A Handwriting Recognition System for Dead Sea Scrolls

Sudhakaran Jain (S3558487))
University of Groningen
E-mail: s.j.jain@student.rug.nl

June 3, 2020

1 Introduction

Historical documents are one of the most valuable information given to mankind from the past. We can see a rise in digitization of such historical resources mainly to facilitate their distribution to promote research remotely. The degradation of these historical documents are one of the major issues in this operation. These resources can be damaged due to tearing of pages due to improper storage, change in color of the pages over time and also because of biological agents. To tackles these problems and digitize these documents, various researchers have come up with different algorithms to recognize what is written on these documents.

Our project focuses to build an automatic handwriting recognition system to recognize one such historical document named $DeadSeaScrolls$[1]. These scrolls are ancient Jewish manuscripts found in the Qumran Caves in the Judaean Desert near the northern shore of Dead sea . These scrolls are written in Hebrew language. They belong to the period of 400-300 BCE which makes them very significant for research. Figure 1 shows how the scrolls which we used look like.

Our approach involved three major processes. Firstly, we had to pre-process the image, which involved extracting the parchment and then binarizing it. We then performed segmentation process where we tried to extract each line in the parchment and further extract words from these lines. The final step is classification, where all these segmented word-images are recognized to get the whole text of parchment.

[1] `https://en.wikipedia.org/wiki/Dead_Sea_Scrolls`

2 Methods

In this section, first, we will give the explanation of our pipeline of the system about how it works overall, including its input, output, components and their functions. Then, we will talk about how these components are implemented in details.

2.1 System Pipeline

The system processes goal images one by one automatically. The input of one assignment of this system is a grey level photo of the Dead Sea Scrolls in a good quality, as shown in Fig 1.

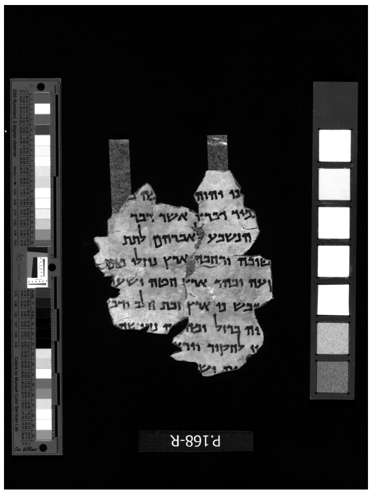


Figure 1: The example picture of the Dead Sea Scrolls as an input

Since there is redundant information on the image, we preprocessed it to isolate the content of our interest which is the parchment. After the isolation of our interested area, we then detected and segmented the text lines on it; for each line, we also segmented it in word levels. The image fragments of words were passed to the classification step. In the classification step, we used sliding window approach to ensure the classification work is done on character level. Each cropped image obtained from the sliding window was passed to classifier for character recognition. Finally, the classified results were saved in a .txt file in the same order as the input image. Note the text of the output file should be read from right to the left, since that is how Hebrew is written. Fig 2 visualizes this system pipeline as an overview.

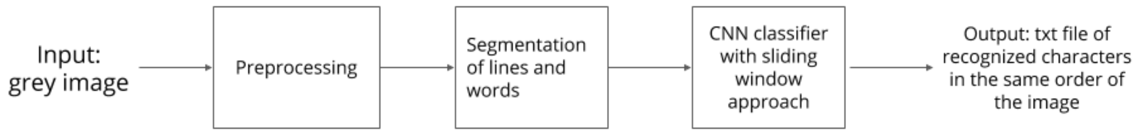

Input: grey image
Preprocessing
Segmentation of lines and words
CNN classifier with sliding window approach
Output: txt file of recognized characters in the same order of the image

Figure 2: Pipeline of the handwriting recognition system

2.2 Pre-processing

2.2.1 Image processing and information extraction

There are five main steps of this part as shown in Fig 3.

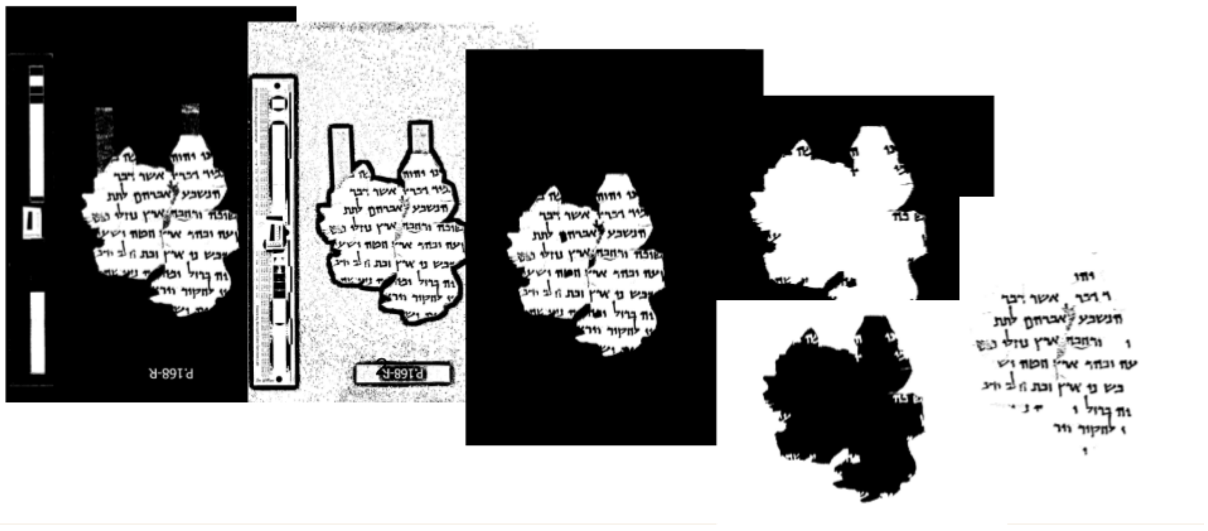


Figure 3: Five main steps of image processing and information extraction. From left to right each image represents: Step 1 - using Otsu binarization on the example image; Step 2 - using Sauvola binarization on the example image; Step 3 - finding the biggest component from the Otsu binarized image; Step 4 - whitening the largest component found in previous step, then inverting its black and white; Step 5 - masking operation of the Step 2 image and the the lower sub image of the Step 4 image.

Otsu Binarization Binarization of an image refers that for the pixels of an image, if a pixel's value is greater than a certain threshold, then it is set to 1 as a foreground pixel, if not it is set to 0 as a background pixel. Determine of the threshold is critical. The threshold could be global as how Otsu algorithm works (Otsu, 1979). Otsu algorithm calculates the optimal threshold for separating the foreground pixels and the background pixels, such that the variance within one class is minimal, and the variance between two classes is maximal. In our practical implementation, we applied the library $threshold_otsu$ [2] from the open source module $skimage.filters$[3].

Sauvola Binarization Sauvola algorithm calculates the threshold for a certain pixel dynamically. Sauvola algorithm centers on the current pixel, calculates its binarization threshold based on the grey level mean and standard variance of its neighborhood pixels within a certain sized window (Sauvola, Seppänen, Haapakoski, & Pietikäinen, 1997). In our practical implementation, we applied the library $threshold\_sauvola$ [4] from the same module.

Connected Component A Connected Component generally refers to an image region composed of foreground pixel points having the same pixel value and adjacent in the image. We need to traverse the pixel matrix of one image. For one pixel, we explore its adjacent pixels (4 directions in our case but could also be 8 directions), to see if they have same values with it: if so, we labeled them as the same with the current pixel; if not, we labeled them differently (Suzuki, Horiba, & Sugie, 2003). In our practical implementation, we imported the library $connectedComponentsWithStats$ from the module $OpenCV$ [5], which could also return the total area (in pixels) of a connected component.

Whitening and BW Invertion We set the value of the pixels within the area of the biggest connected component as white (background is black), then inverted its white part into black and vice versa, for the convenience of using is as a mask.

Masking We compared the pixels of the image matrix one by one of the Sauvola binarized image with the BW inverted mask image from former step. If the pixel in the mask image is white, then we set the pixel with the same location in the Sauvola binarized image into white too. Thus we deleted the noise in the background while kept the text information of our interest.

[2] https://scikit-image.org/docs/dev/api/skimage.filters.html#skimage.filters.threshold_otsu

[3] https://scikit-image.org/docs/dev/api/skimage.filters.html

[4] https://scikit-image.org/docs/dev/api/skimage.filters.html#skimage.filters.threshold_sauvola

[5] https://docs.opencv.org/3.0-beta/modules/imgproc/doc/structural_analysis_and_shape_descriptors.html

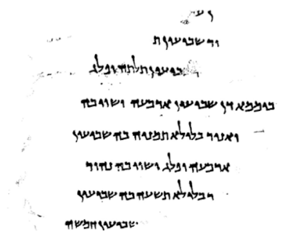


Figure 4: Isolated image rotated by-2 degree

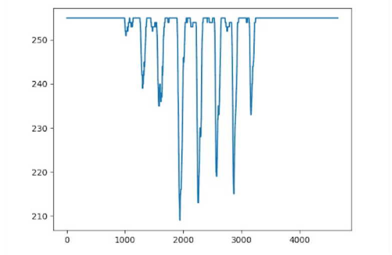


Figure 5: Histogram representing the reduced image matrix with -2 rotated degree

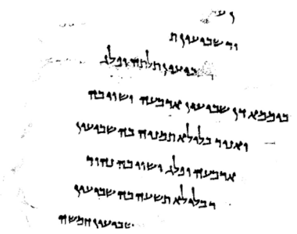


Figure 6: Isolated image rotated by 4 degree

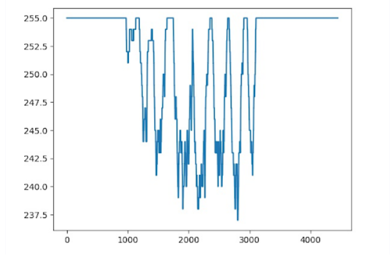


Figure 7: Histogram representing the reduced image matrix with 4 rotated degree

2.2.2 Geometric correction

Though we believe the image was taken as carefully as it can be, tilting of the area of interest could still be a problem sometimes. Thus we needed to revise it into an optimal degree. We firstly rotated the image by certain degrees, for example as shown in Fig 4 and Fig 6 - rotated by -2 degrees and 4 degrees. There are two major steps to find the optimal rotation degree.

Reduce Operation For each image with one specific rotating degree, we reduced the matrix of the image to a 1D vertical vector by accumulating each row. Thus, for one element of this vector, its value can be used to represent the text information load, and its index means which row it was summed from. Finally, we converted the values (we divided them by the length of the row of the image matrix to use it as a mean) and their indices of the reduced vector into a histogram. Fig 5 and Fig 7 are the corresponding histograms of the text images in Fig 4 and Fig 6 respectively. We used the library *reduce* from *OpenCV*[6] in practical implementation. In Fig 5 and 7, the horizontal axis represents the row index and the vertical axis represents the value of the reduced matrix.

[6] https://docs.opencv.org/2.4/modules/core/doc/operations_on_arrays.html#reduce

Peak Detection It is clear to see the pointed strips (look from up to down) indicate the areas with dense black pixels. Since we did grey level conversion for the convenience of further classification, the white value here is 255 instead of 1. Looking at the histograms we can say that more the consistency of text load, more seperated are the stripes and thus better distinguished are the lines. We can see intuitively, by the comparing Fig 4 to Fig 7. We used an open-source library *peakdetect*[7] in our practical implementation: this method, first labels the wave points by looking into the pixel values; then it traverses the wave points determining the peaks according to their 'height' represented by the row index; it also has a parameter `lookahead` as the distance to look ahead from a peak candidate to determine if it is the actual peak or just a jitter. Here we set the `lookahead=50`, as it is the approximate height of a character. For each rotated image, we scored its histogram in this way: we summed the intervals of a pair of highest peak and lowest peak with same list index; when the score is the highest, the corresponding rotation degree is the optimal.

2.3 Segmentation

2.3.1 Segmentation of lines

The segmentation of lines was implemented as a follow-up of to the optimal degree rotation. Here, we looked for the peaks of its histogram to segment the image into fragments each containing one line. Take Fig 5 as an example, where we can see that the peaks are distinctly seperated from one another. Here, the 4^{th} peak around 2000 horizontal coordinate, which is the highest, indeed represents the 4^{th} line of Fig 4 being the longest. We used the locations of these peaks and extracted the pieces from the image.

2.3.2 Segmentation of words

The segmentation of words were implemented with the same approach as the segmentation of lines. There are three major differences compared with line segmentation:
1) the image matrix was reduced to a row vector first;
2) we set `lookahead=30` here as it was the approximate length of one character;
3) since there are also blanks between and inside the characters, that could cause peaks on the histogram not as high as 255. Since we only want to segment the line into words, we increased the value threshold of a peak to be 255 to prevent the problem of over segmentation.

[7] `https://gist.github.com/lorenzoriano/6126450`

2.4 Classification

2.4.1 Implementation of Classifier

The Hebrew language consists of 27 characters in total. So to recognize the characters in the segmented word images, we implemented a Convolutional Neural Network(CNN) (LeCun, Bottou, Bengio, & Haffner, 1998) which acts as a classifier with 27 classes. The architecture of this CNN is as shown in the Figure 8.

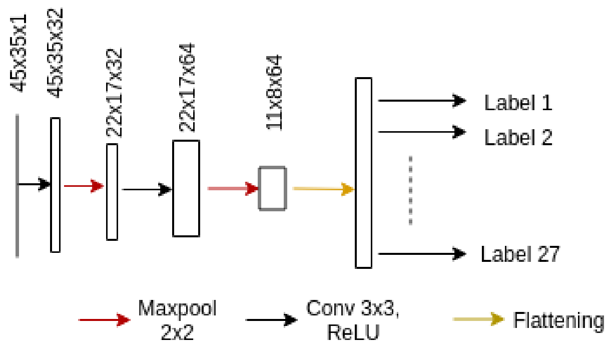


Figure 8: Architecture of our Convolutional Neural Network

We were provided with a dataset for training this CNN which had varying number of samples for each character along with their labels. Also, each sample image was of different resolution. After inspecting these images manually, we realized that all these images had resolution around 45x35x3. Another, careful observation we obtained was the number of channels present in these sample images. These images had 3 channels even though they looked like binary images. But, our preprocessing step converted the scroll images to binary which have single channel. So, in order to train the CNN to recognize characters in these binary images, it has to be trained using sample images which are also binary having single channel. Considering all these above aspects, we converted all the samples in the given dataset to binary images and resized them to resolution of 45x35x1. Hence, this became the input resolution of our CNN as depicted in Figure 8.

The above modified dataset was divided into two parts namely training and test set. The CNN was trained on the training set and the test set was used for evaluating the accuracy. The training is performed by providing the character images of training set as input on one end and respective class labels as output on the other end. The test set is never used in training and therefore represents the unseen data used for evaluating accuracy.

The CNN was trained for 100 epochs and the best accuracy obtained on this dataset was around 92.5%. We intended to improve this accuracy to achieve better performance. We realized that the total size of the dataset was less for training. So, we decided to augment more images. We performed this data augmentation using width-shift, height-shift, shearing and zooming. We did not use horizontal or vertical flipping for augmenting as not all characters were symmetric. The best accuracy we achieved was 93% after experimenting with many parametric values used in our data augmentation. Even though the improvement in the accuracy is less, it makes the CNN to generalize more efficiently therefore increasing its robustness.

2.4.2 Implementation of Sliding Window

As seen in the above sections, the output of the segmentation process is a list of word images segmented from each line in the given scroll image. Each of these word images consist of multiple characters on them which need to be further segmented and given to the classifier. To implement this, we came up with the idea of sliding window. Here, we take a window of particular shape and size and start sliding over the word image. Each time we slide, we take the image inside this window and pass it to the classifier for character recognition. It should be noted that this image is resized to the resolution of 45x35x1 before being passed to classifier. The recognized character is then written on the output text file after transcription.

After implementing this concept, there were many problems which we faced with respect to performance. Firstly, we saw few blank images being cropped by the window. After some inspection, we found that even after 'segmentation of words' procedure was performed, each word image did have some white spaces on both the ends. This made the window to crop the white spaces and send it to classifier. To resolve this issue, we took the average of values of all the pixels in the cropped image. If this value was greater than a threshold, then we omitted the image considering it to be a blank white image. The threshold was decided on a trial and error basis. Secondly, we also found difficulties in deciding the optimal value of step-size for sliding. If it was very low, it would lead to repetitions of cropped images and if it was higher, some characters were omitted from being cropped.

2.4.3 Transcription

The output of the classifier is the string label of character which is predicted. So to write the actual character on the output text file, we had to first convert these labels into their actual characters. This was done by implementing a transcription code which writes the respective characters in the file when given character labels.

3 Results

The result of the project is a complete handwriting recognition system, designed especially for the Dead Sea Scrolls. We can run it easily by a couple of instructions [8]. Note that the images to be recognized should be saved in the input folder under the directory of the system, then after running the instructions, the recognized results would be saved in the output folder as .txt file. We also have three running modes, `normal`, `debug`, and `fast`. In `fast mode`, the system would skip over the intermediary results. The average processing time of one image is about 7 to 11 seconds, depending on the runmode and the machine running the program. Following are the results of each step.

Preprocessing The result example of the preprocessing is contained in the Methods section, as shown in Fig 4. Compared with the original image in Fig 1, the information of our interest got mostly retained. We noticed that it was also slightly rotated to correct the tilted text. Still, a little information was lost during this step, especially around the margins of the scroll image. Also, the noise of the Sauvola binarized image in the background was nicely removed, but few stains on the scroll piece were not specifically dealt.

Segmentation After preprocessing, we used the locations of the peaks in the corresponding histogram to segment the image. It was segmented into several pieces and each piece containing one line. For example, we got the fragment containing the 4^{th} line as shown in Fig 9. Most of the lines were segmented nicely, but it highly depends on the previous step. This is because, if there are stains in the image after pre-processing they can be wrongly segmented as a line sometimes. We used the same approach to segment the words and one result example is shown in Fig 10. Many words were segmented correctly, though it was not as efficient as line segmentation. The issues included a few over-segmentation and under-segmentation cases.

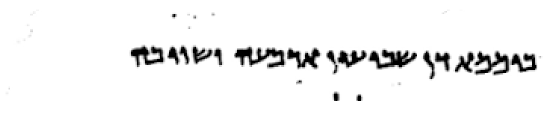


Figure 9: The segmented image piece containing the 4th line

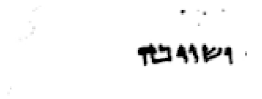


Figure 10: The segmented image piece containing the 1st word of the 4th line of the example image

Classification The results of training the CNN with and without data augmentation are given in the Table 1

[8] The details are posted on https://github.com/atianhlu/HWR2019_Group7/blob/master/README.md

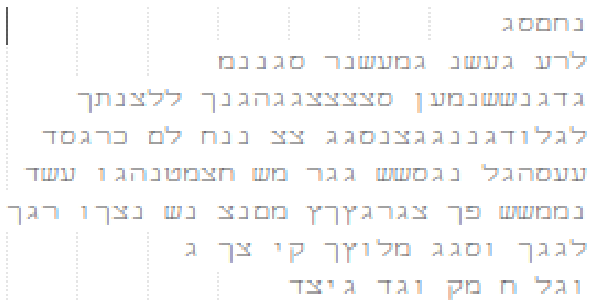
	Accuracy	Epochs
Without Data Augmentation	92.5%	100
With Data Augmentation	93%	100

Table 1: Accuracy of our CNN

After implementing sliding window with proper parameter values of step-size and threshold, we observed that there were still some repeated characters in the output text. One advantage of this setting was that we did not miss out any character in the word images. If we tried to increase the window size further, it would start to omit some characters which is even more worse. Also, while threshold constraint was good enough to avoid minor noises in the image, it failed when it came across significant noises. It therefore sent this cropped noisy image to classifier which resulted in wrong character output.

3.1 Full pipeline results

Running the code for the full pipeline, on the input image presented in Figure 1 resulted in the output presented in Figure 11 below.



```
נחמסג
לרע געשנ גמעשנר סגננמ
גדגנששנמען סצצצגגהגנך ללצנתך
לגלודגננגגצנסגג צצ ננח לם כרגסד
עעסהגל נגסשש גגר מש חצמטנהגו עשד
נממשש פך צגרגץרץ ממנצ נש נצךו רגך
לגגך וסגג מלוזך קי צך ג
וגל ח מק וגד גיצד
```

Figure 11: The output of the pipeline

4 Discussion

In this paper, we presented our approach for automatic handwriting recognition which had three processes mainly - pre-processing, segmentation and classification. Pre-processing involved extraction of parchment using various techniques like Otsu and Sauvola binarization followed by connected-components approach and geometric correction. Segmentation was performed by taking the histogram projection profiles of the extracted text. Finally, classification of the segmented word images were performed using a trained convolutional neural network(CNN).

Our proposed pre-processing step was robust, however, when it comes to segmentation step, we faced few issues after the whole system pipeline was ready and working. Firstly, we observed our system was incompatible with scrolls where the alignment of text sentences were curved. In such situations, the line segmentation step of our system was unable to crop out individual lines efficiently. This indeed affected the further steps performed in our system. As a result, the output text file generated had very low accuracy.

Secondly, the word segmentation step in our system was not highly efficient. On some occasions, we observed that system divided characters of same word into different word-images. On fewer occasions, the system cropped a character itself into two halves putting them into different word-images.

Lastly, we also found difficulties in classification. The main challenge was deciding the step-size of the sliding window. Even though we used trial and error strategy on deciding its value, we still got some repetition of characters in the output text file on few occasion. We found that this issue arised when a character was too small and thus our constant-sized window slided on it more than once. The other issue was when omitting the blank images cropped by our window which we explained in above section. While this worked very well in omitting them indeed, but we were unable to remove the noise even after deciding a perfect threshold value. If the input had some significant noise, the average value of pixels surpassed our condition for the threshold and hence this cropped image was sent to the classifier.

One alternative approach for line segmentation can be using A* algorithm (Olarik Surinta, 2014) which is our future work. Other areas of future work can be the use of lexical meanings of the words or characters. To implement this, we can make use 'Long Short Term Memory(LSTM)' or 'Recurrent Neural Networks(RNN)' combined with N-grams for better results.

To conclude, considering the complexity of the problems with 'dead sea scrolls', we feel that the output generated by our system did produce robust results in the end. We were able to see very clear results for pre-processing but not so fortunate in case of segmentation and classification. Clearly, there still exist scope to improve our strategies using other alternatives in the future.

References

LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998, November). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, *86*(11), 2278-2324.

Olarik Surinta, F. K. J.-P. v. O. L. S. M. W., Michiel Holtkamp. (2014, 5). A* path planning for line segmentation of handwritten documents. , 6. doi: 10.1109/ICFHR.2014.37

Otsu, N. (1979). A Threshold Selection Method from Gray-level Histograms. *IEEE Transactions on Systems, Man and Cybernetics*, *9*(1), 62–66. Retrieved from `http://dx.doi.org/10.1109/TSMC.1979.4310076` doi: 10.1109/TSMC.1979.4310076

Sauvola, J., Seppänen, T., Haapakoski, S., & Pietikäinen, M. (1997, 09). Adaptive document binarization. In (Vol. 33, p. 147-152 vol.1). doi: 10.1109/ICDAR.1997.619831

Suzuki, K., Horiba, I., & Sugie, N. (2003, 01). Linear-time connected-component labeling based on sequential local operations. *Computer Vision and Image Understanding*, *89*, 1-23. doi: 10.1016/S1077-3142(02)00030-9