifications due to close-to-orange objects as well as reflections of the cone. Once I was happy with the performance, I implemented a formula to get the distance of the cone to the camera. This was achieved through the pinhole camera formula seen on the next page:



$$\frac{u}{f} = \frac{X}{Z} \qquad \frac{v}{f} = \frac{Y}{Z}$$

The height of the cone was given as 17 inches and width of the base was given as 7.5 inches, so I already knew that my value for Y was 17 inches. Solving for f in this equation gives me f = (v * Z) / Y, and plugging in my known value for Y gives f = (v * Z) / 17. Because I knew that the camera would be the same for the training images and test images, I knew the focal length of the camera would be the same. My training images have the true distance given, so I used the dimensions from my best bounding box and the true distance given from that image to fill in the values of v and Z in this formula. I found the focal length of this camera to be 55.7657 using this method. Now that I had a focal length, I solved for Z in this formula giving me Z = (Y * f)/ v. Plugging in my given value for Y gives Z = (Y * 55.7647) / by, 'by' being the vertical length of my bounding box. This gives me a predicted value for distance for the rest of my test images.

After this was complete, I had everything in order to perform my final tests.

Results

Running my model on the test images gave me the following results:
On the left you can see my probability mask and on the right my predicted cone identification.

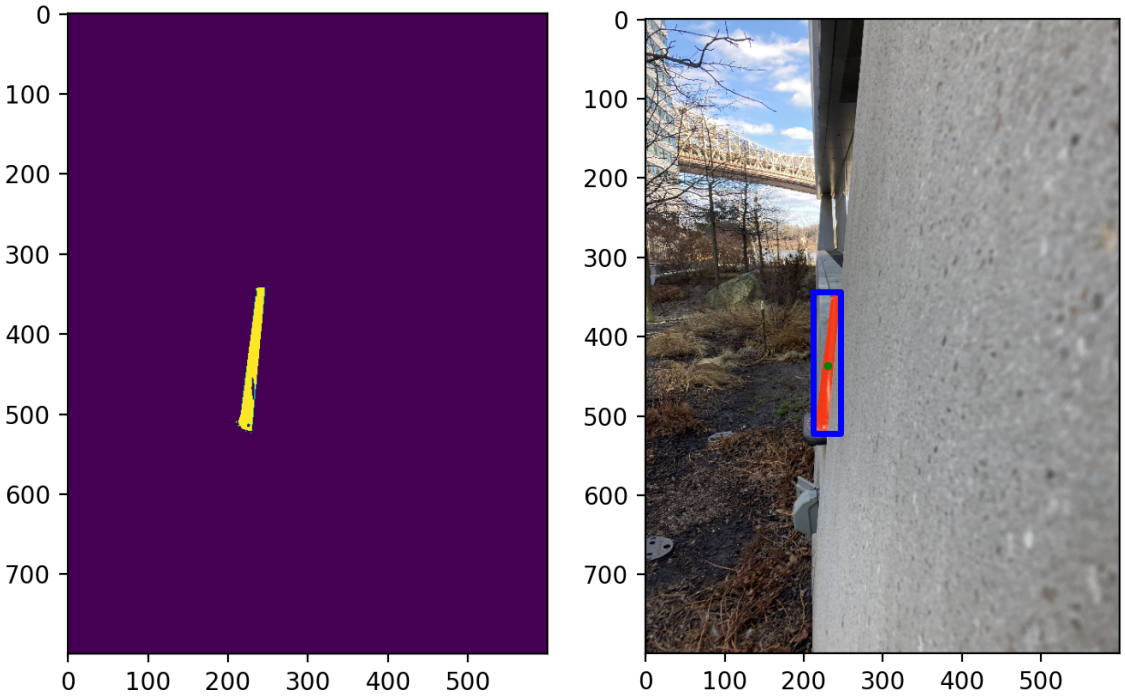


Image No = [01], CentroidX = 230.2330, CentroidY = 436.8775, Distance = 5.296

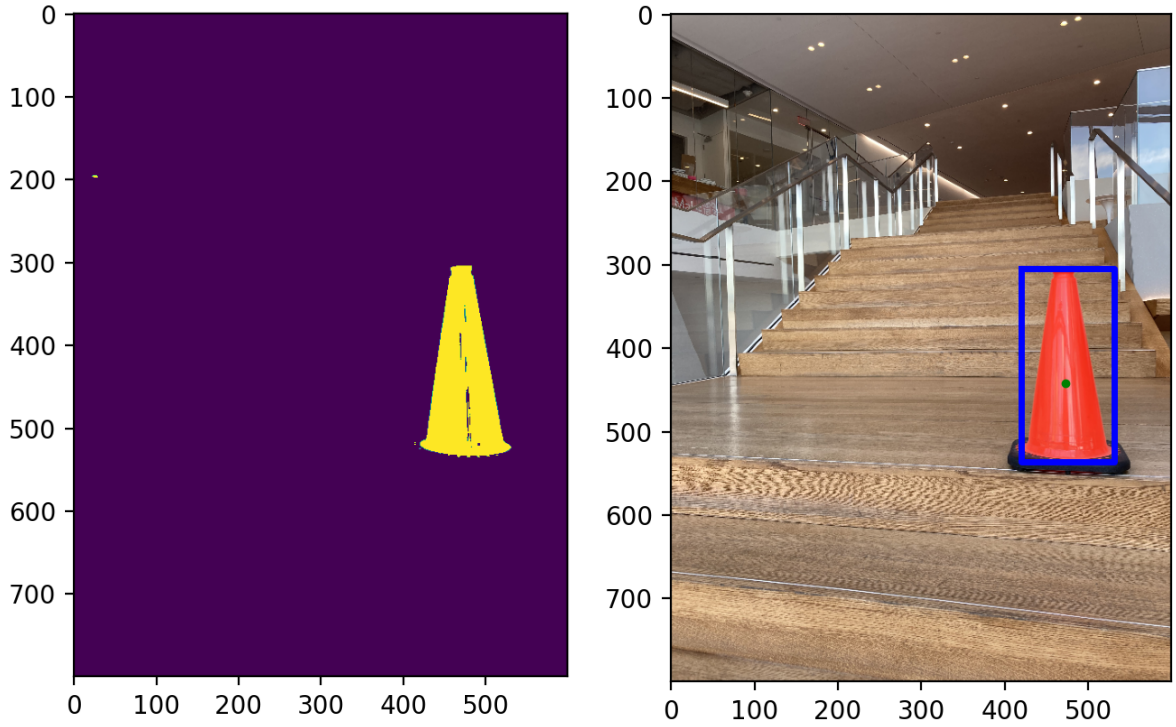


Image No = [02], CentroidX = 472.3413, CentroidY = 441.8007, Distance = 4.104

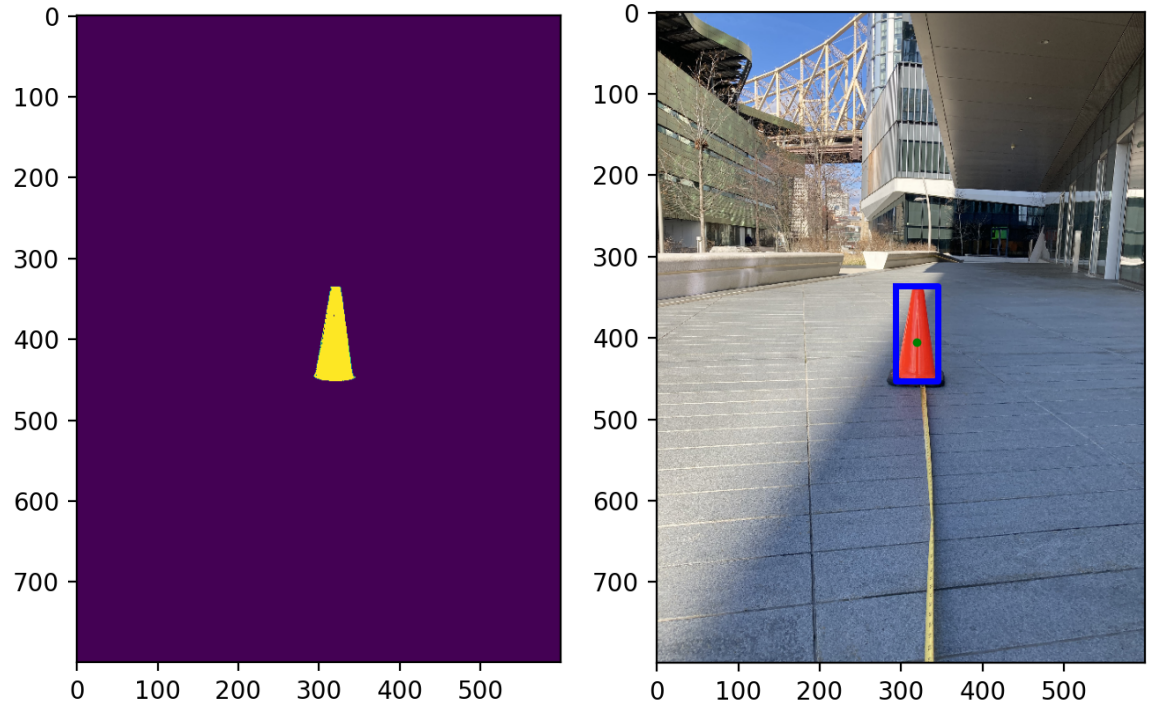


Image No = [03], CentroidX = 318.8365, CentroidY = 405.2625, Distance = 8.103

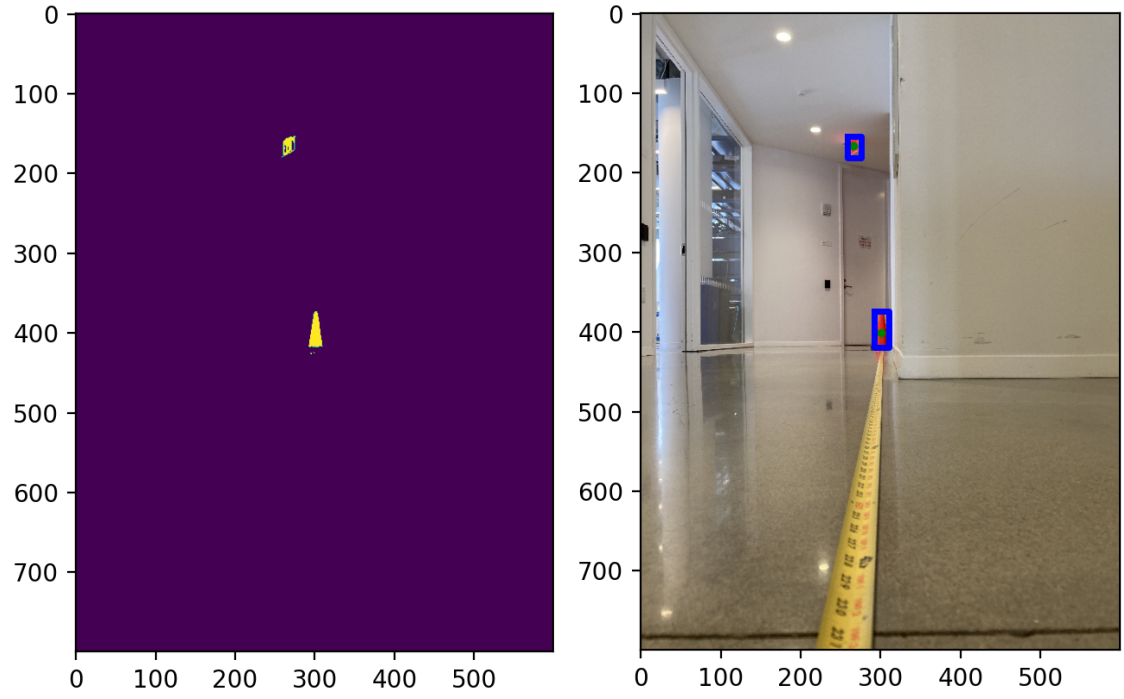


Image No = [04], CentroidX1 = 266.4321, CentroidY1 = 165.8313, Distance1 = 35.111
Image No = [04], CentroidX2 = 300.4969, CentroidY2 = 400.2618, Distance2 = 21.067

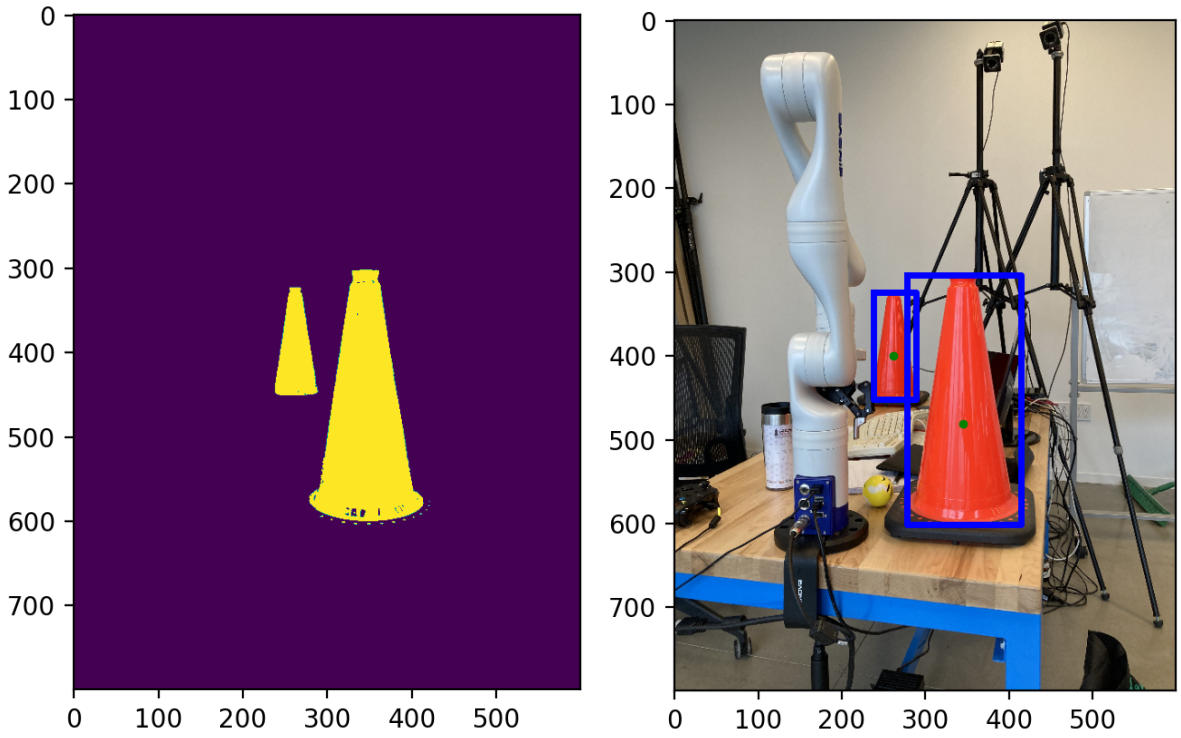


Image No = [05], CentroidX1 = 344.8311, CentroidY1 = 480.4406, Distance1 = 3.181
Image No = [05], CentroidX2 = 261.8766, CentroidY2 = 399.8634, Distance2 = 7.406

Discussion

Looking at all of these results, I can say that my model works fairly well. Looking at the centroid dot and bounding box images shows me that my predictions are fairly accurate, and comparing the results of my holdout set from cross validation (which contains given true distance values) lets me analyze how close to perfect my distance prediction was on that set, which it was fairly accurate on. However my model is not perfect; while it does a great job of identifying the cone/cones in four of the pictures, on test image 4 my algorithm falsely identifies an exit sign as a cone. We can see in the results of image 4 that there are 2 centroids and 2 distances, which is incorrect since there is only one cone in that image. I thought of taking care of this through geometric solutions such as ignoring any bounding boxes where the vertical length is shorter than the horizontal length; however this solution isn't ideal in case there is a test image with a sideways cone. I tried retraining my model by adding another class for exit signs but this ended up hurting the overall performance so I did not use this method. Another way to improve performance would be to implement the erosion/dilation method. This might have taken care of smaller bounding box issues such as the exit sign being detected, however due to time constraints I did not test this method out. The best way to make my model more robust would be to implement a GMM, however again due to time constraints I did not implement this successfully. In the future I want to improve my model with a GMM to see how accurately it can detect cones, without detecting edge cases like exit signs or other objects of similar orange colors.