Adaptive binarization of 17th century printed text

Carl Carenvall
Abstract

Adaptive binarization of 17th century printed text

Carl Carenvall

This work focused on implementing and evaluating an adaptive water flow model for binarization of historical documents, as presented by Valizadeh and Ehsanollah in an article published in early 2012.

While the original method sought an optimal result for all kinds of degraded documents, both on hand written and printed, the work presented here only needed to be concerned with printed documents. This was due to being focused on specific documents scanned by the Uppsala university library.

The method itself consists of several steps, including a couple that uses other methods for binarization to achieve a good result.

While the implementation appears to have been largely successful in replicating the results of the original method, it is very possible that some minor tweaking could result in further improvements. To replicate the results, however, a new parameter had to be inserted in the method. Regardless if this was because of some mistake, or if the sample data used by Valizadeh and Ehsanollah simply differs from the one used here, this may be worth looking more at. In the end of this report are comparisons with a couple of common and state-of-the-art methods for binarization, and this method appear to perform favourably in most cases.
1 Introduction

The university library is scanning scientific reports from between early 1600 and mid 1800, and want to be able to digitize these documents to make them easily searchable and possibly publicly available. Some of the documents are, however, at least partially degraded, making OCR (optical Character Recognition) with high quality difficult. They have done some tests with OCR on the material in question, but were not entirely happy with the results. There is reason to believe some parts of this process can be done better due to recently presented improvements in some algorithms in the field.

The entire process of OCR was not the aim of this project, but rather making the text easier to handle for regular OCR software. The algorithm implemented and evaluated is presented in its entirety in [1]. Some possibilities of improvements have also been examined.

1.1 Previous work

The process of turning an image into digital text (the OCR "pipeline"):

1. Compensating for geometrical errors; An image of a text is not necessarily in straight and proper lines with equally large letters. For instance, the page may have been damaged by water, resulting in an uneven surface. Or it may be written by hand, and the lines may be less than perfect as a result. The first step is about correcting such errors, making some damaged documents easier to process by the following steps.

2. Binarization; The process of making a scanned or photographed image into a binary one, which is more easily handled by the final step where characters are identified. This process concerns itself with segmenting the image into background and foreground. Obviously some issues in this area include avoiding loss of data by identifying text as noise as well as avoiding identifying noise as text. In an optimal result, all black pixels are actual text pixels and all white ones are actual background.

3. Template matching; The last step of the OCR "pipeline" is essentially the process of automatically transcribing an image of a text into actual text data. The higher the quality of the binarized image, the more accurately this step can identify individual characters and digitize them.

This project focused entirely on the binarization step, implementing, analyzing and hopefully improving an existing algorithm. The idea was that commercial OCR software should get better results with more accurately binarized images.

There are many algorithms for binarization, with varying results depending on amount of noise, type of text (printed or handwritten) etc. The algorithm
implemented uses a couple of these in different steps, described in more detail in section 2.

More information about the area in general, as well as some of the concepts mentioned here can be found in [6].

1.2 Other binarization methods

While there are many other methods for binarization, three common and fairly well known ones are presented here. In section 3, these three are compared with the implemented method. One of them is also used as a step in this method.

1.2.1 Niblack binarization method

Niblack’s method was originally presented in [3], and uses adaptive thresholding to separate text from background. It is what is called a local method, since it processes portions of the image with locally calculated thresholds, rather than one threshold for the entire image. It is still a common method, at least to compare new methods with. In the implemented method, this algorithm is included in one of the steps where an estimated binarization is important for calculating some key parameters. Since it uses a window to process smaller parts of the image, local differences impacts the result less than they would with global thresholding methods.

The implementation of the Niblack method used, both in the implemented method as well as for comparison, was found online on mathworks file exchange section, [4].

1.2.2 Otsu thresholding binarization

Otsu’s method is a global thresholding method. There is a windowed version as well, but for the sake of simplicity the global version is used for comparison.

While more advanced versions exists, the basic method itself assumes that an image consists of two classes of pixels (text and background in a black and white text image for instance). It then attempts to find a threshold, which separates these in such a way that the difference in gray level within each class is minimal.

More information about Otsu’s method can be found in [2].

1.2.3 Sauvola binarization method

Sauvola’s method is a version of Niblack’s. And just as Niblack’s, this is also a local binarization method. If the contrast in a local region is high, which is the most common case in text images, the Sauvola method chooses its threshold based on the local mean value. If the local contrast is low, the threshold
should be selected lower than the mean, proportional to the normalized standard deviation for that region.

In general, this method is credited to perform very well. I did not have time to properly calibrate it, but it still performs fairly well. The particular implementation I used was found by a user on an online forum, [8]. That particular version, however, was not a local implementation. Though altered to work as a local method, it did not improve results even after some experimentation with parameters. So the comparison is done with a global version.

2 Implementation

The implementation section briefly describe the algorithm implemented, to later go into details about the important parts, possible areas of improvement etc.

2.1 An adaptive water flow model

The implemented algorithm is the one presented in [1]. Though some minor modifications have been made, the suggested algorithm has generally been followed as thoroughly as possible. The modifications (presented in the end of this section) was mainly focused on calibrating parameters for improved quality, but have generally not taken in to account a possibly reduced quality on handwritten text. The most interesting of these was the inclusion of a multiplier for one piece of data in the process. This was not mentioned anywhere in [1].

There are several reasons why adaptive methods for binarization are likely a very good option, especially when it comes to degraded or low quality images of documents. Optimal threshold values don’t just vary between images, but sometimes even vary from region to region within a single image.

The adaptive water flow model presented in [1] uses several other algorithms to achieve their result. Their work is at least partially based on the water flow model introduced in [5], and focuses on optimization (mainly not having to simulate rain on the entire surface) and quality (automatic criterion for stopping the rainfall).

2.2 Steps and associated data

The algorithm takes in an image either in color (such as the one in figure 1) or in black and white, and if it is in color it will be converted to black and white. A minor change in the algorithm was made at the point when the image has just been loaded. The image is normalized, giving the values the range \( [0 : 1] \in \mathbb{R} \) instead of \( [0 : 255] \in \mathbb{Z} \). The reason for this change was some issues in the last steps of the algorithm where different sets of data would have different types, forcing some awkward changes in data type. Important: all the values suggested in [1] were then altered to match the normalization. Also, please note that the
images in this section serves as illustrations of the steps. Calibration to improve the quality of the final binarized image may slightly alter these results.

**Figure 1:** The original image used in this example. It is turned in to gray scale, normalized and inverted before being run through the rest of the algorithm.

**Figure 2:** Original image in inverted gray scale. The actual data handled is not inverted, but it makes it easier to visually compare it with later steps, since higher colour values are brighter and lower ones are darker.

A few of sets of data, apart from the gray scale image, are required to perform the adaptive water flow step. First of all, a new image with the same dimensions is generated, but this one containing the result of a canny edge detection on the gray scale image. This image will have edge pixels marked as 1, while all other pixels are marked as 0, and typically looks something like the image in figure 3. This image will hence be called the canny image, or edge image. This image will be used to localize the rainfall in the water flow step.

**Figure 3:** Result of canny edge detection. It is worth noting that this image feature relatively little noise. In the rain step of the algorithm, water is "poured" upon these white pixels.

The second piece of data is generated in two steps. First of all, an approximated binarization is generated using Niblack’s method. This image is used
in the second step, which is an approximation of stroke width. Stroke width is used as a general term for the minimum width of a text area (without intermittent background pixels), regardless of the type of text. For the water flow method, the first parameter of Niblack appear to generally give good results at around $-0.5$. This value should be set with caution, however, and an optimal one varies depending on image. The second parameter was found to be optimal at 40, though will probably vary if the resolution of the image differs a lot. The general optimal values for running only Niblack by itself were experimentally determined to be close to $-0.2$ and 40.

![Figure 4: Result of the Niblack algorithm with parameters $-0.5$ and 40. These values appear to give an optimal result for only Niblack, but may in some cases not be optimal for the full water flow method. Note the noise on the left of the image, as well as the relatively clean letters. This image is only used to estimate the width of the text areas, an important parameter in the rain step. The quality of this step has a noticeable impact on the result.](image)

In this step, the entire image is scanned in four directions. When text pixels are found (in the Niblack image, such as the one shown in figure 4), the number of consecutive text pixels following it are counted, and once the other edge is found it backtracks and tags each pixel with this value if it is smaller than the one already there (initially set to a theoretical maximum). The central idea of the stroke width is for it to help estimate the "volume" of the text areas as water is poured on them. Stroke width (hence forth called SW) is a central part of the algorithm, as seen in figure 8, and the quality of this step has a real impact on the quality of the result. The SW image is then smoothed out, doing an average on $25 \times 25$ squares (dimensions suggested in original article). I will refer to the smoothed SW image simply as the SW image, because the original one is no longer needed.

The last piece of data needed is the contrast image. This image checks the contrast on the gray scale image corresponding to the edge pixels on the edge image. This data indicates how long it will rain on each particular edge pixel (number of times water is poured on it). The comparison is not simply made between neighbouring pixels, however. The SW images role in this step is to tell how far apart the pixels being compared are.

The rain step itself uses the SW image, the gray scale image and the contrast image. The contrast image does two things; tell the algorithm where to pour water and how many times it needs to do so. The SW image is used to tell the
Figure 5: Stroke width measured in four directions. This image has been smoothed out with a window of $25 \times 25$ pixels. Compare with figure 4. The brighter the area, the higher the number that will be added on that pixel when water falls there. So brighter areas should fill up significantly faster than darker ones, and black areas should not get any water at all, even if the local minima search of the rain step by chance would end up there (which it is very unlikely to do).

Algorithm how much water is poured on that pixel each time. When water is poured, a trivial search for the local minima is done to find where to deliver the payload of water. This search is done by looking at the values of nearby pixels with in a mask. In [3] a square mask of size $n \times n$ is used, where an optimal $n$ was found through experimentation and, given their set of test data, is suggested to be 3. In [1], however, the mask is reduced to an $n \times n$ cross. An optimal value for $n$ is not suggested there. Through experimentation, I found that an optimal value seems to be around 3, but values around 50 and even higher also seem to work well. The amount of water dumped at the pixel is the value from the SW image for that particular pixel.

Figure 6: This image shows where water have gathered after completing the water flow (or "rain") step. Each connected area is labeled before classification. This means that incorrectly connected letters will be considered a single blob, and evaluated as such. Comparing it with the original in figure 2, we can see that most of the "noise text" (areas where text from another page has rubbed off on this one) has already been eliminated. The last step of the algorithm is to remove remaining noise areas by classification.

The image produced by the rain consists of puddles where the water has gathered. This image will hence be called the ponds image. The idea is that the water will have gathered mainly on text pixels, but some noise is practically unavoidable. Each pond (pixels with water on them connected to each other) is tagged as a unique blob. The tagged image (which I call the blob image) is used for the final part of the process: classification. Each blob is run through
a classifier, which (based on a couple of parameters calculated for each blob) decides if it is text or not. In the end, all that remains is to remove all pixels that are not recognized as text and thus produce the binarized image.

The parameters available to the classifier (as suggested in [1]) are "average water amount" and "average gray level." Figure 20 is a plot of the blobs from the original article based on these two parameters, and figure 21 is an example of my result for comparison. The classifier I ended up using was MATLAB's NaiveBayes classifier. Trained using the results from two images from [7], it appears to perform satisfactory.

Figure 8 shows the relations between the various sets of data, and serves as an abstract illustration of the flow of the algorithm.

2.2.1 Incomplete instructions, assumptions and experimentation

Some constants in the algorithm were not given in the article, so they were experimentally tested for good values to use until possibly better ones could be tracked down.

One of these was the number of cells to look at each step when looking for the local minima during the rain. It was initially set to 2, but when later increased, the result in most areas of the text was noticeably better, and when increased further a slightly worse result in other areas. The optimal range seems to be between either around 3 or around 50, but further testing is required to verify this. My best guess is that varying values are optimal for different regions in the image. In the end, 3 was used for this parameter.

At first, stroke width measurement was implemented only in two directions, which would generate a slightly higher average value than a correct measurement. When four direction measurement was implemented, however, a significant loss of quality occurred. To counter this, tests were made both with reducing the window size of smoothing the stroke width, and with multiplying

Figure 7: The result after classification of the blobs, and removal of all blobs considered to be non-text. The classifier used here is MATLAB's NaiveBayes classifier, trained with two images from [7], but not this particular one. There seems to be very little loss of text, as well as very little noise. The resulting image looks really good, but a more thorough evaluation and comparisons with other methods can be found in section 3.
the smoothed image with a constant. It seems that the best results occur with a multiplier of between 2 and 3, as seen in figure 9. Note that this is based on a window size of 25, a parameter that have a great impact on this particular
result. An optimal value could also vary between images. Though the graph suggests a value around 3.0, the graph was made by testing only one image. Overall, 2.6 appears to give the best result.

![Figure 9](image.png)

**Figure 9:** Testing multipliers between 0.4 and 4.0 for a single page. The results here consists of an evaluation of the binarized image produced by the entire algorithm, compared to the ground truth image of this particular page. For this test, the window size was set to 25. In this case, the optimal value appears to be around 3.0, but it varies from image to image and appears to be between 2.5 and 3.0 or so. The actual comparisons in section 3.2 is run with 2.6. This multiplier is applied to the entire SW image after smoothing. This parameter is not included in [1], but as the results appear to be at par with theirs, this suggests some difference either in implementation or in the data set. If it is the data set, a difference in resolution could explain this result.

One value that was given, but where it proved relevant to experiment with was the window size when smoothing the SW image. Experimentation revealed that the best value was probably around $25 \times 25$ rather than the suggested $30 \times 30$. This variation may be because of different sample data. This is, however, affected by the multiplier applied to the smoothed image. Balancing these parameters properly require further testing, as suggested by figure 10 and figure 11.
Figure 10: As the window size increases, the overall values in the resulting smoothed image decreases. Without a multiplier, optimal results appear to be as close to 0 as possible. This result does not appear to match the one presented in [1]. If the difference is in the implementation or the test data is uncertain. But granted the results in figure 11, it appears a reasonable assumption that the difference is in the test data.

2.3 Tools and work flow

The choices of tools and what to focus on during the work were mainly based on the primary goals of this specific project, and how best to meet them on time.

The entire project have been done in MATLAB and C (mex, run through MATLAB). The reason for this choice was that several parts of the development would be sped up significantly. Most notably, edge detection as well as many matrix operations did not have to be implemented. Focus could be put on the algorithm at hand and its more interesting points, rather than having to build every small part of it from the ground up. Though using this tool does not result in a ready-to-use application, it has allowed for swift testing and evaluation of the algorithm on the specified problem. Making a full application as well as testing the algorithm would go beyond the scope of the project, both in time and complexity.

Step one was to thoroughly read the article by Valizadeh and Kabir [1] to get an understanding of what would be required. The article is well written,
Given a multiplier after the smoothing, the optimal value has shifted to around 25 or so. This is close to the value suggested in [1] (which was 30). and each step in the algorithm is fairly straightforward. Though MATLAB was mainly used for the implementation, it was fairly clear right from the start that at least one or two parts of the algorithm would probably have to be in C (using mex through MATLAB), for performance reasons.

Rather than following a waterfall model while implementing, the aim was to do a basic version of the algorithm straight away while researching each part step by step as it proved necessary.

For each step of the algorithm, the result was made sure to seem at least serviceable. The biggest hurdles in implementation was with mex. At times, the C-code would crash, resulting in MATLAB abruptly shutting down and refusing to start again until the computer was rebooted. Without being able to use any debugging tools other than printing text, fixing some issues took a lot longer than they could have.

The performance boost that came from coding the "rain" step in C instead of MATLAB (which was first tested) was well worth the trouble. A fairly small image that took around 30 seconds with the MATLAB implementation ran in about 5 seconds with the C implementation, on the same computer.
3 Results

Though some more advanced texts yield less than perfect results, the overall quality appears to be really good. There is a possibility that some parts of the algorithm can be fine tuned further for even better results, but for the Alstrin data set that might not be necessary. In its current state it appears to produce stable results of very high quality that compare favourably to all methods included in this comparison, both in terms of text quality and eliminating noise.

3.1 Examples

The images in the following section are taken from two different data sets, and are here primarily used to illustrate what the data for the more important steps looks like. A good or bad result on the example images does not by necessity mean that all images from either set would yield a result of the same quality.

3.1.1 DIBCO 2009 data set

Dibco stands for "Document Image Binarization Contest" and the dibco 2009 set [7] are images used to test binarization methods, and thus a good basis for testing and comparing. Note that these are from the same test run as the images in section 2.

![Figure 12](image.png)

Figure 12: In this image, each blob was compared with the "ground truth" image. All pixels in each square was matched with the same pixels in the ground truth image. If at least 80% of the pixels match, the square is marked green, otherwise its marked red.

3.1.2 Alstrin dataset

Thanks to the Uppsala University library, a sample from the Alstrin data set was available for testing. Since no ground truth was available, and producing it would take more time than could be afforded within the limits of this project, a visual qualitative analysis will have to suffice. The section of text chosen to test here appears to be typical in the kind of text the set consists of, as well as including some broken and/or partially weak letters, which appears to be the most common problem. From the data samples examine, this seems to be fairly typical, though the full data set quite possibly includes more degraded material.
Figure 13: This is a plot of all blobs, each marked with the same color as in figure 12. The plot is based on the "average water level" (AWA) and "average gray level" (AGL) for each blob. The line here represents the simple filter; everything to the left is considered background and erased from the final image, while all blobs to the right of it is considered text and thus included in the final image. In the final version, MATLAB’s NaiveBayes classifier was used for this step.

Figure 14: This is the same image as in figure 12, but after applying the linear filter in figure 13. Note that even though several blobs have been marked as not matching with the ground truth, they are still correctly considered as text. As shown, very little noise has made it to the final result.

While the results of this implementation are no doubt of high quality, Otsu’s method comes in a close second for this data. It is worth noting here that Otsu’s method was observed to take about 0.01 seconds while this implementation took about 7.0 seconds. Also, this implementation seems to scale poorly if there are
Figure 15: This is a small section of a page from the Alstrin data set. The sample chosen appears to be typical in terms of type of text, as well as including some of the more severe problems that appear in the data available for testing. The problems here consists mainly of broken and/or partially weak letters.

Figure 16: This figure is the result of a run with the implemented method. Though some minor breaks are repaired, the overall result pretty much reflects the original. Compared to the other tested methods (figures 17, 18 and 19), this result has almost no noise pixels at all, as well as relatively clean letters.

very large letters in the text, since these require many more iterations in the rain step to fill up. If speed is an issue, Otsu is probably the better candidate for data of as high quality as this. Even though the computer the test was run on is not particularly powerful, the scanning machine can produce up to thousands of pages per hour, so speed could be a factor.

Niblack is notable for its high level of noise, some of it even relatively close to the text. Sauvola does a much better job of cleaning up noise than Niblack, and even manages to repair some broken letters. It appears to perform somewhat worse than both the adaptive water flow model and Otsu's method.

3.2 Comparative analysis

Comparing this implementation with [1] in detail is not particularly easy, as there are fairly little numbers to compare with. It is possible, however, to produced a similar plot of the regions average water level and average gray level. Also, a purely visual comparison of the results can be made.

To test the quality, it can be compared with a few other established methods
Figure 17: This is the result of Otsu’s method. It appears to be the closest candidate next to the adaptive water flow algorithm. The letters appear to be almost exactly as good as in this implementation, but with some extra noise. Due to performing extremely fast compared to the implemented water flow method, Otsu’s method is probably the better choice if the quality of the material is this good and speed is an issue.

Figure 18: Here is the result of the Niblack method. Clearly not performing significantly worse than both Otsu’s method as well as the implemented one. While most of the letters are clean, distinct regions, there is a lot of noise and some of it is fairly close to the actual text.

Figure 19: This image is the result of the Sauvola method. While performing significantly better than Niblack, and in some places repairing a few broken letters, the overall result appear to be somewhat lower than both Otsu and the water flow implementation.

for binarization. Since implementations of these were available, it is possible to generate results with these methods, and a detailed comparison is much easier.
3.2.1 Original work

One way to compare results with the original in [1] is to look at the parameters calculated for the blobs and used by the classifier. A similar pattern in a scatter plot, within similar limits, likely means that the implementation is largely correct.

\[ \text{Figure 20: This is the plot included in the original article, where each dot is a blob and its location is based on "average water level" (AWA) and "average gray level" (AGL). Even though most blobs are easily classified as text or non-text by drawing a line through the middle of the plot, a few red text blobs would incorrectly be labeled as non-text as well as a couple of non-text blobs being labeled as text. A more advanced classifier (than just a line) was stated to be used in [1].} \]

Even though the implementations are tested on different sets of data (it is not revealed exactly which data was used to generate the original plot), the similarity is pretty clear. This particular plot was generated from a fairly large image, to get a clear pattern. Unfortunately, in this case it meant that a ground truth image was not available to compare with, and thus regions can't be reliably tagged as either text or background as in the original plot. Such a plot can, however, be found for a smaller image in figure 13.

All results presented here are with MATLAB's NaiveBayes classifier, trained with sample 1 and 2.

3.2.2 Other methods

Three relatively simple and common methods for binarization were chosen to evaluate the implementation. More complex methods are excluded from this comparison mainly due to lack of time, but could be interesting in the future. The chosen methods were Niblack, Sauvola and Otsu, all of which are briefly described in section 1.2. In the following graphs, results were compiled to how much they matched the ground truth in a few distinct aspects. Thus, the line
Figure 21: This is a plot from the result of a test run on a fairly large image. Since no ground truth exists for it, it can be assumed that at least a couple of text blobs are somewhere in the upper left region with the non-text blobs.

at 100, indicating the optimal result at 100%. All methods were tested against five printed text images from [7]. Note that the AWFB (Adaptive Water Flow Binarization) entry is the implementation of the water flow model discussed throughout this report.

Please note that some of the methods may not be perfectly calibrated. Changes in parameters may alter their results in various ways, particularly Sauvola’s method, which has its only ‘magic value’ set to -0.8.

The graph in figure 22 is most likely a very good measurement of text quality. It goes through all pixels inside the bounding boxes of all regions found in the ground truth images. This excludes all noise from the evaluation, but also potential bleed in the results. To properly compare the methods, the graphs in figures 23 and 24 should be taken into account as well.

The graph in figure 23 shows the results for pixel match in the entire image. As it clearly shows, the AWFB implementation is very close to 100%, only contested in a couple of samples by the Otsu method. While a high result in this regard signals that a method is good at both producing good text as well as clean up noise, a very poor result can be very ambiguous; an image that is either entirely black or entirely white may score well above 50% since a majority of the image is likely to be background. This graph becomes more meaningful when compared particularly with figure 22.
Figure 22: This should be a very accurate measurement of text quality. This graph counts the matching pixels within the bounding boxes of each region found in the ground truth image, thereby limiting the comparison to the actual text and its immediate surroundings. The one caveat of this data is connected letters that are very close to each other, or possible "bleed" from regions in the result, which would be excluded from the comparison. However, noise pixels that are not close to letters are also ignored. Note that Otsu and AWFB both perform well here, but with a noticeably better result for AWFB on all tested images.

A good result when comparing the number of defined regions in the resulting images, as seen in figure 24, is only meaningful when a fairly good result can also be found both in figure 23 as well as figure 22. In such a case, it is reasonable to assume that the found regions are pretty close to the actual letters of the ground truth. A bad result can mean a lot of things, and is not necessarily a measurement of bad text.

Though some methods appear to excel in one area, most of them do poorly when it comes to number of defined regions. Typically this is due to a lot of noise regions. The three results combined give a fairly good measurement of how good each method is. With no exceptions in the sample data included here, the implemented water flow method appears to outperform all other tested methods in every aspect. Some more exact numbers can be found in table 1, 2 and 3.

The tables 1, 2 and 3 give some more detailed information about each methods performance. These tables also confirm that the closest rival among these methods is Otsu. Note also that at its worst, the implemented method (AWFB) is only marginally worse than the Otsus best. There exist couple of possible reasons for the poor performance of the other methods. First of all, it is possible
Figure 23: This graph compares the total amount of matching pixels in the entire image. Here, noise in large background areas does reflect negatively on the result, which can be seen mainly for Niblack. A low difference between this number and the one in figure 22, should indicate relatively low amounts of noise and "bleed" as well as relatively well formed letters. Note, however, that if an image has large background areas, methods that generate little noise in such areas easily appear to perform very well in this graph.

<table>
<thead>
<tr>
<th>Method</th>
<th>Region pixel match %</th>
<th>Number of regions %</th>
<th>Total pixel match %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Niblack</td>
<td>92.4468</td>
<td>2549</td>
<td>91.5607</td>
</tr>
<tr>
<td>Sauvola</td>
<td>90.3770</td>
<td>566.6809</td>
<td>95.0471</td>
</tr>
<tr>
<td>Otsu</td>
<td>97.5691</td>
<td>190.6532</td>
<td>98.5372</td>
</tr>
<tr>
<td>AWFB</td>
<td>99.4316</td>
<td>104.3963</td>
<td>99.7970</td>
</tr>
</tbody>
</table>

Table 1: These are the mean values for each method. Note that AWFB seem to perform really well in all areas, outperforming all other methods. Note also that Otsu is fairly close, at least compared to the other methods.

that the implementations tested here are not optimal. Second, there is a chance that some further calibration of parameters could improve the results. It seems unlikely, however, that the improvement would be so significant that any of them would outperform the AWFB implementation.

A visual inspection of some of the images reveal the actual difference.

The sample image found in figure 25, which is number four in the diagrams in figures 23, 22 and 24, has a region with very low contrast within itself,
Figure 24: This graph compares the number of found regions in the result of each method with the number of regions found in the ground truth. If this value is close to 100 coupled with good results in both figure 22 as well as figure 23, this should indicate that most letters are well defined separate regions. A very high value here indicates high levels of noise and/or broken letters.

<table>
<thead>
<tr>
<th>Method</th>
<th>Region pixel match %</th>
<th>Number of regions %</th>
<th>Total pixel match %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Niblack</td>
<td>99.2889</td>
<td>432.2222</td>
<td>94.4222</td>
</tr>
<tr>
<td>Sauvola</td>
<td>96.2442</td>
<td>144.9541</td>
<td>98.1582</td>
</tr>
<tr>
<td>Otsu</td>
<td>99.6307</td>
<td>111.9266</td>
<td>99.7750</td>
</tr>
<tr>
<td>AWFB</td>
<td>99.8590</td>
<td>99.4444</td>
<td>99.9324</td>
</tr>
</tbody>
</table>

Table 2: These are the best values for each method. The performance of each method varies with each image. These values represent how well they can do given optimal conditions within this test set.

while having quite a large contrast to its surroundings. This poses a fairly great challenge, particularly to global thresholding methods. For a full comparison, look at figures 26, 27, 28 and 29.

This does not tell the whole truth however. While figure 26 was generated using parameter value of $-0.5$ for Niblack’s method before calculating stroke width, so was figure 31. While the large blob is largely broken up in figure 26, there are a significant amount of broken letters in figure 31. Compare those with the images in figures 32 and 33 respectively, which both use 0.0 instead of
Table 3: These are the worst values for each method. It is worth noting that while the other methods perform relatively poorly at least at some point, the AWFB implementation seems to stay very close to the mean value even at its worst.

<table>
<thead>
<tr>
<th>Method</th>
<th>Region pixel match %</th>
<th>Number of regions %</th>
<th>Total pixel match %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Niblack</td>
<td>88.8625</td>
<td>8920.8</td>
<td>87.6920</td>
</tr>
<tr>
<td>Sauvola</td>
<td>83.2928</td>
<td>2098.1</td>
<td>92.7058</td>
</tr>
<tr>
<td>Otsu</td>
<td>95.1808</td>
<td>373.5849</td>
<td>96.3014</td>
</tr>
<tr>
<td>AWFB</td>
<td>98.8537</td>
<td>125.4717</td>
<td>99.6240</td>
</tr>
</tbody>
</table>

Figure 25: This is an example from [7] with some severe degradation. This particular sample is number 4 in the figures 23, 22 and 24. To see the real difference between the methods, look at figures 26, 27, 28 and 29, which show the results of the four methods respectively, and compare them with figure 30, which is the ground truth for this sample.

Figure 26: This figure shows the result of the implemented method when applied to the sample in figure 25. Note especially the region in figure 25 that had very low contrast and compare it with this one. Look also at figures 27, 28 and 29 to compare it with the other tested methods.

Figure 27: This figure shows the result of Otsu’s method when applied to the sample in figure 25.

−0.5 as the first parameter for Niblack’s method. It appears as though −0.5 is a good choice for images with low contrast areas, while 0.0 is a better choice for images with thin and/or weak letters. From this it seems fairly safe to conclude
Figure 28: This figure shows the result of Niblack's method when applied to the sample in figure 25. Note that it fares relatively well in the low contrast region. This is particularly interesting in terms of the stroke width measurement done in the AWFB implementation.

Figure 29: This figure shows the result of Sauvola's method when applied to the sample in figure 25.

Figure 30: This is the uninverted ground truth for figure 25.

that other images may have other optimal values as well.

Figure 31: Generated with the same parameters as figure 26, this image clearly features a lot of broken letters. Compare it with figure 32. The only difference in the parameters is 0.0 instead of −0.5 as the first parameter for Niblack's method.

Note that the image in figure 32, while it looks better than the one in figure 31, scores slightly lower in the evaluations. Probably due to "extra" pixels around the letters due to the "swelling." Regardless, both are relatively clean.
Figure 32: Generated with 0.0 instead of −0.5 as the first parameter for Niblack’s method. Note the reduced amount of broken letters compared to figure 31. Unfortunately this image actually gets a slightly lower score than the image in figure 31 when evaluated for matching pixels, while it looks like it would probably fair better with character recognition.

Figure 33: Generated with 0.0 instead of −0.5 as the first parameter for Niblack’s method. Note that the big blob is not nearly as broken up as in figure 26.

4 Conclusion

In short, my conclusion is that this method in its current state could be used for binarization with favourable results to other common methods, both on the Alstrin data set as well as other, more degraded images. It is much more time consuming than the other methods tested here, so if time is a factor I would suggest looking in to Otsu instead, which performs almost as good. Testing the small sample from Alstrin shown in figure 15 took several seconds for this implementation, while Otsu came in way under even one second. Of course, computer power plays in, and for a large scale project it is very likely that much more powerful machines than the available during this project would be used. Since the scanning machine available to the Uppsala University library can produce up to thousands of pages per hour, it is possible that speed is actually a relevant factor.

I do think that with further testing and experimentation, further increase in quality might be possible. But for the problem at hand, it is most likely already a very good option. Particularly, I think looking at dynamically calculating a good value for Niblack’s method when it is used within the water flow method could prove to be a rewarding area of research in terms of quality.

The use of Niblack to approximate a binarization has a couple of notable consequences. First of all, it produces quite a bit of noise, particularly in areas of the image with low contrast. This does, of course, effect the measure of stroke width (and takes extra time, which may be noticeable in larger images).
however, that Niblack seem to do a very good job of cleaning up the area close to the actual letters, giving them a virtually noise free "aura."

Due to the level of noise, however, I wanted to look a bit a possibly replacing Niblack with a method, which might produce less noise. What I first found was that a simple global thresholding method ran the risk of either looseing text, producing poorer quality, or both. Sauvola seemed like a good candidate at first, with similar quality but less noise. However, the results appeared to have scattered white pixels in the black letters, something Niblack's method typically doesn't have. Since this images' only use is an approximation of stroke width, and these values are measured by counting consecutive black pixels throughout the image, these white pixels in the text would undoubtedly offset the SW data. My conclusion was that using Sauvola would probably produce a less accurate measurement of stroke width than Niblack could, with its reasonably clean letters, even with less than perfect results.

Another important aspect of Niblack is that it requires two input parameters (at least in the implementation I used). These need to be calibrated to optimal values to achieve the best possible results. Such values differ from image to image, which can be seen in figures 26, 33, 31 and 32. Though I did do tests with this, I suggest looking into an adaptive way of calculating them, or finding an alternative method for this.

These parameters appears to be generally more or less optimal at \(-0.5\) and \(40\), respectively. The difference in quality of the whole water flow algorithm is noticeable with well calibrated parameters for this step.

It may be worth pointing out that optimization was not really a priority with this project over all. This means that some parts of the implementation were intentionally coded in a way that takes more time or memory than necessary, but were significantly faster and easier to program. This was mainly to buy more time to work on what seemed to be the important and interesting parts of the algorithm. That said, given the overall quality of the algorithm, optimization may be a valid area of discussion and future work.

Though handwritten text was not taken into account while testing the method, I also did not find any apparent way to simplify the algorithm while keeping the results for printed text at a high quality. At first I thought a possible simplification would be to measure stroke width in less than four directions (see section 2.2), but four directions appear to produce a slightly higher quality of the output.

5 Future work

The two main areas for possible future work that appear the most interesting are quality and optimization.

First of all, I would like to suggest further research of the value with which the smoothed stroke width data is multiplied. This value does not appear in [1],
but has a noticeable impact on the result and optimal values seem to vary from image to image.

Though the algorithm yields what seems to be very high quality of output it requires manually setting a couple of parameters. First of all, there are the parameters that Niblack uses. Though they were experimentally tested for optimal values for the data sets used, they may not hold as a general case. Calculating these adaptively could be important for a general case and easy to use algorithm. I would strongly suggest looking into possibly calculating parameter one dynamically. This value may be based on the over all contrast in the region.

Another approach may be to find a suitable replacement for Niblack. There was not time to properly test this, as it appears to require some recalibration of other aspects of the algorithm, but possibly using the Otsu method when estimating a binarization to measure stroke width on could prove beneficial. For images that don’t have areas with low contrast, it may prove to produce even better results than the method currently does. Examine figures 26, 28 and 27 to see why this may be a problem in changing Niblack to Otsu.

Another parameter that needs to be manually set is the size of the area when searching for the local minima during the rain step of the algorithm. This parameter can seriously impact the result. In previous works [3], this parameter is noted to have been experimentally tested to an optimal value (i am left to assume this is also the case in [1]). From what I can gather, the selected value was 3, a value that appears to be more or less optimal in a general case. Of course, with this parameter there is also a risk of an optimal value varying depending on the data set. Indeed, different values may be optimal within the same image. This suggests that a dynamically set value is probably a valid approach. I suggest looking in to basing this parameter on the stroke width, and to give it a larger value in areas with low stroke width and inversely set it to a smaller value in areas with a larger stroke width.

I would also suggest further testing of the balance between the size of the smoothing window and the stroke width multiplier applied to it after the smoothing. More details in section 2.2.1.

The very apparent bottleneck of the algorithm is the rainfall. Further work aimed at speeding the algorithm up is most likely best focused at this part. One approach to do this is to try to minimize the amount of non-text edge pixels in the edge image. The more of the rain to fall on actual text, and the less to fall on non-text, the better the result. However, the canny edge image seems relatively free of redundant edge pixels. Compare figure 3 with the others in section 2.2 to see this.

It should be noted though, that even if removing noise takes longer than just letting the rain fall on it, this could improve the result of the algorithm by making the job easier for the classifier.

Another part that may be viable to look at from an optimization point of
view is the measurement of stroke width. Reducing noise will speed this up, though the change may hardly be worth it unless it is quite a large image.

The results of increasing the area in which to search for local minima with increased quality in most sections, and decreased in others, suggests that this variable probably can (or even should) be locally and dynamically calculated for an optimal result.

6 Acknowledgements

I would like to thank Uppsala University Library, who kindly provided the Alstrin sample data. This data provided a good real world example to evaluate the implementation, which I hope may serve to help them in some way.

References


