Data and Document Mining project

Pre-processing and layout analysis algorithms

Lorenzo De Luca e Manuel Natale Sapia

Abstract

In this project the goal was to implement the various document recognition algorithms. In particular, we focused on preprocessing and layout analysis.
Starting from the literature, we have implemented the various methods of preprocessing a document by carrying out the various steps: binarization, CCs (to segmentate characters or words), adaptive RLSA, Hough transform for deskew and projections. Subsequently we carried out the layout analysis in particular for the page segmentation, implementing bottom-up methods, such as MST, Docstrum and Voronoi and top-down methods, such as XY-tree. Finally we memorized the coordinates of each paragraph, through Pytesseract OCR we are going to transcribe the content of the paragraphs in a text document with rather satisfactory results.

1 Introduction

Faster computer, large computer memory, and inexpensive scanners fostered an increasing interest in document image analysis. In our work the goal was develop a series of algorithms, known in the literature, useful for this purpose.

Generally, documents image are generated from physical documents digitized by scanners or digital cameras, for this reason, before text-recognition, preliminary steps are necessary so that there are fewer errors: pre-processing and layout analysis.

Preprocessing is necessary to correct the deficiencies in the data acquisition process, therefore it focuses on making the image binarized, eliminating the imperfections of the paper or ink; to find the Connected Components, i.e. the characters, and possibly rotate the image if it is tilted. After this phase, the layout analysis phase begins, which breaks the document into different regions (blocks of text, figures, tables). We are focus on the recognition of textual blocks that can be passed successively to the OCR which will create a *.txt* file with the relative recognized text.

To implement what just described, we used Python 3, and structuring the code in three files:

- *PreProcessing.py*: containing the main methods of the paragraph 2
- *LayoutAnalysis.py*: containing the main methods of the paragraph 3
- *Utils.py*: which contains a number of methods used to make preprocessing algorithms and layout analysis more functional
- *Main.py*: that contains one use example where you can set the binarization and segmentation method, choose to show or not steps in a OpenCV window and some few parameters.

We also used the Jupyter notebook to show the results obtained, there are two files that import the previous files : *projectDDM-Preprocessing.ipynb* and *projectDDM-LayoutAnalysis.ipynb*

In this paper the work done is explained in detail, explaining the algorithms and methods used and showing the results obtained through images; in particular in Appendix A other images have been inserted to illustrate further results changing some parameters.

2 Pre-processing

This step is done to improve the document by removing noise and other distortions in the written material. This step includes binarization, calculation of connected components, detection and removal of the inclination, draw of bounding boxes for characters or words.

2.1 Binarization

Document binarization is one of the preliminary most important steps in most part of system analysis. The goal of binarization is to convert an grayscale or color image in a representation on two levels, i.e. composed only with white and black pixels. Generally the binarization can be done with different criteria and using different algorithm, in our case we choosed to implement three different versions:

- **Fixed Threshold Binarization**, that is a global threshold binarization where each pixel is set to 0 if it is lower than threshold, else is set to 255.
- **Otsu Binarization**, that is a adaptive threshold method generally used for images with background noise; in fact it finds the optimal threshold value so that the variance is minimal.
- **Sauvola Binarization**, that is a local binarization technique useful with images with not uniform background, so instead of calculating a single global threshold for the whole image, it calculate more threshold for every pixel, taking into account the average and standard deviation of neighboring pixels.

This process is very simple to implement thanks to the libraries *OpenCV* e *Scikit-Image*, in fact, these offer functions that achieve the goal very efficiently. In our implementation we obtain the binarized image in two steps using the method `binarization(mode,image)` to which two parameters are passed: the image to binarize and the kind of binarization that we want to obtain ('inverse','Otsu' or 'Sauvola'). In the first step, the image is converted from three levels (Red, Green, Blue) to one level transforming it into grayscale using the OpenCV function `cvtColor()` (only if this is not already in grayscale).

The result of this procedure is then passed as an argument to the *Scikit-Image* method `binarize()` which also receives the threshold corresponding to the type of binarization chosen, *threshold_otsu* and *threshold_sauvola*, and which takes care of blackening the pixels above a certain threshold and making white the pixels that not exceed the threshold, in according to various methodologies.

Fixed-threshold binarization is performed using OpenCV's *threshold* method which has four arguments: the first argument is the source image (in grayscale); the second argument is the threshold value used to classify pixel values; the third argument is the maximum value assigned to pixel values that exceed the threshold and finally OpenCV provides different types of thresholds in the last parameters. In our case the fixed threshold binarization is carried out using `THRESH_BINARY_INV` which, contrary to what is described, makes the pixels above the threshold white and those below black. This method has two outputs: the first is the threshold used (in this case 127) and the second is the effectively binarized image. Since the `binarize()` method returns a binarization consisting of 0 and 1, it was necessary to multiply it by 255 in order to uniform the result with the fixed threshold. For the representation of Otsu binarization we also added a histogram to represent each pixel in order to plot the distribution of the black and white pixels and observe the difference between background and foreground. In Figure 1 the results of the method are shown:

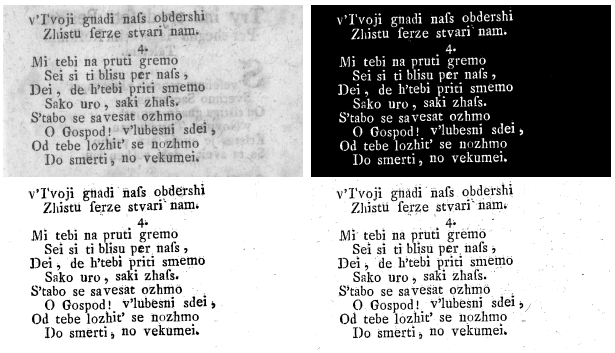


Figure 1: Original image(top left), inverse binarization(top right), Otsu's binarization(lower left), Sauvola's binarization(bottom right)

2.2 Connected Components

In the document image analysis and recognition, the main methods for segmentation are based on the identification of homogeneous regions, or on the search of connected components.
These introduce the concept of **pixel adjacency**: we define 8NN the 8 pixels adjacent to pixel P, and 4NN the closest pixels (2,4,5,7) as we can see in Figure 2

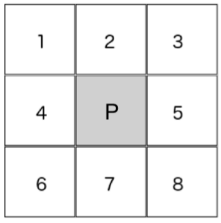


Figure 2: pixel adjacency

Furthermore we define **path** the sequence of pixel $P_1, P_2, ..., P_n$ such that $P_{k-1}(k > 1)$ and $P_{k+1}(k < n)$ are both neighbors of P_k; therefore we can have a "path-4" (using 4NN) or a "path-8" (using 8NN). A set of pixels is connected if, for each pair of pixels A and B in S, exists a path having A as first element, and B as the last one, and if all the path pixels are in S.

In our work the algorithm `ShowCC(binarized_img, connectivity)` calculate the connected components, which thanks to the *OpenCV* library, calls the iterative algorithm `connectedComponents()` that takes the binarized image and the 'connectivity' parameter to define the pixels adjacency. This function returns an int value that indicates the number of connected components in the image and an array with the same size of the image where each component is labeled with a number (from 1 to the number of total components found), and the label 0 represents the background.

To show this, we color each connected component, thanks to the functions that transform the image from binarized (1 channel) to color (3 channels).

As illustrated above, we can see the difference between 8- and 4-connectivity: with 4 will detect a greater number of objects because it will connect only the horizontal and vertical pixels, as shown in the Figure 3; while 8-connectivity will detect fewer objects because diagonal pixels are also taken into account.

Figure 3: 4-connectivity and 8-connectivity

2.3 RLSA

The Run Length Smoothing Algorithm (RLSA) is a commonly used method of grouping characters in a word, words in text lines or lines in a text blocks. The algorithm is applied to each row (or column) in a binary image, and transform a sequence of pixels by the following rules:

1. A white pixel is changed to a black pixel if the number of adjacent white pixels is less than a

predefined threshold C, otherwise it is left white.

2. A black pixel in the input sequence remains unchanged in the output sequence.

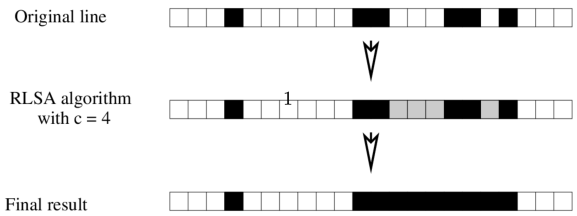


Figure 4: RLSA algorithm with C=4

The selection of the smoothing threshold C is critical:

- low values do not allow to group together characters in words.
- with high values different text regions are merged together.

We must choose C appropriately, in order to group homogeneous regions, for this reason our implementation of the algorithm is adaptive, and we can choose a new threshold based on the document we are analyzing; moreover, often the horizontal and vertical space between components is different, therefore we calculate two threshold for the RLSA in horizontal and vertical direction.

Using the `valueRLSA(binarized_img, vert: bool = False)` method to which we pass the binarized image, we calculate the distance between the centroids using `findDistance(binarized_img, vert: bool = False)`, which returns an inaccurate measure of distance between characters, then subsequently, we calculate the average space occupied by a character with `findMidDistanceContour(binarized_img,vert: bool = False)` and we subtract this value from the distance between centroids. Now we obtained an approximate, but true, average distance between characters.
Once we find the correct threshold value, we pass this value to `rlsa(image, horizontal: bool = True, vertical: bool = True, value: int = 0)` method, a modification of the Vasista Reddy implementation from its repository[2].

To verify the operation, we apply it to an image with the purpose of grouping the words; the following image shows us the result obtained.

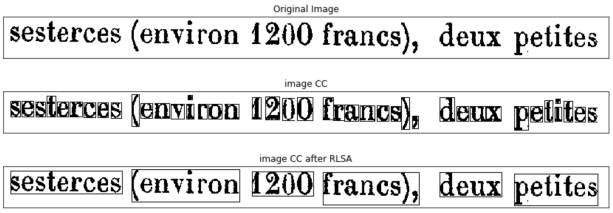


Figure 5: adaptive RLSA

2.4 Hough Transform

Normally the document are digitized through the use of a optical sensor like a scanner, a book scanner or a digital camera, to obtain the digital image. In all this cases it's inevitable to have some inclination grade. The inclination angle is the angle that text lines in the digital image form in the horizontal direction.

This part of pre-processing is very important if we want to do page analysis because the presence of inclination makes elements analysis more complicated overall where there are text column. So we have to estimate and to correct this inclination rotating the image of some grade in the opposite direction. There are more methods for skew estimation like: projection profiles, Hough transform, clustering of the distances of CCs in the page, hypothesize & verify methods.

To reach our goal we have used the Hough transform by the method `HoughLinesP()` of the library OpenCV to make the deskew and by the method `hough_line()` of the library Scikit-Image to show the Hough transform histogram. The method `HoughLinesP()` offer a probabilistic Hough transform receiving as parameter an image with the edges of the components present using the method `Canny()`

of OpenCV, and return the found lines. To make the transform more accurate, we used a utility method `removeFiguresOrSpots(binarized_img, mode)` for removing the figures in the image, because that can generate some vertical lines which make the deskew value wrong. The parameter 'mode' of this method is a string that represent the kind of removal: 'figures','spots','linesVert','linesHoriz', 'linesBoth' (lines in both direction).

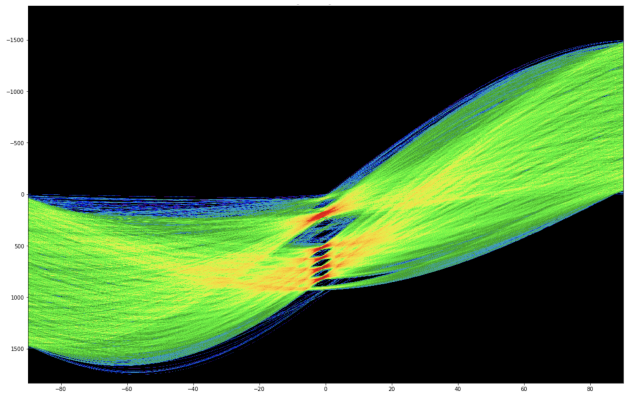


Figure 6: histogram Hough trasform

Once we found the lines we computed the angle of all of this lines and we calculated the average of their values to find the best angle that will be used to make the deskew.

To deskew we got the rotation matrix using the method `getRotationMatrix2D()` that receive as parameter the center of image and the best angle. We passed this matrix to the method `warpAffine()` along with the image size to perform the real deskew both to the original image and the one without figures that are returned by the method `houghTransformDeskew(binarized_img, original_img, plot : bool = False)` that we created. In addition to the original and binarized image, this main method also receives a Boolean variable that enables the plot of the Hough transform histogram.

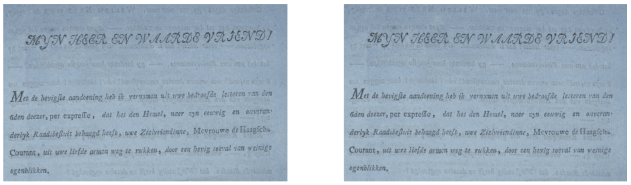


Figure 7: image rotate of -1.26 degrees

2.5 Projection

After rotate the image with the Hough transform, we use a horizontal projection profile to check if the rotation of the image was done well.

The projections are a one-dimensional matrix with a number of positions equal to the number of lines in an image, in which at each position a count of the number of black pixels is stored in the corresponding line of the image. This histogram has the maximum amplitude and frequency when the text in the image is tilted to zero degrees because the number of linear black pixels is maximized in this condition. As we can see from the results obtained in Figure 8 the peaks in the profile calculated from the skew image(top) are lower and less spaced than those calculated in the rotated image(bottom).

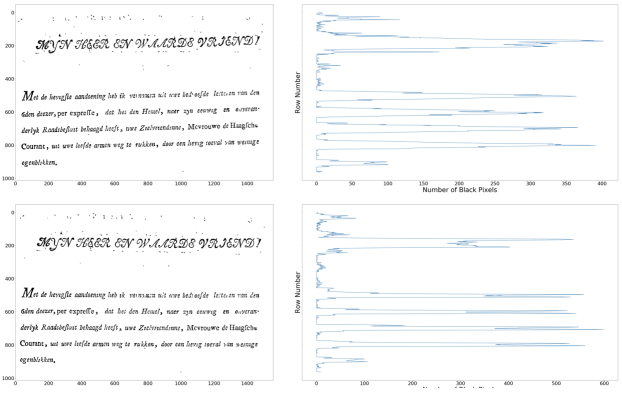


Figure 8: Projection before and after rotation

To implement the projection it was sufficient to

count the number of black pixels in each row from a given binary image, and was done using the `sum()` function of *Numpy*, indicating to add all the black pixels (i.e. those pixels with a value of 0) along an axis, in this case the horizontal, corresponding to the lines. Once the above was done, by calling `projection(img_bin)`, it was sufficient plot the results through a histogram.

3 Layout Analysis

Once that the pre-processing part is done, the next step is to compute some layout analysis in order to find and segment the document in regions with homogeneous content like text blocks, figures and tables. We focus our attention to text blocks because we want to extract from this the content and put it in a text document using the library *PyTesseract* by OCR.

The layout analysis can be physical in order to extract the geometric page structure and logical to extract the logical structure of the document. We focus our attention on the physical analysis where the sequence of operations is similar to the one used in character recognition:

1. Segmentation of the page into blocks finding objects.
2. Extraction of features from that objects.
3. Recognition of these object.

Page layout can be broadly divided into two classes: *overlapping* and *non-overlapping*.
Pages with overlapping layout include a document component whose region overlaps with those of others, while in non-overlapping layout, document components are separated by white space. Subclasses of non-overlapping layout are:

- *rectangular* layout: if all document components can be circumscribed by rectangles whose sides are parallel or perpendicular with one another.
- *Manhattan* layout: if the regions have boundaries consisting of line segments parallel or perpendicular with one another.
- *non-Manhattan* layout: tilted text regions are sometimes contained for highlighting from other upright areas. This makes it difficult to deskew page images.

There are three categories of segmentation algorithms: pixel classification, edge-based segmentation and region-based segmentation. We used the last one and its methods are divided in other 2 categories:

- **Bottom-up:** based on the location of CCs that will be merged in words, text lines, paragraph. RLSA, MST, Area Voronoi and Document spectrum are main example of this methods.
- **Top-down:** based on segmentation of big components into smaller sub-components. Projection profile and XY Tree are main example of this methods.

In this work we used three bottom-up methods: MST, Area Voronoi, Document spectrum and one top-down method: XY Tree and our code works principally on Manhattan layout because our dataset was composed by non-overlapping Mahnattan layout documents but with few changes we can use Area Voronoi for non-Manhattan layouts.

3.1 Minimum Spanning Tree (MST)

A minimum spanning tree (MST) is a minimum-weight, cycle-free subset of a graph's edges such that all nodes are connected. We consider the centroids of CCs as graph nodes and the edges are the edges that connect two CCs by their centroids, so the CCs are merged into a region if this doesn't cause a cycle in the graph. The algorithm outputs multiple disjunct MSTs where each tree corresponds to a text block that can be recognized by the OCR.

To make the right MSTs we need two important things: the CCs centroids and

the distance distribution peak values. The first one is computed by the utility method `findCentroids(binarized_img,output_img)` that receives as parameters the binarized image and an output image where centroids will be drawn. To do this we find the contours of the CCs and for every contour we find the coordinate X and Y using the moments, and puts it in an array of points. The second one instead is computed using the utility method `findPeaks(distances, distance: int=0, plot: bool=False)` that receives as parameter the edges distances of the k-neighbors graph and returns the two higher peaks with a distance between them of at least 20 pixels.

Once we found the points and the two peaks we could compute the MST using the method `minimumSpanningTreeEdges(points, k, peak_values=[])` that receive the points, the k value which is the number of neighbors to consider and the peak values. This method uses the method `kneighbors_graph(points, k)` of the library Scikit-Learn to find the k-nearest neighbors graph which will be passed as a parameter to the method `minimum_spanning_tree()` of the library SciPy. The next step is to delete the edges too much longer, using the higher value of peaks as threshold and to return the full tree and the new edges list. We plots this list using the utility method `plotEdges()` that returns box coordinates in order to send it to the OCR.

3.2 Document Spectrum (DocStrum)

The document spectrum, or docstrum, is a method for structural page layout analysis based on bottom-up, nearest neighbor clustering of page components. The method yields an accurate measure of skew, within-line, and between-line spacings and locates text lines and text blocks. It is advantageous over many other methods in two main ways:

- Independence from different text spacings.
- The ability to process local regions of different text orientations within the same image.

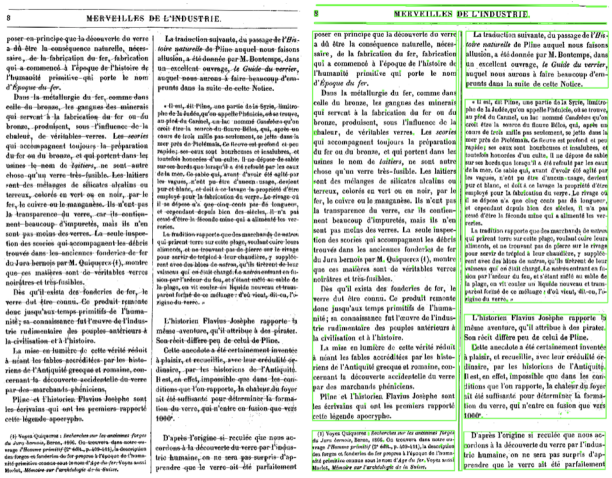


Figure 9: MST(left) and box after MST(right)

To make the docstrum we need to find the centroids list coordinates as we did for MST to find the k-nearest neighbors graph and from this graph we find the edges and their distances to obtain statistical informations. So we used a utility method named `edgesInformation(edges, points, distances)` that receives as parameter the edges list, the points and the distance list. We want to find the horizontal and the vertical edges so this method compute for every edge its angle grade and put it in the corresponding array with an approximation of 20 degrees vertically and horizontally and returns this array.

We use this arrays to compute the docstrum using the corresponding method `docstrum(binarized_img, edges, points, thresh_dist)` that receive an input and output image, the edges, the points and the maximum between the two peak distances used as threshold. Only the distance smaller than threshold will be plotted and written on the output image. This procedure is done the first time to find the lines of text and the second time to find the text blocks. In both cases the corresponding boxes will be created and the coordinated returned as a value in order to send it to the OCR.

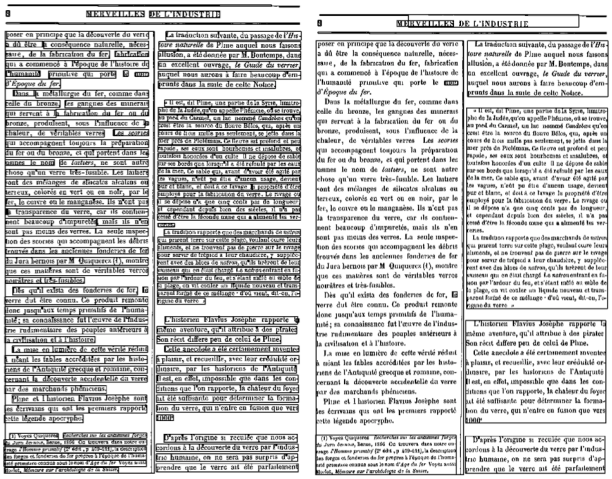


Figure 10: Segmentation of lines(left) and box(right) with Docstrum

3.3 Voronoi

Voronoi area is a method of page segmentation for documents with non-Manhattan layout and a skew as well. A Voronoi diagram is a partition of a plane into regions close of a given set of points called Voronoi cells. In our case, these points are the centroids of the charaters (CCs). We employ the *approximated area Voronoi diagram* which represents the neighborhood of CCs as polygons. Page segmentation can be considered to be the selection of appropriate line segments as boundaries of document components, and for this purpose we need the Euclidean distance and the area ratio 4 calculated from a pair of connected components that are neighbors in the Voronoi diagram.

In order to write Voronoi diagram cells we have to find the Delaunay triangulation that is the dual of the Voronoi Area. In our implementation the `voronoi(points, img, peak_values)` method needs points list to draw the Delaunay triangulation, the image where the triangulation will be drawn along with the points of centroids and the peak_values used in the Voronoi edges selection. The method builds a rectangle with the same size of the original image and creates an empty Delaunay subdivision with the rectangle size where points will be inserted. So it drawn the triangulation using the `drawDelaunay(img, subdiv, delaunay_color)` method that gets the triangle list and draw the matching lines on the image.

In order to draw the full Voronoi diagram and the one with the segmentation, we use the `drawVoronoi(img, subdiv, points, peak_values)` method and three important datas:

- Edges of K-NN graph.
- Distances of K-NN graph.
- List of Voronoi cells coordinates.

With this datas we can build the full Voronoi diagram using the OpenCV `cv.polylines()` method to print every facet in two different images, instead, to reach the segmentation we used the same mathematical criteria of the paper [4], changing a bit the parameters. So for every NN we check if the edge that connect the two CCs,for example A and B, intersect with an edge of the Voronoi cell of both A and B. If it's true so we delete this edge from the Voronoi cell using the OpenCV `cv.line()` method drawing white line and putting index of the two CCs in a new array of edges.

At the end we done some improvements in the Voronoi segmentation diagram in order to make it more easily analyzable. The method `voronoi()` returns the binarized image with trees edges, the full Voronoi diagram and the segmentation Voronoi diagram.

In Figure 11 we can see the complete voronoi diagram (left) and the voronoi diagram after segmentation (right) and in Figure 12 we can see the CCs tree after the Voronoi segmentation.

3.4 XY-Tree

An XY tree is a spatial data structure where each node corresponds to a rectangular region, that is a

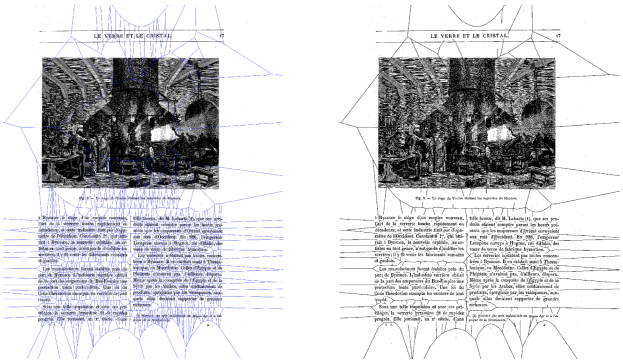


Figure 11: Voronoi Segmentation

portion of the page.
The children of a node represent its splitting in smaller blocks along either the horizontal or vertical directions, infact usually at each level the cutting direction is changed.
The creation of the XY trees is based on the projections, in fact a cut is made when in the projections graph there is a rather wide gap, that indicates the separation between two blocks of the image.

This data structure is very intuitive to construct, however it has some limits, in fact based on the projections, it is important that the images are not inclined, furthermore pages cannot be processed (easily) with regions partially superimposed, because the document can be split only along the whole width (or height) of the sub-image. (Figure 13)

Our implementation builds on two main methods `cutImage(image_bin,nPixel,space,verticalCut: bool = False)` and `cutMatrix(img_name, path, img_bin, info, infoV, XY_Tree)`: The first method do a projection of the image and saves the white lines on a vector (i.e. the lines with a number of black pixels less than a defined threshold), subsequently the white space between the lines are eliminated from this vector. An information vector is then returned and it containing the number of white lines separating two blocks of text, the line number where the white block begins and the line number where it ends. The second method, `cutMatrix()`,

Figure 12: CCs tree after Voronoi segmentation

takes this vector of information in input and cuts the original image matrix. Sub-matrices are then created and the `cutImage ()` method will be applied iteratively to evaluate where to make the vertical cuts. At the end of the procedure the method will return the xy tree.
In Figure 14 we can see the tree(right) relative to the image(left). As we can see from the image, the tree nodes are represented using three colors:

- White, indicates the root, i.e. the page.
- Red, indicates the leafs, i.e. cutted blocks of text.
- Yellow, indicates the intermediate nodes, i.e.

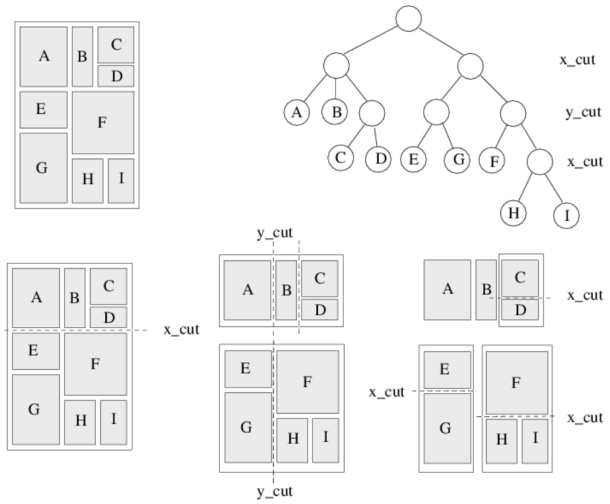


Figure 13: XY tree

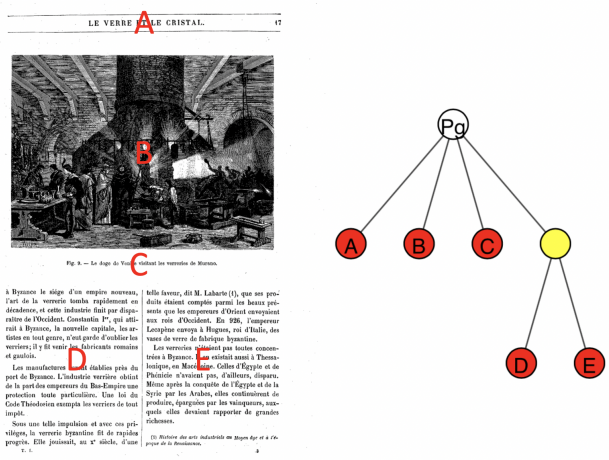


Figure 14: result of *cutMatrix()*

those nodes that are cutted further vertical.

4 PyTesseract

After applied all the preprocessing steps, and used one of the algorithms for the layout analysis, to check if the algorithms used previously worked correctly, we use OCR to recognize the characters of the document. To do this we use the package *pytesseract*, which after reading blocks of images belonging to the same document, located in a folder thanks to the previous methods, transcribes the text of the image thanks to the method `pytesseract.image_to_string(Image.open(file))` in a text file (*.txt*) dividing them by paragraph.
If the paragraph is an image it writes in the text file 'IMAGE'.

The OCR results show quite satisfactory results, however on some occasions there are some errors in the transcription, mainly due to the unsatisfactory quality of the scanned document.

Appendice A

In the next pages we can see a series of further results, obtained using other images and parameters. Furthermore, at the following link dropbox [5] it is possible to view in greater detail all the results obtained.

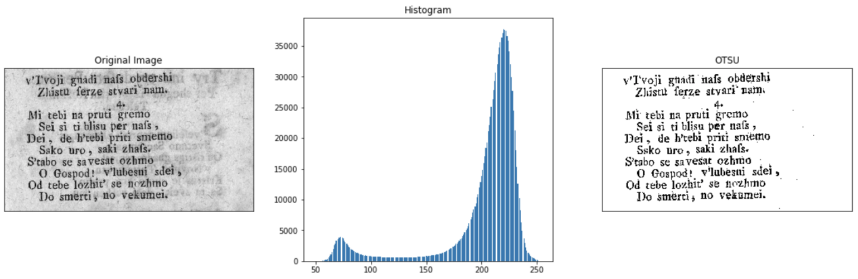


Figure 15: Otsu binarization with histogram of pixel distribution

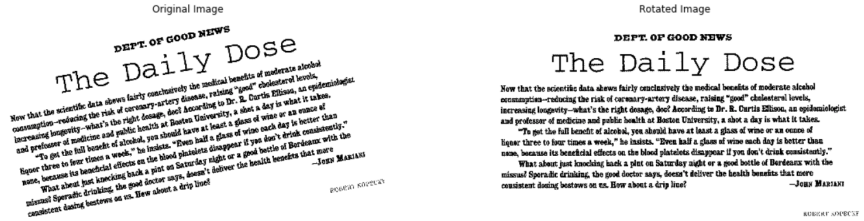


Figure 16: Image rotate of -9.80 degrees

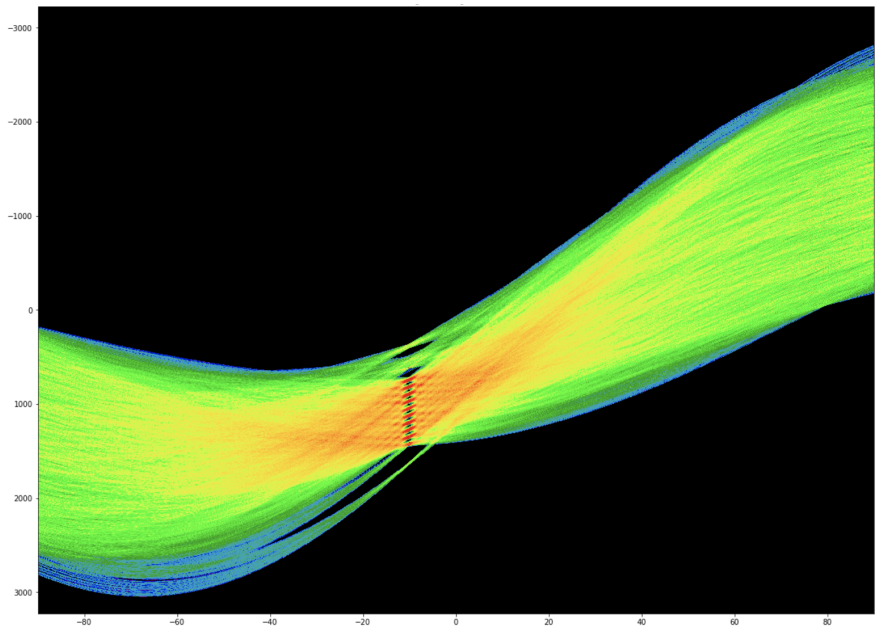


Figure 17: Hough trasform histogram of the image rotate of -9.80 degress

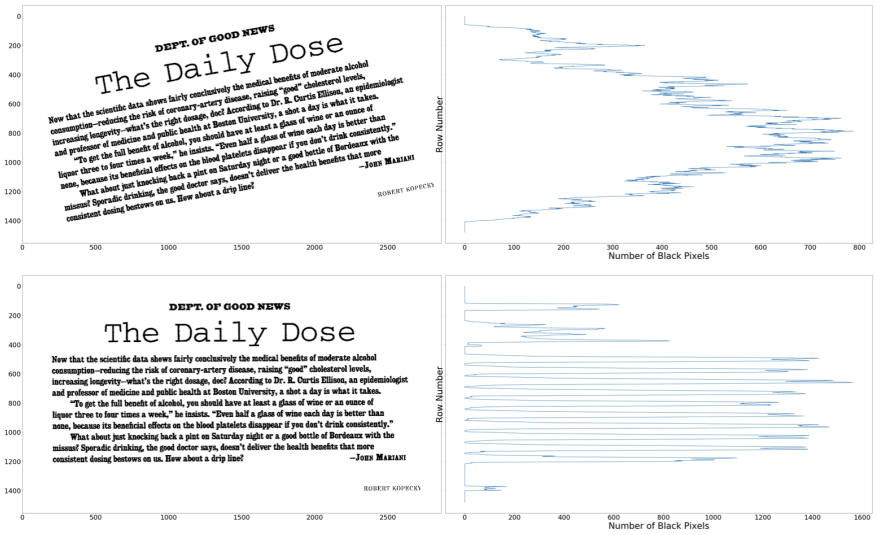


Figure 18: Projection before and after rotation

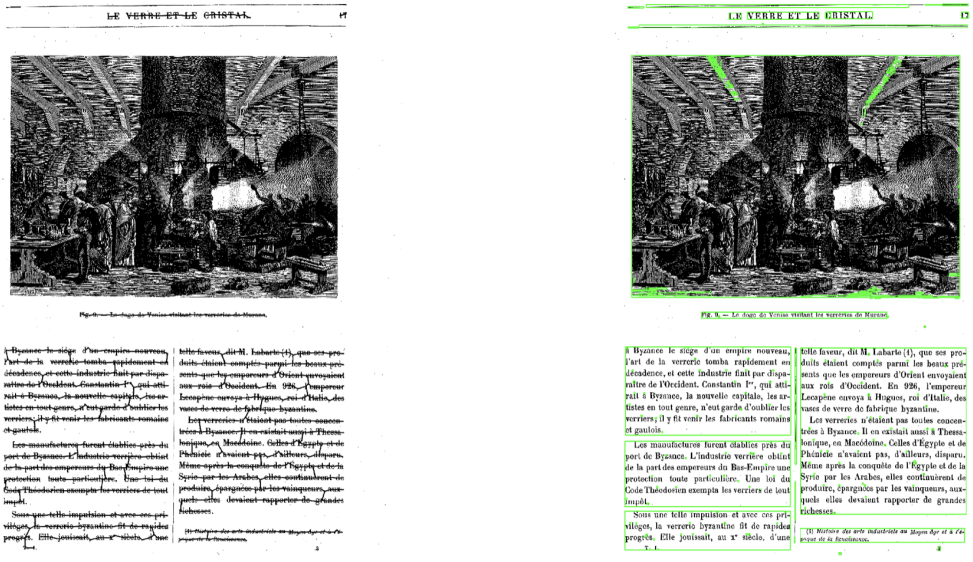


Figure 19: Paragraphs segmentation after MST

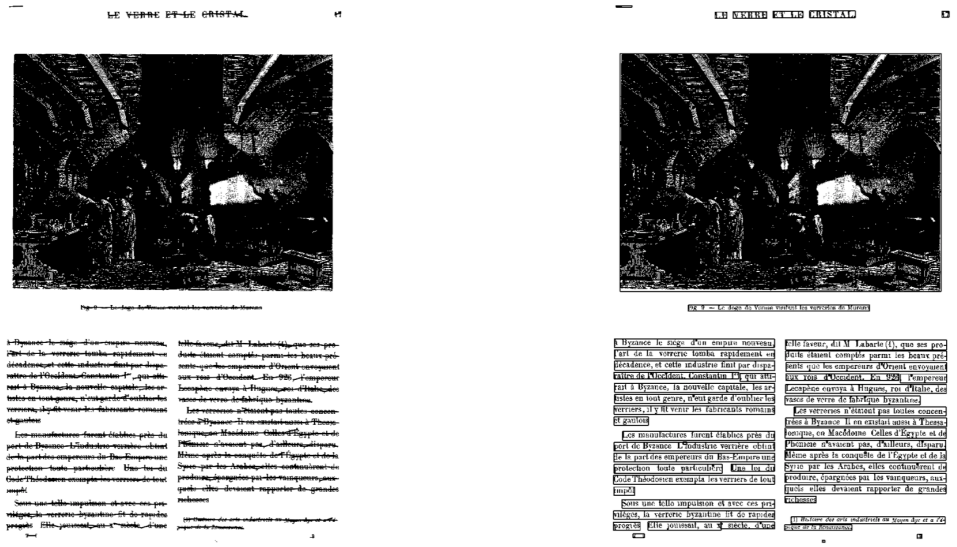


Figure 20: Line segmentation with Docstrum

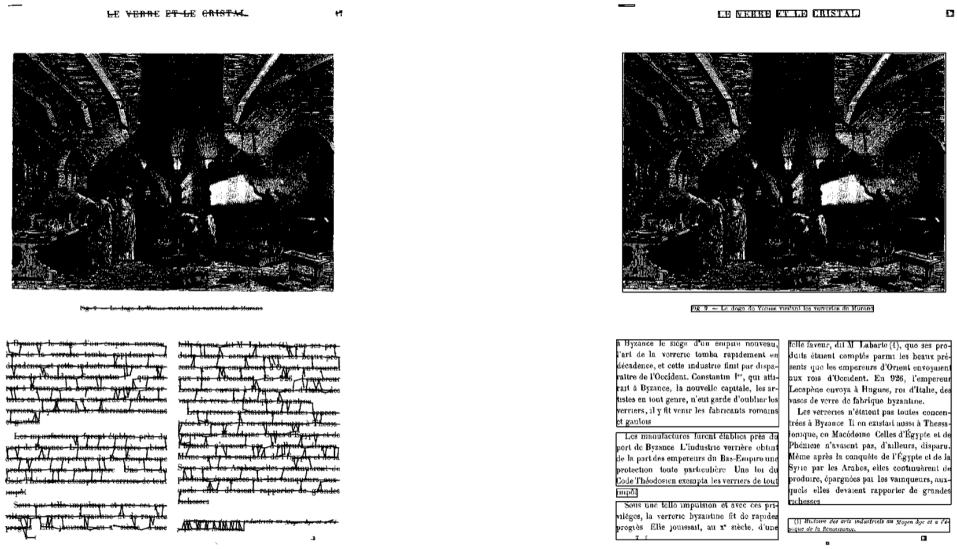


Figure 21: Paragraphs segmentation with Docstrum

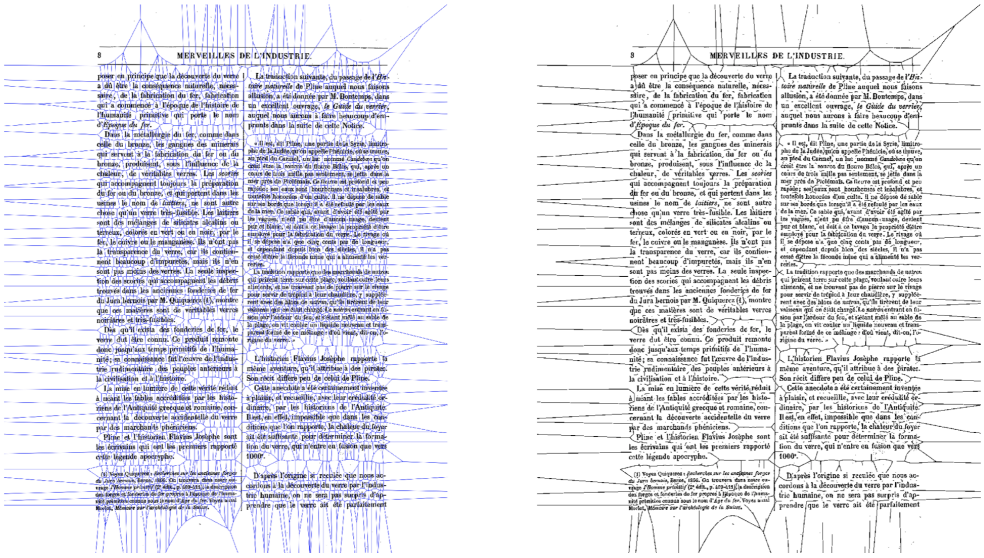
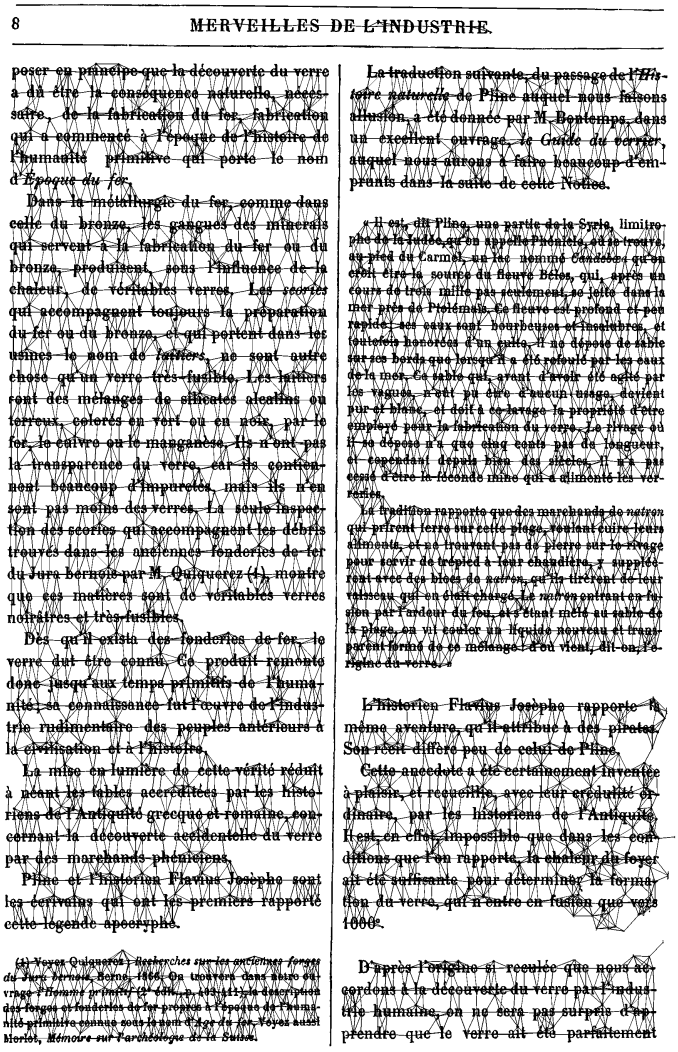


Figure 22: Voronoi Segmentation



8 MERVEILLES DE L'INDUSTRIE.

poser en principe que la découverte du verre a dû être la conséquence naturelle, nécessaire, de la fabrication du fer, fabrication qui a commencé à l'époque de l'histoire de l'humanité primitive qui porte le nom d'*Époque du fer*.

Dans la métallurgie du fer, comme dans celle du bronze, les gangues des minerais qui servent à la fabrication du fer ou du bronze, produisent, sous l'influence de la chaleur, de véritables verres. Les *scories* qui accompagnent toujours la préparation du fer ou du bronze, et qui portent dans les usines le nom de *laitiers*, ne sont autre chose qu'un verre très-fusible. Les laitiers sont des mélanges de silicates alcalins ou terreux, colorés en vert ou en noir, par le fer, le cuivre ou le manganèse. Ils n'ont pas la transparence du verre, car ils contiennent beaucoup d'impuretés, mais ils n'en sont pas moins des verres. La seule inspection des scories qui accompagnent les débris trouvés dans les anciennes fonderies de fer du Jura bernois par M. Quiquerez (1), montre que ces matières sont de véritables verres noirâtres et très fusibles.

Dès qu'il exista des fonderies de fer, le verre dut être connu. Ce produit remonte donc jusqu'aux temps primitifs de l'humanité; sa connaissance fut l'œuvre de l'industrie rudimentaire des peuples antérieurs à la civilisation et à l'histoire.

La mise en lumière de cette vérité réduit à néant les fables accréditées par les historiens de l'Antiquité grecque et romaine, concernant la découverte accidentelle du verre par des marchands phéniciens.

Pline et l'historien Flavius Josèphe sont les écrivains qui ont les premiers rapporté cette légende apocryphe.

(1) Voyez Quiquerez, *Recherches sur les anciennes forges du Jura bernois*, Berne, 1866. On trouvera dans notre ouvrage *l'Homme primitif* (2e vol., p. 171 et 172) la description des forges et fonderies de fer primitives et l'époque de l'humanité primitive connue sous le nom d'âge du fer. Voyez aussi Morlot, *Mémoire sur l'archéologie de la Suisse*.

La traduction suivante du passage de l'*Histoire naturelle* de Pline auquel nous faisons allusion, a été donnée par M. Bontemps, dans un excellent ouvrage, le *Guide du verrier*, auquel nous aurons à faire beaucoup d'emprunts dans la suite de cette notice.

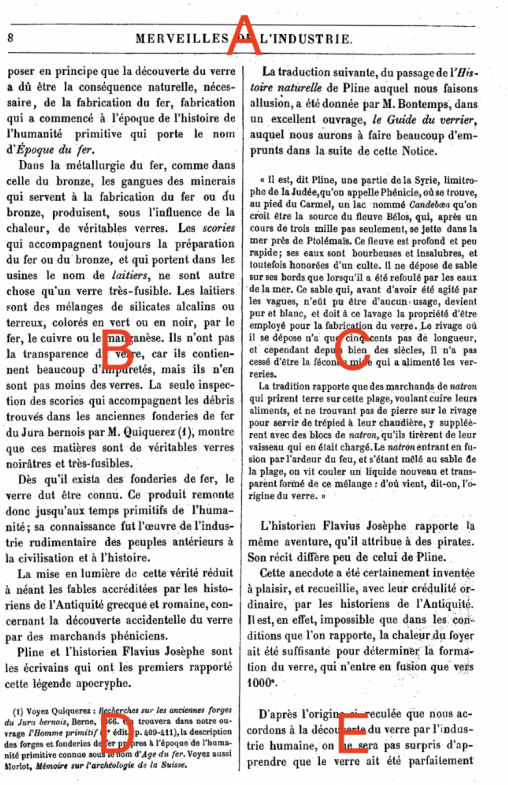
« Il est, dit Pline, une partie de la Syrie, limitrophe de la Judée, qu'on appelle Phénicie, où se trouve, au pied du Carmel, un lac nommé *Cendebia*, qu'on croit être la source du fleuve Belus, qui, après un cours de trois mille pas seulement, se jette dans la mer près de Ptolémaïs. Ce fleuve est profond et peu rapide; ses eaux sont bourbeuses et insalubres, et toutefois honorées d'un culte. Il dépose de sable sur ses bords que lorsqu'il a été refoulé par les eaux de la mer. Ce sable qui, avant d'avoir été agité par les vagues, n'est pu être d'aucun usage, devient pur et blanc, et doit à ce lavage la propriété d'être employé pour la fabrication du verre. Le rivage où il se dépose n'a que cinq cents pas de longueur, et cependant depuis bien des siècles, il n'a pas cessé d'être la féconde mine qui a alimenté les verreries.

La tradition rapporte que des marchands de *natron* qui prirent terre sur cette plage, voulant cuire leurs aliments, et ne trouvant pas de pierre sur le rivage pour servir de trépied à leur chaudière, y suppléèrent avec des blocs de *natron*, qu'ils tirèrent de leur vaisseau qui en était chargé. Le *natron* entrant en fusion par l'ardeur du feu, et s'étant mêlé au sable de la plage, on vit couler un liquide nouveau et transparent formé de ce mélange : d'où vient, dit-on, l'origine du verre. »

L'historien Flavius Josèphe rapporte la même aventure, qu'il attribue à des pirates. Son récit diffère peu de celui de Pline.

Cette anecdote a été certainement inventée à plaisir, et recueillie, avec leur crédulité ordinaire, par les historiens de l'Antiquité. Il est, en effet, impossible que dans les conditions que l'on rapporte, la chaleur du foyer ait été suffisante pour déterminer la formation du verre, qui n'entre en fusion que vers 1000°.

D'après l'origine si reculée que nous accordons à la découverte du verre par l'industrie humaine, on ne sera pas surpris d'apprendre que le verre ait été parfaitement



poser en principe que la découverte du verre a dû être la conséquence naturelle, nécessaire, de la fabrication du fer, fabrication qui a commencé à l'époque de l'histoire de l'humanité primitive qui porte le nom d'*Époque du fer*.

Dans la métallurgie du fer, comme dans celle du bronze, les gangues des minerais qui servent à la fabrication du fer ou du bronze, produisent, sous l'influence de la chaleur, de véritables verres. Les *scories* qui accompagnent toujours la préparation du fer ou du bronze, et qui portent dans les usines le nom de *laitiers*, ne sont autre chose qu'un verre très-fusible. Les laitiers sont des mélanges de silicates alcalins ou terreux, colorés en vert ou en noir, par le fer, le cuivre ou le manganèse. Ils n'ont pas la transparence du verre, car ils contiennent beaucoup d'impuretés, mais ils n'en sont pas moins des verres. La seule inspection des scories qui accompagnent les débris trouvés dans les anciennes fonderies de fer du Jura bernois par M. Quiquerez (1), montre que ces matières sont de véritables verres noirâtres et très-fusibles.

Dès qu'il exista des fonderies de fer, le verre dut être connu. Ce produit remonte donc jusqu'aux temps primitifs de l'humanité; sa connaissance fut l'œuvre de l'industrie rudimentaire des peuples antérieurs à la civilisation et à l'histoire.

La mise en lumière de cette vérité réduit à néant les fables accréditées par les historiens de l'Antiquité grecque et romaine, concernant la découverte accidentelle du verre par des marchands phéniciens.

Pline et l'historien Flavius Josèphe sont les écrivains qui ont les premiers rapporté cette légende apocryphe.

(1) Voyez Quiquerez : *Recherches sur les anciennes forges du Jura bernois*, Berne, 1868. Il se trouvera dans notre ouvrage *l'Homme primitif* (2e édit.) p. 409-411), la description des forges et fonderies de fer primitives à l'époque de l'humanité primitive connue sous le nom d'*Age du fer*. Voyez aussi Morlot, *Mémoire sur l'archéologie de la Suisse*.

La traduction suivante, du passage de l'*Histoire naturelle* de Pline auquel nous faisons allusion, a été donnée par M. Bontemps, dans un excellent ouvrage, *le Guide du verrier*, auquel nous aurons à faire beaucoup d'emprunts dans la suite de cette Notice.

« Il est, dit Pline, une partie de la Syrie, limitrophe de la Judée, qu'on appelle Phénicie, où se trouve, au pied du Carmel, un lac nommé *Candebæa* qu'on croit être la source du fleuve Bélos, qui, après un cours de trois mille pas seulement, se jette dans la mer près de Ptolémaïs. Ce fleuve est profond et peu rapide; ses eaux sont bourbeuses et insalubres, et toutefois honorées d'un culte. Il ne dépose de sable sur ses bords que lorsqu'il a été refoulé par les eaux de la mer. Ce sable qui, avant d'avoir été agité par les vagues, n'eût pu être d'aucun usage, devient pur et blanc, et doit à ce lavage la propriété d'être employé pour la fabrication du verre. Le rivage où il se dépose n'a que cinq cents pas de longueur, et cependant depuis bien des siècles, il n'a pas cessé d'être la féconde mine qui a alimenté les verreries.

La tradition rapporte que des marchands de *natron* qui prirent terre sur cette plage, voulant cuire leurs aliments, et ne trouvant pas de pierre sur le rivage pour servir de trépied à leur chaudière, y suppléèrent avec des blocs de *natron*, qu'ils tirèrent de leur vaisseau qui en était chargé. Le *natron* entrant en fusion par l'ardeur du feu, et s'étant mêlé au sable de la plage, on vit couler un liquide nouveau et transparent formé de ce mélange : d'où vient, dit-on, l'origine du verre. »

L'historien Flavius Josèphe rapporte la même aventure, qu'il attribue à des pirates. Son récit diffère peu de celui de Pline.

Cette anecdote a été certainement inventée à plaisir, et recueillie, avec leur crédulité ordinaire, par les historiens de l'Antiquité. Il est, en effet, impossible que dans les conditions que l'on rapporte, la chaleur du foyer ait été suffisante pour déterminer la formation du verre, qui n'entre en fusion que vers 1000°.

D'après l'origine si reculée que nous accordons à la découverte du verre par l'industrie humaine, on ne sera pas surpris d'apprendre que le verre ait été parfaitement

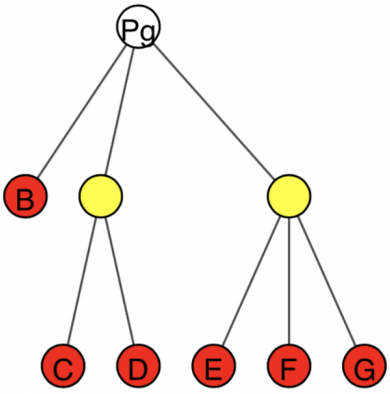


Figure 24: result of *cutMatrix()* for image with four paragraphs

References

[1] GitHub Repository of the project code https://github.com/loredeluca/Document-Recognition.

[2] Vasista Reddy Repository https://github.com/Vasistareddy/python-rlsa.

[3] L. O'Gorman, The document spectrum for page layout analysis, IEEE https://ieeexplore.ieee.org/document/244677.

[4] Koichi Kise, Akinori Sato, and Motoi Iwata - Segmentation of Page Images Using the Area Voronoi Diagramy https://www.sciencedirect.com/science/article/pii/S1077314298906841.

[5] Dropbox folder with results https://www.dropbox.com/sh/y9m144ninc0sz0z/AABh0_Bj1FZO7amEjyiRMgiwa?dl=0.