# DIGITIZE YOUR HANDWRITING

Amrut Shenoy, Computer Science, Rochester Institute of Technology, Spring 2016

**ABSTRACT**:

The objective of this paper is to convert handwritten text in an image into a digital format in the form of a document. For this purpose, Optical Character Recognition (OCR) is developed to meet our demands. A training set containing alphabets of the English language is used as a basis for comparison with the texts in the image. Based on this comparison, the alphabet that closely resembles the handwritten text is chosen as the alphabet that it is most likely to be, and is written into the digital document. An ideal model would be able to represent all the text in the image in a digitized format correctly. The accuracy of this model can be found based on the similarity of how each character should have been represented in the digitized format and how it was actually represented by the model. The more the accuracy, the better the model performed.

**INTRODUCTION**:

An Optical Character Recognition (OCR) system saves a lot of time and human effort by being able to recognize characters based on some differentiating feature. There have been successful attempts in the past for recognition of digital texts and converting them into a document. However, my aim was to enable a user to be able to convert a handwritten text into a digitized

format. This would help the user by saving his/her time and most importantly their effort. My initial plan was to make use of Tesseract[2] as an OCR tool. However Tesseract wasn't working at all because it couldn't differentiate between the pixels, so I had to develop another algorithm in order to implement a recognition mechanism. A roadblock that I came across was not being able to differentiate between what can be classified as a 'text' and what can't be. In order to this, I planned to differentiate between the foreground pixels as the text that was important and the background pixels as something that was unwanted. Once this was done, I wanted to take up each line of text separately and try to segment each letter individually. In this way, each letter can be compared with the letters present in the set of training images. The training images contains English alphabets in a standardized format i.e. all uppercase characters and 24x42 pixels. These images are compared with the letters that we obtained by segmenting. Based on the comparison, the model tries to recognize the characters. I tried using different techniques for comparison such as comparing histograms of the two letters and also to segment the letters in the image but that didn't work. Eventually I came across a model[1] that helped me do this procedure.

**PREVIOUS WORK**:

There have been previous work in this regard. In their paper[3] on OCR for Devanagari script, the authors have tried to state the importance of proper segmentation of characters in the image. They proposed it to work for five different fonts and sizes of Devanagari script. They talk about different approaches in which OCR system can be used. One approach made use of Structural

Analysis to identify the sub-features of an image based on the histograms. Fuzzy-logic can also be used as an OCR by making use of the conventional evaluations like yes/no, true/false etc. They also mention using neural networks as an OCR by sampling the pixels in each image and match them against a known set of image.

Another paper[4] on OCR talks about the usage of Tesseract as an OCR. It mentions about the different steps involved in performing an OCR. It involves scanning the image, performing segmentation of the image. Once this has been done, a preprocessing step is performed to remove noise by smoothing and normalization of the image. This helps to identify the features from the image which forms the next step. Finally, based on the features extracted, pattern recognition is used. This is where Tesseract algorithm was used to recognize characters.

**EXPERIMENTS AND RESULTS**:

1. Applying inbuilt OCR: MATLAB has an inbuilt OCR technique that provides pretty accurate results for recognition of texts. In order to check, I applied OCR on an image provided by MATLAB itself. The results were quite accurate. However, when I applied it on images containing handwritten texts, it failed to recognize any character properly. This might be because of the thresholding that it applies. It is unable to differentiate between handwritten text and the background pixels. So, I had to think of other alternatives to implement OCR.

2. Removal of noise: Removal of noise is an important step as the results will be skewed if we didn't remove noise efficiently. I tried removing noise from the image using average filtering but it didn't improve the quality of the image. Gaussian filter provided better results than the

other filters that I used for removing noise. Hence, I used Gaussian filter to smoothen the image.

2. Text Identification: Initially I tried to run my OCR technique directly on the image provided. However, the system wasn't able to easily differentiate between the handwritten text and the lines on the paper. I realized that it was important for the system to be able to successfully distinguish between the handwritten text present in the image and other non-important pixels/objects in the image.

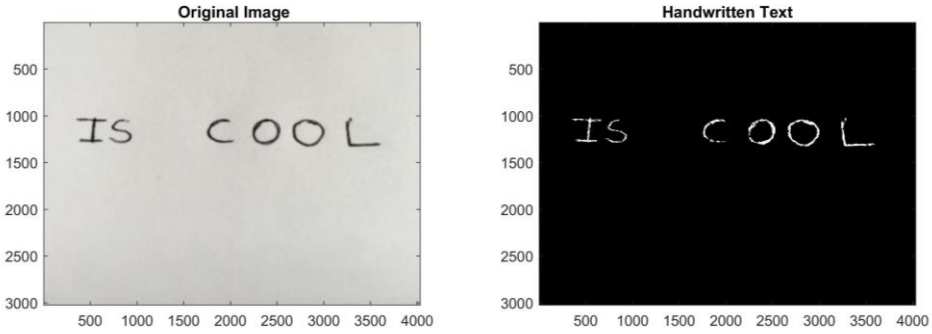


Fig 1: Text Identification on Original Image helps to identify the text.

For this I made use of a technique that made me differentiate between the foreground (handwritten text) and background pixels by finding the Mahalanobis distance of each pixel from the foreground and background pixels.

3. Preprocessing: In order for the OCR system to work correctly, it was important that the image obtained from the text identification step contained a logical image that had a 1 wherever there was a pixel for the text and a 0 for the rest of the pixel. However, majority of the times, if there was even a small mistake in choosing the foreground pixels, then there

would still be some noise in the image that was randomly distributed.

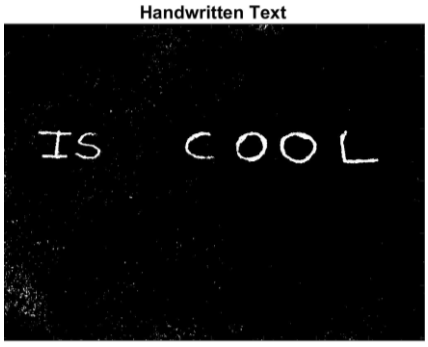


Fig. 2.1: Image with noise

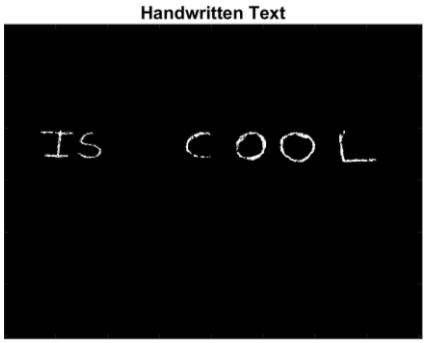


Fig. 2.2: Image without noise

In order to remove this noise, I tried morphological operations on the image such as erosion and dilation but this also affected the handwritten text present in the logical image. Hence, I thought of using the connected components of the pixels. I set a threshold value for the connected components. Any pixel value below this threshold was set to 0. Eventually, I found a proper threshold value for which the noise in the image was removed.

4. Segmentation of Characters: I tried to implement segmentation on the image obtained by various techniques. One was edge detection. But that failed because the characteristics of edge could change and it would not be entirely uniform too. I had planned to use histogram matching but that failed to produce results that would be of any use. Or maybe I didn't apply them correctly. I also applied morphological operations on the image but that was of no use as I had to preserve all the letters and treat each one separately too. This wasn't helping. I thought of using morphological thinning too but again it wouldn't help me segment the characters.

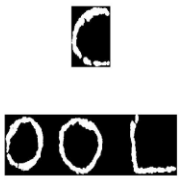

Fig 3: Segmentation of each letter while rest of the characters are yet to be segmented.

I realized that I somehow had to make use of the text present in the image in order to segment the characters efficiently. But I was unable to implement this in practice. I got an idea[1] from another code[1] that I used as a reference from coding perspective. It made use of the boundaries of the texts present in the image. It considered such a boundary that no text in the image exceeded this boundary from any end. Invariably, the length of the highest and widest pixel among all the characters formed the boundary of this image. This helped to segment a line of text. Similar approach was used to differentiate between the alphabets individually. This is how each letter was segmented.

5. Matching features: This involved matching the image in the training set to the image that was obtained (a character). This character had to be checked with each image in the training set in order to determine which character it resembled. I tried using a matched filter but it failed to provide me any significant result. It almost always gave incorrect result. This must have been because the characters didn't exactly match the images in the training set. Finally, I came across the concept of using correlation coefficient based on a code[1] that I used as a reference.

$$r = \frac{\sum_m \sum_n (A_{mn} - \bar{A})(B_{mn} - \bar{B})}{\sqrt{\left(\sum_m \sum_n (A_{mn} - \bar{A})^2\right)\left(\sum_m \sum_n (B_{mn} - \bar{B})^2\right)}}$$

Fig. 4: Formula for calculation of correlation coefficient[5].

Here A and B are the images being compared while m and n indicate indexes of pixel location in the image. For every pixel location in both images, the difference between the intensity values at that particular pixel and the mean intensity of the whole image is calculated[5].

From what I understood, the correlation coefficient represents the similarity between two images with respect to the pixel intensity. The image from the training set that had the maximum correlation coefficient turned out to be the one that most likely represented the letter or the alphabet correctly.

6. Displaying the digitized output: For this I made use of some built-in commands present in MATLAB[6, 7]. I had to basically open a document and keep writing the result of each operation i.e. conversion of each letter to digital format into this document.

7. Final Result:

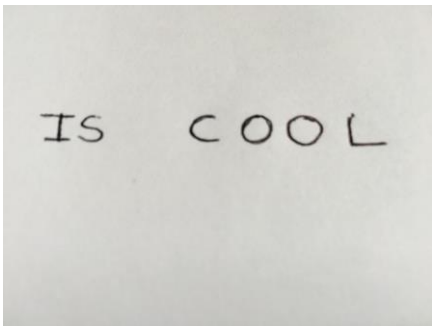


Fig 5.1: Input Image

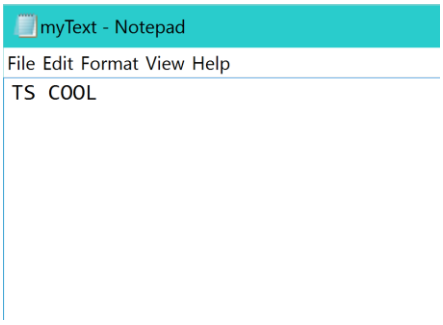


Fig 5.2: Output Text in Digital Format

As can be seen from the image above the model doesn't give 100% accuracy. In places where there is a confusion between two characters, it can choose any one based on whichever character had a higher correlation coefficient. In this case, the model incorrectly identifies 'I' as 'T'. This was because while segmenting, the lower edge of I wasn't big enough to strongly identify it as an edge.

**DISCUSSION**:

I had initially planned to use Tesseract for OCR purpose. However it didn't provide any output at all which meant that I had to plan for implementing the OCR system right from scratch. For this purpose, I referred an implementation of OCR[1] right from scratch. Inbuilt OCR of MATLAB didn't work because it was unable to differentiate between the background and foreground pixels. Hence, I thought of using the Mahalanobis[8] distance to differentiate between the background and foreground pixels. Even after playing around with the parameters, I realized that there was some noise present in the image. I removed it by setting a threshold for connected components. This removed the noise from the image.

I also dilated the image to enhance the edges of the text. This was done because at times the edges of the alphabet weren't connected to each other. For segmenting a line a lot of techniques were tried. Edge detection did work sometimes but it didn't provide satisfactory results. Hence I used the idea provided in the OCR implementation[1] I mentioned above of using the boundary of an entire set of a line of texts.

I obtained a set of images[1] of English alphabets as a training set for comparison of the handwritten text. For comparison of the text I did try histogram matching but it didn't work as

the images could have been similar and not an exact representation. It did give some letters but it wasn't accurate. C was classified as L, I was classified T, R was classified as K. I tried to rectify this but I think it was something that I didn't implement correctly. I tried this many a times and for different images as well but it didn't give accurate results. Finally, I got an idea of using correlation coefficient from the implementation of OCR[1]. This was something new that I learnt and it was fun. To be able to use the correlation coefficient to determine the resemblance of a character to an image present in the training set is something that was a bit difficult to understand initially but eventually was amazing to learn.

Also, I relearnt the importance of height and width of image pixels as I made a silly mistake while writing a 'for' loop that didn't run perfectly. I was stuck up for a long time until I figured out the impact such a small mistake had on my code. It was silly yet an important reminder.

**FUTURE WORK**:

A lot of future work can be done in this regard. Identification of lower case characters along with numbers can be done. Also we can include the recognition of special characters such as "?", "~", ":" or ";". Apart from this, a major improvement would be to identify the cursive handwritten text. The scope would be huge but having a set of handwritten images could be helpful. Basically replace the images in the training set with a set of images for a particular alphabet. This can help to identify a character written in a cursive manner. We can also try to implement it for digital fonts in the image, just like how a standard OCR system works. So, this can work for both handwritten as well as digital text to digital document. It can also be used for

recognizing patterns of handwriting of an individual in different situations by performing an analysis on how the handwriting changes.

**CONCLUSION**:

To summarize, I converted a handwritten text to digital format. For this I made use of Mahalanobis distance to differentiate between the handwritten text pixels and the background pixels. Then I segmented the image on the basis of characters read line by line in an image. Each character was segmented individually and compared with an existing set of training images. Through the use of correlation coefficient, the letters were predicted. The higher the correlation the coefficient, the more likelihood of it being the actual character.

The important learning from this project was how we could make the system smart enough to be able to differentiate between the characters. An important concept that I learnt was determining how to segment the images based on a common property and in some cases, how to not use segmentation on the image. I also learnt about the usage of built-in MATLAB commands such as fopen(),fclose(),winopen(), mat2cell() and most importantly corr2() to find the correlation coefficient. I can only imagine the scope that this project has in the future.

This project was important in more ways than one. It considerably enhanced my knowledge about Computer Vision.

**REFERENCES**:


[1] http://www.mathworks.com/matlabcentral/fileexchange/31322-optical-character-recognition-lower-case-and-space-included- Link for OCR implementation. Last looked up on April 6, 2016 at 10.10 pm EST.

[2] https://github.com/tesseract-ocr/tesseract Link for tesseract. Last looked up on April 6,2016 at 8:45pm EST.

[3] Optical Character Recognition (OCR) for Printed Devnagari Script Using Artificial Neural Network by Raghuraj Singh1 , C. S. Yadav2 , Prabhat Verma3 , Vibhash Yadav4 published in International Journal of Computer Science & CommunicationVol. 1, No. 1, January-June 2010, pp. 91-95.

[4] Optical Character Recognition by Ravina Mithe, Supriya Indalkar, Nilam Divekar . International Journal of Recent Technology and Engineering (IJRTE) ISSN: 2277-3878, Volume-2, Issue-1, March 2013.

[5] http://stackoverflow.com/questions/27343283/explaining-corr2-function-in-matlab Link describing the formula of correlation coefficient. Last looked up on April 29, 2016 at 9:30pm EST.

[6] http://www.mathworks.com/help/matlab/ref/fopen.html?searchHighlight=fopen Link for MATLAB documentation. Last looked up on May 1, 2016 at 11:20am EST.

[7] http://www.mathworks.com/help/matlab/ref/winopen.html?searchHighlight=winopen Link for MATLAB documentation. Last looked up on May 1, 2016 at 4:10pm EST.

[8] Prof. Kinsman's Lecture Code for finding Mahalanobis distance.