

DIP Homework #4

ID #: B04902028

Department & grade: CSIE 3

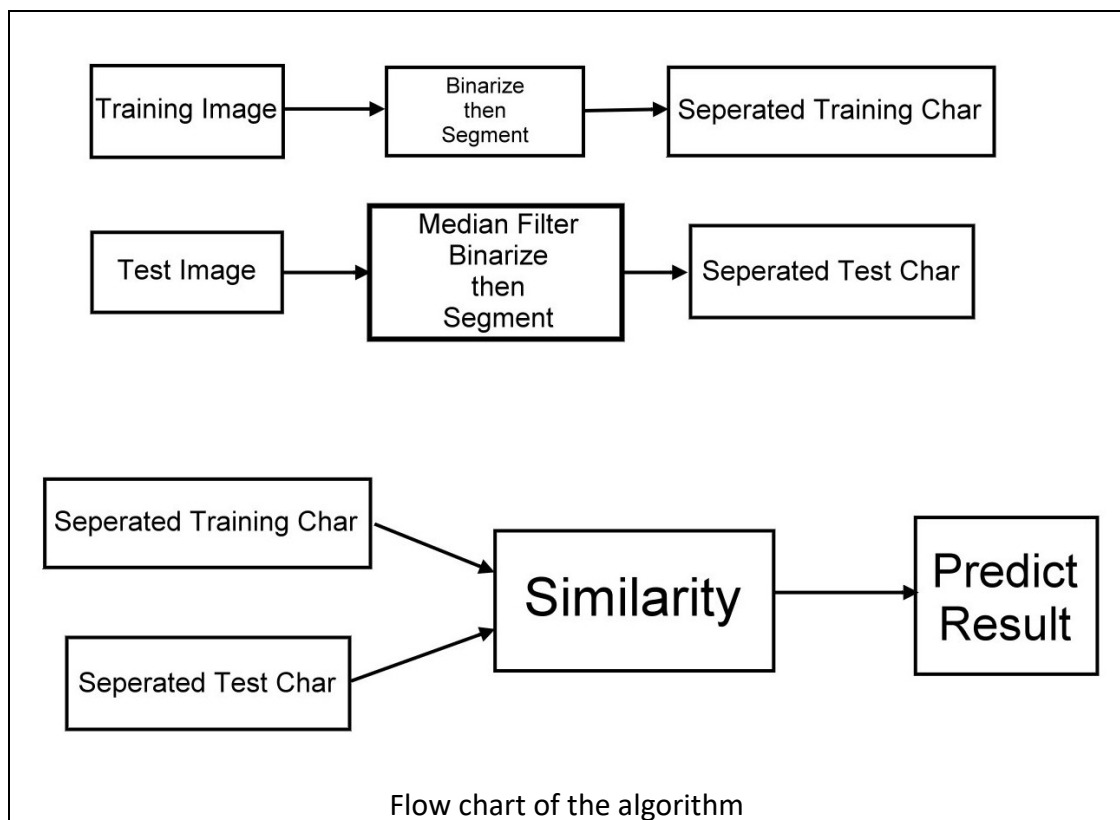
Name: 洪浩翔

Email: b04902028@ntu.edu.tw




Submit time: 05/30/2018

PROBLEM 1: Optical Character Recognition (OCR)

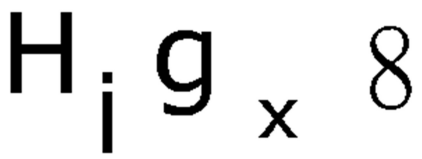
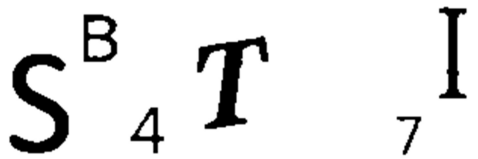


Please design an algorithm to recognize the CAPTCHA images as shown in Fig. 1. The training set is given in Fig. 2. Please describe the proposed algorithm in detail along with a flowchart and discuss the recognition results.



First, I deal with the training set. It is hard to operate on the whole training set without segmentation, so I binarize the training set image and segment the training set into 70 characters first. The detail of the segmentation algorithm is very simple. I separate the training set image into 70 pieces, and remove the redundant part of these pieces by while() loop. In the end, I get 70 characters from the training set image.

 <p>Result of binarized training set image.</p>	 <p>Segmented character 'A'</p>	 <p>Segmented character '\$'</p>
--	---	---

Test images have the same problem of hard to operate. In addition, there is a lot unnecessary information in test image. As a result, I process median filter on test images first and then binarize them. This pre-process can remove the redundant information in test images, making them much smooth. Second, segment the test images into many little character pieces through recursion. The resultant pieces will be very similar with the training set pieces. In the end, I can use these test pieces to compare with training pieces.

 <p>Test image 1 after median filter and binarizing</p>	 <p>Test image 2 after median filter and binarizing</p>
 <p>Test character pieces after segmentation</p>	 <p>Test character pieces after segmentation</p>

The last step is to compare test pieces with training set pieces one by one and output the most likely character. That means each test piece has to compare with 70 character images. The comparison method is very simple. I just calculated the number of same pixels at the same place. If the number is bigger, much likely it is the corresponding character. However, this is hard to conduct because each piece has its own size. As a result, resizing images is needed. Thus, I use nearest neighbor interpolation to deal with the resize problem. In fact, this is very similar with what `imresize()` does. After dealing with resizing and comparing method, the rest is just to

run a double for() loop to compare total 11*70 times and output the most likely character. The output result is in the below table.

Test image	result
Test image 1	H , l(small 'L') , g , X(big 'X') , 8
Test image 2	(some trash) , S , B , 4 , T , 7 , l(big 'l')

Most of the output characters seem to be correct. However, there are still some problems. In image 1, 'i' is predicted as 'l'(small 'L') instead of 'i'. This is a very interesting finding that 'l' can get higher score than 'i' in this case, for 'i' seems to be a character easy to map onto and recognize. The other problem is the program can't distinguish x and X. This is because x and X seem to be the same character after resizing. However, this problem may not be so important because most of the time we can't distinguish x and X without other characters nearby as well. Thus, this may not be the case. In image 2, I found that there is a trash which is a segmented error. It is a very small black point that I can't understand where it comes from. In the end, I have no choice but remove it manually.

Instead of conducting experiment myself, I also compare my result with others. I found that a good interpolation when resizing can lead to a better result. In addition, I also compare the results of using imresize() or nearest neighbor interpolation. Of course I didn't submit the imresize() version. I found that the resultant outputs are the same. This implies that interpolation is enough to deal with this case. There is no necessary to use imresize().