Technical Report - Image Binarization

Andreea Tatulescu
341C5, UPB
Bucharest, Romania
diana.tatulescu@stud.acs.upb.ro

Ilie Burdiniuc
341C5, UPB
Bucharest, Romania
ilie.burdiniuc@stud.acs.upb.ro

Amalia Gruia
341C5, UPB
Bucharest, Romania
amalia.gruia@stud.acs.upb.ro

Gigi-Alexandru Dobre
341C5, UPB
Bucharest, Romania
gigi.dobre@stud.acs.upb.ro

Andrei Padureanu
344C5, UPB
Bucharest, Romania
andrei.padureanu01@stud.acs.upb.ro

***Abstract* - Binarization plays an important role in *document analysis and recognition* systems, which in turn are important for automatic extraction of information from papers. The purpose of this work is to create an algorithm of generating the "ideal" threshold based on 15 different algorithms to get the best result of image binarization.**

Keywords - Image Binarization, Tree, Global Binarization, Local Binarization

I. Introduction

Gray scale image and binary image are the two key variations among digital images. In a gray scale image a particular pixel can take an intensity value between 0 to 255 where in a binary image it could take only two values, either 0 or 1. The procedure to convert a gray scale image into a binary image is known as **image binarization**.[1]

The main goal of image binarization is the segmentation of documents into foreground text and background. The simplest approach to binarization is thresholding.

Image binarization is typically treated as a thresholding operation on a grayscale image. It can be classified as global and local thresholding operations. In the global methods (global thresholding), a single calculated threshold value is used to classify image pixels into object or background classes, while for the local methods (adaptive thresholding), information contained in the neighbourhood of a particular pixel guides the threshold value for that pixel.[1]

The most used binarization algorithms are: Otsu, Niblack, Kapur, Bernsen, Sauvola, Th-mean, and the Iterative Partitioning method.[1]

The binarization methods are also categorized into groups depending on the main criteria they consider in computing the threshold [4]. For example, Otsu's method uses a clustering analysis-based method, methods proposed by Johannsen and Kapur are entropy-based, Savuola's and Niblack's methods are based on image variance, and Bernsen's method uses image contrast.

Next, we describe a few of these in more detail.

The most successful global thresholding method is **Otsu's method**. This algorithm automatically performs histogram shape-based image thresholding. It assumes that the image contains background and foreground layers, based on this it calculates the optimum threshold.

The main situations when single global thresholds are not sufficient are changes in illumination, scanning errors, and resolution. One of the local thresholding algorithms is proposed by **Bernsen**. The adaptive method proposed by Bernsen is based on the contrast of an image. The threshold is computed based on the midrange value. First, the algorithm calculates the minimum and maximum gray values in a local window. The size suggested for the windows is w = 31. Next calculate the contrast → C(i, j) = high(i, j) - low(i, j). Using the comparison between contrast and the threshold k, the algorithm categorizes the pixels in the window as the background or foreground. The problem is that it is dependent on the value of k and also on the size of the window.

Another binarization technique is proposed by **J. Sauvola and M. Pietikainen**. Theirs "Adoptive document image binarization" method [5] first performs a rapid classification of the local contents to the background, pictures, and text. Next based on the new information the algorithm chooses one of two approaches: a soft decision method for background and pictures and a specialized text binarization method for textual areas. The first method includes noise filtering, while the second is used to separate text components from the background in bad conditions. Finally, the output of these two algorithms are combined.

II. Solution Description

As we discussed in the introduction, the simplest approach is thresholding, where we change the pixels of an image. The main parameter is the threshold which we use to compare the brightness. After the

comparison the pixel is assigned to 1 or 0, where 1 means that it is the remaining area and 0 is the object boundary. There are many algorithms which use thresholding, such as: Otsu, Kitler, Lloyd, Sung, Ridler.

Our method is based on generation an “ideal” threshold using the thresholds generated by 15 algorithms: Otsu, Kittler, Lloyd, Sung, Ridler, Huang, Ramesh, Li1, Li2, Brink, Kapur, Sahoo, Shanbhag, Yen, Tsa. Our threshold is computed using a tree which has threshold values and operation on it as leaves.

We tried two approaches in generating trees for computing the ideal threshold.

The first approach (fig. 1) is starting from the leaves (thresholds from other algorithms) and executing random operations on these thresholds until the root contains our ideal value. The operations are executed on 2 thresholds: Cubic Mean, Square Mean, Arithmetic Mean, Geometric Mean, Harmonic Mean, Minimum and Maximum. The order of operations, thresholds and files in the dataset are randomized on every tree generation. Every newly generated tree we try to train on 70% of dataset and save them if their score is more than specified percentage (88% percent we consider good both for speed of training and accuracy of the results), if it is good enough we save the 5 best trees for the next validation and choose the best.

The second approach is starting from root and executing contains 3 types of nodes: atomic node with threshold, if node and for node. IfNode is a node that contains a condition and two thresholds. ForNode is a complex node that contains a set of other nodes and an operation. In this method there are 3 types of operations: Arithmetic Mean, Geometric Mean and Harmonic Mean. A generated tree contains a set of nodes. Each node has an abstract method that calculates the threshold for an image. The tree generation starts from a random value representing the depth of the tree between 5 and 10. The type of the root node will be AtomNode with a random value for a threshold type. The implementation continues with generating two random IfNodes with the conditions true and false. To test the conditions, a random AtomNode is generated, a random operation(‘<’, ‘>’, ‘==’), and a random reference value în [0, 1). The next 6 random nodes are generated having current depth - 1, which represents the set of children nodes and a random operation from the set of means. After a tree is generated, it is being tested on a set of validation tests from the initial data set of files. If a generated tree has a better performance than the previous, then it is saved and so on.

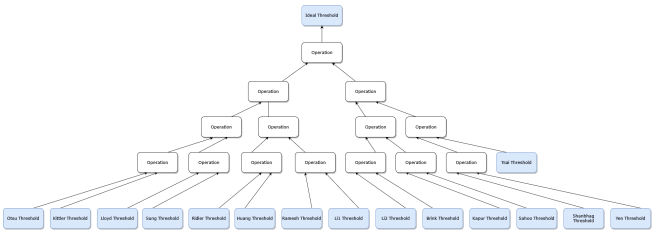

fig.1 - Generating ideal thresholds from leaves to root

For the **local binarization** solution, we kept almost the same approach as the first solution: we generated trees starting from the leaves(containing threshold values) and executing random operations between thresholds until the root node contains a value close to the ideal threshold(fig 1.1).

To apply local binarization, we divide our image into 16 “windows” (the number of windows is equal to the number of threads we used) of pixels. Computing each window will help us obtain a list of threshold values, each type of threshold being calculated using the average of thresholds having the same type. In the input file, each type of threshold is separated by columns. We used multithreading, a thread’s task is to calculate a threshold value for each window. For each window, we take each column of thresholds and calculate their average. To perform operations between thresholds, we used Arithmetic Mean, Square Mean, Cubic Mean, Harmonic Mean and Geometric Mean.

We divided the dataset as before, every newly generated tree we try to train on 70% of dataset and save them if their score is more than specified percentage (54% percent we consider good both for speed of training and accuracy of the results), if it is good enough we save the 5 best trees for the next validation and choose the best.

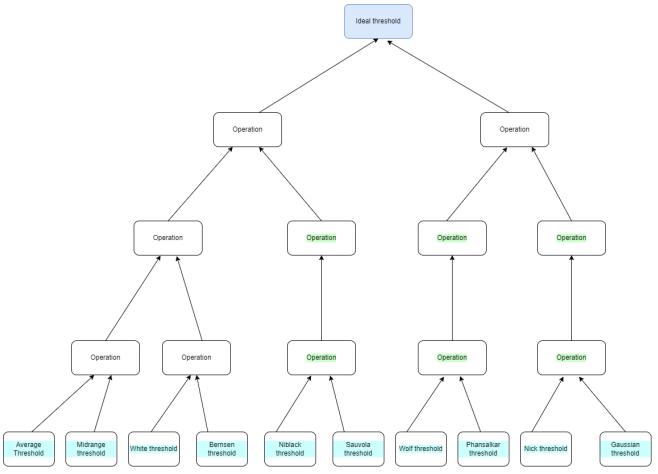

fig 1.1 - Local binarization tree

III. ARCHITECTURE

Both solutions are implemented in Java. To process JSON files for the second approach, we used Jackson library[2].

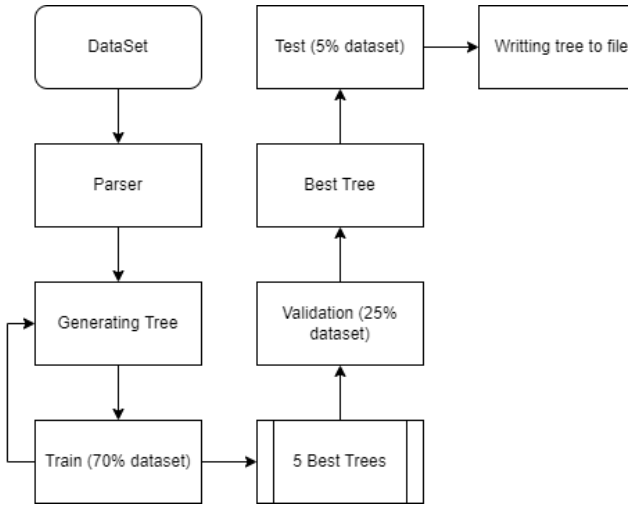

fig. 2 - Project Architecture

They start from reading the thresholds resulted from running 15 different algorithms. First approach starts from generating trees in a loop until the 5 best trees are found, if the score is more than 88% it is saved as the best tree. With generated tree we validate them on 25% of dataset and get the best tree with best score. After that we validate it on 5% of dataset and write it to the file.

The second approach is generating trees with different depths and different operations in a loop until the user stops it, every tree better than older one is written to file.

IV. INTERMEDIATE RESULTS

Global Binarization

While searching for the best solutions, we changed the approach from multiple points of view.

Firstly, the implementation was written in Python. Along the way, we realized that we need to use multithreading to perform operations on the data set in parallel, and python wasn't the best solution to do this job[3]. So we moved the implementation to Java.

The operations we chose in the first place were not the most suitable, as we didn't get the results we expected when using all of them. We decided to eliminate the logical operators '&' and '|'. For the final solution, we kept: arithmetic mean, geometric mean, cubic mean, square mean, harmonic mean, maximum and minimum.

We used multithreading to make each thread iterate on a different section from the initial set of data, without affecting the operations made by other threads. We calculated the start index and end index of the list of files for each thread using the following formula, where ID is the thread ID:

```
int start = ID * (double)N / P;
int end = min((ID + 1) * (double)N / P, N);
```

fig. 3 - Formulas for array indexing

As a performance improvement, we decided to shuffle the list of input files, because we noticed that the score obtained by the best tree is higher when the files are shuffled. Also, we got better results when we used square mean to calculate the average score a tree obtained on a set of files.

Considering the solution where we perform random operations between thresholds and keep increasing the tree level until the root node contains a value close to the ideal threshold, we obtained the following results when generating the best 5 trees:

fig. 4 - Increasing of score of generating trees

Following the graph we can see that our solution is most efficient while generating trees that accumulate a score less than 86(less than 6 seconds). For finding the best 5 trees that accumulate a score of 89 or higher, the execution time extends to 1 min. To isolate the best tree out of the previous 5, we pass each tree through a series of validation tests(25% of the initial data set). This action takes less than 2 seconds in every case. In the end, we test the final tree on 5% of the initial data set of files to see its accuracy.

The best tree is stored in an output file, with the nodes on every level represented as arrays, together with the operations used every 2 thresholds. Here is an example:

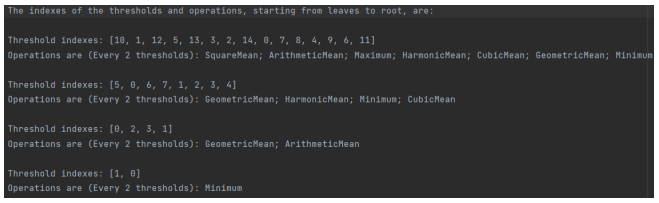

fig. 5 - Output tree - first approach

For the second approach, we need to take into account that the depth of the generated trees is randomly chosen and it might increase the complexity of the program. We calculated the complexity of the program when generating a tree depending on its depth and the operations used(e.g: 'for' has a greater complexity than 'if'). An AtomNode has a complexity of 1, meanwhile we calculated the complexity of an IfNode as 1 + max(complexity(ifTrue), complexity(ifFalse)). A ForNode has a complexity of: 1 + sum(children.map(node -> complexity(node)).

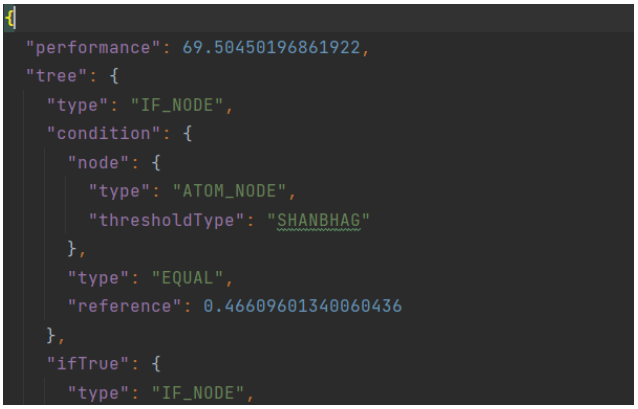

fig. 6 - The output of second approach in json format

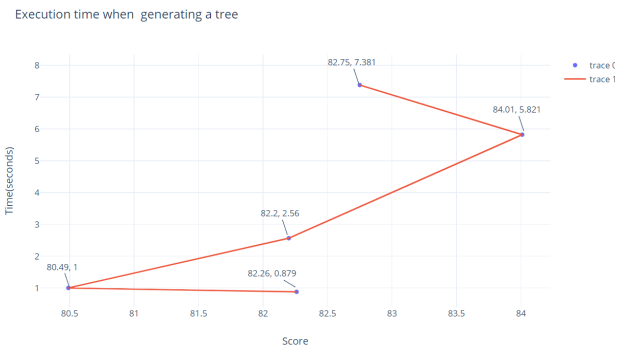

fig. 7 - Scores and execution time of second approach

Following the second graph, we can see that the second solution is unstable. We could not generate trees that accumulated a score higher than 84 because it exceeded the established time limit. For example, for a tree that was successful for 82% of the data set and had a complexity of 6313, the result appears after 3 seconds. But when we generated a tree with greater depth and complexity, it took more than 7 seconds.

Local Binarization

For the local binarization, we used the solution for global binarization as a code base. We made some changes regarding the operations used, we eliminated minimum and maximum and used Arithmetic Mean, Square Mean, Cubic Mean, Harmonic Mean and Geometric Mean.

We used multithreading to iterate and obtain information about every pixel from the file. Also, each thread has the task to obtain threshold values for every type of threshold in the file. Additionally, we used semaphores to allow a single thread to access the list of threshold results. To obtain the thread range, we used the same formula for the global binarization(fig. 3).

For the local binarization, we had to calculate the accuracy for each pixel comparing to their value în the ground truth image, using the following formula:

$$F\text{-measure} = TP/(TP+0.5*(FP+FN))$$

We made tests using timestamps to measure execution time to find the best 5 trees that can reach a range of scores, from 52 to 56:

fig. 8 - Scores and execution time for local binarization

Considering the graph, we can see that our solution works best with generated threes that accumulate a score not greater than 54. For scores lower than 54, the solution takes less than 10 seconds. If we try to generate trees that accumulate a score higher than 56, the execution time extends to one minute, which exceeds our established time limit.

To obtain the best tree out of the previous 5, we ran validation tests(25% of the initial dataset). In every case, this action takes less than 3 seconds. As a final step, we test the best tree we found on 5% of the initial dataset, to validate its performance.

The best tree we generated is stored in an output file, having the nodes of every level represented as arrays, along with the operations used every 2 thresholds(fig. 9).

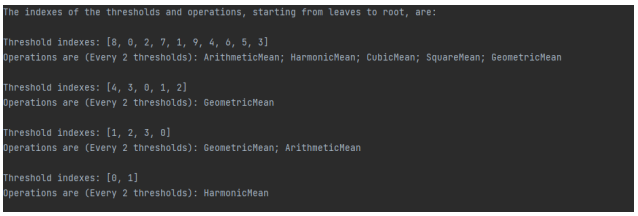

fig. 9 - Output file for the local binarization

V. Conclusions

Comparing the two solutions for the global binarization, we get better results from the one that starts from the leaves and stops when the root node contains the ideal threshold, because we have a fixed depth, and the complexity of the algorithm doesn't depend on depth variations. The second approach could lead to exceeding the time limit for trees with depth close to 10 and complex operations.

Looking at our solution for the local binarization, we could not obtain improvements or at least the same performance compared to the global binarization. This is caused by the computing of the 16 windows of pixels.

VI. References

[1] Sudipta Roy, Sangeet Saha, Ayan Dey, Soharab Hossain Shaikh, and Nabendu Chaki on "Performance Evaluation of Multiple Image Binarization Algorithms Using Multiple Metrics on Standard Image Databases".

[2] https://github.com/FasterXML/jackson

[3]https://www.tutorialspoint.com/python-and-multi-threading-is-it-a-good-idea

[4] Nabendu Chaki, Soharab Hossain Shaikh & Khalid Saeed, "A comprehensive Survey on Image Binarization Techniques"

[5] J. Sauvola, M. PietikaKinen, "Adaptive document image binarization"