Reading Tennessee License Plates

EECE 5354 Final Project Report

Brian Gauch and Shan Lin

12/17/2015

Abstract

We were able to read some Tennessee license plates. We used the fact that these license plates have black text, and detected blackness by combining multiple thresholds with respect to darkness and colorfulness. These methods seemed more effective when used together than any method was individually.

Background

Reading text is one of the major problems in Computer Vision, and has been studied for decades. License plates are of particular interest, both because of the practical applications, and because it is a somewhat more restricted problem in that license plates all have similar fonts, length, etc.

What we were trying to do

Because there are so many different kinds of license plates in the United States, we knew that it would be outside the scope of this class project to write a program that could read all different kinds of license plates. Our goal was simply to recognize all Tennessee license plates. Even this proved too difficult, because, not only do license plates vary from state to state, but also old license plates have different fonts than modern ones. So we changed our goal to identifying all modern (2007+) Tennessee license plates.

What we were able to do

We were able to read the majority of the modern Tennessee license plate images we collected. However, we only dealt with images where the plate was more or less perpendicular to the camera vector.

How we did it

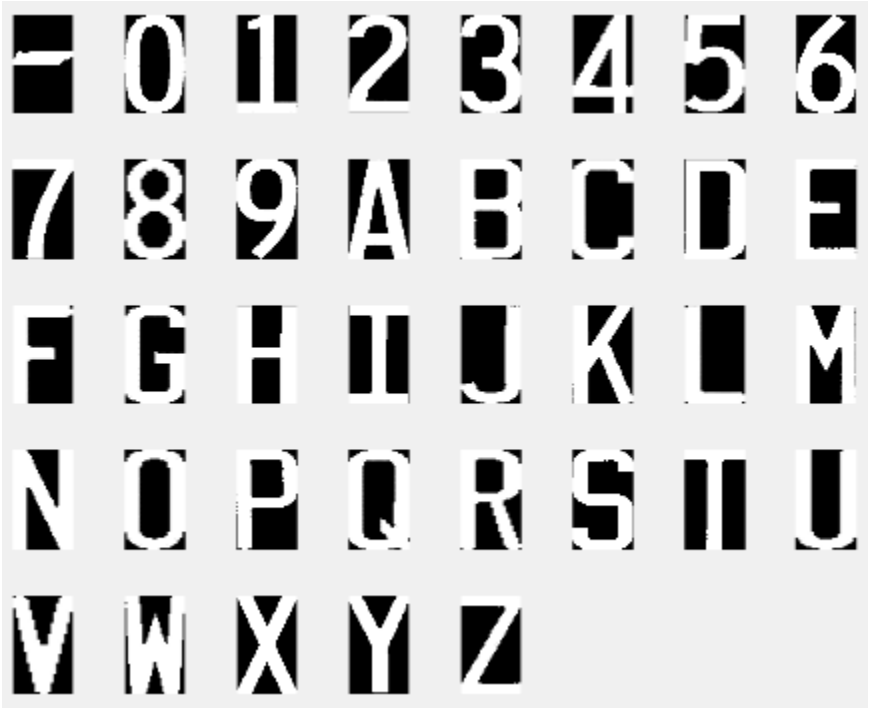
Creating an alphabet

There was no official font information available, so we had to rely on example license plates to determine what each character should look like. To do this, we labeled a number of license plate images (separate from the images being tested) with the correct string, e.g. “273-shd.jpg”. We needed 12 such images to cover a…z and 0…9. We also used “-” to denote the symbol for Tennessee that was in the center of most modern plates, since it was also a black blob. So, there were 37 characters that we could recognize in all. If a character appeared on multiple different images, the larger image was used.

We used a simpler algorithm to find blobs for the alphabet than what we eventually used to match with it, and cleaned the alphabet images up a little bit by hand. The “1” below was one of the worst characters before cleanup.

Before cleanup

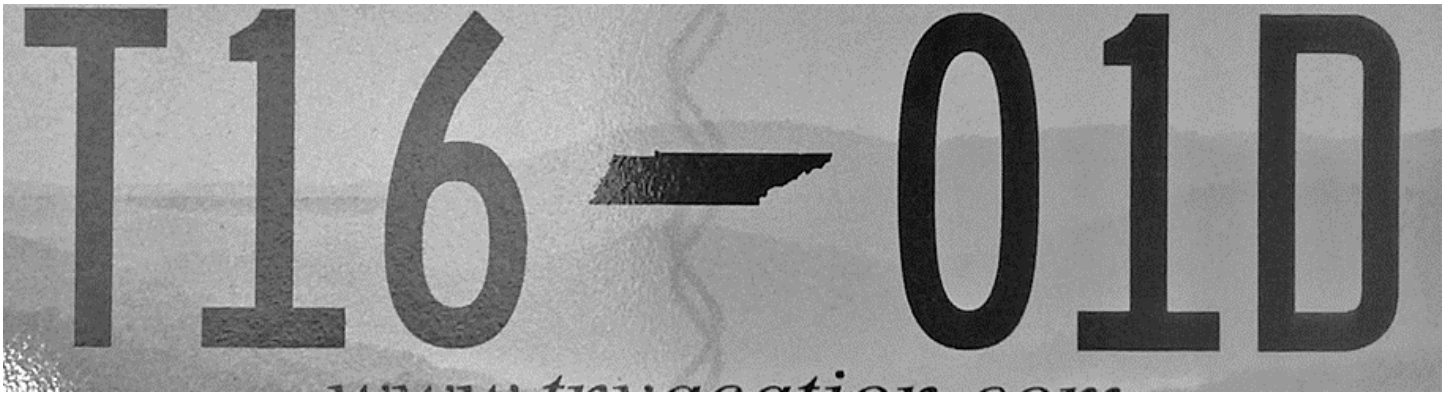
After cleanup



Entire cleaned-up alphabet

Finding blobs using thresholds

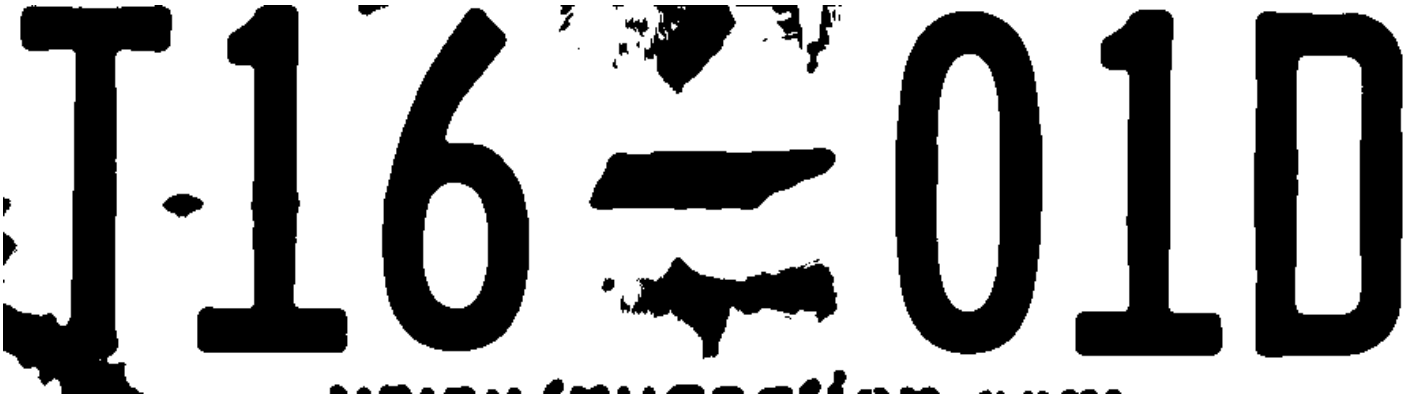
A license plate used for testing (unmodified image)



A license plate used for testing (pre-processed: cropped, grayscale)

Our goal was to identify black characters. Therefore, we looked for darker-than-usual blobs, and also looked for less-colorful-than-usual blobs. We wrote a function called thresh() to choose thresholds for each image, and then apply them to create binary images. In this function, four different thresholding methods are used, and then their results are combined. The first three methods are different ways to find blobs using greyscale intensity, while the fourth considers HSV "value" (i.e., colorfulness).

1) The first thing we used was the Bernsen's local thresholding algorithm[1], an implementation of which we downloaded from http://www.mathworks.com/matlabcentral/fileexchange/40856-bernsen-local-image-thresholding. This algorithm chooses thresholds by considering the local contrast. We used it to find grayscale blobs.

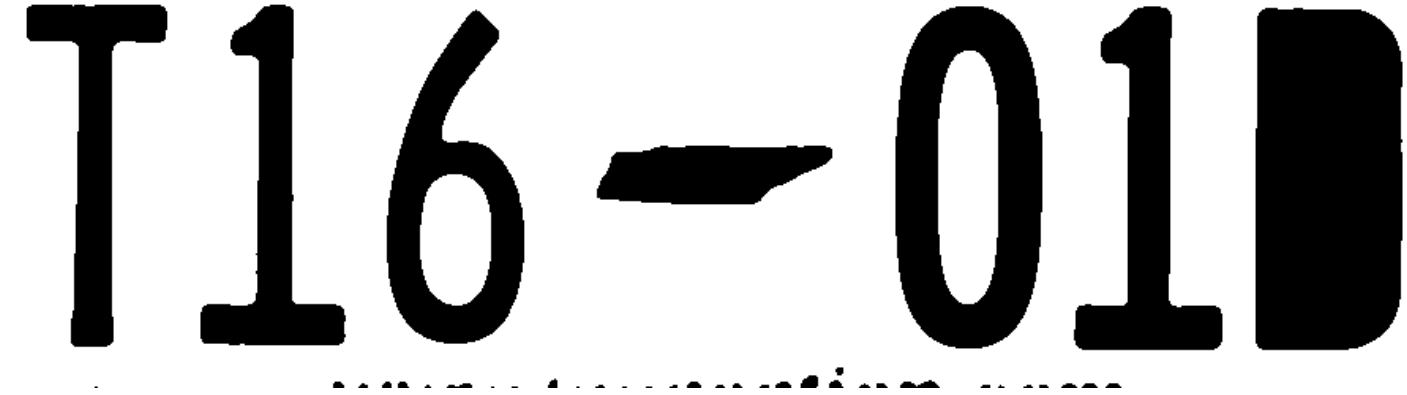


Result of Bernsen's algorithm by itself

2) The second thing we used was Otsu's algorithm[2]. It uses the same thresholds within a single image, instead of adjusting them locally like (1). This algorithm tries clusters into "classes" by examining intra-class variance between them. In our case we simply identified two "classes", the foreground and background, by considering grayscale intensity. The Matlab function graythresh() implements this algorithm, and im2bw() applies this threshold to get a binary image.

Result of Otsu's algorithm by itself

3) The third thing we used was the hysthresh() function from Peter Kovesi, which uses a double threshold (i.e. it uses a high threshold then "grows back" using the second threshold).



Result of hysthresh() by itself

4) The fourth thing we did was separate characters from the background according to their HSV “value” (colorfulness). However, this method did not work very well for some low-quality license plate images we obtained from the internet, where the characters had some color due to noise.

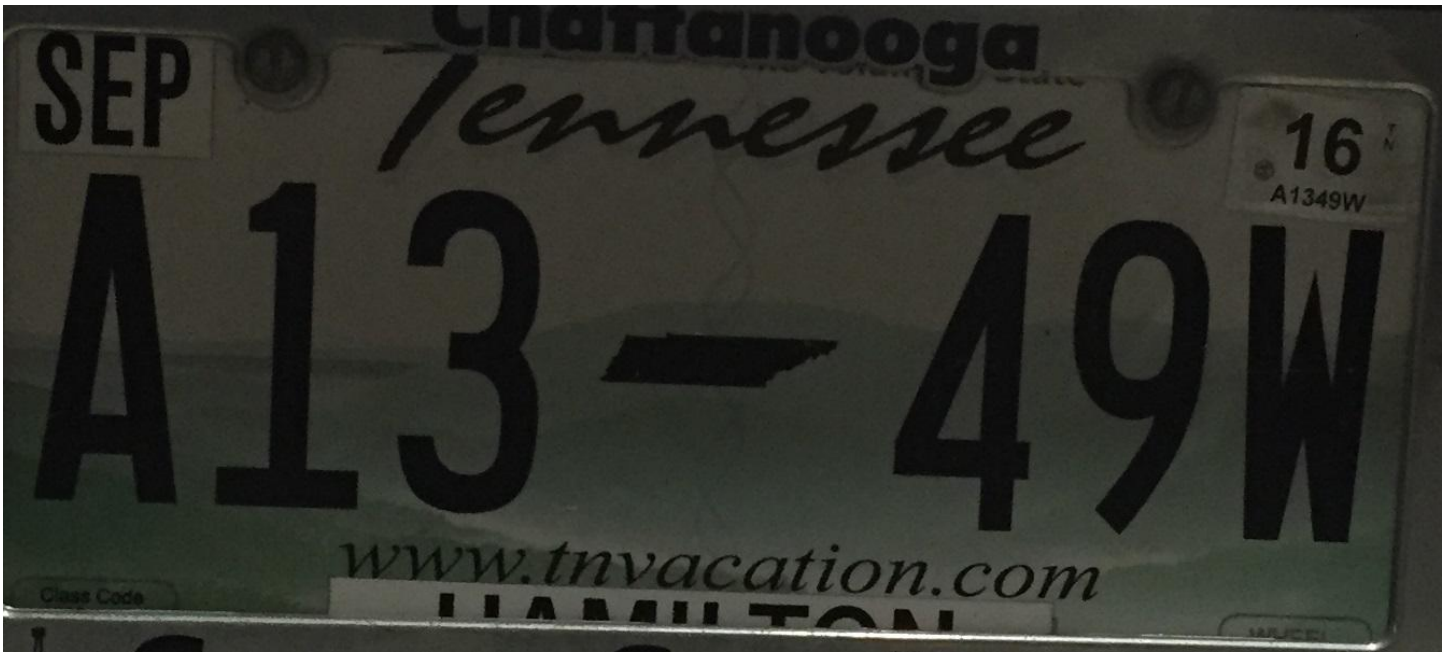
Result of colorfulness thresholding by itself

We combined the results of these four methods by multiplying them. The reason we combined these methods together is that they have different advantages. Bernsen’s algorithm works well for finding the correct outlines for characters, but also finds many other small blobs. Otsu’s method doesn’t introduce many extra blobs, but it sometimes joined characters together as one “blob”. Hysteresis thresholding has similar results, but may fill in the holes in letters (e.g. the two holes in an “8”). The fourth method, which considered colorfulness, was helpful when separating characters from background when light is reflecting off the plate (making black characters appear lighter gray).

Result of all methods multiplied together

Further trimmed to indicate detected blobs

In our running example above, T16-01D was correctly identified, but A13-49W was identified as "AI3-49W"



A different license plate that did not fare as well due to rotation

The recognized blobs for this harder image

Matching blobs with alphabet characters

The algorithm we used to match blobs with alphabet characters was actually comparatively simple. First, we resized each blob image so that it was the same height as all the alphabet images. Then we converted the blob and alphabet images into binary images (white=character/blob, black=non-character/non-blob), and essentially overlaid one on the other - counted the number of matching pixels for each pairing. We needed to avoid matching e.g. F with E or vice-versa, so we considered the strength of a match to be min(percentBlobInAlphabetChar, percentAlphabetCharInBlob). We maximized this measure with respect to translation by essentially performing gradient descent. Perspective and rotation were not taken into account.

What we could do later

We could fairly easily automatically detect how noisy an image is before trying to use HSV “value” to determine character blobs. We could try to identify the rotation of characters (perhaps by simply having multiple template images for each character, although this could be slow). Finally, we could try to identify different kinds of license plates – however, this would mean that we could not use the HSV “value” method at all, since not all plates have black characters.

References

1. Bernsen, J (1986), "Dynamic Thresholding of Grey-Level Images", Proc. of the 8th Int. Conf. on Pattern Recognition
2. Nobuyuki Otsu (1979). "A threshold selection method from gray-level histograms". IEEE Trans. Sys., Man., Cyber. 9 (1): 62–66. doi:10.1109/TSMC.1979.4310076.