BINARY SEGMENTATION

RECONNAISSANCE DES FORMES

Iacob Sergiu

13th February 2020

Contents

CHAPTER 1

INTRODUCTION

1.1 CONTEXT

This report is the result of a university assignment. It aims to prove that the student understood the motivation, goals and means of studying pattern recognition. Therefore, this is a way of summarizing a series of observations and experiments done on images containing shapes.

1.2 MOTIVATION

In order to be able to work with shapes, we must first acquire them. Most of the time, they are not given to us in their simplest form, but found in images with all kinds of backgrounds. They must be extracted as precisely as possible, making sure we differentiate them from the background as accurately as we can. Where there's a need for great precision and accuracy, there's also a need for research, so in this report we'll be looking over some methods we can use to identify objects (shapes).

1.3 GOALS

Having multiple grayscale images, we want to be able to do a *binary segmentation* on them in order to differentiate between the objects and the image's background. Given the fact that we only have gray values at our disposal, we must look into ways of using them in order to state that a certain pixel belongs to either an object or to the background.

CHAPTER 2

EXPERIMENTS

2.1 LOADING DATA

For our following experiments, we will use the following textures:

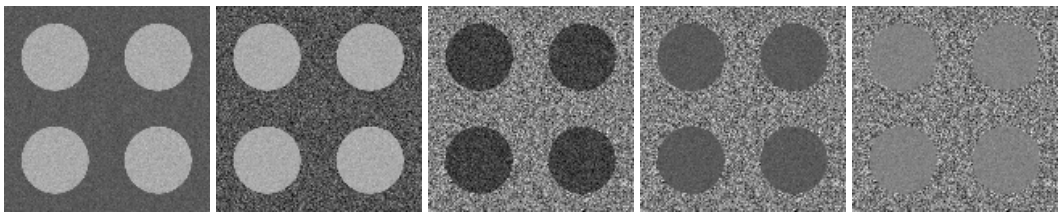

FIGURE 2.1
Images to experiment on

Our reference image (having a clear difference between objects and background) is the following:

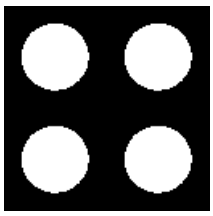

FIGURE 2.2
Reference image

These will be loaded and converted to gray images:

```
rdfReadGreyImage <- function (nom) {
    image <- readImage (nom)
    if (length (dim (image)) == 2) {
        image
    } else {
        channel (image, 'red')
    }
}
```

LISTING 2.1
Loading images

2.2 GRAY LEVELS

2.2.1 GRAY VALUES HISTOGRAMS

A first good step would be taking a look at how the gray values are distributed. We can use a *histogram* for this.

```
1  buildHistogram <- function (nom, nbins){
2      image <- rdfReadGreyImage (nom)
3      h <- hist (as.vector (image), breaks = seq (0, 1, 1 / nbins), main = nom)
4  }
```

LISTING 2.2
Creating histograms of gray values

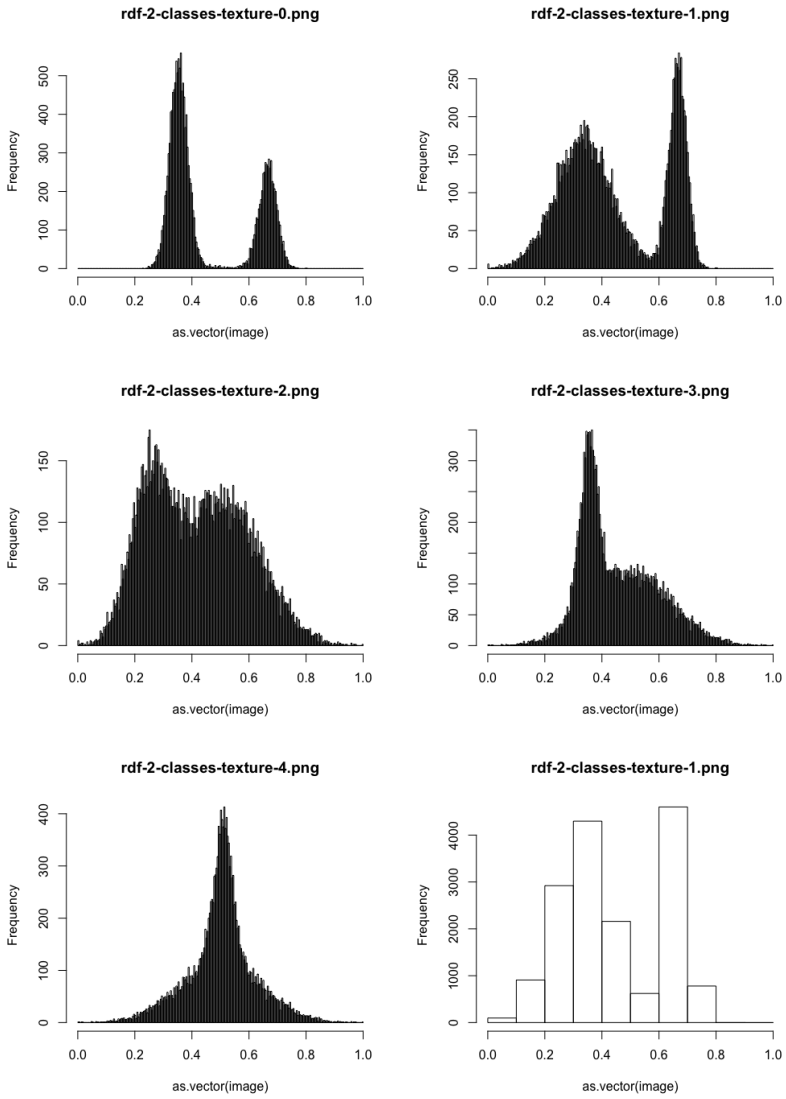

FIGURE 2.3
Histograms of gray values

Since each pixel can have a gray value from 0 to 255, we therefore have a total number of 256 values, so that is going to be our number of *bins* for the histogram. If we were to use, for example, only 10 bins, that would give the number of pixels with gray values between $[0, 0.1)$, $[0.1, 0.2)$ and so on, as it can be seen in the last image 2.3.

2.2.2 THRESHOLDING BASED ON GRAY VALUES

Based on the histograms above 2.3, we may have our first attempt at a simple form of *image segmentation*. By picking the gray value at which (as explained in the figure below), we can classify the pixels on the left as being part of objects, and the others on the right as being part of the background. Cabestaing 2020

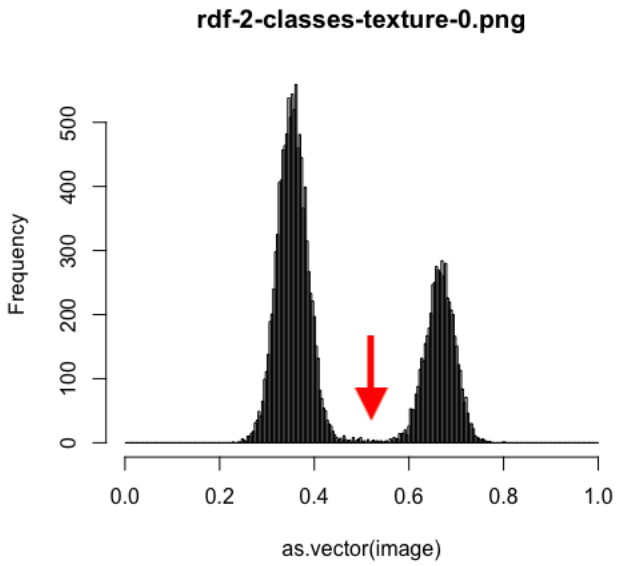

FIGURE 2.4
Choosing the threshold

```
1   buildBinaryImage <- function(nom, threshold){
2       image <- rdfReadGreyImage (nom)
3       binaire <- (image - threshold) >= 0
4       # a "trick" to make the error calculation more precise
5       bin_1 = sum(binaire)
6       bin_0 = dim(binaire)[1] * dim(binaire)[2] - bin_1
7       if (bin_0 < bin_1)
8           binaire = 1 - binaire
9
10      # reference image
11      reference <- rdfReadGreyImage ("rdf-masque-ronds.png")
12      # results
13      error = sum(binaire != reference) / (dim(reference)[1] * dim(reference)[2])
14      error = round (error, 4) * 100
15      info <- sprintf("%s\nthreshold=%s\nerror=%s%%", nom, threshold, error)
16      if (interactive()){
17          display (binaire, "image binaire", method="raster", all=TRUE)
18          legend(x=0.5, y=dim(binaire)[2]/2, info, bg = "lightgreen", box.col = "
    lightgreen", yjust=0.5)
19      }
20  }
```

LISTING 2.3
Segmenting based on the gray threshold

Observation: in the code above, we consider that the majority of pixels are for objects. Therefore, if in our segmentation it's the opposite way, we reverse the image (lines 5-8) so we have a more precise error [1] calculation.

[1] The error represents the percent of missclassified pixels, relative to the reference image

Taking a look at the results, we realise it was not incorrect to call this a *simple* form of segmentation. It's rather effective to reach about the same results as our reference image 2.2 for the first 2 textures, having a low error. However, for the last 2, we don't have a clear "valley" 2.4 in the histogram that we can use as our threshold. For example, the histogram of the last texture resembles a Gaussian distribution, therefore the gray values are evenly distributed relative to their center: $\mu \approx 0.5$.

As a result, we receive an error of about 50% for the last texture when trying to somewhat resemble the reference image. Our lowest error will be achieved by setting the threshold to either 0 or 1, but that will not help us at all.

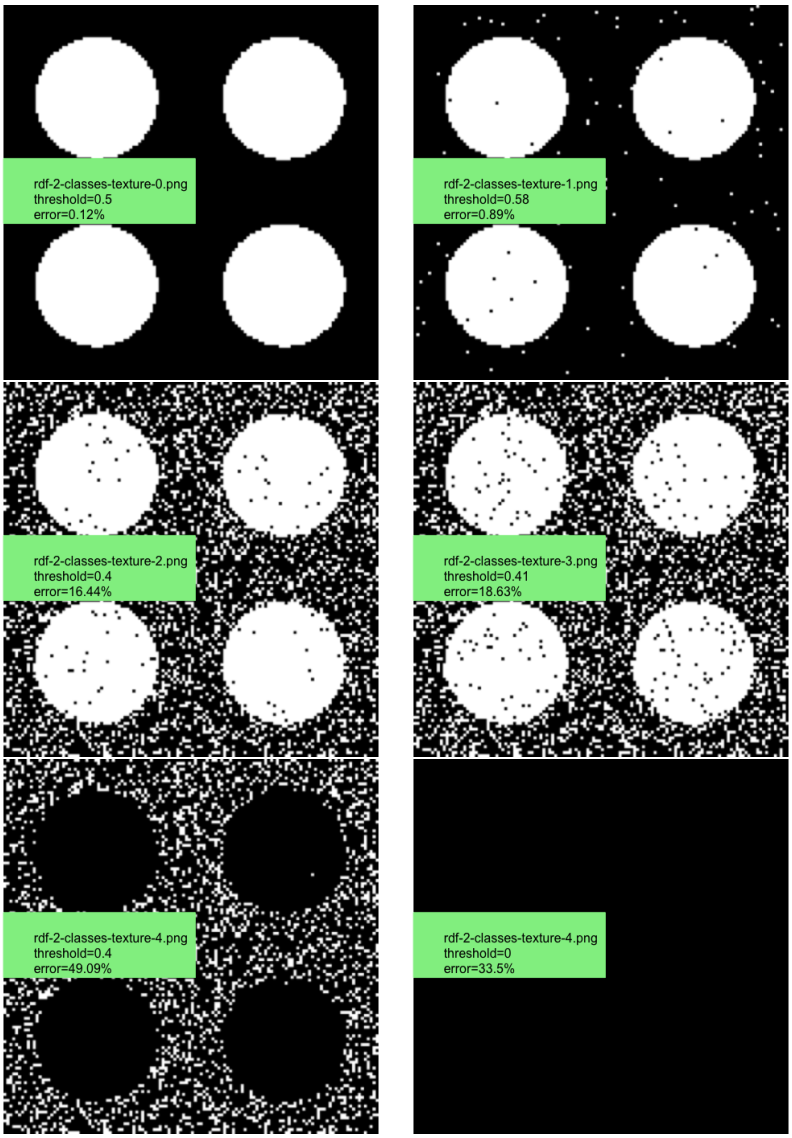

FIGURE 2.5
Image segmentation based on gray threshold

We can conclude this experiment by saying that this method will not suffice, especially for "noisy" textures, as seen above. Even for the simple textures we still have some missclassified pixels, as it's hard (or often times impossible) to get the right threshold for a perfect segmentation. It is intuitive that the single white pixels (for example for the second texture) surrounded by black pixels should also be part of the background, so next we'll focus on trying to use a pixel's neighbours to gain more information.

2.3 TEXTURE LEVELS

2.3.1 CALCULATING TEXTURE LEVELS

We can define a pixel's *texture level* as being the standard deviation of the gray levels for the pixels located in a square neighborhood. Intuitively, a big value for the standard deviation will mean that the nearby pixels have very different gray values. We could therefore say that the current pixel is a *border* between an object and the background.

To calculate this, we first select the *neighbourhood size* we want to work with. That will represent the number of nearby pixels we'll take into consideration when calculating the standard deviation. A neighbourhood size of 2 will give us a square of $(2 * 2 + 1)$x$(2 * 2 + 1)$, so 5x5, since the square will be centered in our original pixel. From there on, we'll just "sweep a neighbourhood window" over each pixel to calculate the average gray level. This can be done with the help of the *filter2()* function from R, which uses fast 2D FFT convolution product. RDocumentation 2020

```
# Moyennage d'une image
rdfMoyenneImage <- function (image, taille) {
    # cote du masque = 2 *taille + 1
    taille <- 2* taille + 1
    masque <- array (taille ^ -2, c (taille, taille))
    # image filtree
    filter2 (image, masque)
}
```

LISTING 2.4
Calculating a pixel's average value

Taking a quick look at the results of the function *rdfMoyenneImage*, with a neighbourhood size of 2 (a square of $5x5$ pixels), we can say that it applies a "blur" over the image. That helps reducing the noise.

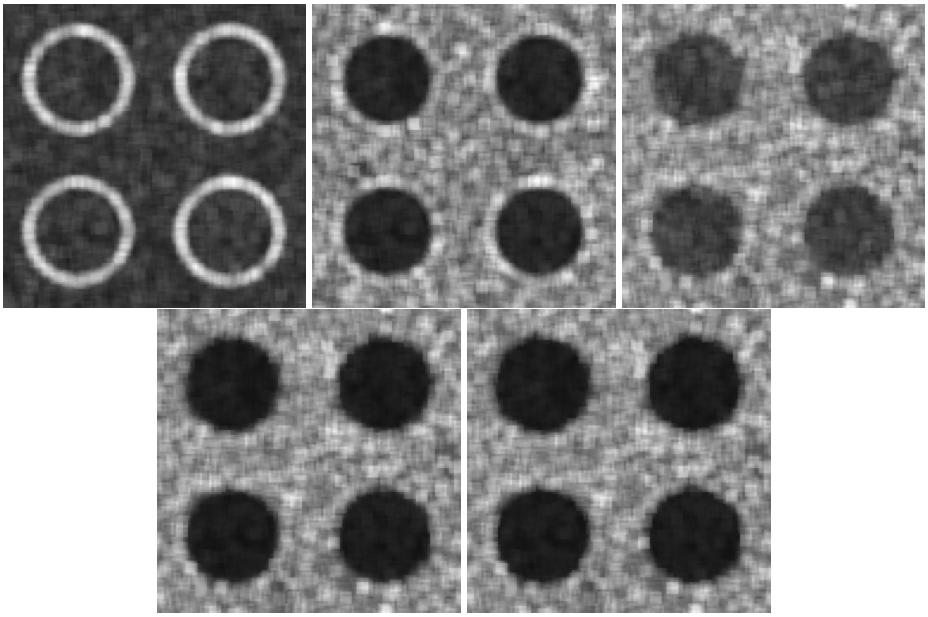

FIGURE 2.6
Result of rdfMoyenneImage() function

Afterwards, we just follow the formula for the standard deviation for each pixel. Observation: the image is normalised to have values between 0 and 1, since we're working with gray values.

```
# Ecart type normalise des voisinages carres d'une image
rdfTextureEcartType <- function (image, taille) {
    # carre de l'image moins sa moyenne
    carre = (image - rdfMoyenneImage (image, taille)) ^ 2
    # ecart type
    ecart = sqrt (rdfMoyenneImage (carre, taille))
    # normalise pour maximum a 1
    ecart / max (ecart)
}
```

LISTING 2.5
Calculating a pixel's average value

2.3.2 HISTOGRAMS OF TEXTURE LEVELS

Applying the methods above (neighbourhood size=2), we can build the following histograms of texture levels:

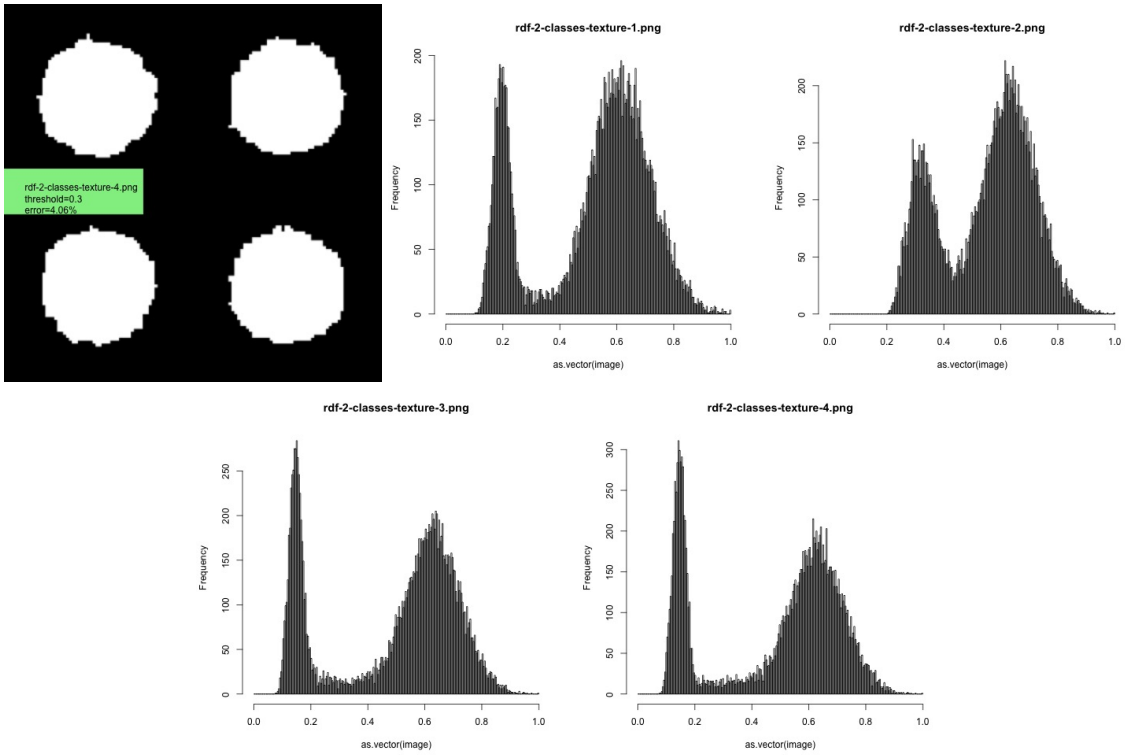

FIGURE 2.7
Histograms of gray values for each texture

An interesting observation is that we now have a "valley" for each texture, even for the noisy ones. Next, we can use the same "thresholding" method as in our previous experiment 2.2.2, but applied on the images obtained from *rdfTextureEcartType()*.

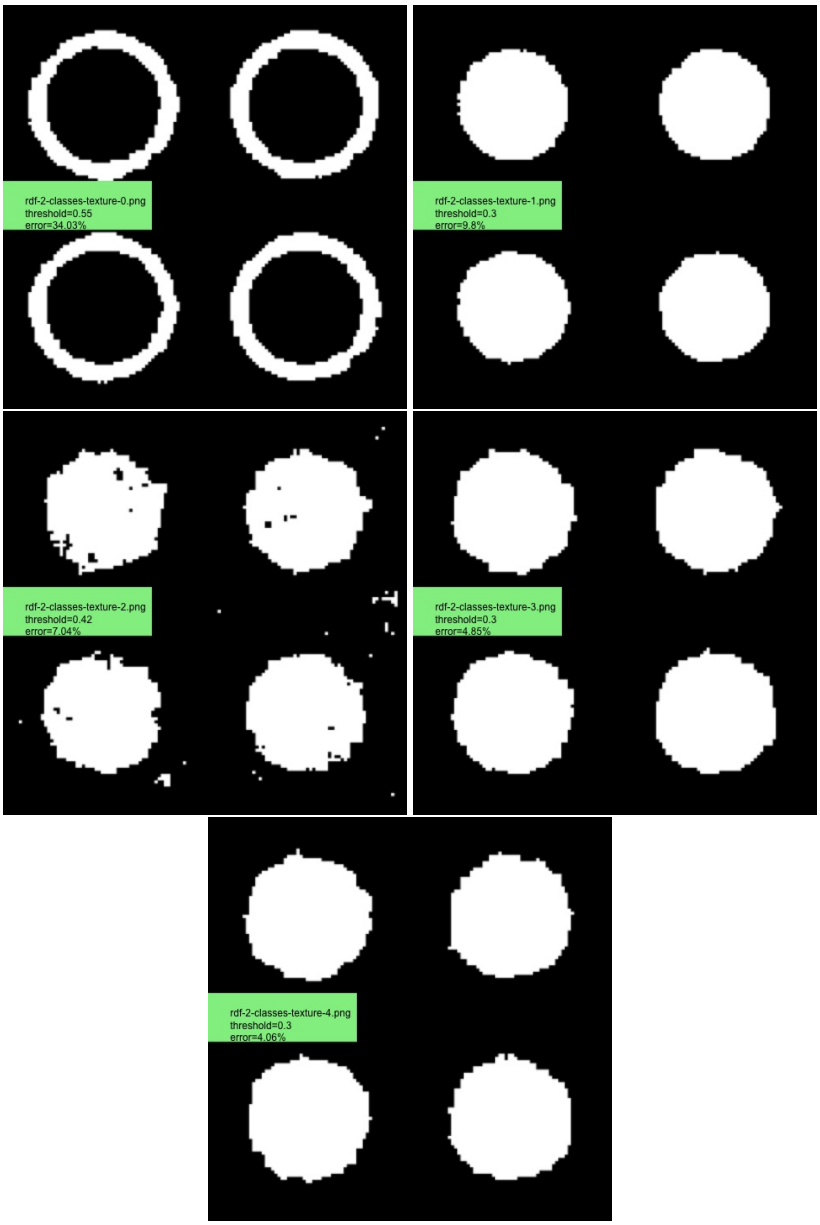

FIGURE 2.8
Thresholding based on texture

Our biggest win here, using texture levels, is that for the last 2 textures, the segmentation works much better, due to the "mathematical blur" we applied by calculating the average gray values. However, this is still far from perfect, especially for the first texture, where our segmentation resulted in... donuts.

As a conclusion for this experiment, it seems like what "gray thresholding" does bad, "texture thresholding" does good and viceversa. Given this, it might be a good idea to try and combine the two methods.

2.4 COMBINING GRAY AND TEXTURE LEVELS

2.4.1 CONJOINT HISTOGRAMS

A useful tool when working with images are *conjoint histograms*: Wikipedia 2020 These help us analyse the difference between images. We can create these histograms by using the original gray images and the ones we derived using the standard deviation. 2.6

```
# Calcul de l'histogramme 2D (log + normalise) de deux images
rdfCalculeHistogramme2D <- function (image1, bins1, image2, bins2) {
    # Bins dans les deux images
    indices1 = findInterval (image1, seq (0, 1, 1 / bins1))
    indices2 = findInterval (image2, seq (0, 1, 1 / bins2))
    # Tableau de contingence
    counts <- table (indices1, indices2)
    # Extension en tableau 2D incluant les valeurs nulles
    liste <- as.data.frame (counts)
    h2d <- array (0, c (bins1, bins2))
    h2d[cbind (liste[,1], liste[,2])] <- liste[,3]
    # Passage en log
    h2d <- log (1 + h2d)
    # normalise pour maximum a 1
    as.Image (h2d / max (h2d))
}
```

LISTING 2.6
Calculating conjoint histograms

The *findInterval()* function assigns each image pixel to its corresponding bin in the context of the single dimension histogram. In our case, the number of bins will be 256 for both. Afterwards, the *table()* function helps us build a *contingency table* that we can use to build our resulted gray image. RDocumentation 2020

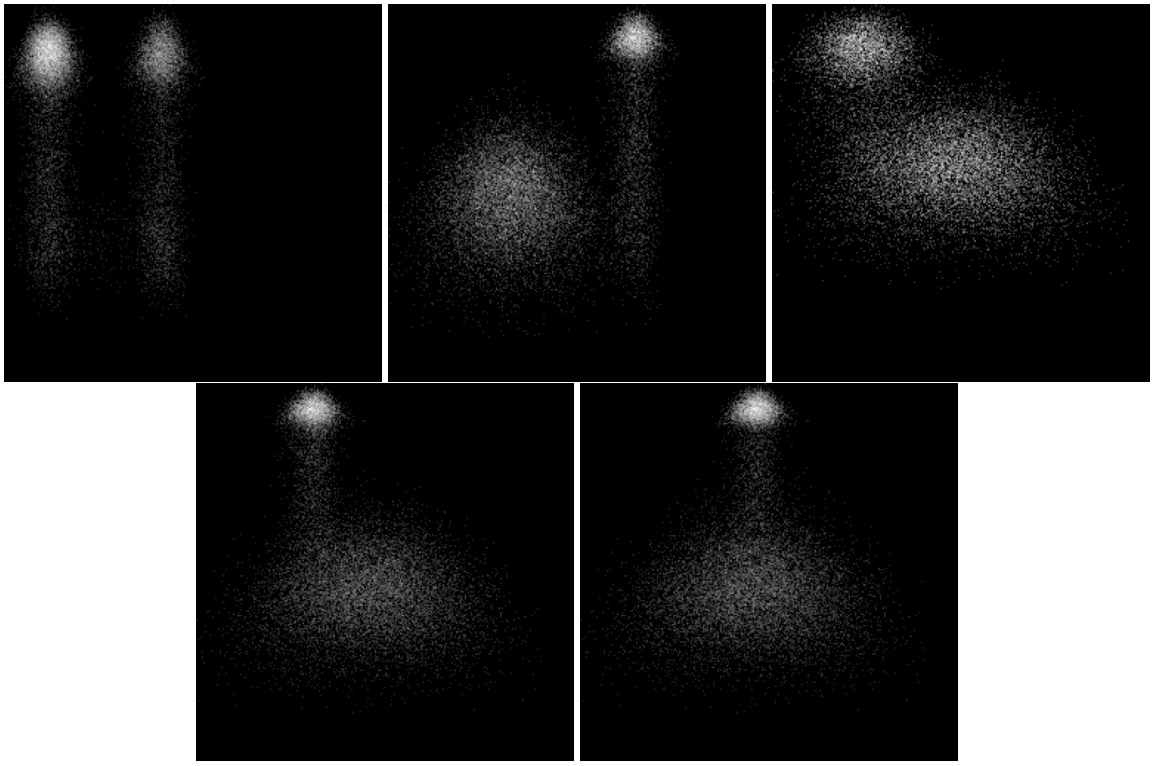

FIGURE 2.9
Conjoint histograms for the textures

Our goal right now is to combine the two image attributes described above (the gray and the texture levels) to properly do the binary segmentation. We can build a third gray image on which we will apply the thresholding method we described earlier. For the third image, each pixel will have the following value:

$$C_{ij} = a * G_{ij} + b * T_{ij}$$

where C is our resulted image, G is our gray image and T our texture image.

Afterwards, we can normalise this image $C_{ij} = C_{ij} / \max(C)$ to obtain a gray one in which we can analyse the gray values histogram. Looking at that histogram, we'll choose the threshold and finally do the binary segmentation.

2.4.2 FINDING THE LINEAR COMBINATION BETWEEN G AND T

In the equation above 2.4.1, a and b correspond to the parameters of a line equation. The line that those parameters represent may be the line that best separates the "conglomerations" of white pixels in our conjoint histogram.

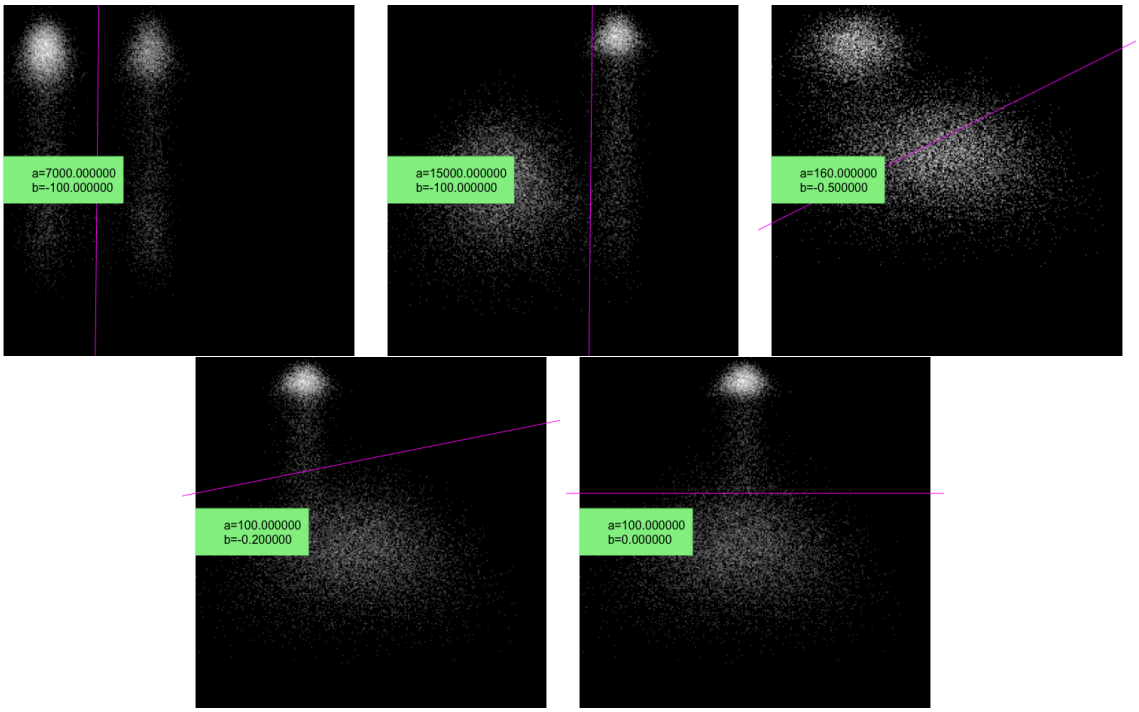

FIGURE 2.10
Separating the attributes with a linear equation

Observation: The values for a and b were chosen "by hand". This won't result in the best results, but still useful to draw some conclusions.

After finding some satisfying values for a and b, we can try and build our final grayscale images on which we'll apply a binary threshold based on the histograms for the gray values.

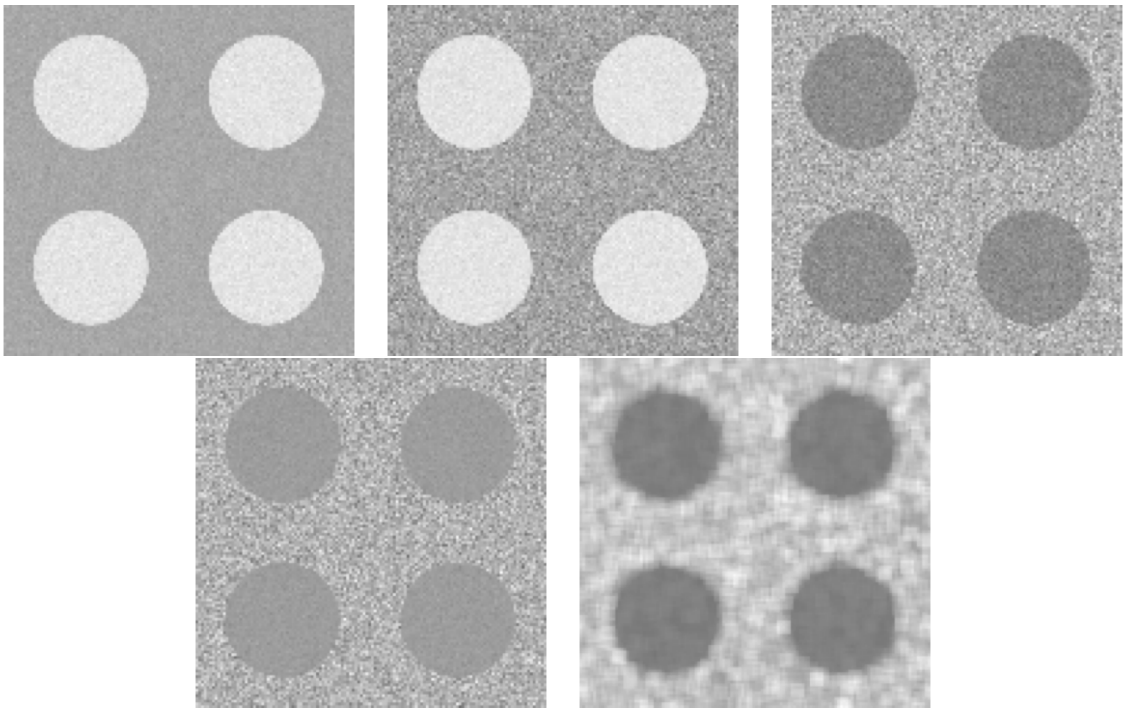

FIGURE 2.11
Final gray images

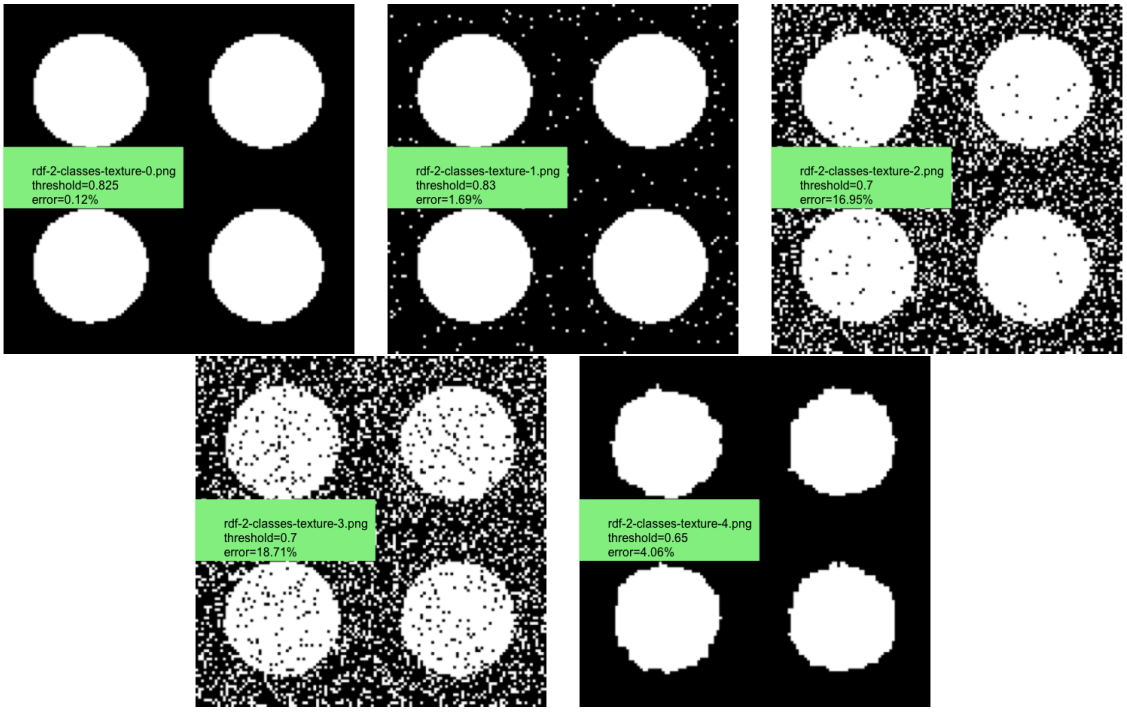

FIGURE 2.12
Binary segmentation on the final images

CHAPTER 3

CONCLUSION

Using only the gray values or the texture values as attributes did not result in a great segmentation. As we've seen in the experiments, the error rate, when using each attribute individually, can be pretty high in certain cases, even though for some textures the methods are really precise.

However, where one of them did bad, the other did well. By combining the two, we have a more precise way of segmenting the image. In conclusion, we may state that the combination of the two attributes is a worthy contender for binary segmentation.

BIBLIOGRAPHY

Cabestaing, François (2020). *Reconnaissance des formes*. Université de Lille. URL: http://master-ivi.univ-lille1.fr/Cours/RdF (visited on 4th Feb. 2020).

RDocumentation (2020). *R documentation*. DataCamp. URL: https://www.rdocumentation.org/packages/EBImage/versions/4.14.2/topics/filter2 (visited on 13th Feb. 2020).

Wikipedia (2020). *Histogramme conjoint*. Wikimedia Foundation. URL: https://fr.wikipedia.org/wiki/Histogramme_conjoint (visited on 13th Feb. 2020).