École Polytechnique Fédérale de Lausanne

Distant seeing
Applying Machine Vision Algorithms to Historical Scientific Images

by Julien Mettler

Bachelor Project Report

Laboratory for the History of Science and Technology (LHST)

Project supervisors
Prof. Jérôme Baudry
Prof. Pascal Fua (CVLAB)
Ion-Gabriel Mihailescu
Semion Sidorenko

EPFL CDH DHI LHST
INN 116 (Bâtiment INN)
Station 14
CH-1015 Lausanne

November 8, 2023

Contents

1 Introduction

With the growth of digital libraries in the past decades, millions of historical data became available in a few clicks to the historians and digital humanists for use in their studies. In particular, this can ease the researches whose aim is to better understand the history of science and technology in a given country, or over some period of time.

However, as the data available is mostly unstructured, as a researcher, building datasets for one specific goal can be time-consuming. In parallel, the flourishing development of computer vision techniques and image classification algorithms has defined more convenient and faster ways to classify and organize images.

In the context of this project, we explored ways in which we could apply techniques from the above two fields to automate the processing and analysis of historical images from scientific textbooks, that date back to the second half of the 19th century. We started from the observation that these data, whether they focus on mechanics, optics, electromagnetism or meteorology, are well suited for the tasks of classification.

Indeed, pages in such literacy are structured around two main elements: text and figures. The latter provide the reader with a visual explanation of the phenomenon described in the text, or a description of the experiment conducted and the empirical observation of its results. The experiment or phenomenon can be depicted in two dimensions or three dimensions, depending of which representation is deemed more suitable for the reader to visualize the concept at stake.

We will first present a processing scheme that automates the extraction of figures in historical scans and evaluate its performance. Then, we will derive a machine learner to classify the constructed dataset into specific categories of figures. The final part will investigate the historical knowledge which can be learnt from our results.

2 Overview of the dataset

As part of this project, we focused our analysis on a set of physics textbooks that were published in France in the second half of the nineteenth century.

Below is shown the information for every textbook that we considered. The links to their original scan can be found by clicking on the title. For conciseness, we will refer in the further sections to a scan by a `codename`, that we assigned to each of them.

- **Pierre-Adolphe Daguin**, *Traité élémentaire de physique théorique et expérimentale*
  - Tome Premier (1855) `daguin_t1`
  - Tome Second (1856) `daguin_t2`
  - Tome Troisième (1860) `daguin_t3`
- **Nicolas Deguin**, *Cours élémentaire de physique*
  - Neuvième Édition, Tome 1 (1854) `deguin_ed7`
  - Précis de Physique (1869) `deguin_ed9`
- **Paul Desains**, *Leçons de physique*
  - Tome Premier (1857)
    - First scan `desains_t1s1`
    - Third scan `desains_t1s3`
  - Tome Second (1865)
    - First scan `desains_t2s1`
    - Second scan `desains_t2s2`
    - Third scan `desains_t2s3`
    - Fourth scan `desains_t2s4`
- **Adolphe Ganot**, *Traité élémentaire de physique expérimentale et appliquée et de météorologie*
  - Treizième Édition (1868) `ganot_ed13`
  - Dix-Septième Édition (1870) `ganot_ed17`
- **Jules Jamin**, *Cours de physique de l'École Polytechnique*
  - Tome Troisième (1866) `jamin_t3`
  - Troisième Édition, Tome Troisième (1879) `jamin_ed3t3`
  - Troisième Édition, Tome Quatrième (1883) `jamin_ed3t4`

- Quatrième Édition, Tome Quatrième (2e partie) 4e Fascicule (1891) `jamin_t2p2f4`
- Premier Supplément (1896) `jamin_s1`
- Deuxième Supplément (1899) `jamin_s2`
- Troisième Supplément (1906) `jamin_s3`

The sources of these scans were mainly found through **Europeana**, an online aggregator that redirects to links towards the hosting institutions.

The scans were mainly taken from two freely available sources:

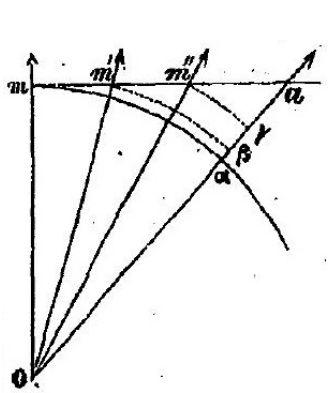
1. **Gallica**, the digital library of the *Bibliothèque nationale de France* (National Library of France), and
2. **Google Books**, the aggregator service which is maintained by Google

As can be observed in the list above, most of the books were published in multiple editions and volumes, and some of them are scanned multiple times. As a result, the dataset contains multiple figures which reappear throughout the editions and volumes of a certain books. It is the case for instance of `desains_t1` and `desains_t2`. These scans may vary in quality, and some as more degraded than others, which will prove useful to test the performance of our extraction algorithm in 3.2.

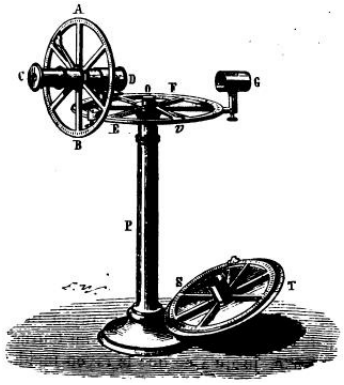
2.1 Assumptions on the data

For this project, we aimed at classifying the extracted figures along three axes:

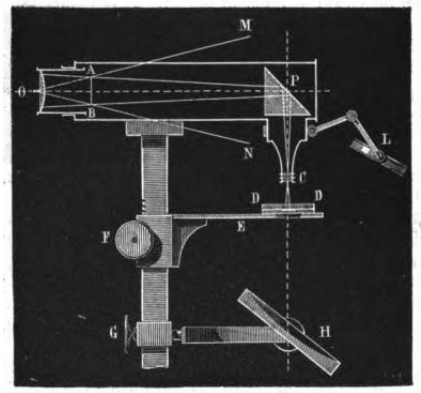
1. **Diagram**. We assume they refer generally to flat and sparse figures, which are structured around thin black lines that may be solid, dashed or dotted, and which are captioned with letters, arrows and other indication symbols. They represent graphs or polygons, but we also consider cartographic maps as diagrams, since they fulfill the aforementioned requirements. An example of a diagram is depicted on the left of the figure below.
2. **Instrument**. On the opposite, instruments represent solid and dense objects. They are generally colored with dark grey or black, and their drawing adds light reflections and shadowing effects on their edges to convey their density and weight, as we see at the center of the figure below. Note that we also consider any solid object in three-dimensions as an instrument.
3. **Black figure**. Such images have a more straightforward structure. They represent figures, like diagrams, that are drawn in a black rectangle. An example is shown at the right below.



(a) A diagram



(b) An instrument



(c) A black figure

For clarity, we displayed above only the "good representatives" of their respective class, but will see in 5 that the dataset constructed will contain many figures whose structure is more ambiguous and requires further analysis.

Based on our assumptions, we will now describe the processing scheme that was applied to each page of the scan in order to extract the figures from them.

3 Image extraction

For the implementation, we will use **OpenCV (Open-source Computer Vision)** library, that implements various image processing techniques. The scheme is implemented in a method `def extract_images_from(filename)`.

We first use it to read the image:

```
def extract_images_from(filename):
    # Load image
    img = cv2.imread(filename)
    img_orig = copy.copy(img)
```

Let us now analyze in detail each of the steps that compose the extracting pipeline. We assume here that we do not apply this scheme on the pages that contain only text.

3.1 Processing pipeline

3.1.1 Grayscaling

The first transformation consists in grayscaling the image. The main reason for this is that the thresholding method in the next step requires a grayscale image as input. It is also a way of normalizing the scans before processing, as it transforms each image color space to the gray color space.

To implement it, we call the `cv2.cvtColor(src, code)` method of OpenCV that applies a conversion from the image color space to another color space. The nature of the conversion is determined by the second parameter.

```
# Grayscale image
img_gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
```

Since we want to convert our original image from the RGB color space (Red/Green/Blue) to the gray space, we use the conversion code `COLOR_BGR2GRAY`.

Concretely, the conversion transforms the image of shape `(height, width, 3)` that has three channels (three pixel values for the Red, Green, Blue colors in the range $[0, 255]$) to an image of shape `(height, width)` that assigns to each pixel only one channel, of color value

$$Y = 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B \in [0, 255]$$

Note that since most of the scans contain dark text and figures on a light background, they are already in the gray color space. In particular, the pixels of the text, figures and of the background generally all have the same value for their three channels $R = G = B$. Hence the conversion $Y = R = G = B$ does not change the value of the pixels. An example is shown on Figure 1. If we zoom in the ball at the top of the jar, we observe that the pixels all have the same value for RGB.

On the other hand, the conversion does make a change for brown pages (Figure 2). We indeed observe that the pixels of a stain contain different values on the three channels. They are thus colored in gray by the transformation.

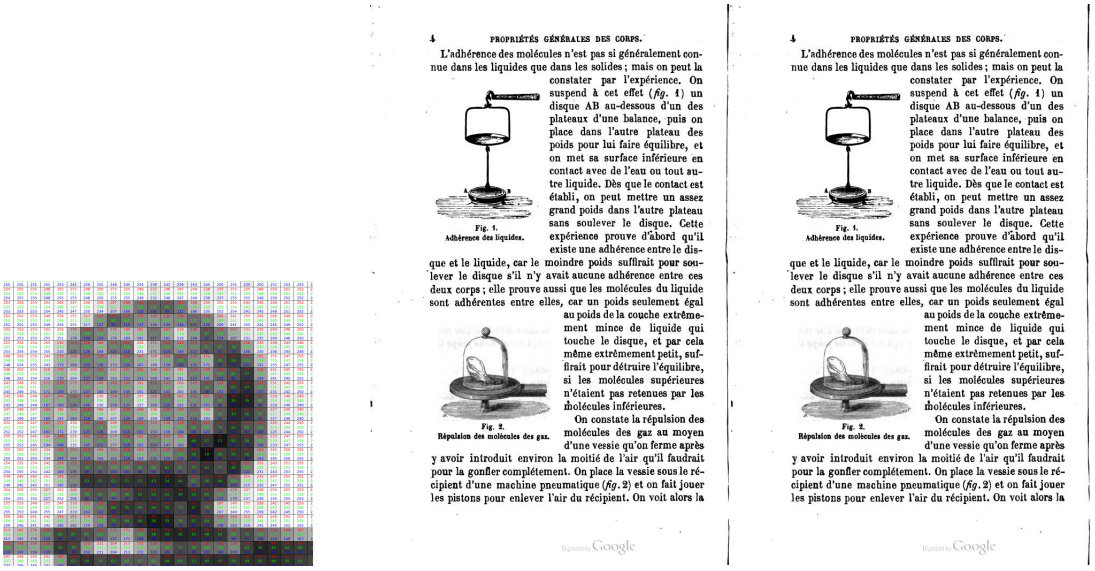


Figure 1: Effect of grayscaling on a "black and white" scan

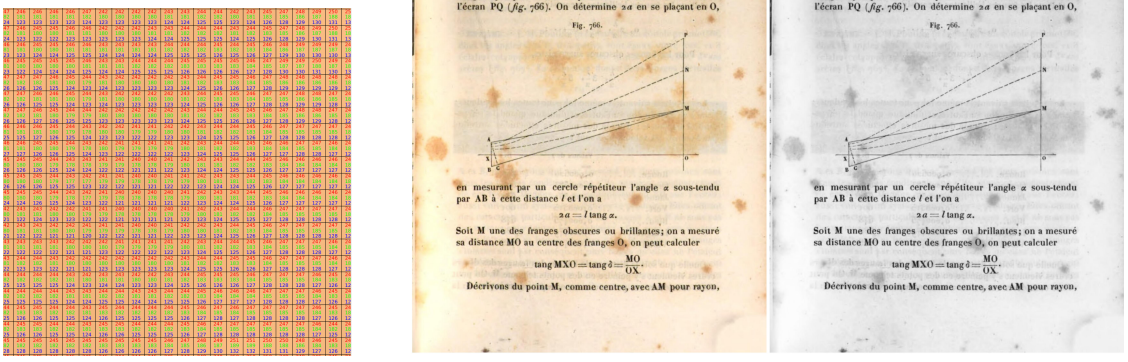


Figure 2: Effect of grayscaling on a brown scan

3.1.2 Binarization

Based on the single-channel image that we obtained at the previous step, we **binarize** the image, i.e., we assign to each pixel a color that can only take two values.

In our case, the contouring method described in the next section finds contours of a white object from a black blackground. In other words, it requires as input a binary image in which foreground (text and figures) and background pixels are labeled 255 (white) and 0 (black), respectively.

To achieve this, we can employ a technique called **thresholding**: it defines a threshold value such that for each pixel, if the value of its channel exceeds the threshold, it is assigned to one fixed value, and if it is smaller, it is assigned to a fixed maximum value. In our case, the first value will hence be 0 and the second value will be 255.

The point of having a binary image is that it allows to clearly segment the image into its core structural elements, represented by the foreground (text and figures) and the background. It also creates a simplified representation of the image, that will ease the further processing steps, as we will see below.

OpenCV allows to perform this segmentation with the method `cv2.threshold(src, thresh, maxval, type)`:

- `src` represents the source image, which must be grayscaled
- `thresh` represents the threshold value that separates the pixels into foreground and background.
- `maxval` represents the maximum value
- `type` represents the type of thresholding

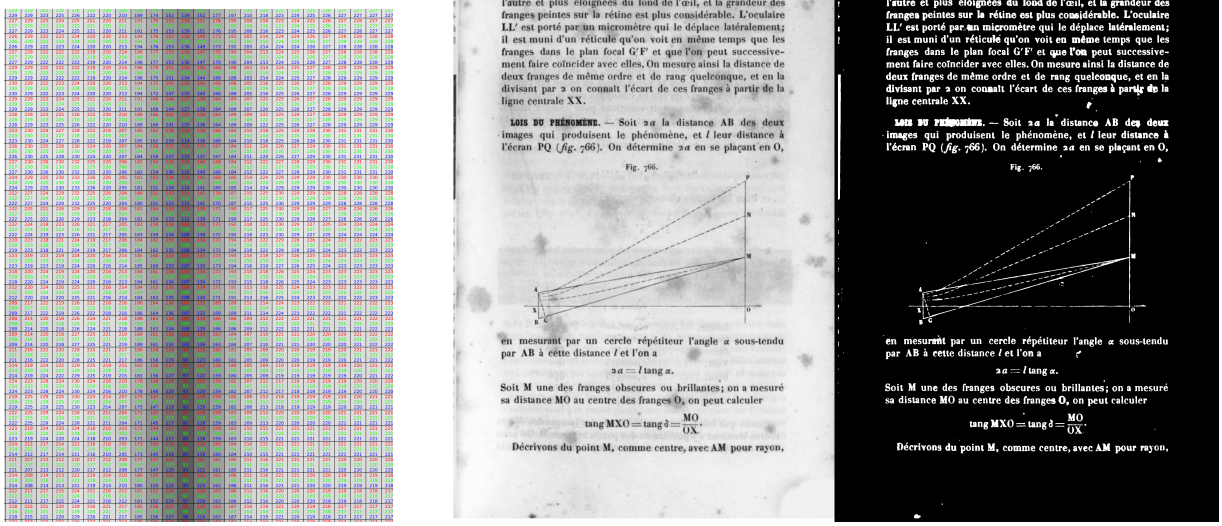
The method returns the threshold value and the thresholded image.

Based on our requirements, we used the **inverse binary thresholding** (`cv2.THRESH_BINARY_INV`), that assigns to each pixel $\texttt{src}(x, y)$ at coordinates (x, y) the value $\texttt{dest}(x, y) = 0$ if $\texttt{src}(x, y) > \texttt{thresh}$, or `maxval` otherwise, which in our case is `255`:

```
# Binary image
thr, img_thresh = cv2.threshold(img_gray, 0, 255, cv2.THRESH_BINARY_INV + cv2.THRESH_OTSU)
```

Furthermore, we passed the extra flag `cv2.THRESH_OTSU` that performs **Otsu's binarization**. This technique finds automatically an optimal threshold value for the image representing the page, thus avoiding us to choose an arbitrary value. We can then put any value for the parameter `thresh`, since it will not be used by the function. In our case, we simply put `0`.

Some results are shown on Figure 3 and 4. On the figure below, 163 was found as the optimal threshold value. If we zoom in to the pixels on the left of the diagram, we observe that the values at the edges are slightly below this threshold. Thus, Otsu's algorithm seems to have adjusted the value to assign the color white to every pixel that is darker than these edges (for instance the text), and the color black to every pixel that is lighter than these edges (for instance the stains).



DES INTERFÉRENCES ET DE LA DIFFRACTION. 521

distance de la vision distincte, vise sur un autre plan F'G' plus rapproché. Les deux images A', B' sont plus voisines l'une de l'autre et plus éloignées du fond de l'œil, et la grandeur des franges peintes sur la rétine est plus considérable. L'oculaire LL' est porté par un micromètre qui le déplace latéralement; il est muni d'un réticule qu'on voit en même temps que les franges dans le plan focal G'F' et que l'on peut successivement faire coïncider avec elles. On mesure ainsi la distance de deux franges de même ordre et de rang quelconque, et en la divisant par 2 on connaît l'écart de ces franges à partir de la ligne centrale XX.

LOIS DU PHÉNOMÈNE. — Soit $2a$ la distance AB des deux images qui produisent le phénomène, et l leur distance à l'écran PQ (*fig.* 766). On détermine $2a$ en se plaçant en O,

Fig. 766.

en mesurant par un cercle répétiteur l'angle α sous-tendu par AB à cette distance l et l'on a

$$2a = l \operatorname{tang} \alpha.$$

Soit M une des franges obscures ou brillantes; on a mesuré sa distance MO au centre des franges O, on peut calculer

$$\operatorname{tang} MXO = \operatorname{tang} \delta = \frac{MO}{OX}.$$

Décrivons du point M, comme centre, avec AM pour rayon,

DES INTERFÉRENCES ET DE LA DIFFRACTION. 521

distance de la vision distincte, vise sur un autre plan F'G' plus rapproché. Les deux images A', B' sont plus voisines l'une de l'autre et plus éloignées du fond de l'œil, et la grandeur des franges peintes sur la rétine est plus considérable. L'oculaire LL' est porté par un micromètre qui le déplace latéralement; il est muni d'un réticule qu'on voit en même temps que les franges dans le plan focal G'F' et que l'on peut successivement faire coïncider avec elles. On mesure ainsi la distance de deux franges de même ordre et de rang quelconque, et en la divisant par 2 on connaît l'écart de ces franges à partir de la ligne centrale XX.

LOIS DU PHÉNOMÈNE. — Soit $2a$ la distance AB des deux images qui produisent le phénomène, et l leur distance à l'écran PQ (*fig.* 766). On détermine $2a$ en se plaçant en O,

Fig. 766.

en mesurant par un cercle répétiteur l'angle α sous-tendu par AB à cette distance l et l'on a

$$2a = l \operatorname{tang} \alpha.$$

Soit M une des franges obscures ou brillantes; on a mesuré sa distance MO au centre des franges O, on peut calculer

$$\operatorname{tang} MXO = \operatorname{tang} \delta = \frac{MO}{OX}.$$

Décrivons du point M, comme centre, avec AM pour rayon,

Figure 3: Stain removal

This last observation also allows us to show that binarization may remove some artifacts that appeared during the scan, like stains, shadows or light reflections. For instance, the second figure shows that the "shadow" of the previous pages is removed after binarization.

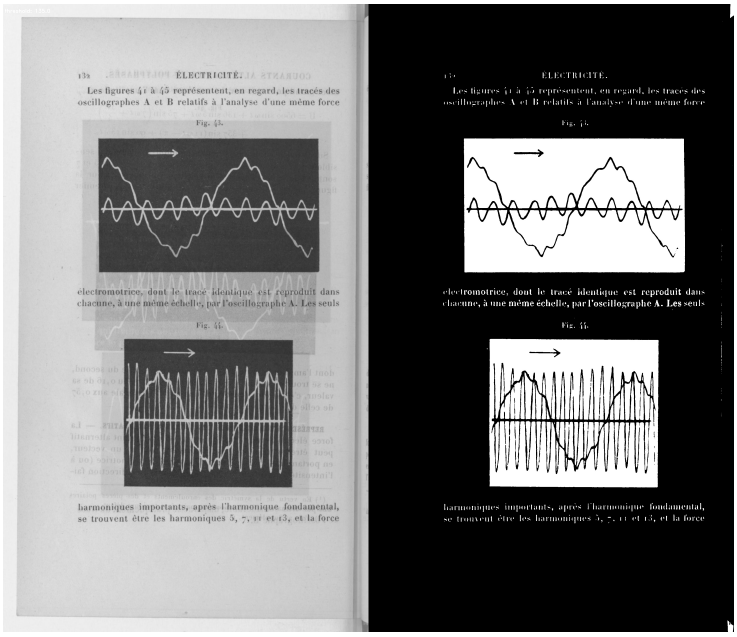


Figure 4: Shadow removal

Hence, grayscaling and binarization allow us to standardize the format of the scans before they are being processed, by converting them to the same color space, and cleaning them from the scan artifacts.

We will now try to determine if the resulting segmented image describes a suitable form for the main step of the pipeline: the countouring step, that defines how the extracted images are being cropped.

3.1.3 Finding contours

For this task, we relied on the method `findContours(image, mode, method)` that is provided by OpenCV.

- `image` represents the source image
- `mode` represents the **contour retrieval mode**
- `method` represents the **contour approximation method**

This method retrieves all the possible contours that it can find in `image`. There can be many ways one can find contours in an image, since contours can be nested in another contours, which creates a hierarchy of contours.

In our case, a diagram, an instrument and a black image all have a shape whose structure is composed of multiple parts, but we were only interested in the whole figure. We thus needed

to consider only the extreme outer contours. We indicated this to the method by assigning `mode=cv2.RETR_EXTERNAL`.

Regarding the contour approximation method, OpenCV proposes for each contour that it finds, either to store all the contour points (`method=CHAIN_APPROX_NONE`) or to store only the points at the corners (`method=CHAIN_APPROX_SIMPLE`). For simplicity, we chose the second method.

We thus have the following:

```
cnts = cv2.findContours(img_thresh, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
cnts = cnts[0] if len(cnts) == 2 else cnts[1]
```

The `findContours` method returns a tuple `cnts` that has contains as a first element a list of all the contours in the image, each one being represented by an array of the corner points (x, y) coordinates. The second element, that will not be of interest for us, returns the hierarchy of these contours. Each array `c` in the tuple defines in fact a bounding box, which we can display for informational purposes. In particular, we used its coordinates in the page to extract a **region of interest (ROI)** and save it as an image:

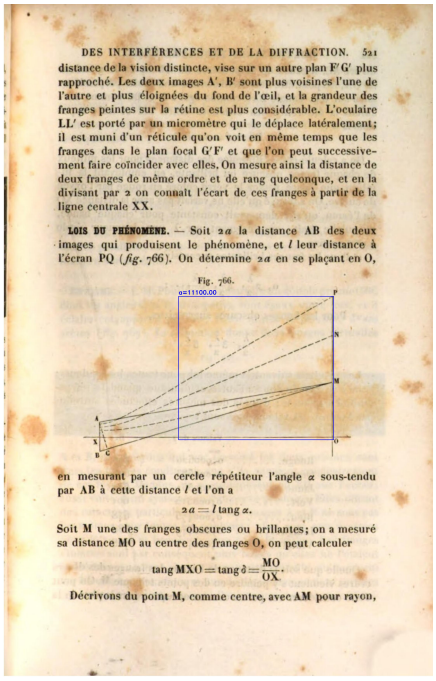
```
COLOR_BLUE = (255, 0, 0)
OUTPUT_DIR = 'images'

# Find contours, obtain bounding box, extract and save region of interest (ROI)
roi_number = 1
cnts = cv2.findContours(img_thresh, mode=cv2.RETR_EXTERNAL, method=cv2.CHAIN_APPROX_SIMPLE)
cnts = cnts[0] if len(cnts) == 2 else cnts[1]
for c in cnts:
    x, y, w, h = cv2.boundingRect(c) # Coordinates, width, height of ROI
    area = cv2.contourArea(c) # Area of ROI found

    if area > 10000:
        ROI = img_orig[y:y+h, x:x+w]
        cv2.rectangle(img, (x, y), (x+w, y+h), color=COLOR_BLUE, thickness=2)
        cv2.putText(img, text=f"a={int(area):.2f}", org=(x, y-5),
                    fontFace=cv2.FONT_HERSHEY_DUPLEX, fontScale=1, color=COLOR_BLUE,
                    thickness=2)

        # Write ROI
        path_roi = os.path.join(OUTPUT_DIR, f"{Path(filename).stem}_{roi_number}.jpg")
        cv2.imwrite(path_roi, ROI)
        roi_number += 1
```

The method `boundingRect(c)` retrieves the x- and y-coordinates of the top-left corner point of the ROI, along with the width and height:

We moreover focused only on bounding boxes whose contour area (retrieved through the method of the same name) exceeds a certain threshold. This relies on the observation that the `findContours` method detects a bounding box for every letter of the text and also for small remaining artifacts like grain dots (with a relatively small area compared to the figures), so they must not be taken into account.

For the remaining few bounding boxes that were not filtered out by the threshold, we used their coordinates to draw the ROI on the original image and display its area, using the methods `rectangle` and `putText`. More importantly, we save the ROI as a separate image file on the disk, in the `'images'` folder.

As shown on Figure 5, this method fails to appropriately catch the entire diagram, as some edges are cut from the ROI. We also observe that the letters at the edges of the diagram are not caught neither by the method:


DES INTERFÉRENCES ET DE LA DIFFRACTION. 521

distance de la vision distincte, vise sur un autre plan F′G′ plus rapproché. Les deux images A′, B′ sont plus voisines l'une de l'autre et plus éloignées du fond de l'œil, et la grandeur des franges peintes sur la rétine est plus considérable. L'oculaire LL′ est porté par un micromètre qui le déplace latéralement; il est muni d'un réticule qu'on voit en même temps que les franges dans le plan focal G′F′ et que l'on peut successivement faire coïncider avec elles. On mesure ainsi la distance de deux franges de même ordre et de rang quelconque, et en la divisant par 2 on connaît l'écart de ces franges à partir de la ligne centrale XX.

LOIS DU PHÉNOMÈNE. — Soit $2a$ la distance AB des deux images qui produisent le phénomène, et l leur distance à l'écran PQ (*fig.* 766). On détermine $2a$ en se plaçant en O,

Fig. 766.

en mesurant par un cercle répétiteur l'angle α sous-tendu par AB à cette distance l et l'on a

$$2a = l \operatorname{tang} \alpha.$$

Soit M une des franges obscures ou brillantes; on a mesuré sa distance MO au centre des franges O, on peut calculer

$$\operatorname{tang} MXO = \operatorname{tang} \delta = \frac{MO}{OX}.$$

Décrivons du point M, comme centre, avec AM pour rayon,

Figure 5: The dotted lines and caption letters are cropped

If we zoom in (Figure 6), we indeed observe holes in these two locations. Such discontinuities break the structure of the diagram, whose boundary is then found much smaller by the contouring method.

To solve this problem, we needed to find a way to fill the holes and join the disparate borders before actually finding the contours, in order to create a continuity.

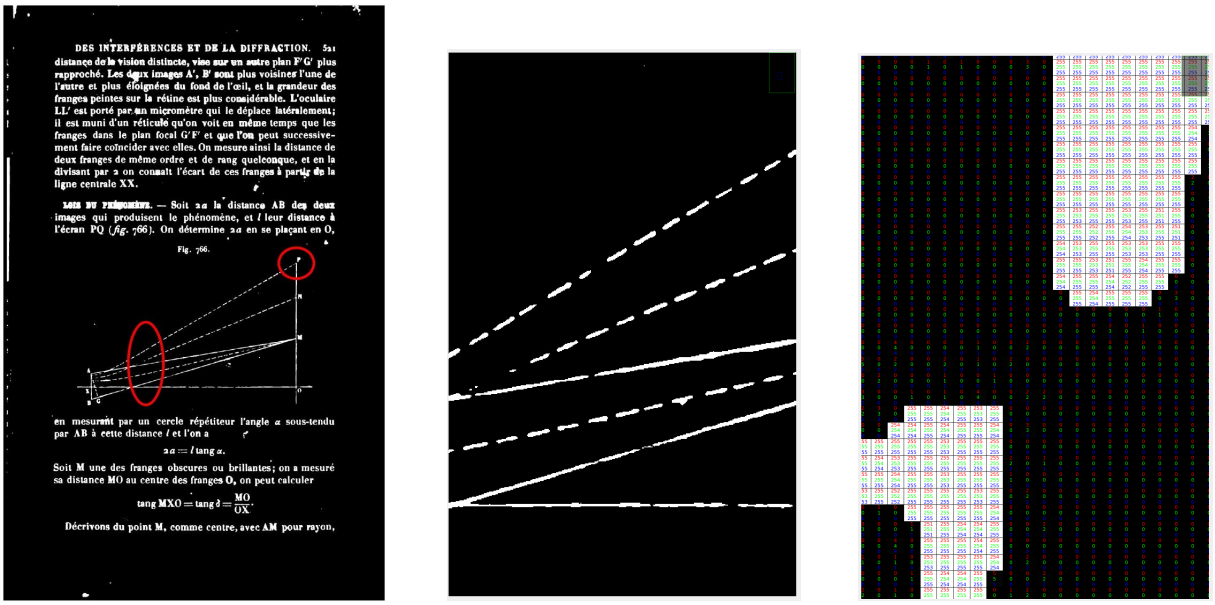


Figure 6: Holes in the diagram's structure

3.1.4 Dilation

To achieve this, we made use of **dilation**, an operation that has the effect of dilating the lighter regions of an image, by scanning it entirely with an element called the **kernel**. The latter takes the form of an $n \times n$ matrix that, when centered on a pixel $\texttt{src}(x, y)$ at coordinates (x, y), replaces its value by the maximal value among its $(n \times n) - 1$ neighbouring pixels in the matrix.

In our inverse-binary image, the pixels at the edges of diagrams take the value 255 while the background pixels, in particular in the hole, take the value 0. By applying the kernel on the image, new white pixels will then be added around the edges, which become wider, but also in the hole, as these black pixels will take the value 255 from the pixels at the borders of the hole. If the kernel is sufficiently large, the hole can be completely clogged.

If we take our previous example again, dilation clogs indeed the hole between the diagram and the letter P, as we see on Figure 7.

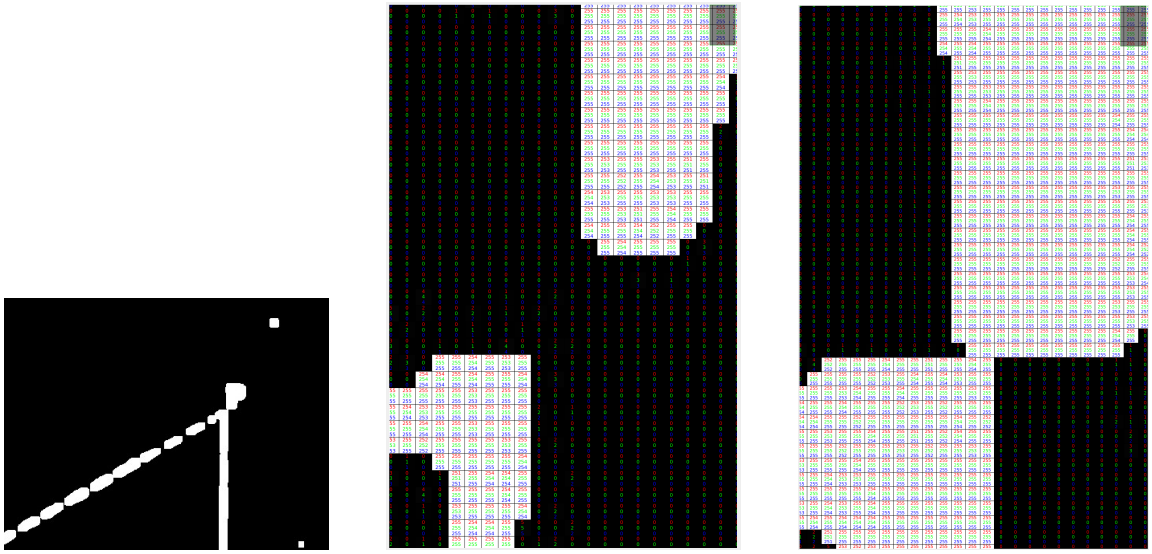


Figure 7: The hole between the diagram and the letter P is clogged after dilation

To add this step into our pipeline, we used the method `cv2.dilate(src, kernel)` of OpenCV, which implements dilation for us. It takes as arguments the source image and a kernel. We apply the dilation in the method `dilate_image`:

```
def dilate_image(img, dilation_size, dilation_shape, iterations):
    kernel = cv2.getStructuringElement(
        shape=dilation_shape,
        ksize=(2*dilation_size+1, 2*dilation_size+1),
        anchor=(dilation_size, dilation_size))
    return cv2.dilate(img, kernel, iterations)

# Dilated image
img_dilate = dilate_image(img_thresh, 9, cv2.MORPH_RECT, 1)
```

To define the kernel, we used the method `cv2.getStructuringElement(shape, ksize, anchor)`. We chose rectangular kernels (`cv2.MORPH_RECT`), that are just matrices filled with ones. For instance, a 3×3 rectangular kernel has the form:

```
array([[1, 1, 1],
       [1, 1, 1],
       [1, 1, 1]])
```

We then give the size of the kernel side; its size must be odd, to keep the kernel symmetric around its center point. We put the anchor point at the coordinates of the center, as a last argument.

The `cv2.dilate` method also allows to perform multiple scans with the kernel by varying the number of `iterations`.

With this value and the kernel size, we have two additional hyperparameters that we can vary, depending on how much a figure needs to be dilated. Some diagrams for instance that are particularly degraded would need more iterations and/or a bigger kernel to reconstruct its broken edges.

The results of dilation on the previous diagram are shown on Figure 8. Dilation has an effect of glueing pieces together, which is particularly useful to associate the caption letters to their diagram. As a result, diagrams are transformed into compact structures with well-defined borders, that are more suitable for contouring methods.

However, this transformation adds other difficulties to the extraction protocol.

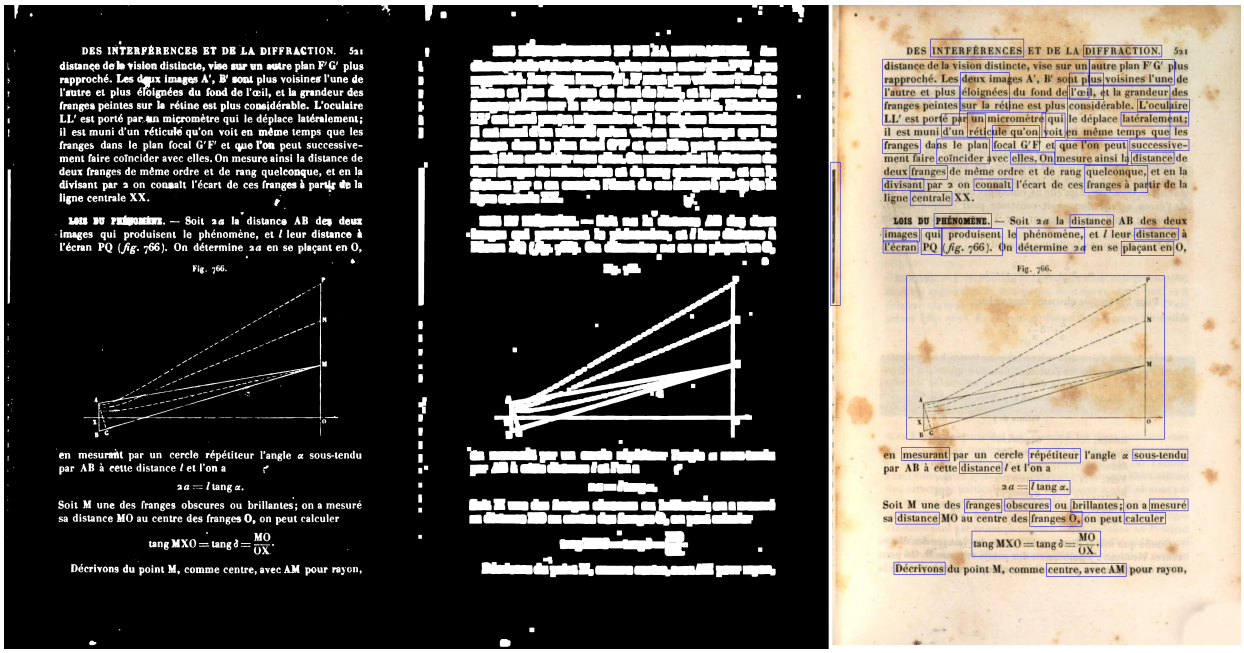


Figure 8: Dilated diagram and paragraphs which are close to each other are glued together

Limitations of dilation As illustrated above, a major problem we face is that dilation also groups letters and words together into chunks, which are detected as valid bounding boxes if their area is sufficiently high. As a result, many useless regions of interest are extracted and their corresponding images must be cleaned afterwards.

Furthermore, for scans of relative bad quality, the edges of the diagrams can be so degraded that they require a bigger kernel size and/or more iterations to fill the holes. This may become time-consuming to vary the parameters for these special cases, and breaks the genericity of the pipeline that we are looking for.

A trade-off hence appeared with dilation: if we increase the kernel size and/or number of iterations, it is more likely that diagrams will be cropped correctly, but on the other hand, larger groups of text will be detected, and sometimes these groups are glued to the diagram above or below it, as we can see on Figure 9.

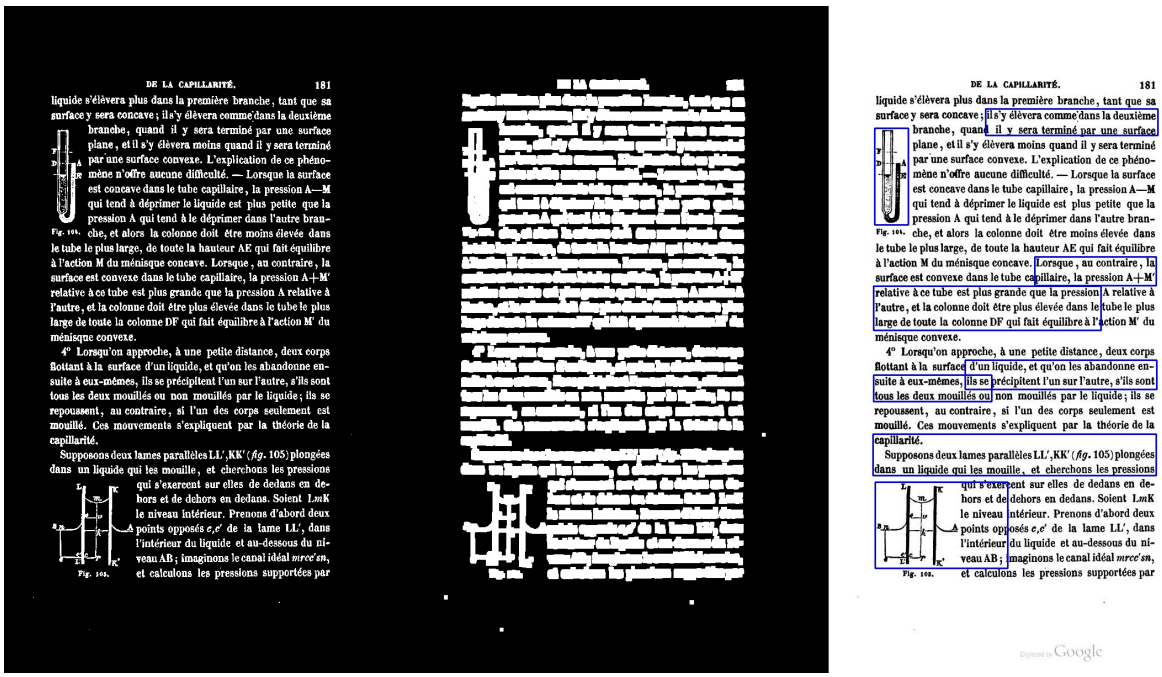


Figure 9: Figures and paragraphs which are close to each other are glued together

To solve these two problems, we needed to apply the dilation only at regions that do not contain text.

3.1.5 Removing text sections

We therefore added a pre-processing step before dilation, which consists in detecting and removing the textual part of the image, by making use of **optical character recognition (OCR)** techniques.

We used the library **Pytesseract**, which is a wrapper in Python of the Tesseract engine developed by Google, that allows to recognize text from images in different languages.

Because we wanted to recognize black text on white background, we changed the binarization mode to **binary thresholding**: the pixel `src`(x, y) at coordinates (x, y) will take the value `dest`(x, y) = 255 if `src`(x, y) > `thresh`, 0 otherwise. To remove text from the binary image, we applied the following steps:

1. Obtain the bounding boxes of the text regions
2. Create a binary mask from the bounding boxes
3. Apply the mask to the binary image

Let us examine in detail how each of these steps are performed.

Step 1: Obtain the bounding boxes of the text regions This step is implemented in the method `find_text_in(img_bin)`.

Since we would like as well to display the original image superimposed with the text boxes found in this step, we returned it in `img_with_boxes`. The binary image is converted from the grayscale to the RGB color space using the code `cv2.COLOR_BGR2GB`, again with `cv2.cvtColor`:

```
def find_text_in(img_bin):
    img_with_boxes = cv2.cvtColor(img_bin, cv2.COLOR_BGR2RGB)
```

To retrieve the bounding boxes, we used the method `pytesseract.image_to_data(image, lang, output_type)` that returns information about each word that it recognized in the image, in the form of a table (Figure 10).

We will only focus on the following columns:

1. `left, top, width, height` give the top-left corner coordinates, width and height of the box, similarly to `boundingRect`.
2. `conf` gives the level of confidence of the recognizer. This value ranges from -1 (no confident at all) to 100 (absolutely confident).

level	page_num	block_num	par_num	line_num	word_num	left	top	width	height	conf	text
1	1	0	0	0	0	0	2134	2888	-1		
2	1	1	0	0	0	758	158	575	76	-1	
3	1	1	1	0	0	717	158	616	76	-1	
4	1	1	1	1	0	758	158	575	38	-1	
5	1	1	1	1	1	717	171	69	13	18	#7:
5	1	1	1	1	2	802	169	79	27	0	$@—
5	1	1	1	1	3	892	168	42	20	48	Du
5	1	1	1	1	4	935	168	159	20	48	monvement
5	1	1	1	1	5	1104	161	24	25	94	ot
5	1	1	1	1	6	1137	166	42	20	95	des
5	1	1	1	1	7	1186	163	94	22	41	forces,
5	1	1	1	1	8	1330	158	3	7	17	‘
4	1	1	1	2	0	891	202	416	32	-1	
5	1	1	1	2	1	891	211	15	18	58	1!
5	1	1	1	2	2	920	204	24	25	63	Dù
5	1	1	1	2	3	953	202	99	27	48	movement
5	1	1	1	2	4	1058	215	19	14	91	en

Figure 10: Information about text recognized by Tesseract

3. `text` gives the text that was recognized.

We can output this table in the form of a dictionary, by setting `output_type = Output.DICT`. Moreover, we set the recognition language to French, by setting `lang='fra'`:

```
data = pytesseract.image_to_data(image=img_bin, lang='fra', output_type=Output.DICT)
```

This dictionary transforms each column attribute that we previously described into a key, whose values correspond to the values of all the bounding boxes for this attribute.

For each text that the OCR engine finds, we then need to add the coordinates of its bounding box to a list `text_boxes` that will then be used to create the mask at step 2.

The coordinates, confidence level and text are first retrieved by looking at the key-value pairs of the corresponding bounding box:

```
n_boxes = len(data['level'])
for i in range(n_boxes):
    (x, y, w, h, c, text) = (data['left'][i], data['top'][i],
                             data['width'][i], data['height'][i],
                             data['conf'][i], data['text'][i])
```

The difficulty of this part is due to the fact that Pytesseract also detects the regions where the images are located as text, as shown on the left of Figure 11. Nevertheless, the text found in these regions has some characteristics that set it apart from the text of the paragraphs.

We made use of that distinction to define a predicate that filters the image regions from the text regions. Every box that passes this filter is considered as being part of a textual region, or more precisely of a ’non-image’ region.

The bounding boxes are then drawn on `img_with_boxes` and their coordinates, width and height are added to `text_boxes` that will be returned by the function, along with `img_with_boxes`:

```
text_boxes = []
n_boxes = len(data['level'])
for i in range(n_boxes):
    (x, y, w, h, c, text) = (data['left'][i], data['top'][i],
                             data['width'][i], data['height'][i],
                             data['conf'][i], data['text'][i])
    if (#TODO: filter):
        text_boxes.append((x, y, w, h))
        cv2.rectangle(img_with_boxes, (x, y), (x + w, y + h), color=COLOR_GREEN, thickness=1)
return img_with_boxes, text_boxes
```

Indeed, we made some observations on the content returned in the dictionary for the text and image regions, and observed some differences.

The first observation we made is that text in image regions is recognized as one or more spacing characters.

We also observed that paragraphs are also recognized as a string of spaces, as well as the bounding box representing the entire page (which explains the black contour of the left image). Hence these will be removed with the filter.

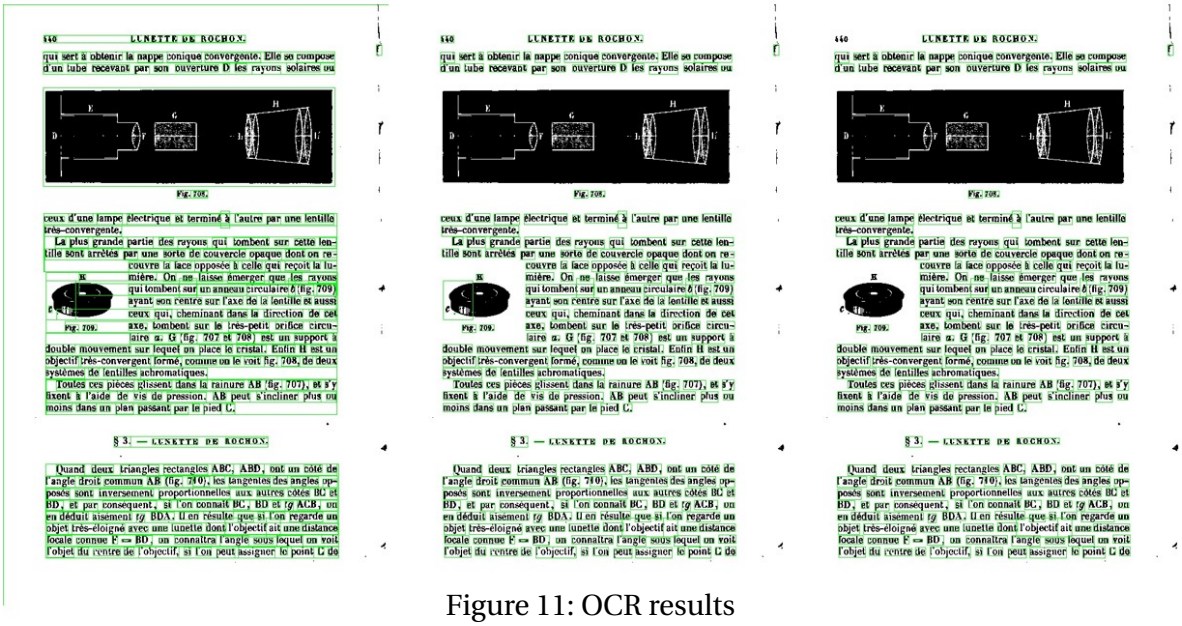


Figure 11: OCR results

To add the condition that a non-image region must have at least one non-space character, we can check if the length of the string with no spaces (returned by the method `strip()`) is nonzero:

```
if (text.strip()):
```

The results are shown in the middle of Figure 11. The top black image now does not pass the filter, yet some part of the bottom instrument is still recognized as text, but it has a confidence level of 0 (Figure 12)

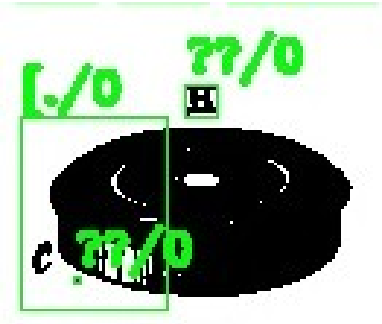


Figure 12: Special symbols are recognized with null confidence

We can therefore add to the filter the condition that the confidence level must be positive:

```
if (text.strip() and int(c) > 0):
```

We also observe that the text recognized in the problematic parts contain special symbols. Ideally, the text recognized for the words and letters should be alphanumeric, i.e., should contain either letters or numbers. We can therefore add the condition that the text recognized should contain alphanumeric symbols, using the function `has_alnum`:

```
def has_alnum(string):
    for c in string:
        if c.isalnum():
            return True
    return False

if (text.strip() and int(c) > 0 and has_alnum(text)):
```

Note that this returns `True` if and only if the string contains at least one alphanumeric character. Using `text.isalnum()` returns `True` if and only if all characters in the string are alphanumeric, which is a too strong requirement in our case. Indeed, in many cases OCR failed to detect accents, and replaced them by "?" symbols. With `isalnum()`, a lot of relevant words would then be filtered out.

The results can be seen on the right of Figure 11. The text and the figures are now clearly separated.

The final code for the method `find_text_in`is shown below:

```
def find_text_in(img_bin):
    img_with_boxes = cv2.cvtColor(img_bin, cv2.COLOR_BGR2RGB)
    data = pytesseract.image_to_data(image=img_with_boxes, lang='fra',
                                     output_type=Output.DICT)
    text_boxes = []
    n_boxes = len(data['level'])
    for i in range(n_boxes):
        (x, y, w, h, c, text) = (data['left'][i], data['top'][i],
                                 data['width'][i], data['height'][i],
                                 data['conf'][i], data['text'][i])
        if (text.strip() and int(c) > 0 and has_alnum(text)):
            text_boxes.append((x, y, w, h))
            cv2.rectangle(img_with_boxes, (x, y), (x+w, y+h), color=COLOR_GREEN, thickness=1)
    return img_with_boxes, text_boxes

# Image with text boxes
img_with_boxes, text_boxes = find_text_in(img_thresh)
```

Based on the bounding boxes that we just found, we created a binary mask to be applied to the image, in order to retain only the image regions.

Step 2: Create a binary mask from the bounding boxes To this end, we created a function `text_mask(img_bin, text_boxes)` that takes as a parameter the binarized image and the bounding boxes coordinates from Step 1.

The idea is to create a white image of the same shape as the image, and on which we draw the bounding boxes in black, as shown on the second image of Figure 13.

```
def text_mask(img_bin, text_boxes):
    mask = np.full(img_bin.shape[:2], 255, dtype="uint8") # white image
    for (x, y, w, h) in text_boxes:
        mask[y:y+h, x:x+w] = 0 # draw bounding box in black
    return mask

# Mask image
img_mask = text_mask(img_thresh, text_boxes)
```

Step 3: Apply the mask to the inverse binary image We now apply this mask to the image.

The binary image is first inverted using the method `cv2.bitwise_not`, which returns the image that will be used in the next step for dilation. Then, we perform an AND operation between the mask and this image:

```
# Apply the text mask
img_thresh_inv = cv2.bitwise_not(img_thresh)
img_no_text = cv2.bitwise_and(img_thresh_inv, img_mask)
```

The motive is that an AND between 0 and any value gives 0. In particular, when applied between the white pixels (255) of the text and the black pixels of the corresponding black bounding boxes (0), the white pixels will be colored in black. As a result, the text will disappear.

The code below and the Figure 13 summarize how text is removed from the inverted binary image, before it is processed with dilation.

```
# Step 1. Image with text boxes
img_with_boxes, text_boxes = find_text_in(img_thresh)

# Step 2. Mask image
img_mask = text_mask(img_thresh, text_boxes)

# Step 3. Apply the text mask
img_thresh_inv = cv2.bitwise_not(img_thresh)
img_no_text = cv2.bitwise_and(img_thresh_inv, img_mask)
```

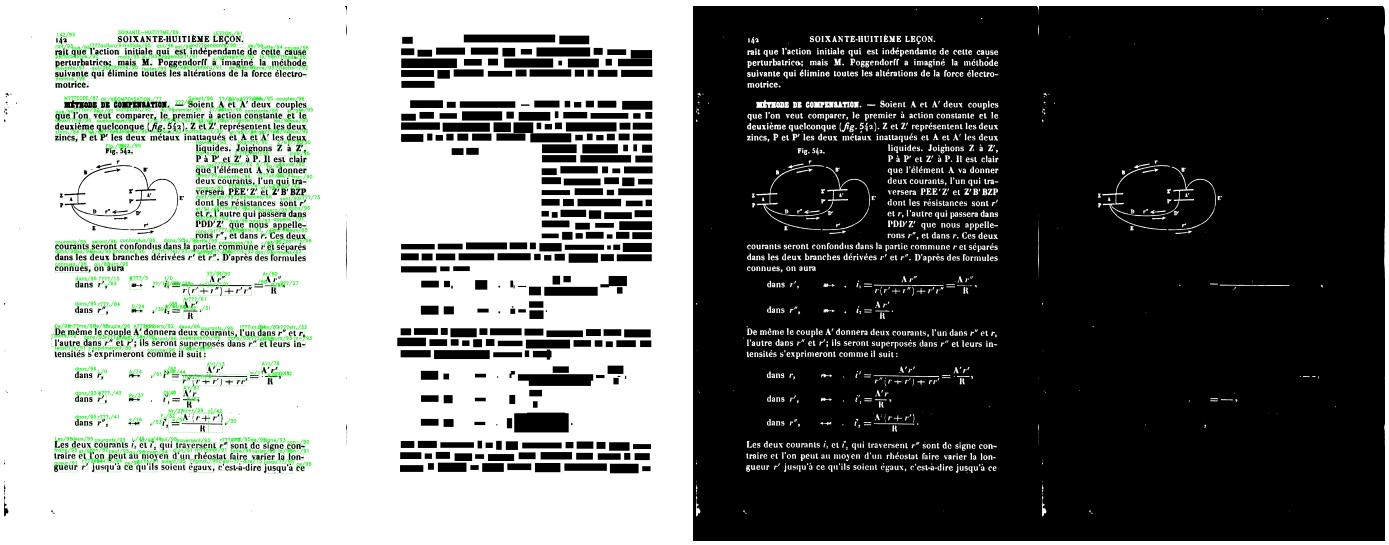


Figure 13: Removing text from an image

Note that adding this pre-processing step is doubly advantageous. Since we obtained a "clean" version of the image, where most of the text is removed, we can more confidently increase the kernel size and the number of iterations without having to deal with text blocks, which improves the genericity of our scheme. As a result, a lot less useless images are extracted, and we can be more certain to detect the diagrams correctly.

On the downside, depending on the quality of the scan, the OCR method can remove some non-text parts of figures before they are dilated, as we will see in 3.3.4.

3.1.6 Summary

To sum up, we display here the whole processing scheme applied to a page containing a diagram and an instrument:

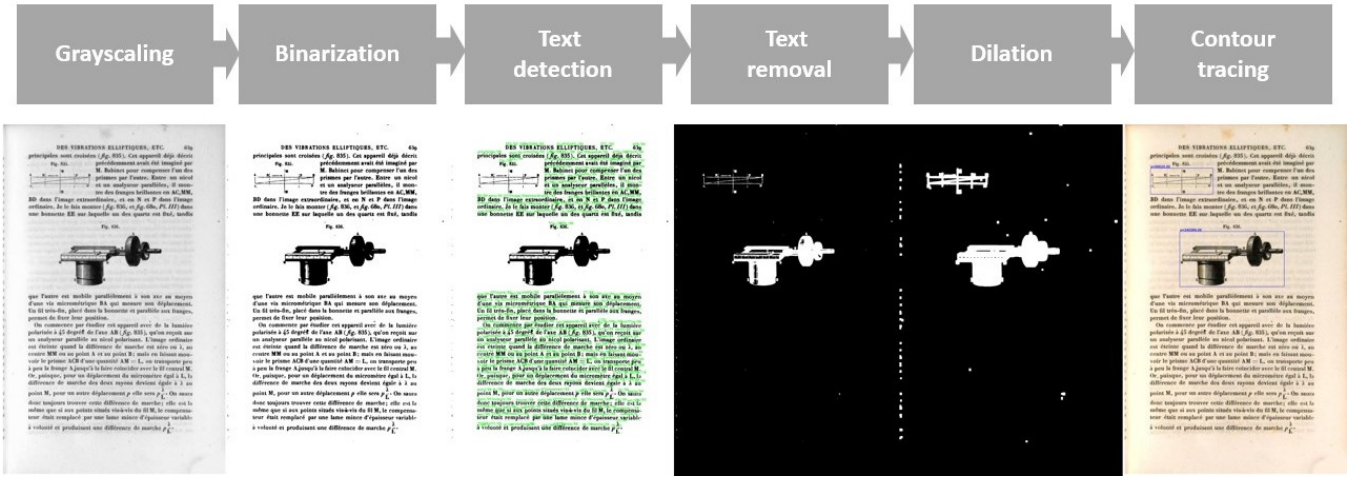


Figure 14: Extracting figures from scientific textbooks

3.2 Performance of the extractor

We can now try to approximate the performance of our pipeline. We based our approximation on the pages that required one or more of their figures that were misdetected to be manually re-cropped. To ease our calculations, we made the assumption that each page contains on average one figure.

Hence the approximation is simply the ratio $\frac{\#\text{pages} - \#\text{handled pages}}{\#\text{pages}}$. Below we show the results for all the books of our set:

Book	**#pages**	**#handled pages**	**Ratio (in %)**
daguin_t1	543	42	**92.3**
daguin_t2	334	34	**98.8**
daguin_t3	403	28	**93.1**
deguin_ed7	301	13	**95.7**
deguin_ed9	199	31	**84.4**
desains_t1s1	311	44	**85.9**
desains_t1s3	313	15	**95.2**
desains_t2s1	375	25	**93.3**
desains_t2s2	392	9	**97.7**
desains_t2s3	381	12	**96.9**
desains_t2s4	392	13	**96.7**
ganot_ed13	519	25	**95.2**
ganot_ed17	548	54	**90.1**
jamin_ed3t3	387	25	**93.5**
jamin_ed3t4	271	20	**92.6**
jamin_s1	36	4	**88.8**
jamin_s2	38	7	**81.6**
jamin_s3	87	21	**75.9**
jamin_t3	355	28	**92**
jamin_t4p2f4	72	17	**76.4**

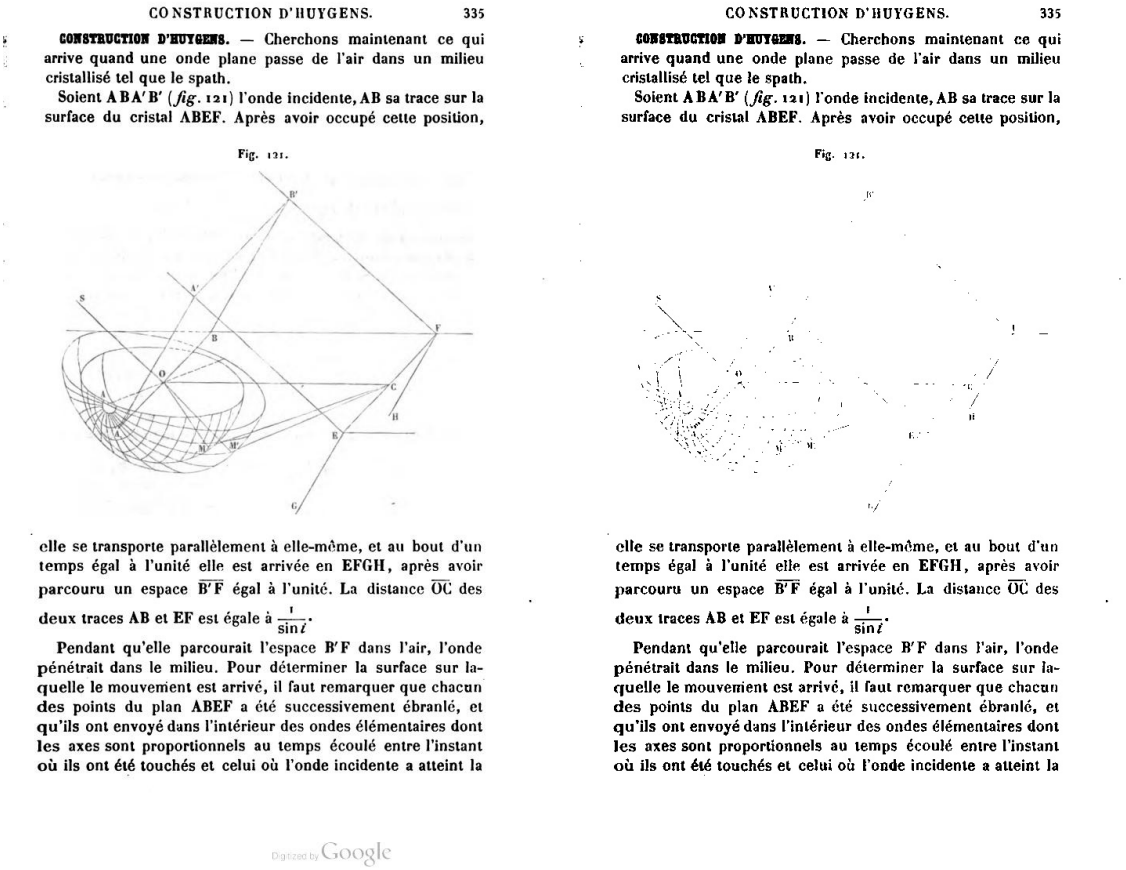
We observe that the performance of the extraction can vary between scans of the same book, as it is the case for `desains_t1s3` and `desains_t1s1`, which has relatively bad quality pages.

Indeed, one major weakness of the pipeline is that its performance in general stays sensitive to the quality of the scans, even with the uniformization obtained after grayscaling and binarization. Below we show some examples that needed to be handled manually and which explain the differences in performance.

3.3 Limitations of the pipeline

3.3.1 At the binarization step

Problems may happen when determining the optimal threshold value. Indeed, we observe on Figure 15 that edges from the diagram were removed in the binary image. The optimal value was found to be 146, whereas the values of the edge pixels range from 150 to 220, so they are considered as white. This problem happens when diagrams are lighter than the text, or when diagram edges are degraded due to to the quality of the scan.



CONSTRUCTION D'HUYGENS. 335

CONSTRUCTION D'HUYGENS. — Cherchons maintenant ce qui arrive quand une onde plane passe de l'air dans un milieu cristallisé tel que le spath.

Soient ABA'B' (*fig.* 121) l'onde incidente, AB sa trace sur la surface du cristal ABEF. Après avoir occupé cette position,

Fig. 121.

elle se transporte parallèlement à elle-même, et au bout d'un temps égal à l'unité elle est arrivée en EFGH, après avoir parcouru un espace $\overline{B'F}$ égal à l'unité. La distance $\overline{OC}$ des deux traces AB et EF est égale à $\frac{1}{\sin i}$.

Pendant qu'elle parcourait l'espace B'F dans l'air, l'onde pénétrait dans le milieu. Pour déterminer la surface sur laquelle le mouvement est arrivé, il faut remarquer que chacun des points du plan ABEF a été successivement ébranlé, et qu'ils ont envoyé dans l'intérieur des ondes élémentaires dont les axes sont proportionnels au temps écoulé entre l'instant où ils ont été touchés et celui où l'onde incidente a atteint la

CONSTRUCTION D'HUYGENS. 335

CONSTRUCTION D'HUYGENS. — Cherchons maintenant ce qui arrive quand une onde plane passe de l'air dans un milieu cristallisé tel que le spath.

Soient ABA'B' (*fig.* 121) l'onde incidente, AB sa trace sur la surface du cristal ABEF. Après avoir occupé cette position,

Fig. 121.

elle se transporte parallèlement à elle-même, et au bout d'un temps égal à l'unité elle est arrivée en EFGH, après avoir parcouru un espace $\overline{B'F}$ égal à l'unité. La distance $\overline{OC}$ des deux traces AB et EF est égale à $\frac{1}{\sin i}$.

Pendant qu'elle parcourait l'espace B'F dans l'air, l'onde pénétrait dans le milieu. Pour déterminer la surface sur laquelle le mouvement est arrivé, il faut remarquer que chacun des points du plan ABEF a été successivement ébranlé, et qu'ils ont envoyé dans l'intérieur des ondes élémentaires dont les axes sont proportionnels au temps écoulé entre l'instant où ils ont été touchés et celui où l'onde incidente a atteint la

Figure 15: The effects of an underestimated threshold value on binarization of a diagram

The thresholding algorithm is then influenced by darker elements in the scan, in our case the text (whose pixel values are strictly smaller than 100) to determine the optimal value, which filters out many pixels from lighter diagrams.

The thresholding algorithm can also overestimate the threshold. This is the case in Figure 16,

which shows the dilation and contouring steps. Here the pixels of the shadow at the right border of the page have been considered in the second step as black pixels, hence it impacts afterwards the dilation step, which glues the map to the margin, and as result detects the entire page as ROI.

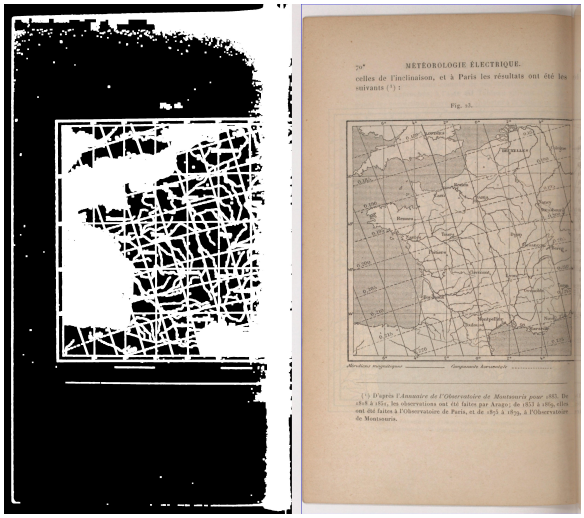


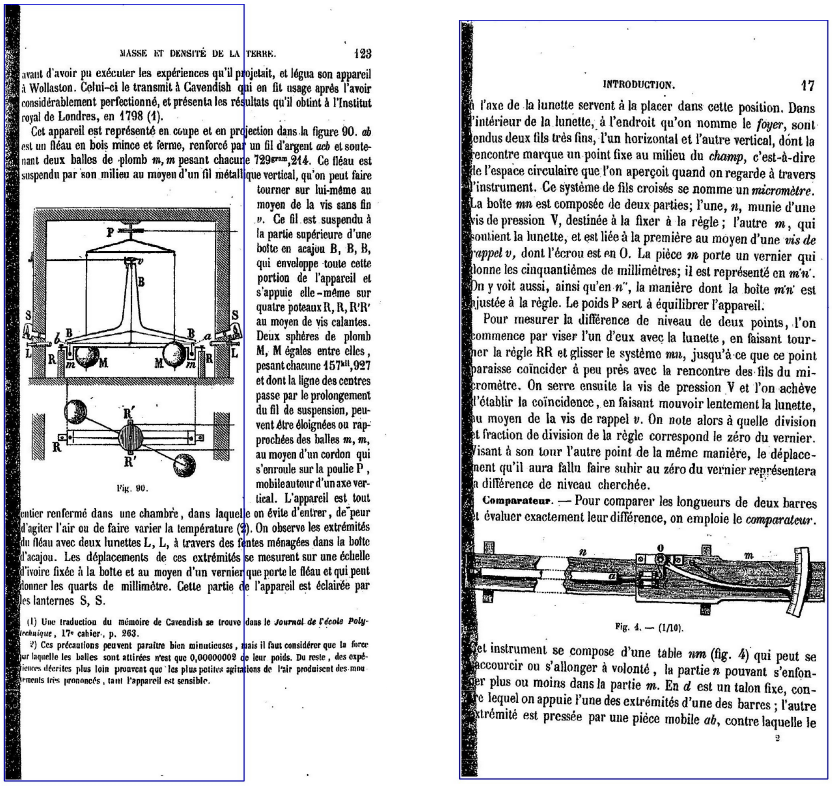
Figure 16: The effects of an overestimated threshold value on binarization of a diagram

3.3.2 At the dilation step

This problem happened especially in the `daguin` scans, that have made a black margin appear at the left of the image (Figure 17). Figures that are close to the margin can hence be glued to it during dilation, which results in a merging figure of the same height as the margin, as seen in the previous section. Moreover, the scan has sometimes produced the effect that we see on the right of the figure, where a figure's edge is directly glued to the margin. For these cases, the pipeline can never succeed.

3.3.3 At the contouring step

This problem relates to the diagrams whose understanding relies on elements that are distant to each other. For these cases, the distance between the elements is too high for dilation to "build" the diagram in its entirety, hence the contouring algorithm interprets the dilated figure as multiple separate regions (Figure 18).



MASSE ET DENSITÉ DE LA TERRE. 123

avant d'avoir pu exécuter les expériences qu'il projetait, et légua son appareil à Wollaston. Celui-ci le transmit à Cavendish qui en fit usage après l'avoir considérablement perfectionné, et présenta les résultats qu'il obtint à l'Institut royal de Londres, en 1798 (1).

Cet appareil est représenté en coupe et en projection dans la figure 90. *ab* est un fléau en bois mince et ferme, renforcé par un fil d'argent *acb* et soutenant deux balles de plomb *m*, *m* pesant chacune 729gr.m,214. Ce fléau est suspendu par son milieu au moyen d'un fil métallique vertical, qu'on peut faire tourner sur lui-même au moyen de la vis sans fin *n*. Ce fil est suspendu à la partie supérieure d'une boîte en acajou B, B, B, qui enveloppe toute cette portion de l'appareil et s'appuie elle-même sur quatre poteaux R, R, R'R' au moyen de vis calantes. Deux sphères de plomb M, M égales entre elles, pesant chacune 157kil,927 et dont la ligne des centres passe par le prolongement du fil de suspension, peuvent être éloignées ou rapprochées des balles *m*, *m*, au moyen d'un cordon qui s'enroule sur la poulie P, mobile autour d'un axe vertical. L'appareil est tout entier renfermé dans une chambre, dans laquelle on évite d'entrer, de peur d'agiter l'air ou de faire varier la température (2). On observe les extrémités du fléau avec deux lunettes L, L, à travers des fentes ménagées dans la boîte d'acajou. Les déplacements de ces extrémités se mesurent sur une échelle d'ivoire fixée à la boîte et au moyen d'un vernier que porte le fléau et qui peut donner les quarts de millimètre. Cette partie de l'appareil est éclairée par les lanternes S, S.

Fig. 90.

(1) Une traduction du mémoire de Cavendish se trouve dans le *Journal de l'école Polytechnique*, 17e cahier, p. 263.

2) Ces précautions peuvent paraître bien minutieuses, mais il faut considérer que la force par laquelle les balles sont attirées n'est que 0,00000009 de leur poids. Du reste, des expériences décrites plus loin prouvent que les plus petites agitations de l'air produisent des mouvements très prononcés, tant l'appareil est sensible.

INTRODUCTION. 17

à l'axe de la lunette servent à la placer dans cette position. Dans l'intérieur de la lunette, à l'endroit qu'on nomme le *foyer*, sont tendus deux fils très fins, l'un horizontal et l'autre vertical, dont la rencontre marque un point fixe au milieu du *champ*, c'est-à-dire de l'espace circulaire que l'on aperçoit quand on regarde à travers l'instrument. Ce système de fils croisés se nomme un *micromètre*. La boîte *mn* est composée de deux parties; l'une, *n*, munie d'une vis de pression V, destinée à la fixer à la règle; l'autre *m*, qui soutient la lunette, et est liée à la première au moyen d'une *vis de rappel v*, dont l'écrou est en O. La pièce *m* porte un vernier qui donne les cinquantièmes de millimètres; il est représenté en *m'n'*. On y voit aussi, ainsi qu'en *n''*, la manière dont la boîte *m'n'* est ajustée à la règle. Le poids P sert à équilibrer l'appareil.

Pour mesurer la différence de niveau de deux points, l'on commence par viser l'un d'eux avec la lunette, en faisant tourner la règle RR et glisser le système *mn*, jusqu'à ce que ce point paraisse coïncider à peu près avec la rencontre des fils du micromètre. On serre ensuite la vis de pression V et l'on achève d'établir la coïncidence, en faisant mouvoir lentement la lunette, au moyen de la vis de rappel *v*. On note alors à quelle division et fraction de division de la règle correspond le zéro du vernier. Visant à son tour l'autre point de la même manière, le déplacement qu'il aura fallu faire subir au zéro du vernier représentera la différence de niveau cherchée.

Comparateur. — Pour comparer les longueurs de deux barres et évaluer exactement leur différence, on emploie le *comparateur*.

Fig. 4. — (1/10).

Cet instrument se compose d'une table *nm* (fig. 4) qui peut se raccourcir ou s'allonger à volonté, la partie *n* pouvant s'enfoncer plus ou moins dans la partie *m*. En *d* est un talon fixe, contre lequel on appuie l'une des extrémités d'une des barres; l'autre extrémité est pressée par une pièce mobile *ab*, contre laquelle le

2

Figure 17: Black margins affect the extraction

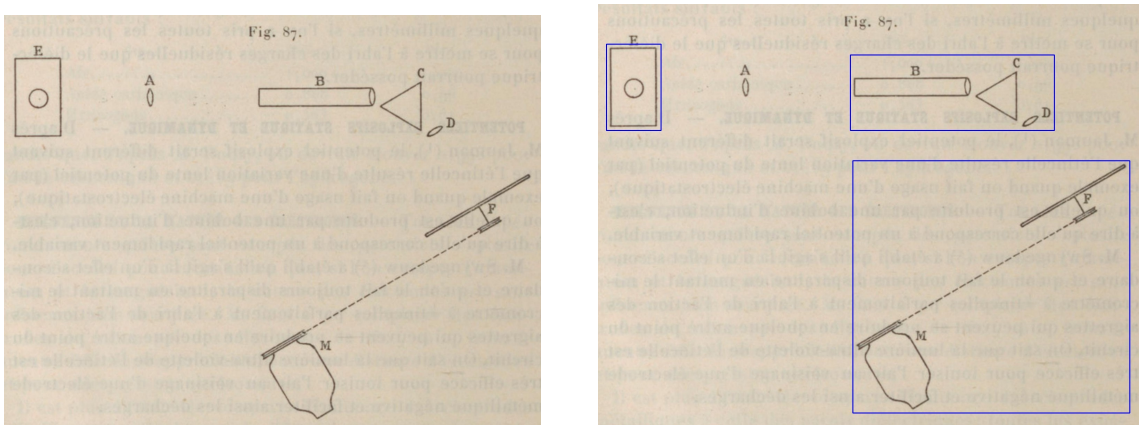


Figure 18: Spread out diagrams are broken by the contouring algorithm, which modifies their interpretation

3.3.4 At the text removal step

We finish by the most frequent errors, which relate to the text recognition. In many cases indeed, Tesseract recognized letters or even words from the non-caption parts of the figures, as shown below. As a result, these parts are removed by the mask, which shortens the boundary of the object and hence leads to a bad cropping.

We show below an example from `desains_t1s1` that illustrates this phenomenon, as well as the same example but from `desains_t1s3`, which has a better quality scan.

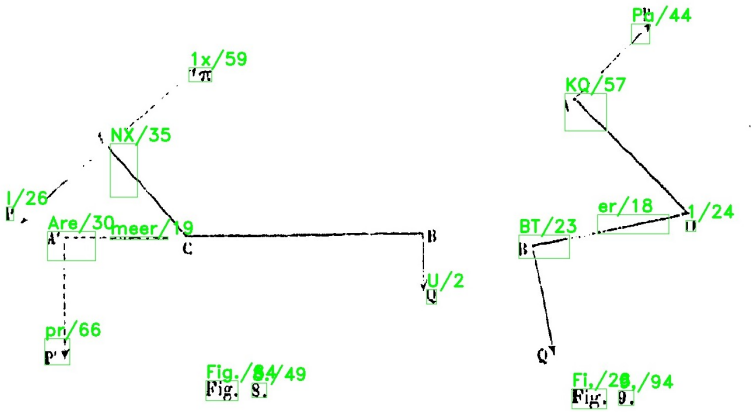


Figure 19: Words are detected in the edges of the diagram

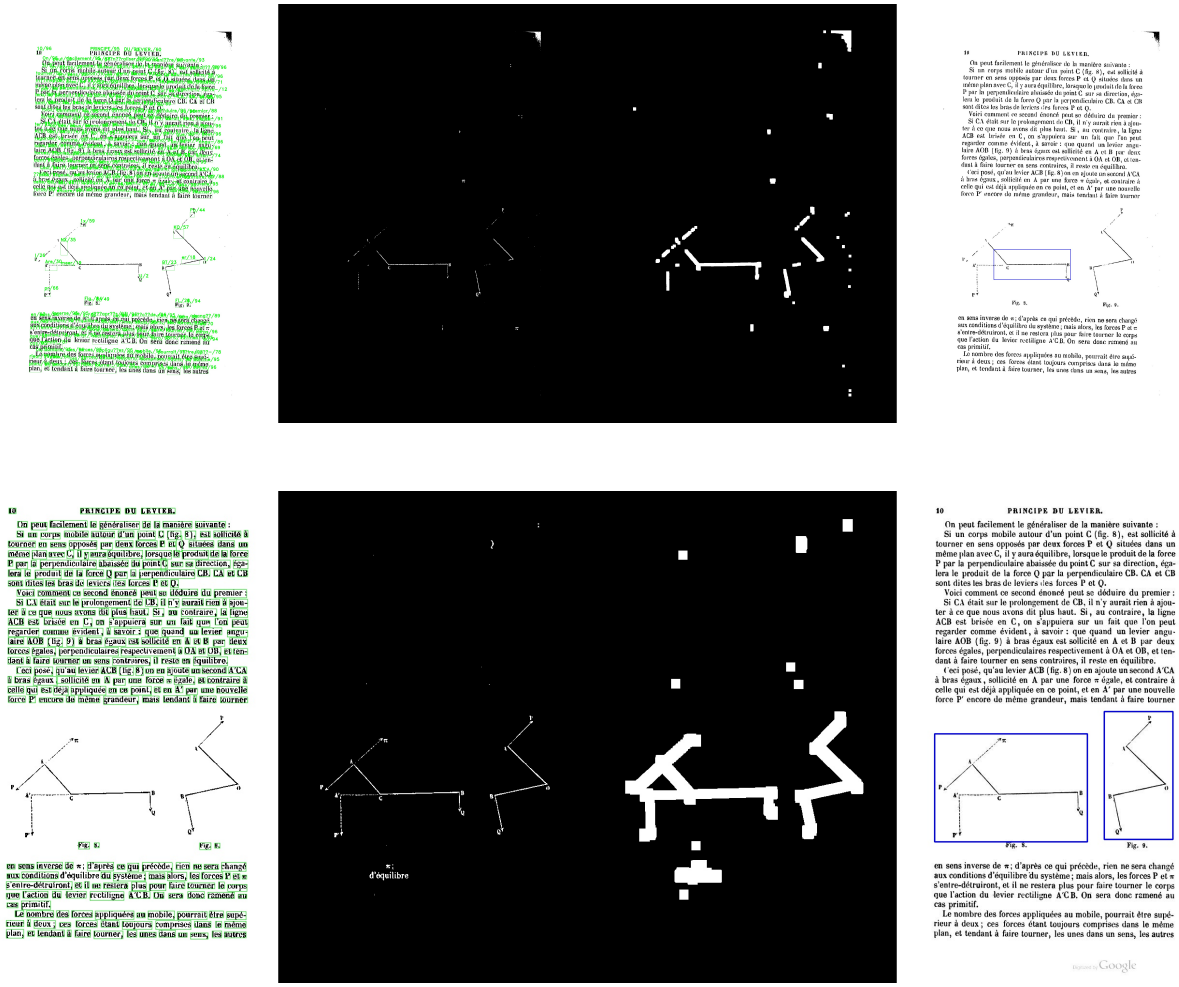


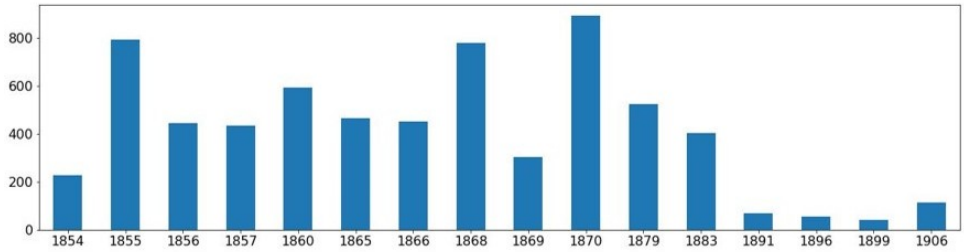
Figure 20: Quality of the scan impacts the extracting results for the same figure

Overall, we find that the performance of our pipeline still depends on the quality of the scan, which can induce different extracting results even for the same figure, as shown above.

Now that we extracted the images from the scans, we classify them into the classes that we are interested in: diagrams, instruments and black figures.

3.4 Extracting results

The dataset obtained after extraction comprises 8466 images, from books whose publication dates span from 1854 to 1906. Figure 21 below shows the counts per year, where we removed from the count the ones from redundant scans of Volumes 1 and 2 of Paul Desains's book (1857 and 1865): `desains_t1s1`, `desains_t2s1`, `desains_t2s2` and `desains_t2s4`.



Year	Extracted images
1854	229
1855	793
1856	446
1857	436
1860	593
1865	467
1866	451
1868	778
1869	303
1870	893
1879	523
1883	404
1891	69
1896	56
1899	41
1906	115

Figure 21: Year counts

Figure 22 shows the counts per author, where we kept the redundant scans in the counts:

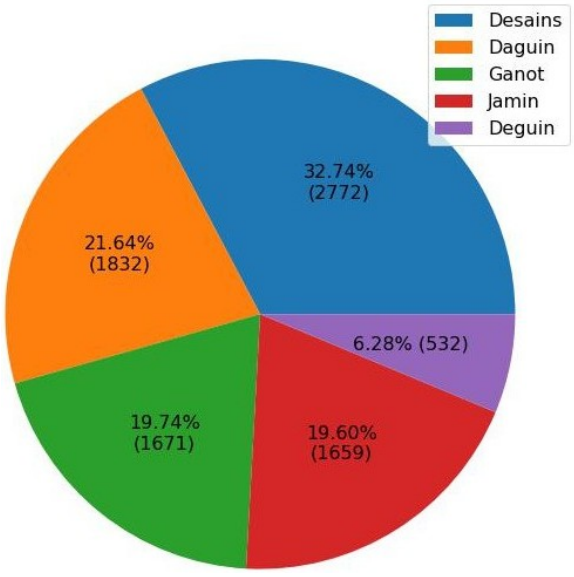


Figure 22: Author counts

Our goal now is to classify the dataset into the three classes that we are interested in: diagrams, instruments, and black figures.

4 Image classification

To this end, we applied knowledge from Convolutional Neural Networks (CNN), in particular we trained a CNN on a subset of the extracted images and measured its performance. We then applied the trained classifier to the rest of the dataset.

4.1 Constructing the training and test sets

In total, 1028 images were used, 798 for training and 230 for testing. The training set divides into 340 diagrams, 340 instruments and 118 black figures. As for the testing set, it is composed of 100 diagrams, 100 instruments and 30 black figures.

As we will see in 5.2, we consider some figures to be ambiguous, i.e., it is more difficult for them to determine visually to which class they belong to, in particular the difficulty appeared between diagrams and instruments.

Consequently, we chose to compose the training and test sets only with images that are "paradigmatic" of their class. The idea was to not mislead the classifier with too many ambiguous samples or samples who have a unique structure in their class, as the calculation of the weights could have impacted future predictions.

To help us implement this task, we relied on the **PyTorch** framework which provides many built-in functions for creating our own datasets and machine learning models.

One useful functionality of PyTorch is `ImageFolder`, which allows to build custom `Datasets` from the contents and structure of a directory. In our case, we organized the training and test samples in the following way:

```
dataset
├── train
│   ├── 0_diagram
│   ├── 1_instrument
│   └── 2_black
└── test
    ├── 0_diagram
    ├── 1_instrument
    └── 2_black
```

```
# Create datasets
train_data = ImageFolder(
    os.path.join(os.getcwd(), "dataset", "train"),
    transform=Compose( # data augmentation
        [Grayscale(num_output_channels=3),
         Resize((224, 224)),
         RandomHorizontalFlip(),
         RandomVerticalFlip(),
         ToTensor()
        ]
    )
)

test_data = ImageFolder(
    os.path.join(os.getcwd(), "dataset", "test"),
    transform=Compose( # data augmentation
        [Grayscale(num_output_channels=3),
         Resize((224, 224)),
         RandomHorizontalFlip(),
         RandomVerticalFlip(),
         ToTensor()
        ]
    )
)
```

The above code creates the training and test sets from the directory files, and automatically finds the annotations of the files through the subfolder they are located in. Samples in directory `'0_diagram'` are labelled 0, the ones in `'1_instrument'` are labelled 1 and the ones in `'2_black'` are labelled 2.

In addition, we performed **data augmentation** by applying minor transformations to the images. They are grayscaled and are randomly flipped, horizontally and/or vertically. We moreover resized them to square images of shape $224 \cdot 224 \cdot 3$, which is the shape required in the first layer of the CNN we will use.

We can then prepare our training and test data:

```
# Specify corresponding batched data loaders
train_loader = DataLoader(train_data, batch_size=len(train_data), shuffle=True)
test_loader = DataLoader(test_data, batch_size=len(test_data), shuffle=False)

# Training and test data
```

```
X_train, y_train = next(iter(train_loader)) # (798, 3, 224, 224)
X_test, y_test = next(iter(test_loader)) # (230, 3, 224, 224)
```

We are now able to feed our training and test data to a machine learner, in order to calculate the appropriate weights and derive a model/classifier for our dataset.

4.2 Training the model with active learning

For this task we relied on a specific case of learning called **active learning**.

4.2.1 Active learning

Supervised models generally require large amounts of data to be trained efficiently. This implies that a lot of samples must be annotated manually by a human, which may be time-consuming to do when constructing large datasets.

Active learning provides techniques that make this step much easier to handle. In the context of our task, we looked at one category of active learning, which is **pool-based sampling**.

In this method, the model is first trained on a small set of labelled samples, that we call the **initial data**. Then it makes predictions on a much larger set of unlabelled samples called the **pool**. Among these samples, the active learner selects the one it is the least confident about with the prediction, and asks an **oracle** (i.e., a human) to label this uncertain sample. The sample and its annotation are thus added in the dataset, on which the model is retrained. In essence, active learning is therefore a cooperation between the human and the machine, where the latter asks the human to annotate samples that it is unsure about. This way, the human only needs to label the data that is the most beneficial to the machine.

By definition, the active learner needs to have a notion of "confidence" about the samples it is working on, and must be able to assign a "confidence score" for each one of them. Different scores define different **query strategies** to the oracle. In our case, we chose the **entropy sampling** strategy; i.e., we took as confidence score the **entropy function**.

Depending on the context in which it is applied, the entropy can have multiple significations. Here, the entropy represents the level of uncertainty of the probability distribution which is given by the prediction vector P at the last layer of the CNN. The probability distribution hence will be of size 3, with each element being the probability $p_i \in [0,1], i \in$ [diagram, instrument, black] that the predicted image belongs to class i.

The value of the entropy for the prediction vector P of a sample is given by:

$$H(P) = -\sum_i p_i \cdot \log_2(p_i),$$

where $p_i = p(\text{"sample belongs to class } i\text{"})$

For instance, a sample for which the model predicts with 99% confidence that it is a diagram will have a small entropy/uncertainty:

$$P = [0.99, 0.005, 0.005]$$

$$H(P) = -0.99 \cdot \log_2(0.99) - 2 \cdot 0.05 \log_2(0.05) \approx 0.09$$

For a sample that it classifies with 100% confidence, the entropy is zero:

$$P = [1.0, 0.0, 0.0]$$

$$H(P) = -1.0 \cdot \log_2(1.0) = 0$$

In fact it is the minimum entropy possible, there is no uncertainty at all.

On the opposite, the uncertainty is maximal when the classifier cannot decide between all three classes:

$$P = [\frac{1}{3}, \frac{1}{3}, \frac{1}{3}]$$

$$H(P) = -3 \cdot \frac{1}{3} \cdot \log_2(\frac{1}{3}) \approx 1.58$$

To implement the pool-based sampling strategy, we used the **modAL** framework, which provides tools to create and customize our own active learning workflows in only a few lines of code, as we will see. This framework is built on top of the **scikit-learn** library. To make its functions compatible with our PyTorch training and test sets, we used **Skorch** that wraps PyTorch models around a scikit-learn interface.

4.2.2 Assembling data

As a first step, we assembled the initial data by taking a fixed number (here 128) of random samples from our training data, which represents approximately 16% of the 798 training samples.

```
# assemble initial data
n_initial = 128
initial_idx = np.random.choice(range(len(X_train)), size=n_initial, replace=False)
X_initial = X_train[initial_idx] # (128, 3, 224, 224)
```

```
y_initial = y_train[initial_idx] # (128,1)
```

Then, the pool is in our case generated from the remaining 670 samples. Their annotation hence has remained unknown to the active learner.

```
# remove the initial data from the training dataset
X_pool = np.delete(X_train, initial_idx, axis=0) # (670, 3, 224, 224)
```

4.2.3 Loading a pre-trained model

We based our classifier on the **ResNet18** architecture, and initialized our weights with the ones that were pre-trained for the ImageNet database.

```
model_ft = torchvision.models.resnet18(pretrained=True)
```

The last fully connected layer has 1000 output neurons since it has been trained on the ImageNet task, which has 1000 image classes. But we would like to perform ternary classification. Therefore, we replaced the last fully-connected layer to suit our needs (three output units):

```
in_features = model_ft.fc.in_features
out_features = len(train_data.classes) # 3 ('0_diagram', '1_instrument', '2_black')
model_ft.fc = nn.Linear(in_features=in_features, out_features=out_features))
```

4.2.4 Initializing the active learner

```
device_type = "cuda" if torch.cuda.is_available() else "cpu"
device = torch.device(device_type)
# create the classifier
classifier = NeuralNetClassifier(model_ft,
                                 criterion=nn.CrossEntropyLoss,
                                 optimizer=torch.optim.Adam,
                                 train_split=None,
                                 verbose=0,
                                 device=device_type)
```

We then wrapped our PyTorch model `model_ft` into a scikit-learn classifier by using the class `NeuralNetClassifier` from Skorch. Now any function of scikit-learn can be called, hence we can use the modAL framework.

The learning rate `lr` and the batch size `batch_size` can be determined by a **grid search** on the initial data:

```
param_grid = {
    'lr': [1e-5, 1e-4, 1e-3, 1e-2, 1e-1],
    'batch_size': [128, 64, 32, 16, 8, 4]
}

gs = GridSearchCV(estimator=classifier,
                  param_grid=param_grid,
                  scoring=make_scorer(f1_score, average='micro'),
                  refit=False,
                  cv=3,
                  verbose=2)

gs.fit(X_initial, y_initial)

classifier.set_params(lr=gs.best_params_['lr'])
classifier.set_params(batch_size=gs.best_params_['batch_size'])
```

This will compute the **accuracy** = $\frac{\#\text{samples who were correctly classified}}{\text{Total }\#\text{samples}}$ for every combination of `lr` and `batch_size` (which amount to 30 here), and return the couple that yields the highest accuracy.

We can now initialize our active learner with **modAL**:

```
learner = ActiveLearner(
    estimator=classifier,
    query_strategy=entropy_sampling,
    X_training=X_initial, y_training=y_initial
)
```

The learner automatically trains on the initial data that is passed as an argument. We also specified that it should use entropy sampling as a query strategy, which is already implemented in modAL.

4.2.5 The active learning loop

The main part of the training consists in asking `n_queries` times the oracle to label the sample with the highest entropy, and then retrain the model with the new dataset. At each iteration, the sample is displayed, next to a graph that shows how the accuracy evolved so far during the active learning (Figure 23).

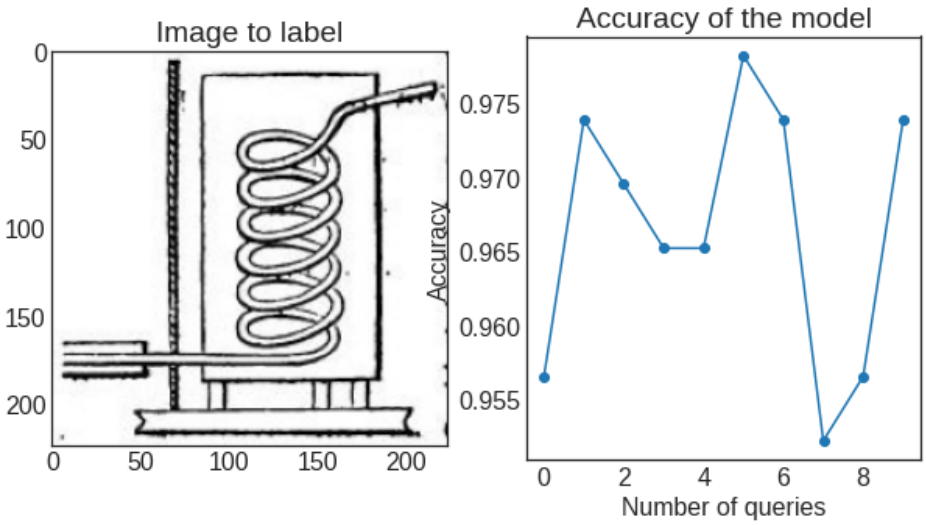
```
accuracy_scores = [learner.score(X_test, y_test)]

n_queries = 10
for i in range(n_queries):
    query_idx, query_instance = learner.query(X_pool) # (1, 3, 224, 224)

    # Plot the sample to label and the accuracy of the model so far (not shown)

    # Ask label to user
    print(f"Query no. {i+1}: What is the class of this image?")
    y_new = np.array([int(input())], dtype=int)

    # Re-train model
    learner.teach(query_instance, y_new)

    # Remove queried instance from pool
    X_pool = np.delete(X_pool, query_idx, axis=0)
    accuracy_scores.append(learner.score(X_test, y_test))
```

The method `query(X_pool)` applies under the hood the entropy sampling function and returns the resulting highest-entropy sample, along with its index in the pool dataset.

The user is then asked to label the displayed sample, by entering the index of the class.

The available training data is then augmented with the queried sample and its label `y_new` entered by the user, then refits the `classifier` to this new training set. In the backend, `learner` indeed keeps track of the training data it has seen during its lifetime. All of the above is done in the method `teach(query_instance, y_new)`

The queried sample is thus removed from the pool and the new test accuracy is added to a list, which is displayed on the right plot of Figure 23.




```
Query no. 10: What is the class of this image?
 1
Re-initializing module.
Re-initializing criterion.
Re-initializing optimizer.
  epoch    f1_score    train_loss     dur
-------  ----------  ------------  ------
      1      1.0000        0.0006  0.2887
      2      0.9928        0.0230  0.2700
      3      1.0000        0.0010  0.2683
      4      1.0000        0.0000  0.2681
      5      1.0000        0.0000  0.2677
      6      1.0000        0.0000  0.2687
      7      1.0000        0.0000  0.2665
      8      1.0000        0.0000  0.2687
      9      1.0000        0.0000  0.2681
     10      1.0000        0.0000  0.2683
```

Figure 23: The oracle interface

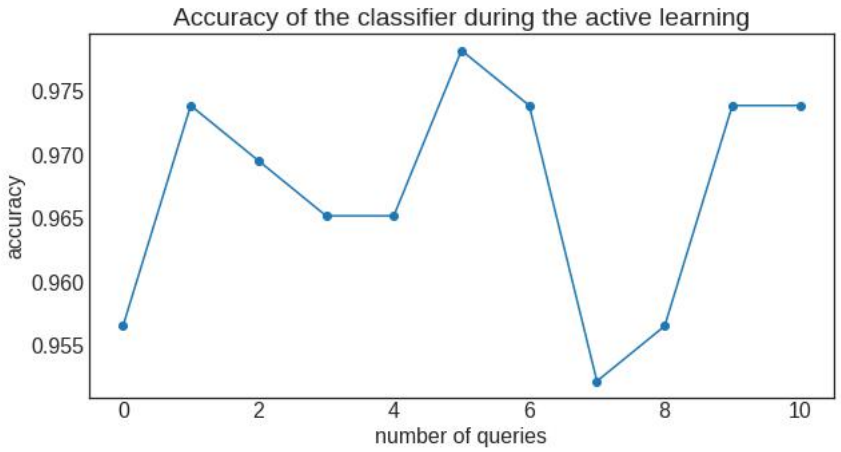


Figure 24: Evolution of the accuracy during the active learning

4.3 Performance of the classifier

To evaluate our classifier, we relied on the confusion matrices obtained for the training set and the test set.

As displayed on Figure 24, the classifier's final test accuracy is 97%. We can retrieve this value by displaying the confusion matrix obtained on the test set `X_test` and `y_test`:

```
from sklearn.metrics import confusion_matrix

preds = classifier.predict_proba(X_test)
y_pred = np.apply_along_axis(lambda pred: np.argmax(pred), axis=1, arr=preds)
confusion_matrix(y_true=y_test, y_pred=y_pred, labels=[0, 1, 2])
```

We obtain the following table, where the rows correspond to the predicted class and the columns to the actual class. The order of both rows and columns is "diagram", "instrument" and "black".

```
array([[98,  2,  0],
       [ 4, 96,  0],
       [ 0,  0, 30]])
```

We indeed retrieve the accuracy $\frac{98+96+30}{98+2+4+96+30} = \frac{224}{230} = 0.9739 \approx 97\%$.

For the train accuracy we have the following:

```
preds = classifier.predict_proba(X_train)
y_pred = np.apply_along_axis(lambda pred: np.argmax(pred), axis=1, arr=preds)
confusion_matrix(y_true=y_train, y_pred=y_pred, labels=[0, 1, 2])
```

```
>>> array([[335,   5,   0],
           [ 12, 328,   0],
           [  0,   0, 118]])
```

Likewise, we found a train accuracy of $\frac{781}{798} = 0.9786 \approx 98\%$.

Based on the weights that were calculated with active learning, we applied our newly-trained model on the remaining 7442 images of the set. We will now examine the results of this classification.

5 Analysis of the dataset

This final section will be dedicated to studying the lessons we can learn from our classified dataset, giving some statistics about it and explaining how it could prove useful for future applications.

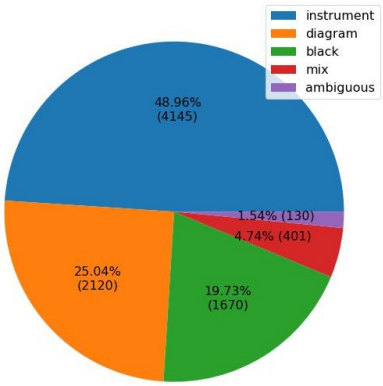


Figure 25: Distribution of the figures in the classified dataset

When applying our trained classifier on the remaining 7442 images, we also keep track of the `entropy` of the prediction and the probability vector `prediction`.

To ease the following calculations, we used the **Pandas** library to store the results in a table (a `DataFrame`), with other useful information such as the year or the author. Figure 26 below shows a little snippet of this table, with rows sorted by the entropy in decreasing order:

	filename	entropy	prediction	predicted_class	misclassified	real_class	year	author	book
8459	ganot_ed17-601_1.jpg	1.500669	[0.1901204, 0.34962684, 0.46025276]	ambiguous	False	ambiguous	1870	Ganot	ganot_ed17
8464	ganot_ed17-106_1.jpg	1.482158	[0.49477264, 0.18950227, 0.31572512]	ambiguous	False	ambiguous	1870	Ganot	ganot_ed17
8420	desains_t1s3-520_7.jpg	1.420480	[0.21884738, 0.566638, 0.21451463]	ambiguous	False	ambiguous	1857	Desains	desains_t1s3
8422	desains_t1s1-071_1.jpg	1.403330	[0.11894093, 0.47562638, 0.4054327]	ambiguous	False	ambiguous	1857	Desains	desains_t1s1
8451	daguin_t1-0369_2.jpg	1.344321	[0.09173174, 0.49817345, 0.41009483]	ambiguous	False	ambiguous	1855	Daguin	daguin_t1
...	...	...	...	...	...	...	...	...	...
3052	desains_t2s1-249_2.jpg	0.000000	[0.0, 0.0, 1.0]	black	False	black	1865	Desains	desains_t2s1
3045	jamin_t3-805_1.jpg	0.000000	[0.0, 0.0, 1.0]	black	False	black	1866	Jamin	jamin_t3
3044	desains_t2s3-457_1.jpg	0.000000	[0.0, 0.0, 1.0]	black	False	black	1865	Desains	desains_t2s3
6329	desains_t2s1-382_2.jpg	0.000000	[0.0, 1.0, 0.0]	instrument	False	instrument	1865	Desains	desains_t2s1
5619	daguin_t3-0342_2.jpg	0.000000	[0.0, 1.0, 0.0]	instrument	False	instrument	1860	Daguin	daguin_t3

8466 rows × 9 columns

Figure 26: Classification data

The rows at the bottom of the table represent the images we used in our training and test sets, so they have minimum entropy and their prediction vector is the most certain probability

distribution. Moreover we added two classes to help us through the statistics, `mix` and `ambiguous`, that we will explain in 5.2.

5.1 Misclassifications

Let us now see what we can draw from the misclassified samples.

In total, 450 samples were misclassified by the model. Figure 27 show the nature of these misclassifications. They happen in their majority between diagrams and instruments, at 95%. More precisely, 59% of the misclassifications relate to instruments that were classified as diagrams and 35% for the inverse. The notation `class_1,class_2` indicates that a sample was classified as `class_1` whereas its real class was `class_2`.

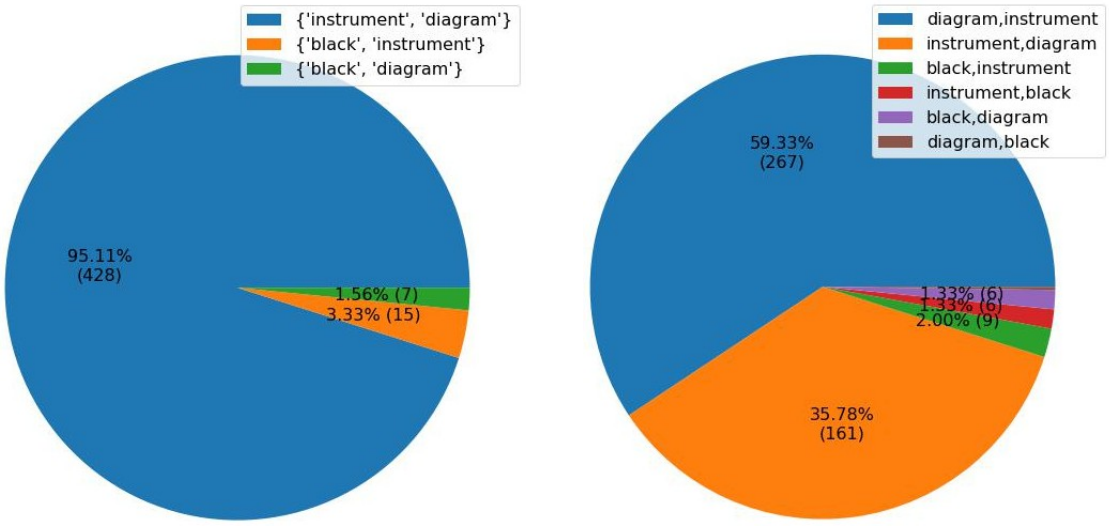


Figure 27: Distributions of the misclassifications

Instruments classified as diagrams We observe on the figure below some of the instruments who were misclassified with the lowest entropy as diagrams. What could explain the decision made by the classifier is that each of them indeed does not convey the "solidity" of normal instruments, and are here represented with very few details, with thin edges as in diagrams, and are depicted in profile, in two dimensions.

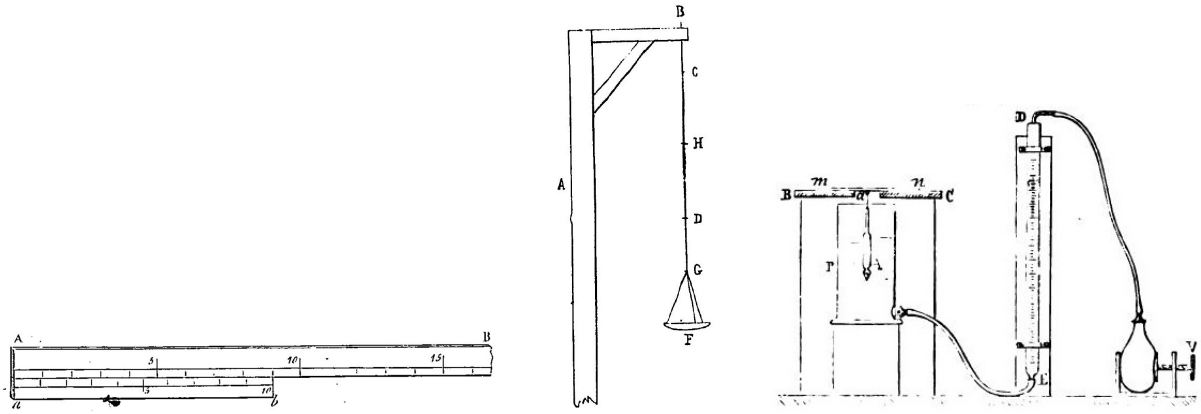


Figure 28: Some instruments who were classified as diagrams

Diagrams classified as instruments Conversely, on the figure below we observe diagrams that were misclassified with a low entropy as instruments. This time, the classifier seems to have been mislead by the "texture" of the figures. The sectional view of an eye on the left depicts well the thickness of the eye, which in addition is highlighted by the dark colors. The middle figure, that represents molecules who are polarized around an iron bar, is again depicted with a dark-ish texture, as well as the shape of a lens on the right.

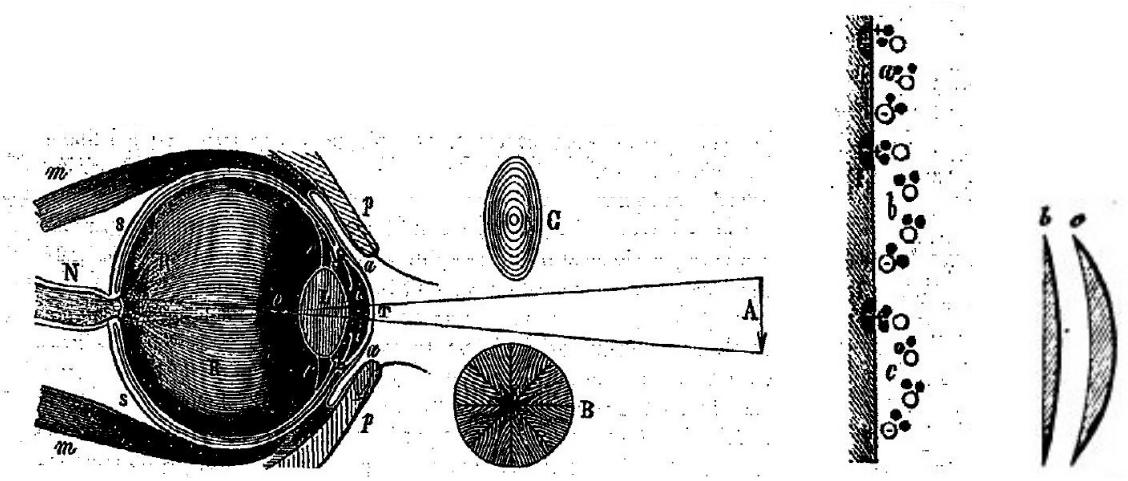


Figure 29: Some diagrams who were classified as instruments

From these observations, we see that diagrams and instruments come in many various shapes, textures, and size. They can even be confused, in some figures that share attributes of both classes. We will analyze in the next section such images who are defined by this ambiguity.

5.2 Ambiguous and mixed figures

As we constructed the training and test sets only with images that are "good representatives" of their class, and because the images in the whole dataset come in many various forms, the classification yielded interesting results for the images who standout from the representatives.

5.2.1 Ambiguous images

Indeed, we first focused on images that yielded the highest-entropy in the set. We consider an image as having "high entropy" if the entropy is at 0.9 bits. This value was set empirically, by looking at the probability distributions yielded by the classifier, for different ranges of entropy.

For images with entropies belonging to $[0.8, 0.9[$ for instance, we found that the most common distributions were $(0.7, 0.3, 0.0)$ and $(0.75, 0.25, 0.0)$ (or $(0.3, 0.7, 0.0)$, $(0.25, 0.75, 0.0)$, etc.). In other words, the classifier predicted the class with 70 to 75% confidence. However, we are more interested in distributions who tend to the uniform distribution $(33\%, 33\%, 33\%)$ or to distributions like $(50\%, 50\%, 0\%)$.

These kinds of distributions appear for the first time for images of entropy ≥ 0.9, which we call **ambiguous images**. 130 ambiguous images in total were found in the classified dataset. For those, the ambiguity happened primarily between diagrams and instruments, for which the confidence levels span from $(60\%, 40\%, 0\%)$ to $(50\%, 50\%, 0\%)$ (Figure 30)

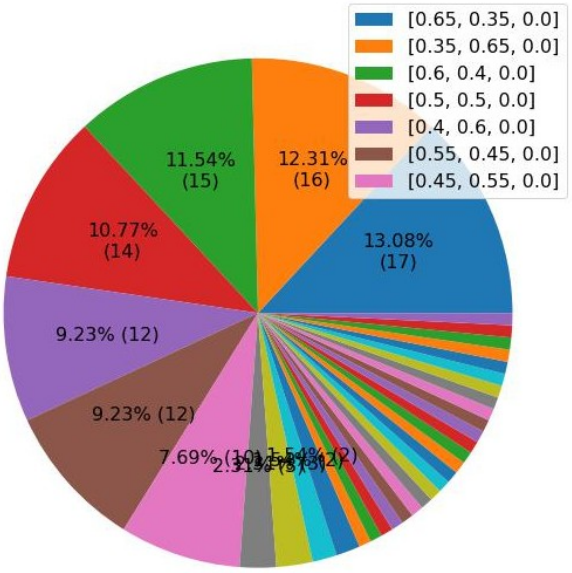


Figure 30: Confidence levels (rounded to the closest 0.05) obtained for the ambiguous images

As explained in 4.2.1, this uncertainty is mathematically conveyed by the entropy of the proba-

bility vector that is outputted by the classifier.

For instance, let us look at the image whose prediction was the most uncertain (Figure 31). The model indeed predicted this image as an instrument, a diagram and a black figure with 19%, 35% and 46% confidence, respectively. This corresponds to an entropy of 1.50 bits, which approaches the maximal value of 1.58 bits.

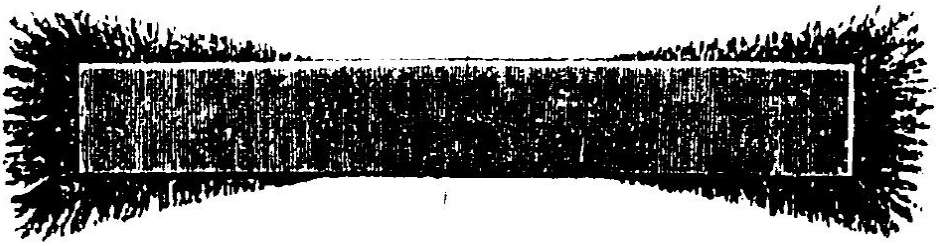
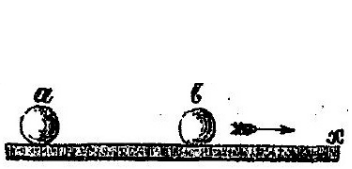
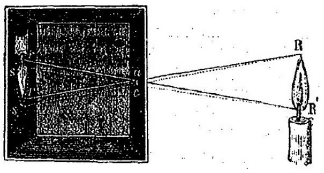


Figure 31: The most uncertain image (19% diagram, 35% instrument, 46% black figure)

As a matter of fact, we can observe that this image has the attributes of all three classes. This figure depicts the magnetic force exerted on a bar magnet, which is represented by these bristling spikes at the ends. Hence it can be seen as an instrument because of the solidity of the bar magnet and as a diagram because of the spikes. Finally, the dark color palette that predominates on the whole figure, and which is a side effect of the scan, leads the classifier to decide it is a black figure. The quality of the scan thus seems to have influenced the classification.



(a) $(32\%, \mathbf{68\%}, 0\%)$



(b) $(19\%, 14\%, \mathbf{67\%})$

Figure 32: Other examples of ambiguous images

Other ambiguous images are shown above. The first one represents a very general idea in physics, the shock of two inelastic masses. We observe in fact that the organization is very simple, and the shapes used are basic, i.e. two balls on a rectilinear surface, which can also be represented well with a simple diagram. The experiment on the right involves again the three classes, with the candle and box as instrument, the light rays as a diagram, and the box as a black figure.

Thus, among the images that the classifier identifies, are the figures that "mix" the attributes of multiples classes. In the next section we will study the most recurrent form of mixed images, which are between diagrams and an instrument, and interpret some examples that could be useful for our understanding of the dataset.

5.2.2 Mixed images

The first major group of mixed images relates to optics. Many figures of experiments conducted in this field indeed describe light rays, that are reflected by objects such as mirrors.

The first image on Figure 33 for instance depicts the making of a "magic bouquet", an optic illusion where a spherical concave mirror is used to reflect an inverted image of a vase placed on a pedestal. The second image shows an experiment that compares the reflective powers of different substances placed on a mirror.

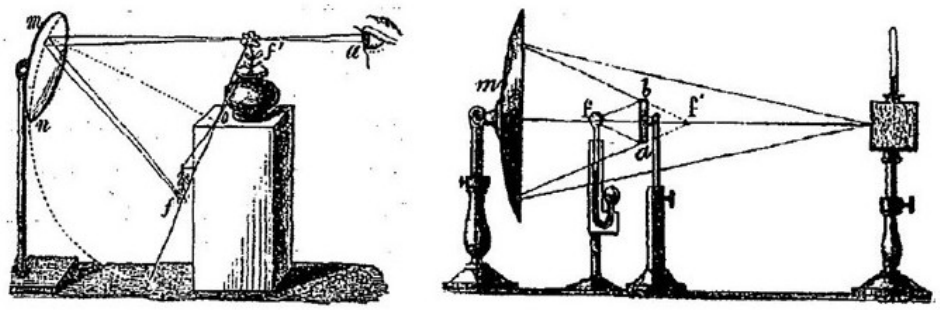


Figure 33: Optics experiments

Electrodynamics is another major field of studies that is likely to give this kind of figures. This time, the diagram aspect of the figure is conveyed by the arrows that mark the direction of the electric current, as we can see on Figure 34.

Both images show the direction of the current induced between two brass tubes.

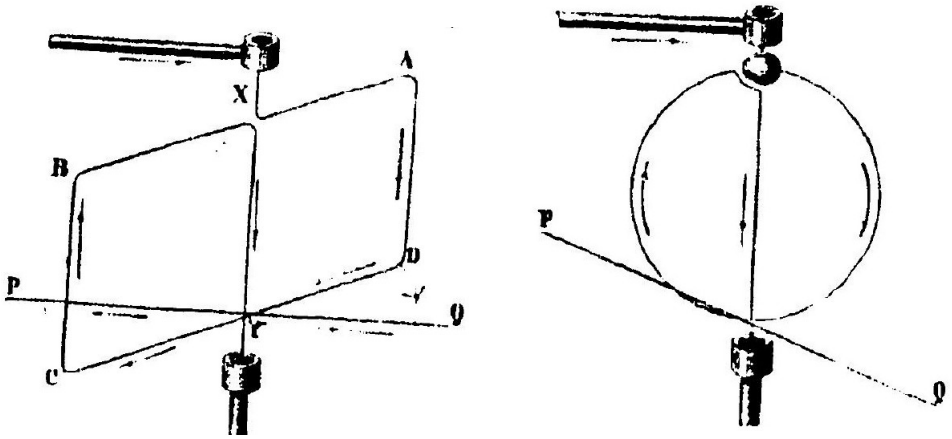


Figure 34: Electrodynamics experiments

In both cases, the core ideas that the figures convey to the reader can only be understood by combining the representation of the experiment setup in three-dimensional space, with the morphology of the physics phenomenon that is at stake.

5.3 Statistics

In this section we resume the results obtained after classification and addition of the `mix` and `ambiguous` classes to our dataset. We plotted below the distribution of the five classes, by year and by author. As shown on Figure 25, the dataset is dominated by instruments, which represent around 49% of the images, followed by diagrams, black figures, mixed images and ambiguous images.

This hierarchy is found again on both plots in general, except in the case of Desains (1857, 1865) which tends to use more black figures than the other authors, and in the case of Jamin (1866, 1879, 1883, 1891, 1896, 1899, 1906) which uses more diagrams, comparatively to other authors, especially towards the end of the century (1896, 1899 and 1906).

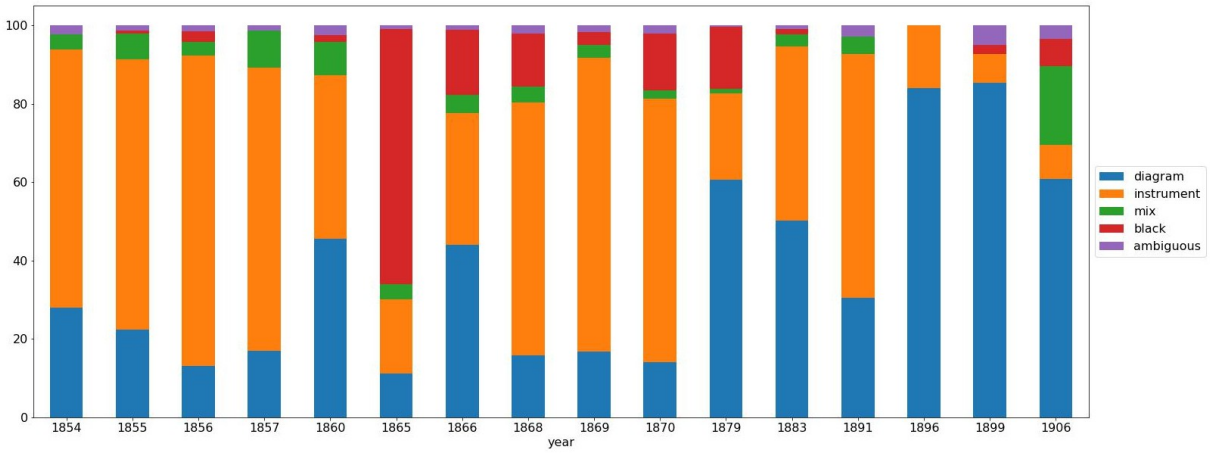


Figure 35: Distribution of the classes (in $\%$) per year

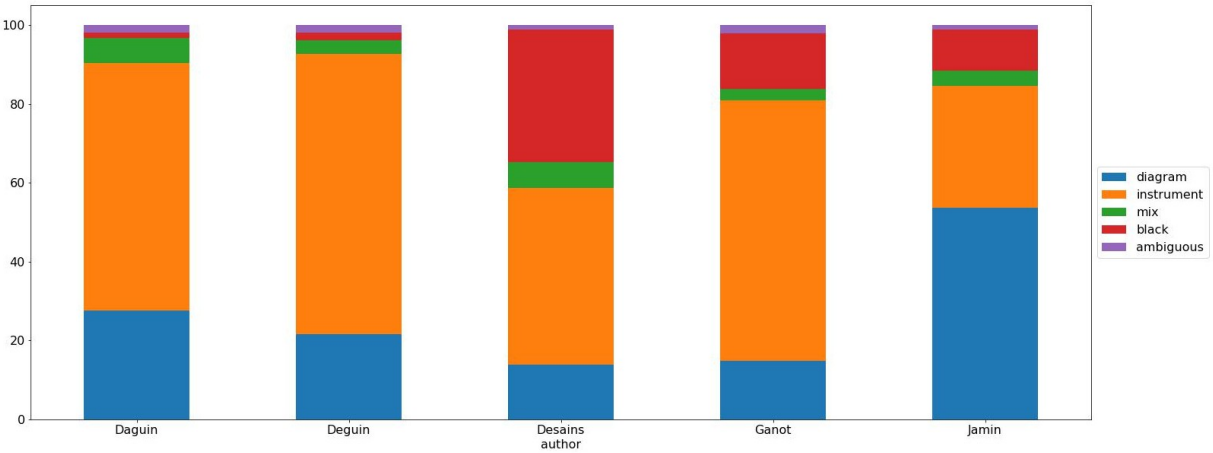


Figure 36: Distribution of the classes (in $\%$) per author

From these few statistics, we already gain a little insight on the habits of some authors of the set and on the evolution of the use of the classes throughout the years. More statistics could be applied to the dataset and reveal interesting habits and how they change over time. In particular, in the last section we will focus on some extensions that we could apply to our dataset to increase our actual knowledge in the field of history of science for this period.

5.4 Future extensions

5.4.1 Identifying visual patterns and features

As we navigated through the dataset, we noticed in particular that some features were used extensively in different books from different decades. One can for instance consider hands (Figure 37), that guide the reader into how to manipulate an instrument for an experiment. Or we could also consider the drawing of the eyes (Figure 38), who give a visual indication for optics experiments, mainly.

One may consider for instance identifying these special features, by applying the extraction and classification algorithms on books from other years, and/or orthogonally from other countries, to construct a little database containing these features, along with the information about the years and the authors. This could then help a researcher/historian that studies, for instance, the propagation of reused patterns in scientific textbooks over different periods of time/different countries.

We may also reuse the extraction and classification to create different datasets for each country over the period we spanned, such as United Kingdom, and then compare for instance which instruments were used both in France and in United Kingdom, and which were not, in some given field. We could conclude from that whether one country was more technologically advanced that another on, say, a given year. This could be even more relevant to compare in the years that span the epoch of the Industrial Revolution, and see which new technologies appeared and when.

5.4.2 Group figures by their description and field of studies

In our dataset, we only used OCR to remove text from the images during processing. We could also try to use it to extract the visual description of each figure. We could take advantage from the location of the text where the author refers to the figure, by recognizing texts such as "(Fig. ..)", or "As shown in Fig. ...". This would involve technologies such as text mining.

And then, based on the descriptions we could refine our classification by creating classes for specific kinds of instruments or diagrams, such as "mirror", "beaker", "tube" and group them by the field of study they are related to. For instance, the class "coil" may be grouped in the category "Electromagnetism".

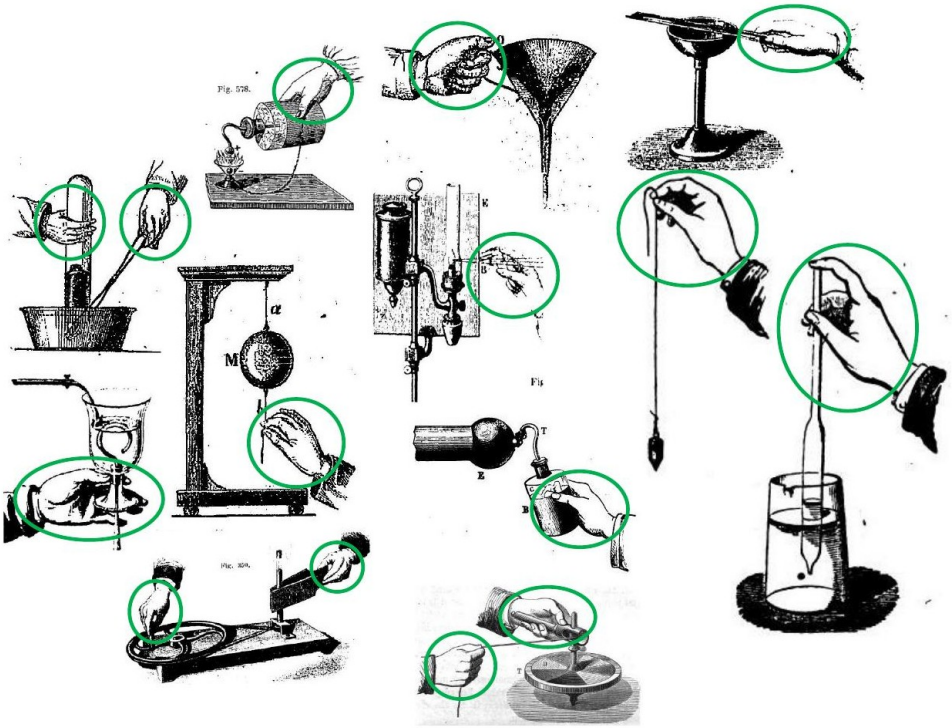


Figure 37: Hands

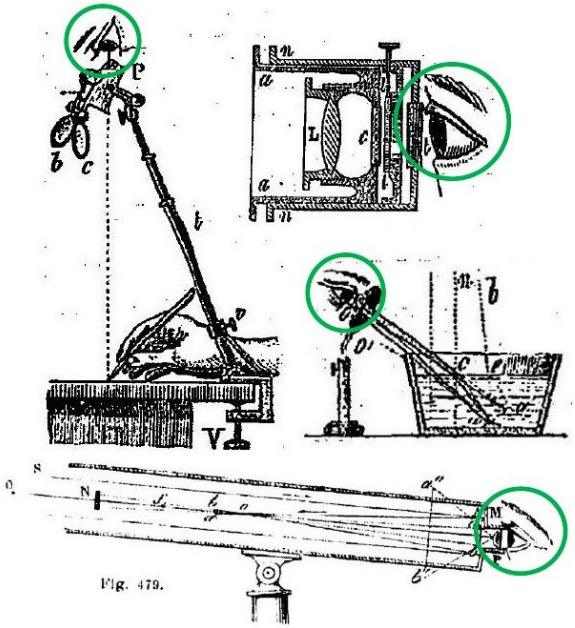


Figure 38: Eyes

Deriving this dataset could again ease the work of an historian, that wishes for instance to study the evolution of the knowledge in a particular domain in a given country.

6 Conclusion

As part of this project, we first built a processing scheme to extract figures from scientific scans, and applied it to a set of French textbooks from the second half of the nineteenth century. Then, we made use of neural networks to build an active learner targeted at image classification, and applied it to the extracted images. Finally, we analyzed the results of the classification and gave an insight of the future applications the dataset could be involved into, as part of extending the actual knowledge in the history of science.

By our evaluation of the extraction scheme, we showed that the errors due to variations in the quality of the scans prevented the pipeline from being completely generic. This could be improved for instance by using optical character recognizers trained for scientific books, to prevent some edges to be recognized as groups of letters.

By analyzing the results of the classification, we had a small glimpse of the overall trend in the usage of figures in the scientific literacy in France, as well as among authors as among years. Through the results, we also managed to understand more precisely the links that can happen between diagrams and instruments. To reduce the ambiguity between these two classes, we could retrain the model on larger datasets from other books, to classify with more confidence the figures.

We finished by reflecting about how this dataset could be expanded further and be of use for future analyses that are dedicated to better understanding the state of the technology over a given period or country.