CENG466 - FINAL

1st Bengisu AYAN
Derpartment of Computer Engineering
Middle East Technical University
Ankara, Turkey
e223697@metu.edu.tr

I. QUESTION 1

A. Introduction

In this question we are required to classify the numbers containing and not containing loops from a dataset of street numbers.
For me, at first glance the biggest challenge in the question was to separate the digits from the background, which was step 1. If I failed in this part, I doubted the rest of the procedure would run smoothly. In the dataset, there were many strict lines in the background, since these sharp background changes would create problems in both global thresholding and edge detection based methods, I thought they would cause difficulties in step 1. Furthermore, the numerals in the first three images were mostly dark and thin while the numerals in the last two images were fairer and thicker. This property would make it harder to come up with a general algorithm.

During the literature survey, I generally encountered machine learning based approaches, however I did not have enough data to train a model and it was out of the scope of this course. [1] proposed an interesting approach to detecting licence plate characters, which in my opinion is a similar process, as in both cases background noise is a problem. It suggested using Sobel edge detector to remove the border elements and Otsu's method for thresholding the characters. I decided to try to remove the border elements, as in our dataset the numbers were generally towards the middle of the image. [2] proposed a heuristic binarization algorithm based on two different thresholds. The dataset was quiet different from the one given to us, however it gave me the idea of trying adaptive thresholding.

In my algorithm, I got inspired by [1], as I also first removed the boundaries from the background. In my algorithm, I applied a double sequential thresholding method (First adaptive thresholding, then Otsu's method after limiting the area that might contain numbers), which was not in the above mentioned papers. I applied this method because neither adaptive thresholding nor Otsu's method alone responded to my needs. The results of adaptive thresholding was very noisy to separate numerals without loss of information. The results of Otsu's method was what I ultimately wanted, but it was hard to get rid of some of noise present in the image, and it made it impossible to distinguish the digits in images like "2.png" which has very close dark values.

B. Theoretical Presentation

In my heuristic, I mainly used Otsu's method, adaptive thresholding, morphological operations and smoothing techniques.

1) Steps to my heuristic:

1) Apply median blurring with a kernel size of 3.
2) Apply Gaussian filter with a kernel size of 5.
3) Apply adaptive thesholding
4) Inverse the image to make the background black and objects white
5) Remove elements touching the borders
6) Crop % 15 of the image from the sides by making them black.
7) Remove objects smaller than 30 pixels from the image
8) Find the general boundaries of the remaining objects in the image. (At this step, mostly just the numerals stayed in the dataset)
9) Apply Otsu's thresholding inside the boundary
10) Inverse binarize the image if the background is not black
11) Find the bounding box of each connected component
12) If there are more than one connected component in the box, dilate until there is one.
13) Find the bounding box again
14) Find Euler's number to detect the numerals with loops

I first applied median filtering with a kernel size of 3 in this heuristic, since it significantly improves the results for adaptive thresholding. However, I tried not to keep the kernel size too big, since it resulted in some loss of information in some images and I could get rid of the remaining noise by opening.

I also applied Gaussian filter with a kernel size of 5 to further get rid of the noise. If I further increased the kernel size, there was the risk of the numerals becoming unreadable in the first images since they are very thin. However a kernel size of 3 was not enough while removing the noise.

In this heuristic, I apply adaptive thresholding to detect the region that is most likely to contain the numerals. I first tried edge detection for this purpose, but adaptive thresholding responded better to my attempts of removing the noise and the changes in the parameter of Canny edge detection resulted differently in each image of the dataset.

After I get rid of the background noise, I dissect the the image checking iteratively the left, right, top and bottom boundaries of the connected components and find the general boundaries for the numerals as a whole. To this subsection of the original image, I apply Otsu's thresholding. This way, the numerals are a lot clearer and not at all connected to each other. If we skipped the boundary extraction step, some of the numerals were not readable since Otsu's method tried to find one global threshold for the image and it was not this precise.

I managed to extract the skeleton of the numerals, however it did not work the way I wanted in some of the images such 1.png. It lost a significant portion of the 1. Thus, while deciding if the numeral has a loop or not, I did not use the skeletonized versions of the images. During the skeletonization process, I used the algorithms in [3] While skeletonizing, I used a cross shaped kernel as it very common in the literature.

While deciding if the number in the bounding box has a loop or not, I used Euler's number. According to [4] the number of holes H and connected components C in a figure can be used to define the Euler number E:

$$E = C - H \tag{1}$$

As we can consider "loops" as hole in the connected components of our image, we can take the euler number of the connected component in each individual bounding box, which we found when we used $connected\_component\_with\_stats$ function of Opencv. It returns left most, top most pixels of each component and width and height. This information is enough to analytically define a bounding box. In my implementation, this function returns 2 components for each bounding box, counting both the background and numeral. Therefore, if the Euler's number is less than 0, the numeral contains loops and if it is greater or equal to zero, the numeral does not contain any loops.

Algorithm 1 Step 1 - Heuristic

$image \leftarrow Median\_blurring(image)$
$image \leftarrow Gaussian\_blurring(image)$
$image \leftarrow Adaptive\_threshold(image)$
$image \leftarrow Binary\_inverse(image)$
$image \leftarrow Remove\_elements\_on\_borders(image)$
$image \leftarrow Crop(image)$
$image \leftarrow bwareaopen(image, 30)$
$box \leftarrow find\_general\_bounding\_box(image)$
$otsu \leftarrow otsu\_threshold(box)$

C. Experimental Results

1) Step 1: During step 1, I applied a median filter of kernel size 3x3, Gaussian kernel of size 5x5, adaptive thresholding to eliminate the background and objects touching the borders, Otsu's thresholding to the big bounding box containing the numerals. For the last 2, there was no need for explicit values, which I believe made the algorithm more generic. I also cropped %15 percent from the sides, deducting from the dataset that the images are mainly situated around middle of the image. It helped remove some of the bigger background components. Results of Step 1 can be seen in Figure 1

Algorithm 2 Find the general bounding box for all numerals

$connected\_components, stats \leftarrow get_c onnected_c omponents\_stats(image)$
$left \leftarrow image.shape.x$
$right \leftarrow 0$
$top \leftarrow image.shape.x$
$bottom \leftarrow 0$
for `i in connected_components` **do**
  $row \leftarrow stats[i][1]$
  $col \leftarrow stats[i][0]$
  $width \leftarrow stats[i][2]$
  $height \leftarrow stats[i][3]$
  if $row + height > bottom$ **then**
    $bottom \leftarrow row + height$
  end if
  if $row < top$ **then**
    $top \leftarrow row$
  end if
  if $col < left$ **then**
    $left \leftarrow col$
  end if
  if $col + width > right$ **then**
    $right \leftarrow col + width$
  end if
end for
$draw\_rectangle(left, right, top, bottom)$

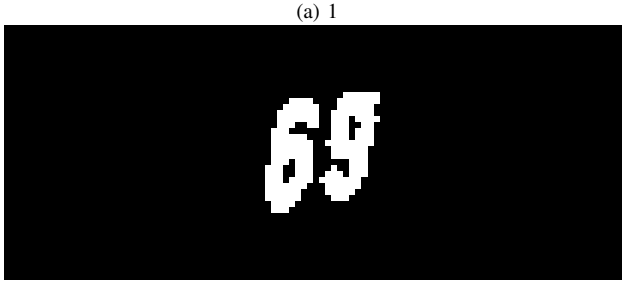
Algorithm 3 Step - Skeletonize

while $image \leftarrow NOT\_EMPTY$ **do**
  $skel \leftarrow skel|(img\&!open(img))$
  $image \leftarrow eroded(image)$
end while

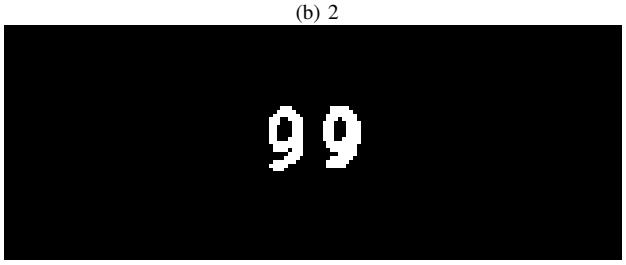
2) Step 2: Opencv offers a function called connectedComponentsWithStats, which provides us with the boundary information of the connected component. Thus, drawing the rectangles is trivial. However in certain images such 5.png some numerals were disconnected from each other, which resulted in them appearing in different bounding boxes (For examle 5 appeared in 2 separate bounding boxes which had intersections with each other in 5.png) Thus, I made an assumption that there was enough space between numbers horizontally and I dilated the connected components until there was only one connected component in each bounding box. It worked for the 5 images in the dataset. However, it might cause problems in other images. For the dilation I used a kernel size of 3x3 rectangular kernel, to make the dilating process slow. Results of Step 2 can be seen in Figure 2

3) Step 3: I am not very satisfied with the results of step 3. I used a generic algorithm available in the book with a cross shaped kernel of size. I did not used the results of this step, in

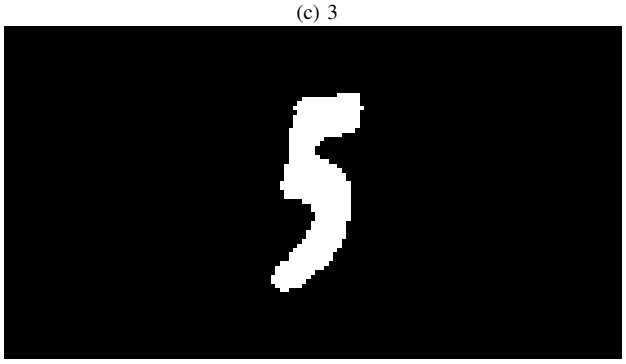
(a) 1



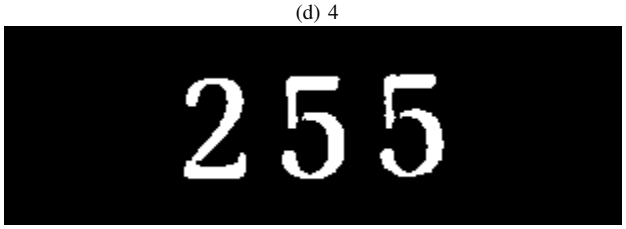
(b) 2



(c) 3

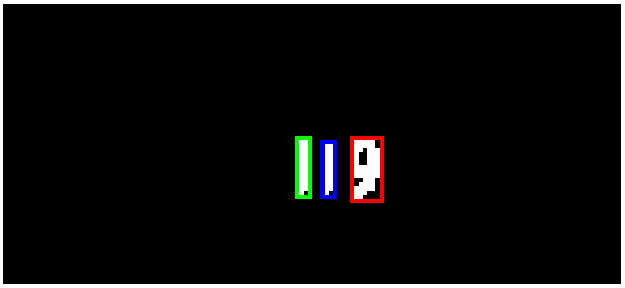


(d) 4

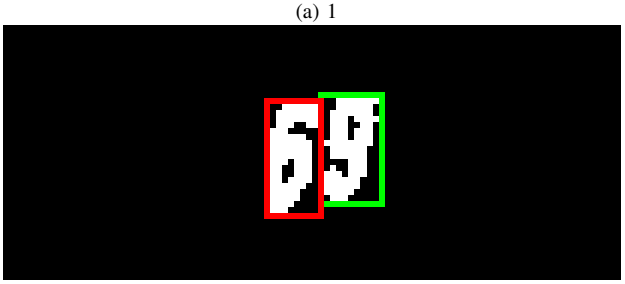


(e) 5

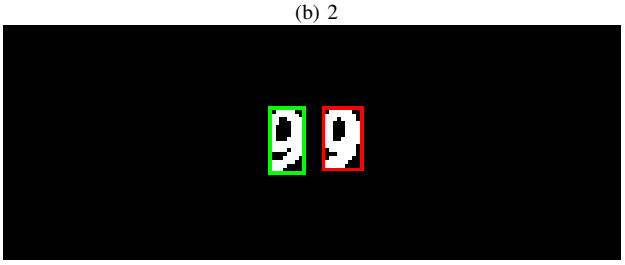
Fig. 1: Images after Step 1



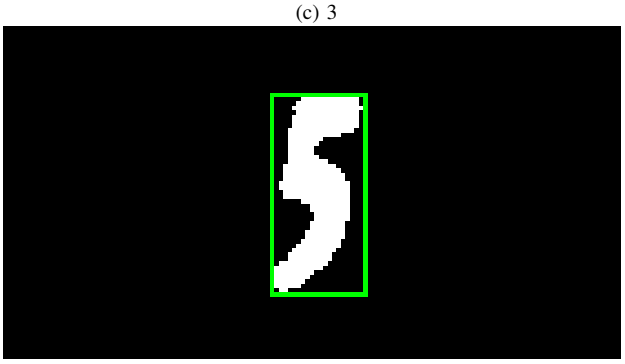
(a) 1



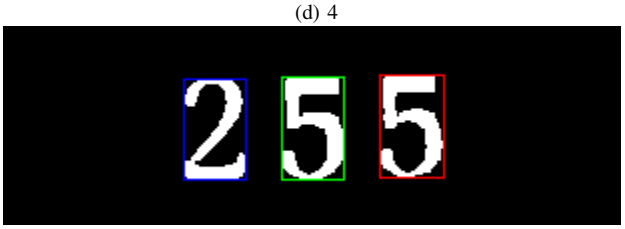
(b) 2



(c) 3



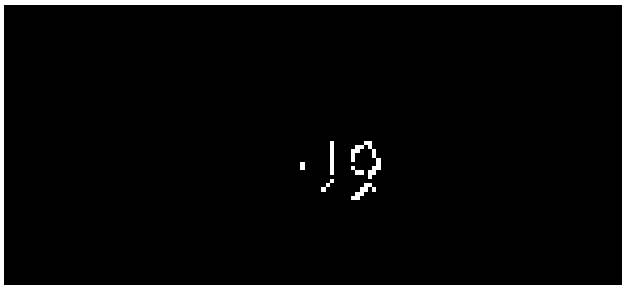
(d) 4



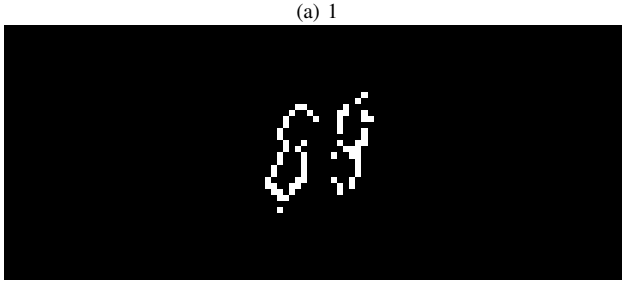
(e) 5

Fig. 2: Images after Step 2

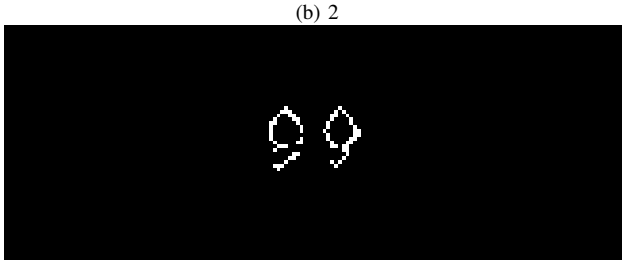
step 4 while classifying the numerals as they resulted in many disconnected components. The results can be seen in Figure 3



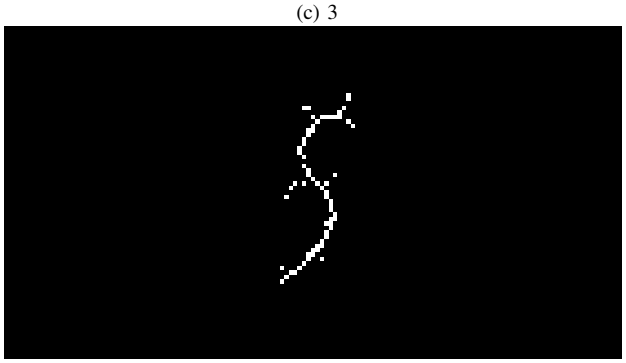
(a) 1



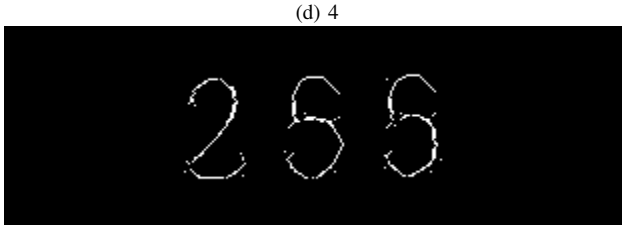
(b) 2



(c) 3



(d) 4



(e) 5

Fig. 3: Images after Step 3

4) Step 4: In 1.png: 1 numeral containing loops, 2 numerals without loops
In 2.png: 2 numerals with loops
In 3.png: 2 numerals with loops
In 4.png: 1 numeral without loop
In 5.png: 3 numerals without loops

During this step, I used Euler numbers as explained in the theoratical section. When the images were preprocessed properly, this method always worked with the dataset. I had trouble finding a function that calculates this characteristic for images, therefore I used a computer vision library called "mahotas".

5) Step 5: I manually calculated the performance of my results as I could not know if the algorithm worked without visually checking the results. However, in this dataset my Performance was 1. All 10 numerals were classified correctly.

6) Tools: I implemented my algorithm in python because I was more familiar with it from the previous homeworks. However I encountered certain problems while working with it. First of all, I had to import many libraries as the functionalities I needed was not available in separate ones. I used cv2 for morphological operations and read/write operations. However, there was no bwareaopen or clear_border functions in cv2 so I had to import them from skimage or write them myself. Furthermore, I imported Mahotas to calculate Euler characteristics of an image. While researching I realized most of these functionalities are available in matlab, but it was too late to switch platforms. I also used numpy.

D. Conclusion

Even though my algorithm works with the given images in Dataset 1, there are several problematic assumptions I made from the given 5 images, which might not necessarily be true for the test dataset. First of all, as I assumed the numerals are situated around the middle of the image, I cropped %15 of the image from the sides. Secondly, I made the assumption that the numerals would not touch the borders and removed the components touching the border, which also might cause problems. Furthermore, while deciding on the bounding boxes, sometimes there existed more than one bounding intersecting with each other enclosing different parts of the numeral. I made the assumption that these intersections would not be with other numerals and dilated the components inside this bounding box until there was only one connected component in it. I think this might cause problems in the test dataset. This part of the algorithm can be seen in Figure 4. However, overall my algorithm works pretty well with the given dataset.

The complexity of the algorithm is restricted with the complexity of thresholding and morphological operations.

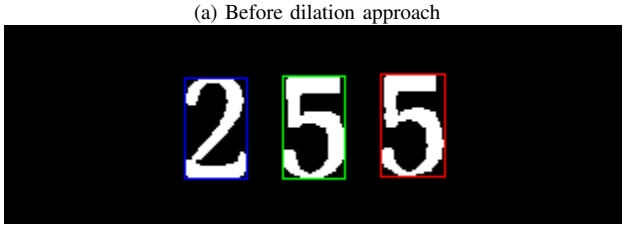
II. QUESTION 2

A. Introduction

In this part, we are required to segment animals from their backgrounds.

For me the most difficult parts of this question was the theoretical foundation of mean shift and n-cut segmentation. They have very mathematical explanations, therefore it can be very hard to find the right parameters for these algorithms.

(a) Before dilation approach



(b) After dilation approach

Fig. 4: Images after Step 3

Furthermore, it is also difficult to predict how the results should look like. For example, when looking at a graph, I have difficult making any sense of it related to the image. Thankfully the zebra images in [6] gave me a rough idea on how the end image should look like.

While deciding how to apply mean shift clustering algorithm, [5] gave me a very good idea about which features of the image I have to feed the algorithm. Before reading it, I had only used pixel values for 3 channels, but then I added spatial information and I believe obtained better results.

B. Theoretical Presentation

1) Step 1: Mean Shift algorithm is an advanced clustering based segmentation algorithm. It is basically a mode searching algorithm, which searches for the mean value of a Gaussian distribution. There are several steps to be taken to segment an image using mean shift clustering algorithm:

1) Find the features of the image.
   In my implementation I used the spatial data and color information while determining the feature of the image. According to [5], I created 5 dimensional input for the Mean Shift clustering algorithm. 3 of these dimensions are the color information since it is an 3 channel RGB image. In different papers, CIELAB color space was suggested for mean shift segmentation but I preferred RGB color space to respresent color range information simply because it is easier to implement. 2 of the dimensions are the spatial data.
2) Initialize windows at individual pixel locations
   Then the mean shift clustering algorithm initializes seeds at pixel locations with the initially set radius for the chosen kernel. This is basically the initialization step for mean shift algorithm.
   The choice of bandwidth was the most crucial parameter of the clustering algorithm. When I chose bandwidth to be a large value, I get less segments in Step 1. However, since this is an oversegmentation to create superpixels, the number of segments should not be too low.
3) Perform mean shift for each window until convergence. For each window, the center of mass of the window is found and a mean shift vector is calculated from the center the window to the center of mass. Then the center of the window is shifted to the center of mass. This process is done until center of mass converges with center of the kernel.
4) Merge windows that end up near the same peak.
   Then, windows with near modes are merged with each other and these windows create segments.

Mean shift clustering algorithm works on Gaussian distributions as Gaussian distributions have a peak and towards the peak, the probability density function values increases. Coming from this, mean shift clustering algorithm initializes a window in a random location. Then finds the center of mass of this window and creates a mean distance vector from the center of the window to center of mass. Finally it moves the center of the window to this center of mass. Using this approach, the window keeps getting close to the peak. The algorithm iterates this way until the center of mass and center of the window converges since we reached the peak. Using a vectors we have calculated before, it is possible to estimate the variance of the Gaussian distribution. As variance in a Gaussian distribution shows how dispersed the distribution is.

2) Step 2: In this step, we were required to define an attributed graph whose edge weights were the texture similarity between the superpixels obtained in Step1 using mean shift segmentation. According to [6], edge weights are calculated using the following formula:

$$w_{ij} = e^{\frac{-\|F(i)-F(j)\|^2}{\sigma_x}} * \begin{cases} e^{\frac{-\|X(i)-X(j)\|^2}{\sigma_y}} & \text{if } |X(i)-X(j)\| < \text{r} \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

However, I changed the formula a bit because I got really small values in every other case:

$$w_{ij} = e^{\frac{-\|F(i)-F(j)\|}{\sigma_x}} * \begin{cases} e^{\frac{-\|X(i)-X(j)\|}{\sigma_y}} & \text{if } |X(i)-X(j)\| < \text{r} \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

Basically F(i) and F(j) values represent the texture information which will later be explained. X(i) and X(j) values represent the spatial information. Since we do not want to find similarities between superpixels that are very far, we make their similarity component 0.

To find the texture similarity, I used Gabor filters at different frequencies and orientation values. Gabor filters are extensively used for texture analysis and analyzes whether there is any specific frequency content in the image in specific directions in a localized region around the region of analysis. I created filter banks for this consisting of different sigma, theta and frequency values and convolved each superpixel with each filter. Then I multiplied the results with the mean intensity values of each superpixel. (Not sure the implementation is fully correct) However, I believe there was an ambuguity about what to do with the results of each pixel in [6], as I could not

fully decide what to do with them later to convert them into weights. Thus I decided to sum them like in [7]. Basically, I tried to following formula in [6]:

$$F(i) = [\|I * f_1\|, \|I * f_2\|, ...\|I * f_n\|](i) \tag{4}$$

where the f_i are the results of Gabor filters. (instead of DOOG filters as in [6]).

I also tried to use mean, variance, kurtosis and skewness values for determining texture similarities, however could not obtain desirable results. I settled on the one with Gabor filters because [6] relied on a similar technique and got the best results with Gabot filters.

3) Step 3: In the previous step, we had created a region adjacency graph whose weights are based on spatial and texture similarities between superpixels (we have obtained in step 1). By applying normalized cut algorithm to them, we basically want to partition the graph into subgraph that each have maximum weights by cutting the dissimilar edges. This is a min cut problem, however min-cut does not work properly in this case because it has a bias of cutting small isolated partitions. Therefore, [6] introduces normalized cuts which aims to overcome this bias during segmentation. They define a dissociation and association measure to partition the graph. Normalized cuts show how tightly on average nodes within the cluster are connected to each other. However minimizing normalized cut become an NP-complete problem. Therefore they decide to relax the problem into a generalized Eigenvalue system defined by:

$$(D - W)\lambda = D\lambda \tag{5}$$

.

Then, we use the eigenvector with the second smallest eigenvalue to bipartition the graph. Then we repartition recursively if necessary.

Oversegmentation is the process of segmenting the image to more segments than we want. By oversegmenting, we split the image into many subgroups which have similar characteristics depending on the similarity measures of the segmentation. This is generally a preliminary step before extracting the segment we want. By oversegmenting we also partition the objects we wanted. However in the later steps of the algorithm, we can reassembly these steps. In out final, applying the mean shift is the oversegmentation process. I basically partition the image into segments depending on their position and color first, then reassemble them according to their texture.

A superpixel can be defined as a group of pixels that share common characteristics (in our case the common characteristics are pixel values and positions) They generally carry more useful information than a single pixel. For example, like in our case, we can define texture similarity measures between different superpixels. Furthermore, they faciliate the computational complexity.

A texel is a group of pixels that carries valuable information about the texture of the region. Structural properties of texels are very important for texture segmentation. In our case we are trying to extract texels from the superpixels that were based on pixel values that we obtained before.

There is no definite value for the number of segments k in Step 3. As the k increases, the area of superpixels decreases and it can be hard to find correct texture similarities with very small segments. However, if k is too small, unnecessary regions are interconnected with each other. Therefore, I think it is hard to find a definite number for k.

C. Experimental Results

1) Step 1: For computing the mean shift clustering, I used MeanShift function of Scikit learn. The function internally uses a flat kernel for clustering.

Normally, during mean shift algorithm, a window is placed in each pixel, however to make the implementation run faster, I set the bin_seeding = true. On this mode, initial kernel locations are not locations of all points, but rather the location of the discretized version of points, where points are binned onto a grid whose coarseness corresponds to the bandwidth. I also set the max_iter to 100 hoping it would speed up the process but it did not change much. I chose bandwidth to be 30 by experimentation. The results can be seen in Figure 5

2) Step 2: As mentioned in the theoretical section I got very small values for similarity measure, probably due to a problem in the implementation, that is why I had to scale my results with σ values of similarity measures. I chose very high σ values simply to get similarity weights other than 0. (1000000 for texture, 100000 for spatial similarity). The r value for spatial similarity is 100 pixels obtained by experimentation. Furthermore, the parameters for my Gabor filter banks are as follows: sigma values between 1 and 3, theta values between 0 and 4, frequency values between 0.05 and 0.25. I chose these values according to the tutorial of skimage in [7]. The results can be seen in Figure 6.

3) Step 3: I only gave the graph obtained in Step 2 and segmented image obtained in 1 as input. The results can be seen in Figure 7. As can be seen, in images 1, 3 and 4 the results are promising. However images 2 and 5 are not very close to what we want.

4) Tools: For this part of the final, I used skimage for N-cut segmentation, scikit learn for mean shift clustering, cv2, numpy, scipy and matplotlib for intermediary operations.

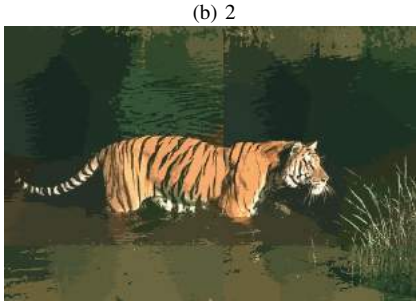
5) Complexity Analysis: Mean shift algorithm tends towards $O(T * n^2)$ according to scikit learn. N-cut algorithm is $O(VE^2)$. The operations I made for constructing the graph is also $O(n^2)$. Thus this algorithm has a complexity of $O(T*n^2)$ which is pretty slow.(Sorry Hazal Hocam, please be patient with my code)

D. Conclusion

Overall, my algorithm worked better for the Dataset 2 than I hoped it would. However, I believe there is a fundamental problem with the way I calculate similarities as I do not think sigma values should be this high. For the texture similarity part, I should have found a better approach to combining the results of the filter banks then summing the difference,

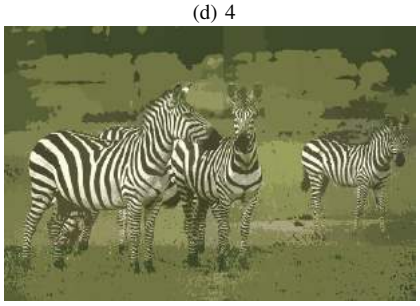
(a) 1

(b) 2



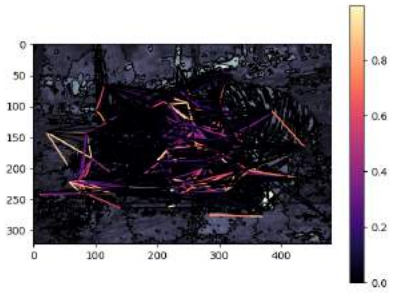
(c) 3

(d) 4

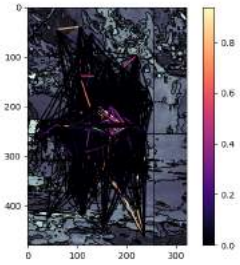


(e) 5

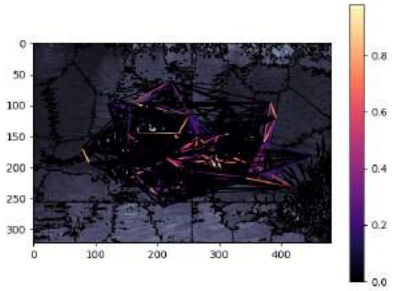
Fig. 5: Mean Shift segmentation results



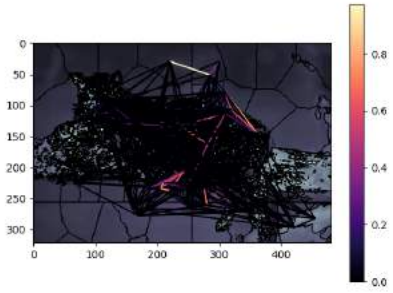
(a) 1



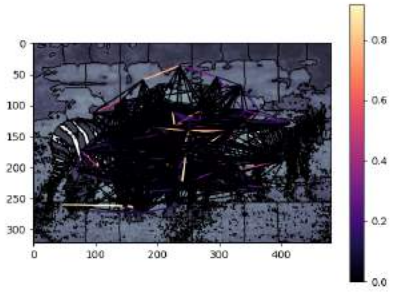
(b) 2



(c) 3



(d) 4

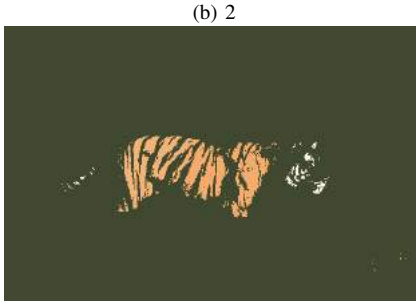


(e) 5

Fig. 6: Region Adjacency Graphs

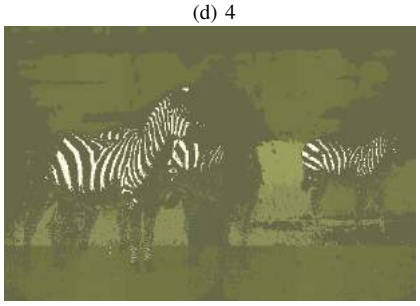
(a) 1

(b) 2



(c) 3

(d) 4



(e) 5

Fig. 7: Normalized cut results

however in any case it works better than the approach I tried with mean, variance, kurtosis and skewness values. The algorithm is pretty slow, it might be better to use "quickshift" which is a similar algorithm to meanshift clustering in the way that they are both mode seeking algorithms. However, in comparison quickshift was much faster.

III. QUESTION 3

A. Introduction

In this part, we are required to extract the roads from the images.

I think the hardest parts seemed to be separating roads from the rest of the image. I decided to apply Color space segmentation in HSV color space for step 1. In the literature, CIELAB color space was also widely used for this purpose.

B. Theoretical Presentation

1) Step 1: In this section I decided exploit the color information of the images. The roads in this dataset and in general are gray. For this reason I decided to get remove the blue, green and white colors as much as possible. As removing certain color from the image is easier in the HSV color space, I first converted the RGB image into HSV color space. Then I implemented a mask containing certain shades of green, blue and white and applied it to the image. I mostly decided which values correspond to which colors by checking out the spectrum of HSV. Then I experimentally tried to find the best range for these values.

Then I applied respectively closing and opening to these images to connect some of the roads together and get rid of small objects. Then I further got rid of objects smaller than 150 pixels. While deciding on the kernel sizes of opening and closing, I did not want to choose very big values since they might result in very big changes in different images. Thus, I just chose a kernel size of 3X3.

At last I applied Canny edge detection to extract the edges of the roads. As this was already a binary image, the parameters of Canny edge detection did not make a difference in this case.

Algorithm 4 Step 1 - Heuristic

$green\_mask \leftarrow filter\_green(image)$
$blue\_mask \leftarrow filter\_blue(image)$
$white\_mask \leftarrow filter\_white(image)$
$final\_mask \leftarrow green\_mask + blue\_mask + white\_mask$
$image \leftarrow apply\_final\_mask(image)$
$image \leftarrow apply\_closing(image)$
$image \leftarrow apply\_opening(image)$
$image \leftarrow bwareaopen(image)$
$image \leftarrow canny(image)$

2) Step 2: I used progressive probabilistic Hough transform introduced in [8] to detects the lines in the image. This algorithms tries to minimize the computational cost of Hough transform. For this purpose, the algorithm repeatedly selects

a random point for voting . Then tries to decide if this point is due to the random noise. Using this transform instead of normal Hough Line transform, I got less arbitrary lines in my images since I was not able to completely get rid of the noises in the image that does not contain roads.

3) Step 3: Could not complete this part.

C. Experimental Results

1) Step 1: I used HSV color space to filter the colors. For the colors to be masked, I chose to following intervals in HSV space:
white: range of [0, 0, 200] to [145,60,255]
blue: range of [80, 0, 200] to [125,60,255]
green: range of [35, 0, 200] to [86,60,255]

For kernel value I chose 3X3 to not lose a lot of information during opening and closing. I also got rid of the pixels smaller than 150, which might cause problems later. The results can be seen in Figure 8

2) Step 2: I used cv2 HoughLinesP function for this step and chose experimentally 40 for minLineLength and 10 for maxLineGap. Results can be seen in Figure 9

3) Step 3: Could not complete this part

4) Step 4: Results can be seen in Figure 10

5) Tools: I used cv2, numpy and math libraries for this part.

D. Conclusion

Since I could not complete after step 3, It did not work properly.However the first 2 steps were better than I expected. I worried the implementation would not work if I could not extract all the noise around the roads however, progressive probabilistic Hough transform managed to find the main roads.

REFERENCES

[1] X. He, L. Zheng, Qiang Wu, Wenjing Jia, Bijan Samali and M. Palaniswami, "Segmentation of characters on car license plates," 2008 IEEE 10th Workshop on Multimedia Signal Processing, Cairns, Qld, 2008, pp. 399-402, doi: 10.1109/MMSP.2008.4665111.

[2] Cavalcanti, G.D., Silva, E.F., Zanchettin, C., Bezerra, B.L., Doria, R.C., Rabelo, J. (2006). A Heuristic Binarization Algorithm for Documents with Complex Background. 2006 International Conference on Image Processing, 389-392.

[3] R. C. Gonzalez, Digital Image Processing, 3rd ed., Prentice-Hall, Inc., 2006, pp.651-653.

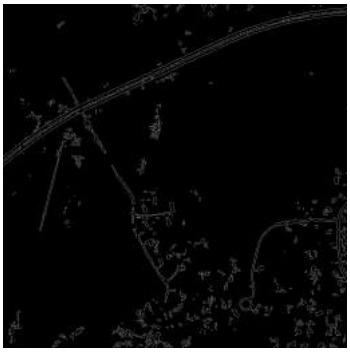
[4] R. C. Gonzalez, Digital Image Processing, 3rd ed., Prentice-Hall, Inc., 2006, pp.823.

[5] W. Tao, H. Jin and Y. Zhang, "Color Image Segmentation Based on Mean Shift and Normalized Cuts," in IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics), vol. 37, no. 5, pp. 1382-1389, Oct. 2007, doi: 10.1109/TSMCB.2007.902249.

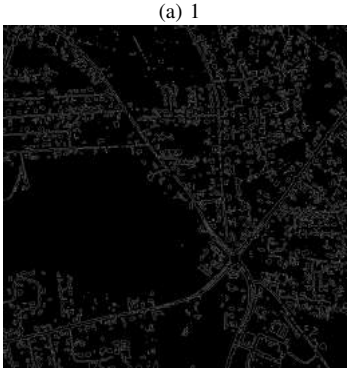
[6] Shi, Jianbo & Malik, Jitendra. (2002). Normalized Cuts and Image Segmentation. IEEE Transactions on Pattern Analysis and Machine Intelligence. 22. 10.1109/34.868688.

[7] https://scikit-image.org/docs/stable/auto_examples/features_detection/plot_gabor.html

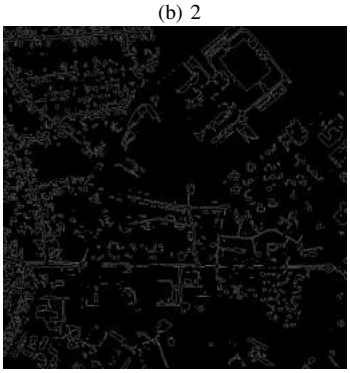
[8] Matas, Jiri & Galambos, C. & Kittler, J.. (2000). Robust Detection of Lines Using the Progressive Probabilistic Hough Transform. Computer Vision and Image Understanding. 78. 119-137. 10.1006/cviu.1999.0831.



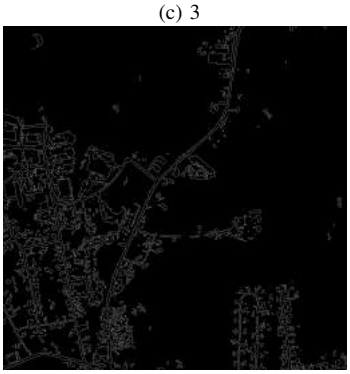
(a) 1



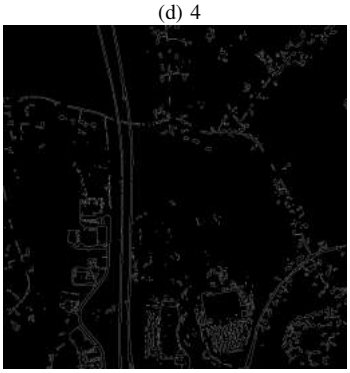
(b) 2



(c) 3

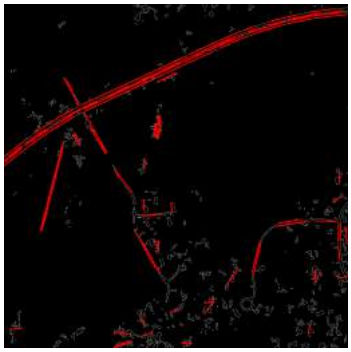


(d) 4

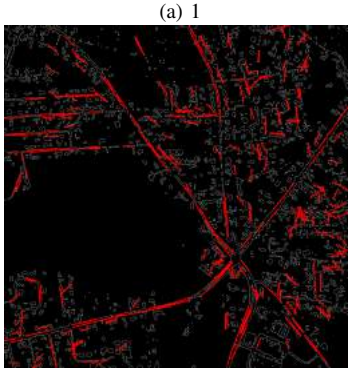


(e) 5

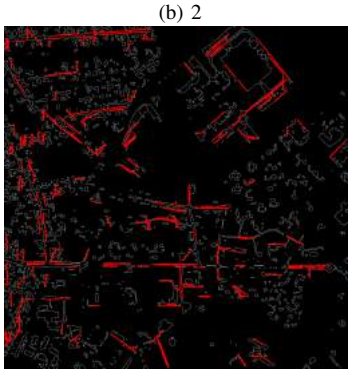
Fig. 8: Images after Step 1



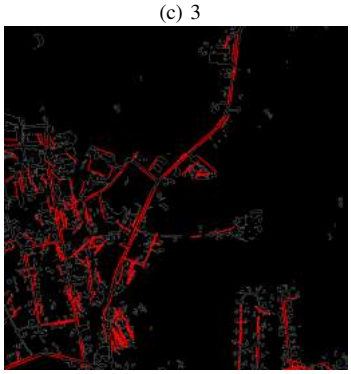
(a) 1



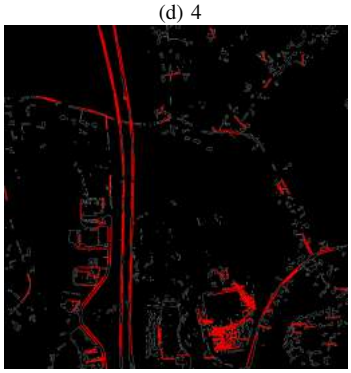
(b) 2



(c) 3



(d) 4

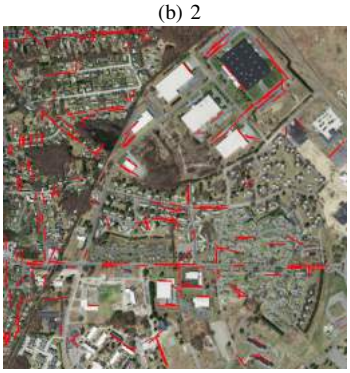


(e) 5

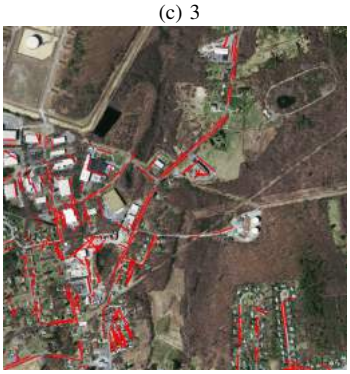
Fig. 9: Images after Step 2

(a) 1

(b) 2



(c) 3



(d) 4

(e) 5

Fig. 10: Images after Step 4