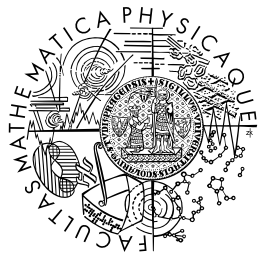


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Lucia Tódová

OCR for tabular data

Department of Software Engineering

Supervisor of the bachelor thesis:	Mgr. Miroslav Kratochvíl
Study programme:	Computer Science
Study branch:	Programming and Software Systems

Prague 2019

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

I would like to express my sincere gratitude to Mgr. Miroslav Kratochvíl, the supervisor of this thesis, for his time, guidance and patience over the last year. I would also like to thank my boyfriend, my family and my friends for their constant support.

Title: OCR for tabular data

Author: Lucia Tódová

Department: Department of Software Engineering

Supervisor: Mgr. Miroslav Kratochvíl, Department of Software Engineering

Abstract: Table recognition is an important tool for digitalizing documents that contain tabular data, which often occur in areas of administration, finances and education. This thesis re-uses existing optical character recognition software to construct a new table recognition algorithm that aims to simplify the digitalization of diverse document types. The resulting algorithm achieves comparable or better results than currently available open-source software. Thesis additionally reviews common methods of OCR software implementation, and measures the influence of image preprocessing quality on the outcome of the table recognition.

Keywords: optical character recognition, document digitalization, table recognition, image processing

Contents

Introduction

Recently, digitalization of documents has become an important part of information systems and related workflows. By digitalization, documents from governmental, administrative, educational and publishing workflows are being processed to a more accessible, searchable and manageable form.

To convert the printed text documents back into digital text form, digitalization tools employ *optical character recognition* (OCR). OCR engines are used to recognize individual text elements of a scanned document and output them in a more suitable format, e.g. as rich text or a searchable PDF file.

Specialized OCR algorithms exist to handle various specific input document layouts where general OCR would produce sub-optimal or unusuable results; e.g. for tickets, passports, car plates or postal envelopes. Tabular documents form a wide, useful category of structured input — table recognition algorithms must be able to process various features found in the tables, including inconsistent border formatting, different alignment, spanning cells, or misaligned columns and rows. Currently, the available table-recognition software is, in most cases, derived from text-oriented OCR algorithms and recognizes only limited amount of all possible table features.

The main goal of this thesis is to explore the existing OCR techniques and software, and create a new table recognition OCR software that can provide better results than the currently available tools. The resulting software uses an external OCR engine and image-processing libraries to preprocess the input images and extract simple text elements, which are then heuristically combined into words, lines, table columns and whole tables. Finally, it produces an easily readable JSON-formatted output that describes the layouts of tabular and textual elements in the processed document. The generalized JSON output may be eventually converted to other, more specific formats.

Related work

There exist several softwares that focus on table recognition. For example, Tabula [Ari] is an under development open-source project focusing on the extraction of tables from PDF files. This software uses the help of the OpenCV library [IC00] and often results in errors when presented with scanned documents, multi-line rows and complex tables. Another examples are OCRSpace [Gmb] and OpenCV [IC00]. Although the focus of these engines is mainly character recognition, they also provide table recognition of specific layouts (e.g. receipts, tickets). Other approaches to table recogni-

tion were presented by Yildiz, Kaiser, and Miksch [YKM05], Kieninger and Dengel [KD98] and Hu et al. [Hu+99]. Even though they proposed new techniques, they are now either outdated or do not provide results of sufficient quality. Satisfactory results are provided by commercial softwares like ABBYY FineReader [ABBa].

Table recognition is also implemented as a feature of the Tesseract library [SS10]. We discuss it in detail later in this thesis (section 2.2.1).

Layout of this thesis This thesis is structured as follows: In Chapter 1, we review the obstacles that complicate character recognition and show existing solutions for some of them; after that we describe several techniques and heuristics for text recognition. In Chapter 2, we describe the implementation of existing table recognition algorithms. Chapter 3 details the implementation of our software, including the heuristic used for table detection and recognition. Performance measurements, results and summary of the improvements are presented in Chapter 4. After the last chapter, we conclude with a brief overview of possible future work.

1. Text recognition algorithms

Text recognition forms the basis of any complex OCR software. Its task is to transform an input image into its segmented form, with segments containing information about individual characters and their positions.

We begin this chapter by presenting various aspects that impact the quality of the input image, which we have observed to affect the results of the text recognition algorithms. To reduce the effects of these aspects, we also focus on the related task of preprocessing the input image. After that, we analyze the individual steps of the text detection process and review the existing techniques used for this purpose.

1.1 Common obstacles of successful text recognition

Even for human perception, the task of recognizing text in a complex document might become difficult due to various factors. Text recognition engines work similarly or, in some specific cases trivial for human perception, even worse.

Common difficulties faced by text recognition algorithms have been reviewed by Bieniecki, Grabowski, and Rozenberg [BGR07]. These can be categorized into following major classes:

Font face variability Text documents contain a lot of different fonts and styles, including different headings, body text, footer and header sizes along with bold, italics and underlined characters, differently colored fonts and even custom-made fonts. Furthermore, when it comes to handwritten text, each character is slightly different and the text contains a lot of inaccuracies. To hold the information about every possible font and style would therefore be impractical for any OCR software, and, in case of custom-made fonts or handwritten text, even impossible. Therefore, an OCR software needs to use heuristics to match the actual symbol it recognized to its expectation of the character shape. This does not always produce the desired results. In specific cases, especially in decorative fonts or handwritten text, characters like S or 5, 0 or O and I or l are hard to distinguish even by human perception. This often results in the failure of text detection algorithm for these specific characters.

Scan quality The optimal results of image acquisition techniques, like scanning, are high-contrast images with no inaccuracies. However, the scanning process often disrupts the input image, which results in low contrast, insufficient sharpness, pixelation, noise and disruption of lines. This is usually due to a low resolution scan or human made mistakes during scanning. We use the techniques of preprocessing to try and reconstruct this error prone image into a suitable input for text recognition engines.

Skew problems Photographing or scanning an image often results in minor human made skew and rotation inaccuracies. However, most of the text recognition engines already assume the input to be properly vertically and horizontally aligned for the simplicity and lower time complexity of its algorithm. This causes the engines to fail on skewed images. Therefore, we attempt to deskew the input images during preprocessing with the use of deskewing techniques.

Colors To distinguish characters from the image background, OCR engines use various techniques based on the color values of the individual pixels, their contrast, brightness, etc. When it comes to monochrome one images, to determine this division of image pixels is a simple task. With colored inputs, however, many complications arise. For example, recognition of green text on a red background fails when presented to a method based on contrast. This and improved time complexity of the algorithm is the reason why images often undergo the process of binarization.

These difficulties motivate the common OCR implementations to work with optimistic assumptions about documents, such as that the document has only one column, that it is not handwritten, and that its lines are horizontally aligned.

Several existing image processing techniques may be used to eliminate the effects of most of these obstacles in the attempt to increase the performance of OCR engines. We review the most important of them in the next section.

1.2 Preprocessing for OCR

Many of the existing OCR engines are equipped with several simple built-in preprocessing methods. It is, however, often beneficial to preprocess the image using specialized tools in order to obtain a better input for the OCR. In this section, we discuss the most effective and commonly used image transformations for this purpose.

1.2.1 Scaling

In computer graphics, scaling refers to the process of changing the resolution of a digital image while preserving its data content. In OCR, the purpose of this method is to provide an input of viable size for existing OCR engines. For example, OCR engines likes Tesseract [Pac96] or ABBYY FineReader [ABBa] encourage their users to use 300 dpi images — this is due to the fact that the classification of newly recognized characters requires a large set of training data which they are then compared to; which are, in this case, most often obtained from 300 dpi images. 300 dpi is claimed to be a resolution high enough for a clear recognition of character features in common text, but also low enough to maintain a reasonable execution time of the algorithm.

If the user has provided us with an image of a different resolution, the image therefore needs to be either upscaled (if the dpi of the input image was lower) or downscaled (if the dpi of the input image was higher). For these purposes, interpolation-based algorithms are used.

Interpolation attempts to minimize the aliasing effect of the resulting image, so the scaled picture does not appear visibly pixelated and its edges are not jagged. It is based on using known data to estimate values at unknown points. It specifically approximates the color and intensity of the resulting pixel based on the values of surrounding pixels. When downscaling, this technique therefore computes one color value for each set of pixels, which are the color values of the pixels in the new image. When upscaling, it computes color values of the new pixels by approximating the values of the old pixels around them.

Existing interpolation algorithms can be grouped into two categories: adaptive and non-adaptive.

Adaptive methods change the way they treat various pixels depending on whether they are a part of a smooth texture or a sharp edge. They are primarily designed to minimize the presence of interpolation artifacts in regions where they are most apparent. Examples of adaptive methods include e.g. Spline, Sinc, Lanczos or Discrete Wavelet Transforms [Fad14].

Non-adaptive methods, on the other hand, treat all pixels equally. Their complexity and classification depends on the size of pixel neighborhood they interpolate [Fad14]. The effects of several of these methods are detailed in fig. 1.1.

Although the results of adaptive methods are more accurate, they are not generally used for OCR preprocessing due to their high time complexity. Therefore, the most popular decision for general cases is the use of Bicubic interpolation method [Han13].

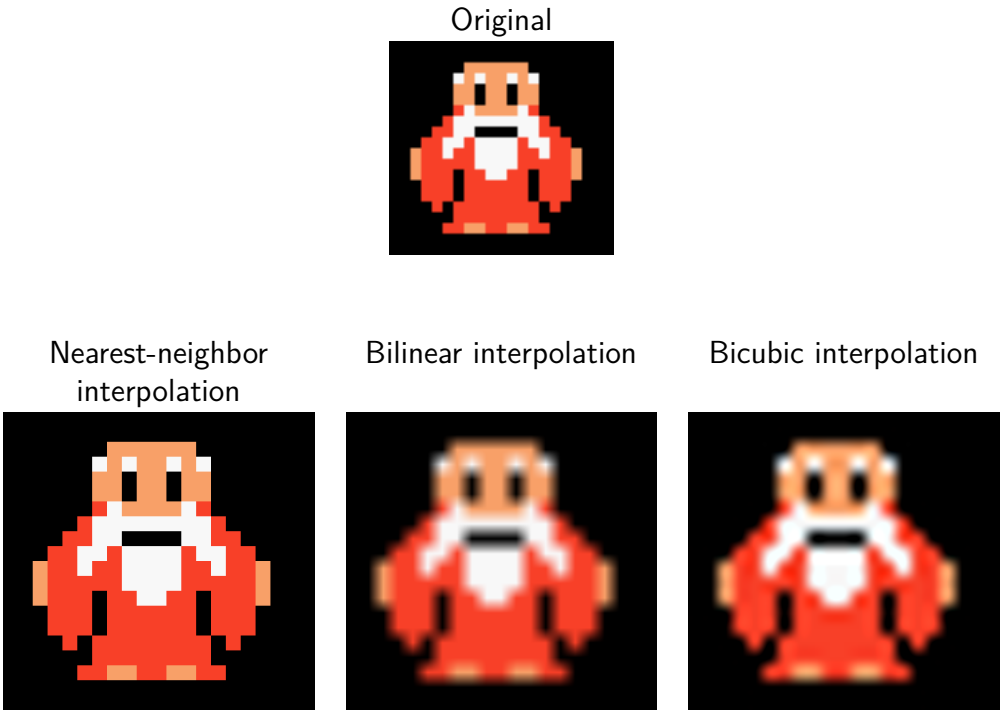


Figure 1.1: Comparison of results of several non-adaptive interpolation methods, scaled 4× (image by Demofox [Dem]).

1.2.2 Contrast enhancement

Contrast of an image is the difference in luminance or colour that makes objects in the same field of view distinguishable. Low contrast often results in blending of edges during recognition, which decreases the accuracy of OCR.

An image histogram is used to improve this aspect. A histogram is a representation of the distribution of data and, in case of image histograms, it represents the tonal distribution of an image — x-axis stands for all available tonal levels, and y-axis represents the number of pixels for each tonal level. In colored (RGB) images, the histogram is represented by three separate histograms, one for each color component. By spreading out the most frequent intensity values in the histogram, as shown in fig. 1.2, the contrast of the image increases.

Histogram stretching methods are categorized as either methods of linear or non-linear stretching [AaKK10], depending on the forms of functions they use. Although linear stretching methods are easier to implement and do not cause the objects in the original image to lose their brightness values, non-linear stretching methods were shown to produce better results in gen-

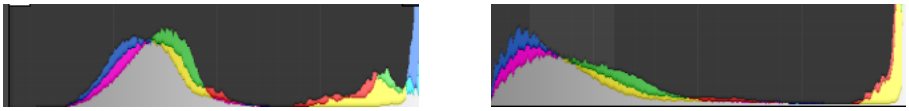


Figure 1.2: Histogram-based comparison of image contrast. Left: low-contrast image histogram, right: high-contrast image histogram.

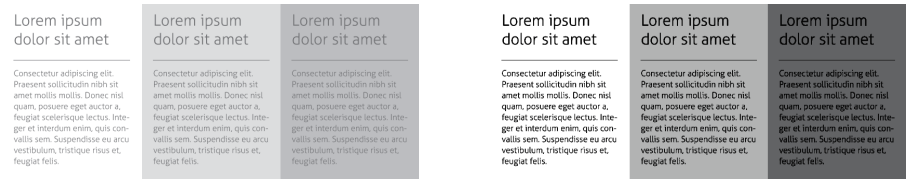


Figure 1.3: Results of histogram equalization. Left: original image, right: processed image.

eral [CJP]. Despite of that, linear stretching methods still deliver satisfactory results on images with Gaussian or near-Gaussian histograms.

The most popular choice for contrast enhancement for OCR is a non-linear stretching technique called *histogram equalization* [Dor+15]. Although it produces unrealistic and over-enhanced effects in photographs, the over-processing actually suits the text data. The typical output of histogram equalization on textual data is displayed in fig. 1.3.

Suganya, Gayathri, and Mohanapriya [SGM+13] describe other methods of contrast enhancement, like CLAHE, Dynamic Stochastic Resonance and generic transforms, such as Fourier, Discrete Cosine, and Wavelet. For complex text recognition cases, CLAHE method can be used with satisfactory results.

1.2.3 Binarization

To recognize characters in an image, an OCR software must distinguish the background pixels from the actual pixels that belong to the characters. To determine the boundaries between these two groups in a colored image is a complex task due to the need of processing the whole RGB information of each pixel. However, there might be a loss of information when converting an RGB image to grayscale/monochrome. Therefore, the approach to the conversion can not be naive and must consider unusually colored images (e.g. green font on a red background).

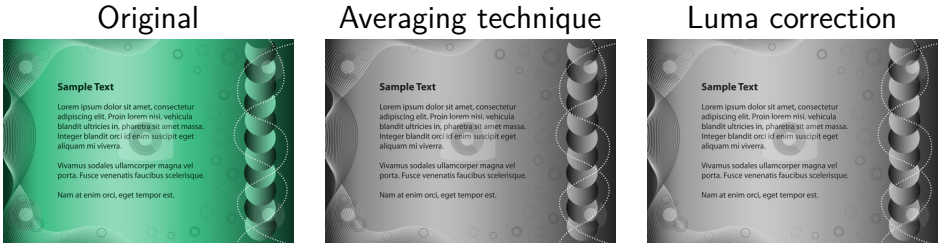


Figure 1.4: Comparison of grayscale techniques.

Grayscale conversion

Grayscale conversion is the process of assigning each pixel of an image a gray value, while preserving as much information of the original image as possible. The simplest approach to grayscale conversion is the averaging approach, which assigns each pixel the average of its R, G and B values. However, this approach causes the blending of colors of similar luminance contrast (such as the already mentioned greens and reds). Additionally, human perception of colors is non-uniform — green color is perceived more strongly than red, and red more strongly than blue. For these reasons, a correction formula (often called *luma* [KC12]) that weights each color component differently is used. Slight differences in the results of these techniques are demonstrated in fig. 1.4.

Other more complicated methods were reviewed by Ĉadík [Ĉad08].

Threshold-based binarization

Binarization of an image is usually performed on grayscale images, which we obtain from a colored input by previously mentioned techniques. Most of the binarization methods work on the basis of a *threshold* — a value that determines whether the pixel should be black or white in the resulting binarized image. Various binarization methods differ in the way that this threshold is calculated and used.

Global thresholding [LKR13] is the simplest approach to binarization. It works by choosing a threshold value and iterating over the whole image, comparing the value of each pixel to this threshold. However, as the brightness and contrast of different images vary, it is generally impossible to choose a threshold suitable for all images.

More complex methods that aim to correct the problems of global thresholding exist. Some of these include:

- Otsu's method for global threshold estimation [You11]

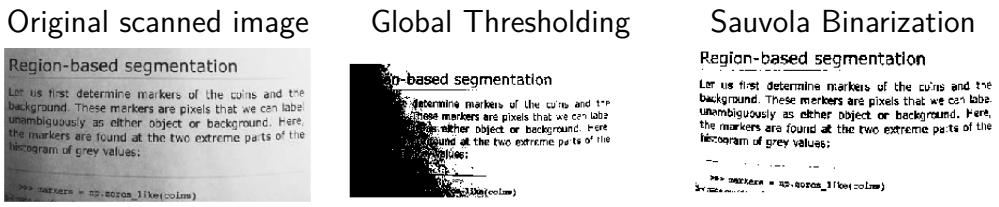


Figure 1.5: Comparison of the outputs of binarization methods on an unevenly lit image (images by Pandey [Pan]).

- Local-thresholding methods [SS04]:
  - Niblack method
  - Bernsen method
  - Sauvola method

Otsu's method statistically computes a global threshold by a dynamical classification of image pixels into background and foreground pixels. Although simple and easy to implement, but, as any global method, it is inherently unable to fix high variance or uneven spots in an image background. Local-thresholding methods aim to solve this issue by allowing different threshold values in different parts of the image. They differ in the way the existing thresholds are computed — for example, Niblack's method [SP00] uses mean and standard deviation of surrounding pixels for each pixel, Sauvola method improves it by checking for blank regions and Bernsen's method by optimizing Niblack's computations.

Visual differences between the mentioned methods are displayed in fig. 1.5. Comprehensive reviews of other, more complex methods are also available [SS04].

1.2.4 Deskewing

Skewed images are probably the most common problem that many OCR engines struggle with. They usually appear when an image is scanned or photographed from an angle.

Negative effects of skewed images are most apparent in results of page segmentation (see section 1.3.1), as this process often relies on the characters to be properly vertically and horizontally aligned.

In this section, we present various methods of image deskewing. All of them work (for more accurate results) after binarization of the image and generally assume the skew angle to be no more than 20°. Although algorithms that work on greater skew angles also exist [Ros+14a], they are not as widely

Method	Performance	Max. skew	Image handling	Setbacks
Hough transform	slow	90°	sufficient	–
Projection profiles	slow	10°	problematic	–
Cross correlation	relatively fast	10°	problematic	works only on simple documents
Connected component clustering	clustering-dependent	20°	good	requires high-quality input

Table 1.1: The advantages and disadvantages of different deskewing methods.

used for the purposes of deskewing scanned documents, which are generally only slightly tweaked.

The most effective and popular algorithms include the following [Ros+14b]:

- Hough transform [SKK16]
- Projection profile method [Ros+14a]
- Cross correlation [Ros+14a]
- Connected component clustering (or nearest-neighbor clustering) [LT03]

Hough transform applies feature extraction to the image, takes the line with the highest pixel count, and computes the skew of the whole document using this line. Projection profile method rotates the image multiple times and computes the histogram along horizontal scan lines. The skew of the rotated image where the histogram has the most peaks and valleys is determined to be the skew of the input image. Cross correlation method is based on applying the projection profile method to multiple vertical segments, and Connected component clustering method is based on finding the coherency between connected components of the image and determining their common skew. We present the overview and comparison of these methods in table 1.1.

Other methods of deskewing may also be used. These include Fourier [SK13], Wavelet Decomposition, Radon Transform, PCP, one-pass or multi-pass skew correction, morphology algorithms, etc.

1.2.5 Noise reduction

Image noise is an usually unwanted random variation of brightness or color information in images, similar to film grain in digital cameras. It is often

Figure 1.6: Image deskewing illustrated. Left: original image, right: deskewed image.

caused by the inevitable physical variance in amount of light recorded by a scanner or a digital camera.

The presence of noise negatively affects the quality of the image by e.g. disrupting sharp edges of elements. This causes the process of specific OCR steps, such as edge detection, to output less accurate data, which can lead to the failure of character recognition.

Common image processing filters are used [MK11] when performing noise reduction. These can be categorized as either linear or non-linear, depending on their linearity relationships.

Linear filtering, such as mean filtering, Gaussian filtering or Wiener filtering, processes each pixel by linearly combining the values of its neighborhood pixels. This, however, often results in blurred edges and smoother fine details. As these methods require only a small amount of computations (which leads to increased speed and simple implementation), they are still widely used.

Non-linear filtering, on the other hand, renders an output which is not a linear function of its input. Methods based on non-linear filtering used for denoising purposes include filters like *median filter*, which computes the value of the resulting pixel from a local median, or *non-local means method* [BCM05], which is based on taking a mean value of all pixels in the image weighted by their similarity to the target pixel. Both methods produce sharper and clearer images than linear filters at the expense of increased run time.

We provide a comparison of the above mentioned methods in fig. 1.7.

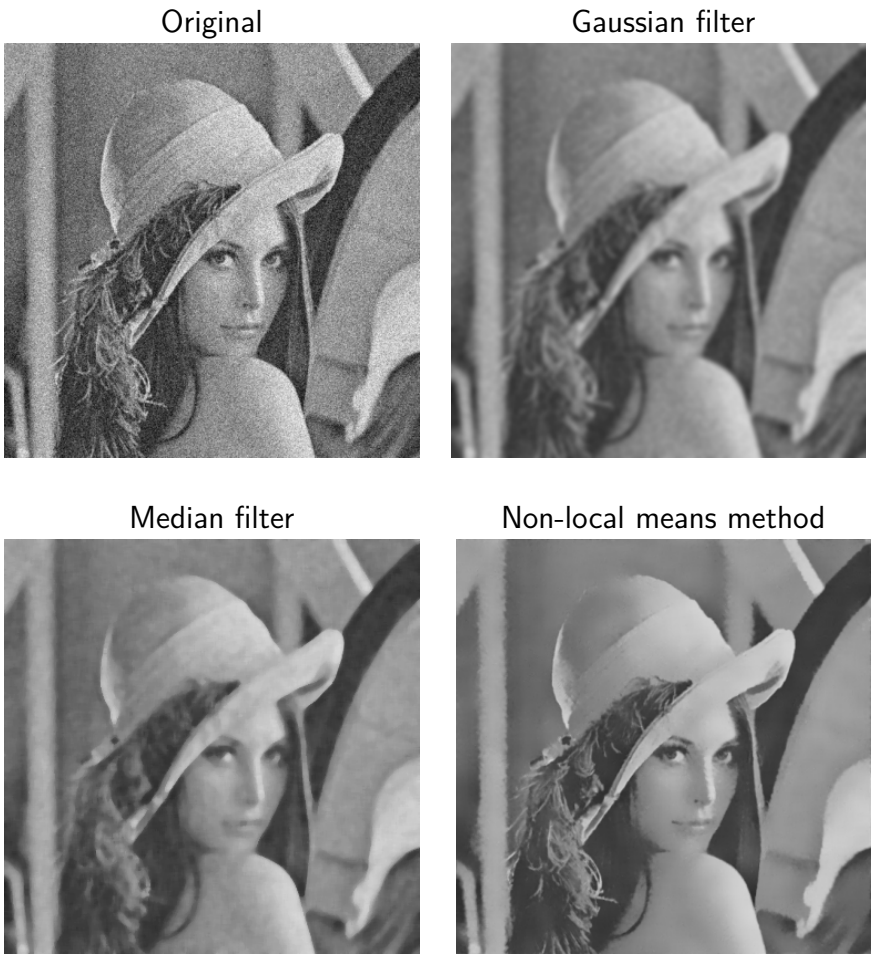


Figure 1.7: Comparison of denoising techniques.

1.2.6 Scanning border reduction

Scanned documents often have visible dark borders (*marginal noise*). These originate either during scanning (due to the presence of neighboring pages) or are an unwanted result of the binarization process.

Marginal noise can cause negative effects on the results of OCR, as it may either be interpreted as a part of character symbols due to its structure, or it may interfere with other page elements.

Algorithms focusing on marginal noise reduction often perceive it as a dark connected component of the page. Therefore, they use connected component extraction to recognize and further remove these components. An example of a connected component based scanning border reduction algorithm was presented by Shafait and Breuel [SB09].

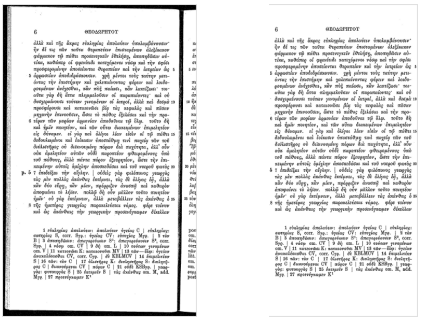


Figure 1.8: Removal of scanning artifacts, as presented by Packard [Pac96]. Left: original, right: removed scanning borders.

Another slightly different approach was described by Peerawit and Kawtrakul [PK04]. It is based on edge density calculations, and the fact that text areas have generally a much lower density than edge areas. This approach uses an edge detector to examine the presence of both vertical and horizontal noise. Its removal is then performed by filters or other heuristic algorithms.

We demonstrate the effects of scanning border reduction in fig. 1.8.

1.3 Text detection process

The goal of text detection is to recognize and classify individual textual elements of an image document and interpret them as textual contents of the document.

There exist two main approaches for text detection in OCR:

1. heuristic-based approaches

2. neural-network-based approaches

In this paper, we concentrate on the heuristic-based approach of the OCR problem, as we use it in our implementation. However, neural network recognition is a promising field of study and also deserves a mention.

Although OCR engines greatly differ in the way their heuristic approaches work, they generally contain the following steps:

1. *Page segmentation* — logical and topological division of the page into multiple smaller segments.

2. *Feature extraction* — simplification of the extracted elements of the image to small, informative inputs viable for standalone recognition.

3. *Character classification.*

We analyze these steps in the individual following subsections.

1.3.1 Page segmentation

The task of page segmentation is to extract page sections that contain uniformly-formatted bodies of characters (e.g. columns or paragraphs) that can be easily passed to later steps of text recognition.

Page segmentation can be performed in three different ways [SKB08] — either we start with an image as a whole and recursively divide it into parts until certain criterion is met (top-down), or we start with individual pixels and group them according to similarity of their attributes (bottom-up), or we use a combination of these methods.

In this section, we mostly focus on a method used by the Tesseract engine [Smi09]. For comparison, we also overview several other existing techniques.

Tesseract page segmentation

In this section, we are concerned with a page segmentation technique proposed by Smith [Smi09]. This method is based on detecting tab-stops.

Tab-stops are used in word processors to enable obvious alignment of text, and are present in documents as margins, column edges, or indentations. All of them are placed at fixed x-positions, at which edges or centers of text lines are vertically aligned.

Following are the individual steps of the tab-stop based page segmentation algorithm (fig. 1.9):

1. Obtain the input image document (fig. 1.9a).
2. Find the connected components of the document image.
3. Select candidate tab-stop components from connected components by analyzing their alignment.
4. Group the discovered candidate tab-stop components into vertical lines (fig. 1.9b).
5. Create tab-lines by examining the alignment of the vertical lines.
6. Create *column partitions* from the connected components of the layout analysis so they do not cross any tab-lines and are of the same type (e.g. text, image, form) (fig. 1.9c).

7. Merge column partitions into columns (fig. 1.9d).

8. Based on the positioning of columns and heuristics (different types of fonts, line spacing), merge or divide existing column partitions into segmented blocks. (fig. 1.9e).

This method is implemented and used by the Tesseract engine [Pac] with the help of the Leptonica library [Blo01]. It claims to yield satisfactory results, given the input document is properly preprocessed.

Other page segmentation techniques

In this section, we briefly overview the principles of the most popular page segmentation techniques, already described by Shafait, Keysers, and Breuel [SKB08], such as RXYC algorithm, Smearing algorithm, Voronoi diagram based algorithm, Docstrum algorithm, Whitespace analysis and Contrained text-line detection. Their core ideas greatly differ and therefore, their behaviours vary depending on the input.

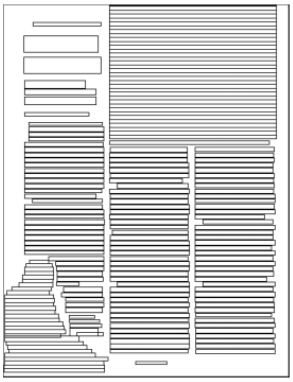
For example, RXYC algorithm perceives the input page as a tree and recursively splits its nodes, dividing the page into smaller segments. Smearing algorithm, on the other hand, is a bottom-up approach based on linking black areas together. Voronoi and Docstrum algorithms are both based on the extraction of connected components. However, Voronoi algorithm determines the individual page segments from the edges of these components, while the Docstrum algorithm focuses on identifying fonts and styles and tries to heuristically determine the segments. Whitespace analysis and Constrained text-line detection both fixate on detecting as much of the white background as possible and removing it to uncover the existing characters. They differ in the way they detect the background.

The most accurate option for segmentation of diverse text documents was determined to probably be Constrained text-line detection, as it has provided the best results on a heterogeneous collection of documents. However, Docstrum and Voronoi algorithms both produced satisfactory results when presented with homogeneous documents. Voronoi algorithm also seemed to work well on documents with non-Manhattan layout.

Other approaches did not render as accurate results in general cases. However, they are still used for specific purposes, e.g. Smearing algorithm for vehicle plate recognition.

(a) The original input image.

(b) Connected components grouped into vertical lines. Upon close inspection, we can see that each page column contains multiple connected components.



(c) Column partitions created from the connected components and individual tab-lines.

(d) Column partitions merged into columns. Upon close inspection, we can see that the recognized columns are almost identical to the actual page columns.

(e) The final segmented blocks.

Figure 1.9: The process of Tesseract table recognition [Smi09].

1.3.2 Feature extraction

Upon extracting elements by page segmentation, we are left with an unnecessary amount of information. Trying to run a text recognition algorithm on all of it would be significantly time consuming. Additionally, as the character recognition is based on specific features for each symbol, the extra information might not suit the character features of the correct candidate. The candidate might get undesirably discarded, which might result in decreased accuracy.

For these reasons, a set of features is extracted for each element that distinguishes it from other element classes while keeping characteristic differences. The set of features that represents each characters needs to be picked carefully, so that the features define the shape of a character precisely a uniquely, but are also the smallest set possible to prevent increased memory usage and time complexity.

There exist three methods for performing feature extraction, described by both Chaudhuri et al. [Cha+17] and Kumar and Bhatia [KB14]. They differ in the ways the features are derived from the character, in reconstructability (whether the image can be reconstructed to its previous version solely from its features), in invariance to transformations and many more. We describe these methods in the following list:

- *Global transformation and series expansion*

  These include methods like Fourier or Gabor transform, Fourier Descriptor, Wavelets, Moments and Karhunen-Loeve expansion, and are based on the representation of continuous signals by a linear combination of simpler functions. The coefficients of the linear combination provide a compact encoding known as transformation or series expansion. Therefore, deformations like translation and rotations are invariant under these methods. The downside of these approaches is an increased time complexity, as these methods require a number of non-trivial computations.

- *Statistical representation*

  Unlike transforms, these features are based on the statistical distribution of points in an element. Their extraction is fast and they also usually take care of different font and style variations. Methods based on this representation are e.g. *zoning* (where the frame containing the character is split into different zones, which are then analyzed for densities of points and strokes), *crossings and distances* (crossings — the number of crossings of a character along vertical and horizontal lines;

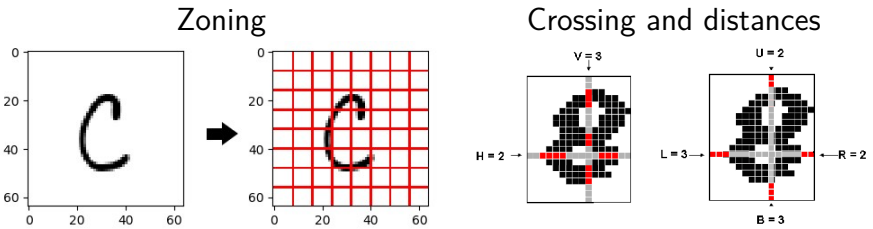


Figure 1.10: Example of two feature extraction techniques based on statistical representation (images by Luc [Luc18]).

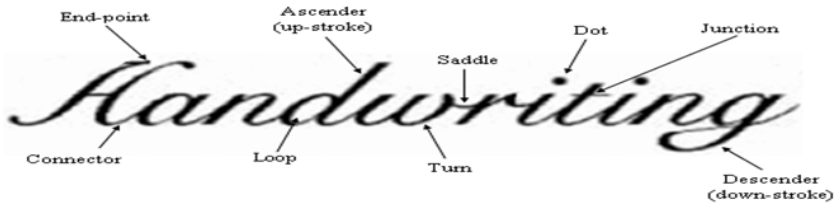


Figure 1.11: Example of several geometrical and topological features, as presented by Abdul-Rahaim [AR15].

distances — the distances of character pixels from frame boundaries), and others, like projections, characteristic loci etc. We display a few of these techniques in fig. 1.10.

The downside of this approach is the invariance to transforms — if a character is only slightly skewed, the feature extraction results greatly differ.

- *Geometrical and topological representation*

  These features are based on the geometrical and topological properties of the individual elements, like basic line types that form a character skeleton, or features of the character contour like extreme points, maxima, minima, cross points, line ends, isolated dots, different types of strokes and many more. We display a few of them in fig. 1.11.

  They highly tolerate various distortions and style variations, and their output can be assigned into a feature extraction vector.

1.3.3 Character classification

Character classification is the process of obtaining a digital form of a character from its features. To classify the characters, OCR engines widely use

the methodologies of pattern recognition, which assign an unknown sample into one of its predefined classes. These approaches are not necessarily independent of each other and OCR engines frequently combine subsets of them to achieve the most accurate results.

Classification is often approached by one of the following types of techniques:

- Template matching
- Statistical techniques
- Techniques inspired by machine learning

In this section, we discuss their core principles.

The first mentioned approach to classification is by *template matching* [Mud+07], visualized in fig. 1.12. It is based on the existence of a database of predefined *templates* — multiple bitmaps containing characters of the alphabet. Improved version of this method have extended this database to include numbers and special characters. Once a character is detected, it is passed to the algorithm and for every existing template, its similarity ratio is calculated. The template with the greatest ratio is then assumed to be the recognized character. This method has various implementations depending on how the ratio is calculated — for example, cross correlation, normalized correlation or euclidean formula can be used.

Although the implementation of this method is very simple, even small disfigurements and noise can greatly affect its efficiency. Also, in case of template matching, a feature extraction would be unnecessary as all templates must be created manually.

The second approach mentioned was by using *statistical techniques* [DM11], like Bayes classifier, Hidden Markov Modeling or Nearest Neighbor. These methods are based on feature extraction for each character. To classify an unknown character, its extracted features are compared to the features of training data with the help of statistical decision functions. We show some of the possible extracted features for these methods in fig. 1.13.

The problem with these methods is that they have no information about whole-part relations. Moreover, they greatly rely on the correctness and sufficiency of training data.

The last approach discussed are techniques inspired by *machine learning* [Seb02]. These are based on neural networks, and their ability to "learn" and, over time, build a model of training data that is further used for categorizing their input. There is no requirement for manual categorization of individual data features, as the neural networks are capable of performing the feature extraction from training data on their own.

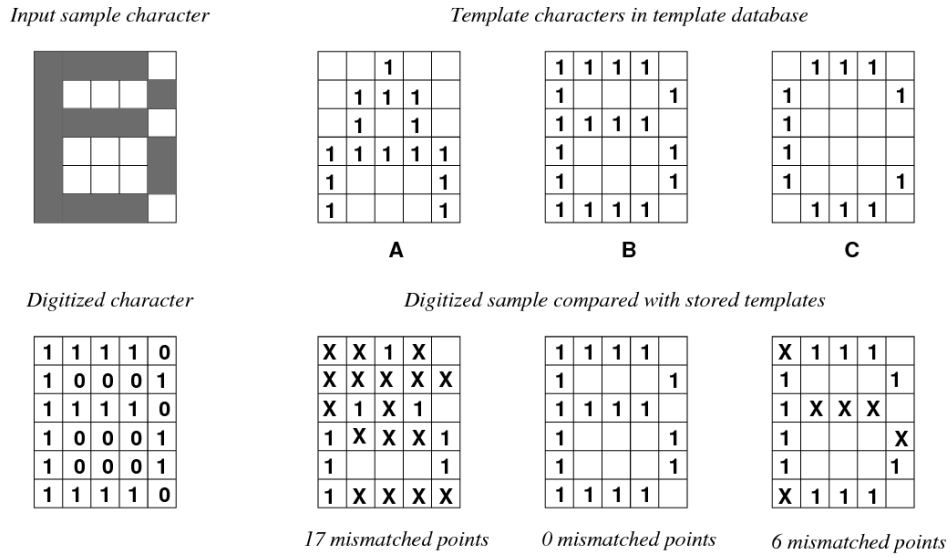


Figure 1.12: Examples of template matching technique as described by Ning [Nin93].

In the following list, we present an overview of some of the most popular machine learning algorithms, as described by Bhavsar and Ganatra [BG12]:

- *KNN classification algorithm* — the algorithm chooses K objects from the training data that are the most similar to the input image. After that, the image is assigned to the class most common among its K nearest neighbors, as shown in fig. 1.14.
- *Support vector machine algorithm* — given a set of data, each marked as belonging to one of two categories, this algorithm builds a model based on complex mathematical functions that is capable of assigning a newly received data into one of the categories.
- *Decision tree* — builds a tree from top to bottom, with each node representing a characteristics feature and its children the values of the feature (e.g. number of horizontal lines).

Other machine learning algorithms are e.g. Naive Bayes [NJ02], Logistic Regression [NJ02], Random Forest [Seg04], etc.

1.4 Available OCR software

OCR engines greatly differ not only in the accuracy of their recognition, but also in the features that they provide. Although OCR engines focusing only

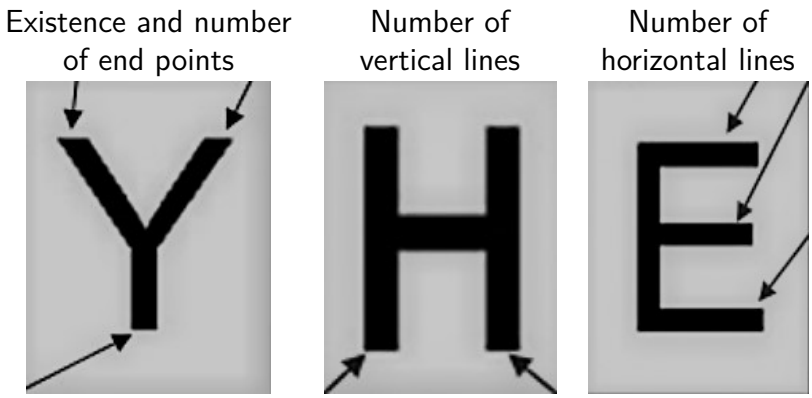


Figure 1.13: A few of the possible extracted features in statistical techniques [VK15].

Name	**Year**	**Code**	**Languages**	**Layout analysis**	**Neural nets**
Tesseract	1985	C++, C	100+	•	•
Ocrad	2003	C++	Latin alphabet	•	–
GOCR	2000	C	20+	–	–
CuneiForm	1996	C, C++	28	–	–
OCRopus	2007	Python	Latin script	–	•

Table 1.2: Overview of available OCR software

on text recognition exist, most of the modern software also contains recognition of other document elements, like images, forms, tables and many more. Further differences include the presence of preprocessing options, support for various file extensions, recognition of different fonts, including handwritten documents, etc. Furthermore, there exist OCR engines specialized in processing only specific types of images, such as automatic number plate recognition engines or ticket validation engines.

In table 1.2, we overview the most popular and widely used free OCR engines. Commercial software used for the purposes of OCR, such as ABBYY FineReader [ABBb], ReadIris [Gro] and OmniPage [Com], also deserves a mention. However, although some of these engines produce overall better results than the free software, they are unusable for the purposes of this thesis.

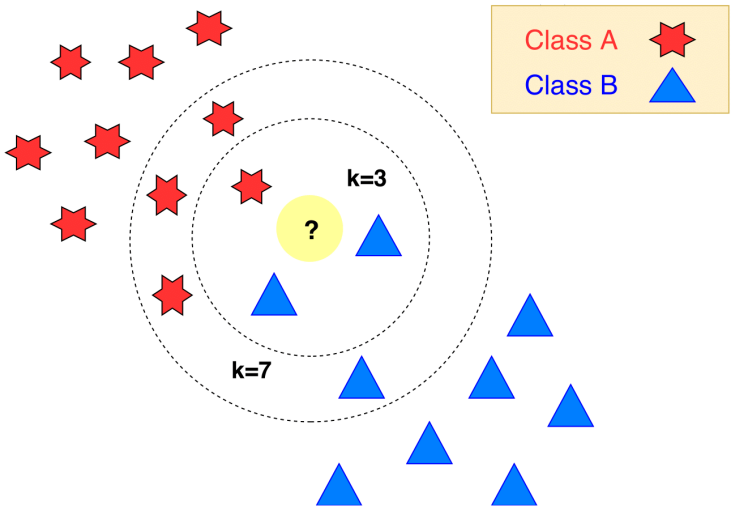


Figure 1.14: KNN algorithm: with $k = 3$, the algorithm assigns the new element to class B; with $k = 7$ to class A.

1.4.1 Tesseract

Originally developed by Hewlett-Packard Company around 1990 [Pac], Tesseract is one of the most robust and accurate open-source OCR engines. In the beginning, Tesseract could only accept TIFF images containing simple one-column text in English language. Since then, it has undergone a lot of improvements and added many features. As of today, Tesseract supports the processing of multi-columned documents, claims to support over 100 languages (including right-to-left text such as Arabic or Hebrew), works on different input and output image formats (with the help of the Leptonica library [Blo01]) and is available for Windows, Linux and even Mac OS. In its latest version (4.0.0), Tesseract also added a new neural network (specifically a LSTM network) focused on line recognition.

Tesseract does not have a GUI and works as a command line program. It is used mostly for development purposes and provides an OCR engine (libtesseract) that gives developers a chance to create their own applications using tesseract API. Also, Tesseract contains no preprocessing algorithms. It advises users to preprocess the input images themselves [Pac96].

Based on our observation, Tesseract provides the most features and in general cases, provides the best accuracy of element recognition among all the other open-source OCR engines. Moreover, we found its documentation, information about individual functions and even examples of use cases well-organized and easy to understand.

This is the reason why we use the recognition and features of this engine

in our implementation.

Tesseract character recognition

In this section, we present an overview of the way Tesseract detects symbols and so-called *textlines*, which are logical rows consisting of character symbols. Thoroughly analyzed by Smith [Smi07], this algorithm has the following steps:

1. *Page segmentation*

   Already discussed in section 1.3.1, the Tesseract page segmentation algorithm is first performed to provide us with text regions of roughly uniform text size.

2. *Line finding*

   Upon obtaining text regions, heuristics like height filter and median filter are used to estimate the height of characters and remove noise, punctuation and other obstacles. The filtered so-called *blobs* are then merged into lines by their y-coordinate and position, with focus on slight possible skew inaccuracies.

   Once the blobs are assigned to lines, they are merged based on heuristic criteria.

3. *Baseline fitting*

   This steps provides a baseline for each line, as shown in fig. 1.15. These baselines are fitted using a quadratic spline, as curved baselines can also be present.

4. *Character chopping*

   Upon determining lines, Tesseract tries to chop the existing words into characters. This is done by a determined constant pitch when handling fixed-pitch text. In case of non-fixed-pitch text (fig. 1.16), Tesseract tries to determine the size of a character by measuring the gaps between its baseline and upper boundary. However, due to possible inaccurate assumptions, the decision can still be changed after word recognition.

5. *Word recognition in case of non-fixed-pitch text*

   This step proceeds to further chop blobs that might have produced unsatisfactory results in the last step. It does so by using various heuristics, and although its results are satisfactory, this process often

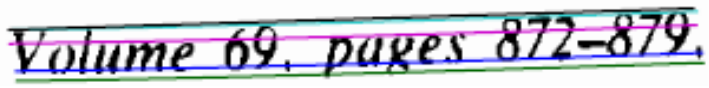


Figure 1.15: An example of a fitted baseline (dark blue) along with helper lines used for baseline fitting and later, character chopping [Smi07].

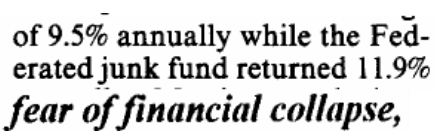


Figure 1.16: Non-fixed pitch text spacing issues [Smi07].

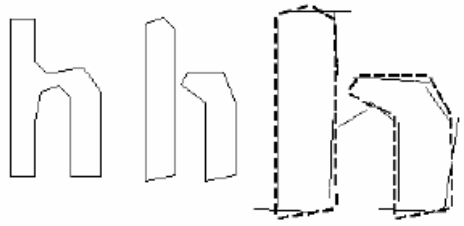


Figure 1.17: Left: original letter h; middle: broken character; right: merging of multiple smaller features [Smi07].

results in higher number of chops than required. Although an associator that already has knowledge of existing characters tries to resolve this issue, the occurrence of slight mistakes can not be excluded.

6. *Character classification*

   Earliest versions of Tesseract used a static character classifier based on topological features. However, this approach was not robust to damaged characters in real-life images. This was solved by extracting multiple small features of fixed length from the unknown and merging them to match more robust features of a character, as shown in fig. 1.17.

   Classification itself is then performed by matching the characters to a training data set and outputting the character that has the greatest number of similar features (where the "greatest number" computation is based on heuristics).

2. Layout recognition for tabular data

Layout recognition is the process of extracting individual image elements and determining their logical relations. When applied to tabular data, its goal is to determine the presence and content of image tables.

The extraction of table structures from a page poses various difficulties. Complications mostly arise when the input document does not correspond to the typical one-column, graphics-free layout. This may cause the spaces between individual page columns to be interpreted as table column spaces, which may lead to a complete rejection of any tables present in page columns. Additionally, complex layouts often cause difficulties when determining the reading order of the document. Therefore, in most cases, a *layout analysis* first needs to performed.

2.1 Layout analysis

Layout analysis is the process of identifying and categorizing image document elements, such as figures, tables, forms, math symbols, headers, footers or simple paragraph text (*geometric layout analysis*) and semantically labeling them according to their logical roles (*logical layout analysis*).

In the previous chapter, we already covered the basics of geometric layout analysis, which is the same process as page segmentation (section 1.3.1). The output of this process is a data structure containing information about the detected elements. A logical layout analysis is then applied on this output.

Logical layout analysis is used to determine the reading order of the image document. It adopts the idea of *labels*, which provide an information about the semantic order and type of individual document elements. For example, a label might be just a number indicating the reading order (as presented in fig. 2.2), or it could contain more complex information, such as "table header", "page footer", "image caption", etc.

The result of logical layout analysis is therefore often in a form of mapping of each element to its corresponding label. However, the determination of correct labels is sometimes hard even for human perception, as presented in fig. 2.1. With various differently aligned columns with different font sizes, or with image captions appearing on different sides of the image in every document, people often determine which elements belong together only according to their intuition (e.g. when reading about a recent earthquake,

Figure 2.1: Reading order problems. Determination of correct labels is sometimes hard even for human perception.

caption saying "Rescued puppy" probably belongs to the picture of a dog instead of a flooded beach, even though it can be placed right in the middle of these two images).

Computers have no notion of such things. This is often a cause of many errors and a reason why a lot of OCR engines claim to work on only documents with specified layouts, e.g. single-column, non-graphical, etc.

Various heuristics are used for the determination of labels [DS14]. In the following list, we overview several of the most widely used:

Templates The most simple and basic approach is the technique of the already mentioned templates. It is based on a limited number of predefined document layouts (*templates*), which already contain the information about the structure of individual elements. An input document is then matched to these layouts. In the OCR engines that focus solely on processing a single type of documents (such as ticket validation, recipe or passport recognition, recognition of forms filled out by patients in hospitals), this process yields almost perfect results, even though it is

Figure 2.2: An example of layout analysis [Had+04].

a naive approach.

Rule-based approaches A human reader often determines the logical succession of document elements by font settings and locations of the elements. Rule-based approaches take advantage of this fact and create heuristic "rules" that determine the type of the element. For example, a rule for a page header could be "has the smallest y-axis value, has font size above 22pt, is bold, and is the only element on its line".

Syntactic methods These methods present the structure used for element labeling in a form of a set of formal (usually context free) grammars. These grammars contain rules for aggregating pixels into more structured entities until they form logical objects. Parsers for a syntactic analysis are automatically obtained from these grammars. They are then used to perform the actual labeling of the detected elements.

Machine learning Already mentioned in this thesis, a non-heuristic approach

to logical layout analysis is the use of neural networks. Given enough information and time for training, the networks are able to determine the labeling on their own.

There exist various techniques of machine learning, distinguished by the way the neural networks are trained. For example, a neural network can be given a set of rules and input images, which leads the learning process to produce results similar to human observation. Also, it can be solely reliant on raw physical data and itself.

Worth mentioning are also techniques like *Blackboard system*, *Description language* or methods based on *Hidden Markov Models* [Cat+98].

Layout analysis is a crucial part of almost every OCR engine. If either geometric or logical layout analysis fails, the input of the recognition engine might contain corrupted data. This might lead to a significantly lower accuracy of the recognition process.

Table recognition utilizes the output of layout analysis and extracts table related information, such as table cells, headers and footers, from it. Other elements are discarded (e.g. floating text, graphics), as they are considered worthless for the purposes of table recognition.

2.2 Table recognition

The goal of table recognition is to determine if a table even is present on a page, and if yes, where and what its content is. This process is often divided into two parts — *table detection* and *table decomposition*, with table detection determining the presence and placement of a table, and table decomposition analyzing its content and producing a meaningful structure of its representation. In our thesis, we approach the table recognition problem as a whole, as usually, both parts benefit from reusing each others outputs.

Although obtaining the elements of a simple $m \times n$ grid is an easy task achieved by a simple line detection algorithm (e.g. Hough transform [SKK16]), the absence of cell borders requires more complicated heuristics. Moreover, tables that contain cells of different sizes, various numbers of cells in different rows, complicated headers or footers, multi-line cells and many more also require more complex solutions. We present a few of these possible obstacles in fig. 2.3.

In this section, we overview some of the existing table recognition algorithms. For the purposes of this thesis, we specifically focus on the table recognition implementation of the Tesseract engine.

DESCRIZIONE	Consistenza 1/01/2004	Acquisizioni	Spostamenti dalla voce nella voce	Alienazioni	Arrotondamenti	Ammortamento 2004	Consistenza 31/12/2004
Costi di ricerca, sviluppo e pubblicità	38.948	30.240	-	-	-	16.203	52.985
Concessioni, licenze e marchi	5.751	12.595	-	-	-	4.572	13.774
Altre immobilizzazioni immateriali	78.452	4.164	-	-	-	16.245	66.370
Immobilizzazioni in corso	0	46.137				0	46.137
TOTALI	**123.151**	**93.136**	**-**	**-**	**-**	**37.021**	**179.266**

Figure 2.3: Several common problems occurring during table recognition: missing horizontal and vertical lines; missing information in cells; multi-line cells in the first column and header row; different alignment of header and content cells.

2.2.1 Tesseract table recognition

The Tesseract engine has been originally used for character detection. Over the years, many features have been added, including a TableFind algorithm for table detection and recognition.

Shafait and Smith [SS10] based the TableFind recognition algorithm on already existing Tesseract's features, including layout analysis (described in section 1.3.1) and character detection (section 1.4.1). Following are the individual steps of the algorithm, along with their visualization in fig. 2.4:

1. *Layout analysis*

   First, layout analysis is performed by *tab-stop detection* (section 1.3.1) included in the Tesseract library. The results of this step include not only a list of segmented blocks, but also the column layout (fig. 2.4a) and column partitions (sequences of connected components of the same type — like text, image — that do not cross any tab-line, presented in fig. 2.4b).

2. *Identification of table partitions*

   With the page layout available, TableFind tries to identify text column partitions that could possibly belong to a table — *table partitions*. This process is based on heuristics, by identifying column partitions that contain at least one large gap between their connected components, consist of only one word, or overlap with other partitions along the y-axis.

   This stage of the algorithm is performed quite aggressively, so although this process returns the desired table partitions, it also produces a lot of false positives, such as considering section headings, page headers,

footers, equations, etc., as tables. Most of these unwanted partitions are then removed by a heuristic filter. However, as presented in fig. 2.4c, the presence of minor mistakes is not completely eliminated.

3. *Detection of table columns*

   As shown in fig. 2.4d, vertically aligned partitions are grouped into a single column. Columns with only one partition are then removed.

4. *Table construction*

   The goal of table construction is to group table columns into a table. Here, TableFind assumes that flowing text does not share space with a table along the y-axis. Therefore, boundaries of table columns are expanded along the y-axis to the borders of the page columns that contain them. In result, bounding boxes are created for whole tables. Tables that span across multiple page columns are detected only if a table column exists that belongs to all of these page columns.

5. *Removal of false positives*

   Because of relatively greedy heuristic used in the previous step, non-tabular content may be falsely identified in a table. Therefore, TableFind removes tables with only one column. This produces the final result (fig. 2.4e).

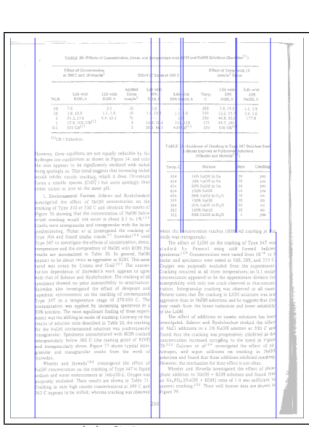
The algorithm has been proved to have a 86% precision. The biggest problems have shown to be full-page tables, often resulting in over or under-segmentation, partial detection or detection of false positives [SS10].

Another problem with TableFind is that currently, there exists no simple command that a user could run to see the output of this algorithm. To actually detect a table, a user must first write its own program that uses the functions of the added Tesseract table recognition files. Then, he needs to process the data the library returns and output them in a meaningful format, which requires a non-trivial knowledge of the Tesseract implementation.

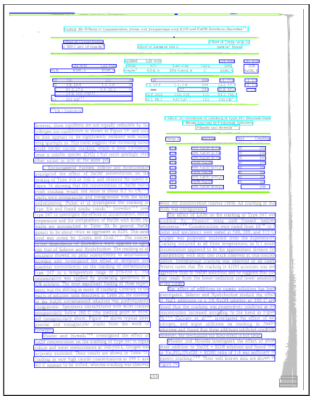
We further present the results of this algorithm in section 4.5, where we compare them to our implementation.

2.2.2 Other existing approaches

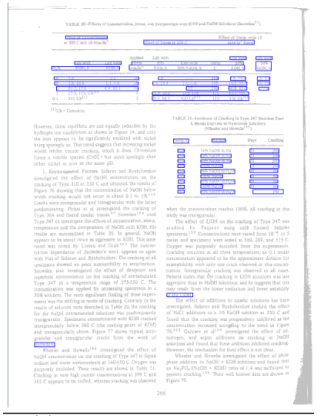
In addition to the TableFind algorithm, there exist various heuristic approaches for table detection. In this section, we overview several of them,



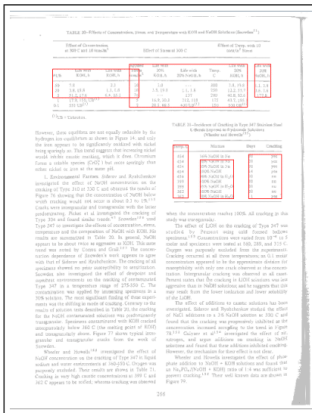
(a) Column layout.



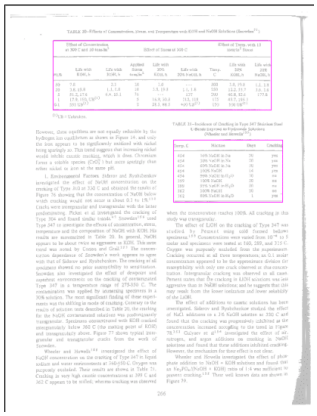
(b) Column partitions.



(c) Candidate table partitions.



(d) Table columns.



(e) Detected table regions.

Figure 2.4: The process of Tesseract table recognition [SS10].

including their advantages and disadvantages. Specifically, we focus on T-Recs table recognition system [KD98], Medium-independent table detection [Hu+99], pdf2table project [YKM05] and an approach based on a hierarchical MXY tree page representation [Ces+02]. However, multiple other approaches also exist. Worth mentioning is also, for example, sparse line detection [LMG08], which already uses principles of machine learning. Some of the other methods are briefly reviewed by Jahan and Ragel [JR14] or Bansal, Harit, and Roy [BHR14].

One of the first approaches to table recognition was presented by the *T-recs* table recognition system [KD98], which is based on a bottom-up approach of clustering word bounding boxes and building a "segmentation graph". This segments the page into different regions, which are then evaluated by certain heuristic criteria for tables. Although widely popular in the 90s, this technique has several setbacks. T-Recs is controlled by a set of numerical parameters, which need to be adjusted manually by the user according to the layout of the page. Moreover, it yields unsatisfactory results on multi-column documents.

Another algorithm was described by Hu et al. [Hu+99]. In single-column documents, a page can be easily segmented into individual textlines. The table detection problem is then perceived as an optimization problem, where the start and end textlines that belong to a table are identified by optimizing some quality function. However, this approach fails on multi-column documents, and on documents that contain more than one table.

Cesarini et al. [Ces+02] describe an approach based on hierarchical MXY-tree-like representation. The presence of a table is determined by searching the tree for parallel lines that contain white spaces, and other perpendicular lines between them, which indicate cell borders. Located tables are merged on the basis of proximity and similarity criteria. However, this approach fails if no lines in tables are present.

Method presented by the *pdf2table* project [YKM05] is based on assigning each text object of the page its positional attributes. By the evaluation of these attributes, text objects are then merged into single-lines (lines with only one text object), multi-lines (lines with more than one text object) and multi-line blocks (multiple multi-lines merged together). The table detection algorithm then merges the multi-line blocks that may belong to the same table, with the help of a heuristic threshold that determines the greatest number of single-line objects between two multi-line blocks possible. This method also assumes the input to be a single-column document. Despite of that, the user can manually adjust the information about the number of columns, which yields more accurate results.

3. Table recognition implementation

This chapter gives us an overview of procedures and tools used to create a table recognition software.

The project is written in C++ and compilable using CMake (tested on MSVC 2017 and g++ 7.4). We chose CMake for its cross-platform support and its ability to work with external dependencies (e.g. Tesseract, Leptonica).

The implementation is divided into two main parts — *preprocessing*, which prepares the input images for recognition, and *tabular OCR*, which applies our heuristic algorithm for table recognition. Both parts use the features of the Leptonica library and tabular OCR utilizes the character and textline recognition of the Tesseract engine. The image processing in the program is schematized in fig. 3.1.

3.1 Preprocessing

The preprocessing part is basically a wrapper around the Leptonica library. Based on the input arguments of our software, the preprocessor calls relevant Leptonica's methods with specified parameters. Not all of the Leptonica's preprocessing features are currently supported in our implementation. However, it is a simple task to add the rest of them. Currently supported methods are the following:

- *Contrast enhancement*, specifically histogram equalization (see section 1.2.2), a method of non-linear stretching called gamma correction [Rah+16], and a simple method of linear stretching based on an arctangent function.
- *Greyscale conversion*, specifically the averaging technique and luma correction technique (both mentioned in section 1.2.3). Supported are also two techniques from the Leptonica library based on the selection of either the maximal or minimal value from the three RGB components.
- *Binarization*, specifically the support of Otsu and Sauvola binarization techniques (see section 1.2.3).
- *Deskewing*, which does not contain any options. This is due to the fact that Leptonica contains only one deskewing function, which is based on

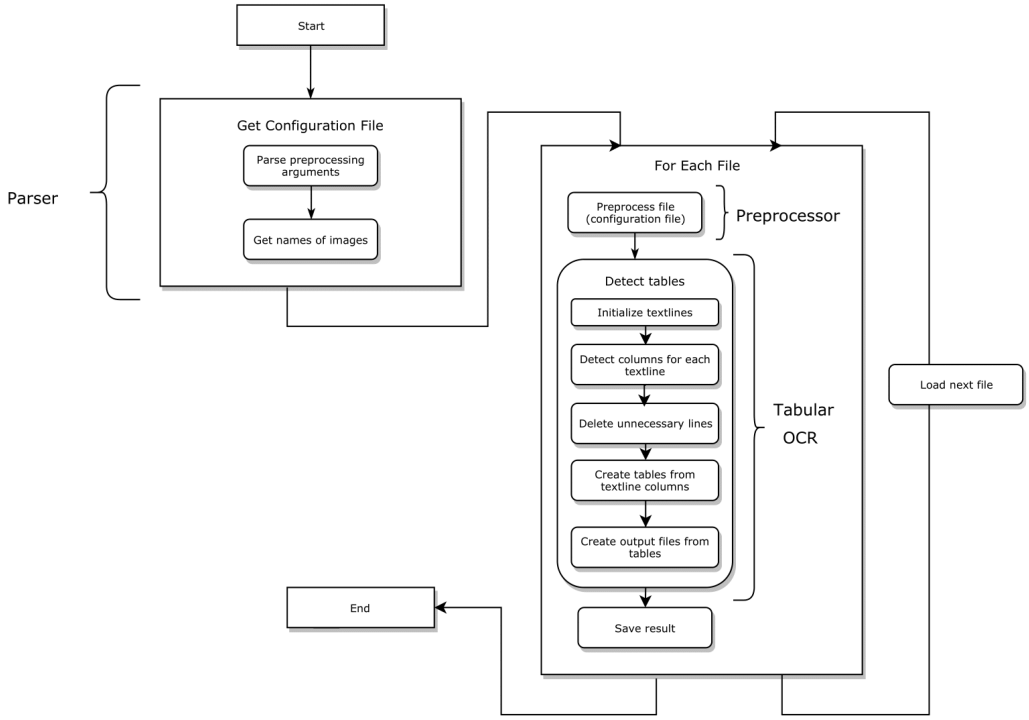


Figure 3.1: Program flow diagram.

the calculation of projection profiles (see section 1.2.4) with the help of Hough transform [Blo].

The initial idea for preprocessing was to provide more complex functions which are included in the OpenCV library [IC00]. However, this was not the goal of the thesis. Therefore, the preprocessing part stayed very simple and mostly demonstrative and the user is still advised to preprocess the images manually.

3.2 Tabular OCR

After the preprocessing part is done, the preprocessed image is passed to tabular OCR for table recognition. The individual steps of our heuristic table recognition algorithm are executed in the main `process_image` function.

In this section, we analyze this algorithm step-by-step and overview the functions used. During the process, we use the concept of *textlines* — rows of the image document that contain recognized character symbols.

Our algorithm is based on the already mentioned Tesseract symbol and textline recognition (see section 1.4.1) and moreover on whitespace detection. Upon detecting whitespaces between individual symbols, we try to heuristically estimate the whitespaces between words and, furthermore, columns, for each textline of the image. Once we have all the textlines separated into columns, we try to merge consecutive lines with similar column layout into a table.

Following are the individual steps of the algorithm, also visualized in fig. 3.5:

1. *Textline initialization*, which includes the Tesseract recognition process, and from its output obtains all the required information about individual textlines.

2. *Deletion of unnecessary lines*, which removes textlines with no valuable information (e.g. textlines with no symbols, false positives).

3. *Column detection*, which determines the column segmentation for each existing textline.

4. *Column merge into tables*, which merges the existing textlines into tables based on their column layout.

5. *Output determination*, which processes the information in each recognized table and outputs it in a meaningful structure.

3.2.1 Textline initialization

The whole process begins with the detection of individual characters and their merge into textlines (fig. 3.5a).

For the purposes of the character recognition, we use the Tesseract engine. We initialize the Tesseract API without the use of neural networks and obtain both lines and symbols. We decided to not use the neural networks due to the fact that they did not produce as accurate results as simple heuristic recognition at the time of testing. However, their performance may increase over the time. Therefore, this setting might need to be manually changed in the future.

After this step, we do not use the features of the Tesseract engine anymore, and rely solely on our heuristic functions.

The symbols and lines obtained from Tesseract are represented as simple boxes, with the symbols also containing textual information. To gain a more detailed information about individual textlines, we therefore iterate over all the lines and symbols and assign symbols into their lines. The result of this function is therefore a list of all textlines containing the information about their individual symbols, like positioning and their actual character value in UTF8.

This function runs in $O(m \cdot n^2)$. The initial idea for this implementation was to firstly sort both the symbols and lines by their y coordinates (which is simply $O(\log n)$ and $O(\log m)$). The assignment of individual symbols into their lines would then be performed by simply iterating symbols and jumping to another line once the current symbol does not fit in the current line, with the iteration therefore having a $O(m \cdot n)$ time complexity. This would present a significant execution time improvement.

However, a problem occurred with this approach. By default, Tesseract's recognition creates a lot of false positives, including noise recognized as dots, white spaces, and, most importantly, horizontal and vertical lines, like footer or header separators, underlinings of words, table borders, etc. These false positives disrupt the reading order of the document, which is a factor that this approach relies on.

Although we tried to remove these false positives (for example, removal of empty characters was trivial and required only checking for symbols that contain no textual information), we found no criterion that would suffice all of these obstacles. Therefore, in our current approach, we chose to sacrifice time complexity in favor of accuracy.

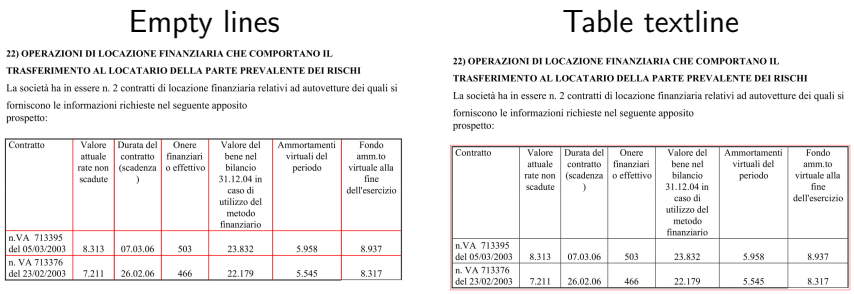


Figure 3.2: Types of unnecessary lines that Tesseract might have recognized as textlines. We remove these lines by heuristic algorithms.

3.2.2 Deletion of unnecessary lines

As already mentioned, Tesseract's textline recognition algorithm may include false positives. In this step, we delete all unnecessary lines (fig. 3.5b), specifically:

- *Empty lines*

  Like horizontal and vertical line segments, borders or other lines that contain no UTF8 symbols are of no use and are therefore removed from the textline list.

- *Table textlines*

  Table textlines are parts of the image that are bordered by visible lines, which usually represent either a table, form or even a graphics image. Tesseract often recognizes these parts as single textlines. These textlines therefore contain multiple other textlines, and are significantly greater in height. We remove them by a simple heuristic based on the comparison of their height and the font of their symbols.

We present both cases of unnecessary lines in fig. 3.2.

Once this step of the algorithm is done, we are left only with lines that contain symbols and can be a part of the table.

3.2.3 Column detection

Upon obtaining individual textlines, we try to determine their segmentation into columns by analyzing the positions of symbols they contain. This is done for each line individually. Symbols are merged into words and then into columns (fig. 3.5c). Although we could simply merge symbols into columns

and ignore the whole processing of words, this would leave us with no information about word spaces and would therefore lead to merging of text.

We start this process by obtaining all the spaces between individual symbols and sorting them by size. The resulting list of spaces is then used to determine word space size using a *multiplication factor*.

Multiplication factor is used to decide whether an element of the sorted space sizes list differs enough from its predecessor to be a word space. From the results of the detection of spaces between individual symbols, we have observed that the greater the current space is, the less the multiplication factor should be. Based on this, we have tried many different values and curves for the determination of the multiplication factor. First observations from these attempts led to the estimation that the best curve to use would be logarithmic. However, the current implementation seemed to worked well enough and was therefore left as it is.

The determination of the column whitespace was based on a similar idea with slightly altered parameters.

Upon determining the column and word whitespaces, we then merge symbols by these whitespaces into words and columns respectively, as shown in fig. 3.3.

The determination of whitespaces was probably the most difficult part of the algorithm. There have been multiple different ideas for the implementation. The one that has been preferred most of the time was the idea of separating textlines according to their font sizes. The ones with similar fonts were assigned to the same *font category*, and the whitespace was then determined for the whole category. The word whitespace recognition worked slightly better with this approach. However, the determination of columns had a higher chance of failure, as the sizes of column spaces greatly differ in each line. Therefore, the current simpler approach was chosen.

Another initial idea was to simply determine the size of the column space by a constant, e.g. $word\_whitespace * constant = column\_whitespace$. Surprisingly, the results of this approach were comparable to those of the current implementation. However, it was prone to errors, especially when the input image contained small fonts or full-page tables, and had no room for improvement in contrast to the current approach.

3.2.4 Column merge into tables

Once we have the information about columns for each textline, we can start searching for tables. The table detection algorithm is performed by a simple $O(n)$ algorithm by iterating the textlines from top to bottom and merging two consecutive textlines with similar column layout.

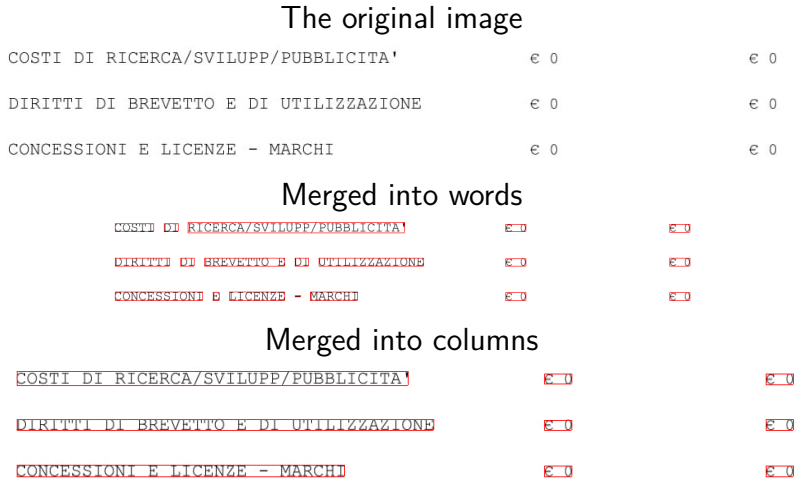


Figure 3.3: The process of merging symbols of a textline.

The determination of whether two textlines have similar column layout is performed according to algorithm 1.

From the recognized tables represented by textlines, we create cells by simply overlaying rows and columns and saving their common areas. The problem arises with the existence of multi-line rows, that is, rows that often span over multiple textlines. In our implementation, a simple constant-based algorithm is added to recognize at least some of them and therefore merge multiple textlines into one row.

3.2.5 Output determination

Once we obtain the individual cells, the only thing left is to create a user-friendly representation of the recognized data. Here, the user has two options according to the parameter he sets in the command line environment.

The first option is a simple image output. Recognized cells are bordered by colored boxes (fig. 3.5d) in the original input image and saved in a PNG file.

The other option is a JSON structure of the recognized cells, which also contains text within each cell. The JSON structure is demonstrated in fig. 3.4.

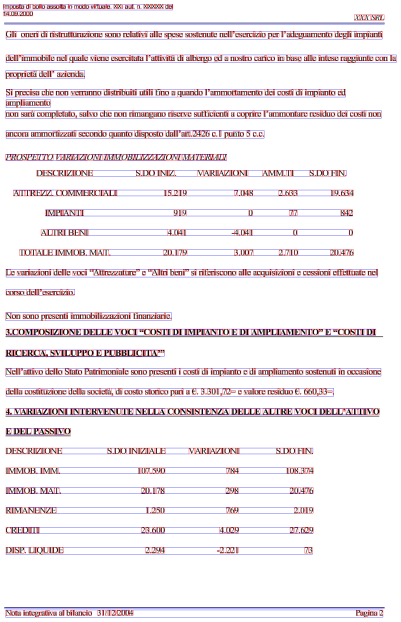
By default, both output options are selected and therefore two files are saved in the newly created *results* directory next to the executable.

Algorithm 1 Are textlines in same table

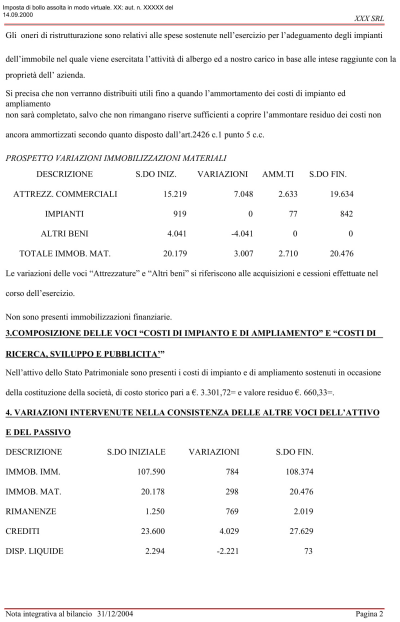
```
Require: first_col          ▷ represents the current place we are when iterating over columns of first line
  second_col                ▷ represents the current place we are when iterating over columns of second line
  while true do
    if either first_col or second_col is at the end of their line then
      if at least one pair of columns was found that should be merged then
        merge
      else
        create a table if there are any merged lines
      end if
      break
    end if
    if current columns do not overlap then
      if first_col has lower x-axis than second_col then
        first_col = first_col + 1
      else
        second_col = second_col + 1
      end if
      continue
    end if
    if found columns do not overlap with other existing columns then
      save the position of overlapping columns for future merging
    else
      create a table if there are any merged lines
    end if
    first_col = first_col + 1
    second_col = second_col + 1
  end while
```

```
{"all_tables": {
  "cols": number of columns,
  "rows": number of rows,
  "table_repres": {
    "h": height of table,
    "w": width of table,
    "x": the x-coordinate where the table starts,
    "y": the y-coordinate where the table starts
  },
  "cells": [
        {
        "box": {
           "h": height of current cell,
           "w": width of current cell,
           "x": the x-coordinate where the cell starts,
           "y": they y-coordinate where the cells starts
        },
        "cols_no": in which column of the table the cell is,
        "rows_no": in which row of the table the cell is,
        "text": the UTF8 text displayed in the cell
        },
        ...
        other cells
    ]
  }
  ...
  other tables
}
```

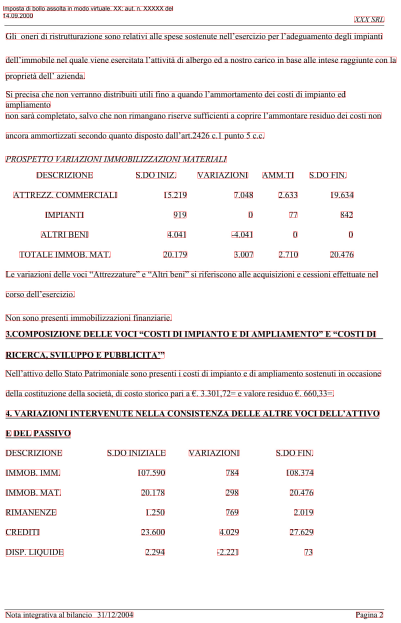
Figure 3.4: A JSON structure used for the representation of tables recognized from an input image.



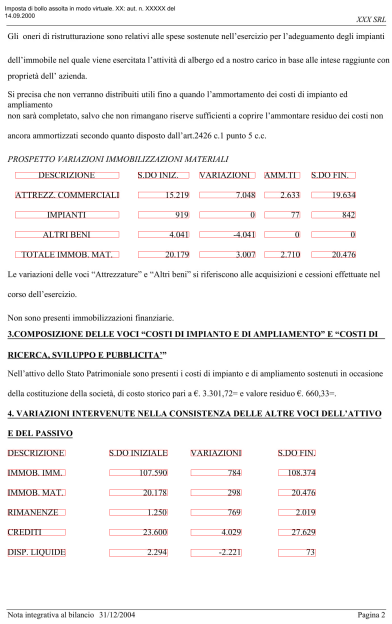
(a) Recognized textlines (blue) and their symbols (red) by the Tesseract recognition. These are merged so the symbols are assigned to their textlines.



(b) Unnecessary lines recognized from the image. These are removed during the algorithm.



(c) The results of segmentation of each textline into columns.



(d) Result of our algorithm: individual bordered cells that represent tables.

Figure 3.5: The process of table recognition.

4. Results

In this chapter, we analyze the performance of our software and compare its results to those of the Tesseract TableFind algorithm. For these purposes, we use our own testing set of around 140 images containing one or more tables. We deliberately chose images that are not scanned or disrupted in any way, as this might negatively affect the results of Tesseract recognition, which both our implementation and Tesseract's TableFind rely on.

4.1 Performance measurements

As already mentioned, Tesseract is an engine heavy on resources. Its recognition and its functions therefore take a great amount of time, which is also the reason why the time complexity of our implementation is significantly higher.

For comparison, when we run our software on a single image (1654×2339, of size 675 KB), the absolute time of recognition is 11.6 seconds. Tesseract's API initialization takes 9.6 seconds of this time (with its `recognize` function performing the character and line recognition in 9.2 seconds), and our functions for saving results (which use the calls of Leptonica) take 0.5 seconds. About 1.4 seconds is consumed by our `init_textlines` function, which also uses the calls of the Tesseract API. However, this function needs to iterate over all symbols and textlines multiple times, which is the reason for the higher time complexity. The time complexity of the other functions (which is below 0.1 seconds) is therefore negligible.

We ran a performance test on all our images to provide a better concept of the amount of time that our software spends on each function. We did not add any preprocessing, as it is a part of the Leptonica library and therefore does not affect the performance of entirely our functions. Furthermore, upon running a few tests on the preprocessor functions, they seemed to barely affect the performance.

The results were as follows: the calls of Tesseract API took 65.94% of the overall time, our textline initialization 30.26%, result saving 2.9% and the other functions solely 0.9%. Despite these results, we have observed that time complexity of the textline initialization does not exceed 20% in most cases. However, a few images, usually those containing full-page tables, spent even more time analyzing textlines than with the actual recognition, which significantly affected the overall results.

In the following sections, we discuss the options of reducing this time

complexity in favor of the accuracy of results. This includes the discussion about the effects of the quality of an image on the results and time complexity, as well as the effects of the amount of text in an image on the time complexity.

4.2 Effects of the amount of text

In this section, we provide an overview of the effects of the number of recognized symbols and textlines on the execution time of our algorithm. As we already mentioned in section 3.2.1, the time complexity of our `init_textlines` function is $O(m * n^2)$ (where m represents the amount of textlines and n the amount of symbols). Therefore, the greater the amount of textlines and symbols, the more time this function will consume. However, the time complexity of the Tesseract recognition also greatly depends on the amount of textlines and symbols it eventually finds. We tested these dependencies on a subset of 18 of the already mentioned images. We presented all of them in various (about 15) different resolutions, ranging from 25 to 800 dpi, to our recognition system. This gave us a reasonably diverse set of different images. We used the results from the tests to present the mentioned dependencies in figs. 4.1 to 4.6.

By observing the testing results and utilizing the above mentioned graphs, we may conclude the following statements:

- *For our table detection algorithm to detect a table, Tesseract needs to recognize above 800 symbols and 40 textlines.*

  If our algorithm is presented with either smaller amount of symbols or textlines, then either the Tesseract recognition system failed to recognize most of the page content, or the page has so little text information that we can assume it does not contain a table. Our table detection algorithm is therefore presumed to fail.

- *The time of the execution of our `init_textlines` function grows with the number of symbols and textlines.*

  As already expected, the time complexity of this function grows exponentially in both the textline and symbol case. We present this dependency in fig. 4.1 and fig. 4.4.

- *The time of the Tesseract recognition greatly depends on the number of symbols, not so much on textlines.*

  The recognition of symbols and textlines is performed concurrently (section 1.4.1). However, what Tesseract focuses on during this process

is the merging and recognition of the individual characters, while the textlines are only a side product. Therefore, the dependencies in graph fig. 4.5 could be completely different and the graph does not contain any meaningful information.

4.3 Effects of image resolution

The quality of the input image greatly affects Tesseract's recognition system and therefore our recognition system. Tesseract advises its users to use at least 300 dpi images, as it claims that lower resolution images are more highly likely to fail during recognition. In this section, we tested the effects of image resolution on both the speed of the algorithm (presented in figs. 4.7 to 4.9) and the accuracy of Tesseract (by which we understand the number of recognized symbols and textlines, as Tesseract outputs barely any false positives, presented in figs. 4.10 and 4.11). For these purposes, we used the same different resolution images as in section 4.2.

4.3.1 Effects on time complexity

From the observation of the results in fig. 4.8, it is evident that the DPI of the image does not have a direct effect on the `init_textlines` function. This is because the time complexity of this function is directly dependent on the number of symbols and textlines. As a higher DPI image does not necessarily mean that more symbols and textlines will be present in the image (due to false positives, incorrect splits between textlines in lower resolution images, splitting of characters and more), the dependencies shown in fig. 4.8 could greatly vary.

However, when it comes to Tesseract recognition, displayed in fig. 4.7, we can clearly see that its execution time increases with increased DPI. The exception to this is when the image has around 300 dpi, where the execution time suddenly decreases. This is due to the fact that most of the Tesseract training data are of a 300 dpi resolution (see section 1.2.1).

This implies that the overall time complexity of our algorithm will also be dependent on the DPI of the image, with the best results obtained at the dpi of around 300 (shown in fig. 4.9). Although the recognition time is exponentially lower with dpi under 150, the results from images with this resolution are in most cases useless, as they produce many undetected characters.

4.3.2 Effects on Tesseract accuracy

From the observation of the results in fig. 4.10, we can clearly see that the number of detected symbols rises rapidly up to the DPI of around 300. However, once we increase the DPI of the image above the value of 300, the recognition system barely recognizes any new characters. Sometimes, at significantly higher resolutions, the recognition even decreases in accuracy. This leads us to the assumption that the 300 dpi resolution is sufficient enough for symbol recognition.

As we already mentioned in this chapter, the fact that there is an increased amount of textlines does not necessarily mean that the accuracy of Tesseract has increased. Therefore, the effects of DPI on the amount of textlines in an image are unpredictable and the results displayed in fig. 4.11 do not contain any meaningful information.

4.4 Effects of preprocessing

As mentioned multiple times in the previous chapters, preprocessing is a crucial part of any OCR engine, including Tesseract. In both fig. 4.12 and fig. 4.13, we present its importance along with a few examples of how the Tesseract symbol recognition results change when only slight tweaks in an image are made.

4.5 Comparison to Tesseract's TableFind

We already mentioned Tesseract's TableFind algorithm in section 2.2.1. It uses heuristic algorithms to find tables from the segmentation Tesseract already offers. It also contains a simple recognition class that tries to analyze the found tables for rows, columns and cells. It is a bottom-down approach, in comparison to our algorithm, which starts from the individual characters and builds up a table.

Although this algorithm has a pretty impressive results when finding tables (*table detection*), it does not provide any information about the text in various cells and has no output except for a bounding box around the found table. Therefore, there is almost no support for table recognition. As this was the goal of our implementation, it is difficult to compare its results to ours. Moreover, TableFind is mainly used as a tool for developers. It has no command line interface or GUI and the user needs to understand its API to obtain its results.

The only comparable results that both algorithm produce are the recognized tables. We ran both TableFind and our algorithm on all of our testing files. By observing the results, we concluded the following:

- Our algorithm considers noticeably different fonts to probably not belong to the same table, which is often true. TableFind does not make this assumption.
- In contrast to our algorithm, TableFind often results in over-segmentation (fig. 4.14).
- TableFind achieves more accurate results than our algorithm on full-page tables and heterogeneous page layouts (fig. 4.15).
- TableFind has no support for handling footers and often includes them in a table, in comparison to our algorithm.
- TableFind considers horizontal and vertical lines and therefore renders satisfactory results when it comes to bordered tables.

4.6 Open problems and future work

In fig. 4.20, we provide a few sample results of our algorithm. Following are the most visible problems that our algorithm encounters, along with an outline of their solutions:

- *Tables bordered by horizontal and vertical lines*

  Presented in fig. 4.16, this most basic example of a table is something we completely ignore in our algorithm. Once we assume that a textline recognized by Tesseract is either a simple horizontal or vertical line, or even a border around a segmentation element, we automatically remove it as it may disrupt our further recognition process.

  Rather than ignoring these lines, we could first implement a heuristic algorithm trying to combine these lines together and determine whether they might not be representing a table.

- *Word whitespace detection*

  Although column detection seems to work quite well, word whitespace detection has great room for improvement (see fig. 4.17). The estimated whitespace is often greater than the real one, which results in merging of words that should be separated. As already mentioned

in section 3.2.3, the approach with various font categories provided better results of word whitespace estimation. We replaced this approach by the current. This was because of the complications that arose when calculating column whitespace, and because of the increased time complexity of the algorithm, as calculating the font categories involved a number of other iterations of symbols and textlines. To increase the accuracy of word whitespace detection, the better practice would be to use the old approach for word whitespace detection, and the new for column whitespace detection.

- *Multi-row columns*

  Clearly visible in fig. 4.18, the detection of multi-rows does not work very effectively. This is because of our constant based algorithm. We could try to improve this by an implementation of a smarter algorithm based on heuristics. For example, one of the possible implementations could try to get all of the gaps between lines in a table and determine the spaces similarly to our whitespace detection (section 3.2.3). Also, when every third cell's text starts with an upper case letter and all the others start with a lower case, it is a pretty clear indicator of the presence of a multi-row cell.

- *Multi-column layouts*

  Our algorithm has no support for multi-column layouts. This results in either merging of tables from different columns into one, or recognizing each table from different column as a single column table. This can be solved by firstly checking whether the page is multi-column (by finding wide whole-page columns that contain no symbols), and then applying our algorithm to each of the recognized columns.

- *Support of different languages*

  In our implementation, we initialize Tesseract with a testing data set suitable for characters of English language. Tesseract still recognizes symbols from other languages (like é, ô, ñ and many more), but instead of outputting the symbol we wished for, it returns a combination of non-alphabetical ASCII characters. Although this does not interfere with the image output of our table recognition, it notably affects the JSON output.

  This can be solved by adding other testing data from the Tesseract open source repository and implementing their usage.

- *Support of only simple table structure*

Our system does not recognize complicated table structures. It relies on tables having a simple grid layout. Moreover, the structure of our algorithm was not designed for handling other kinds of layouts (although our JSON output file supports them). To solve this problem, a lot of serious changes need to be made in our function that merges lines into tables, such as isolating the tables, checking for their headers, adding the support for segmentation of the table and maybe even a semantic analysis.

Our algorithm encounters multiple other types of errors. These include unrecognized headers of tables, table splits (see fig. 4.19), recognition of graphics as tables, no support for vertical text and others.

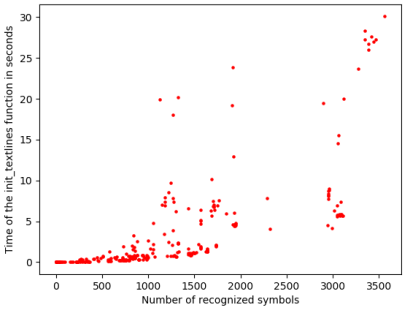


Figure 4.1: The time of execution of the `init_textlines` function with respect to the number of recognized textlines.

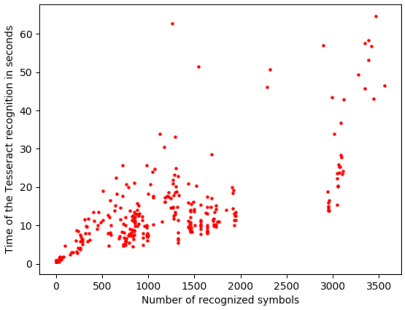


Figure 4.2: The time of execution of Tesseract recognition with respect to the number of recognized textlines.

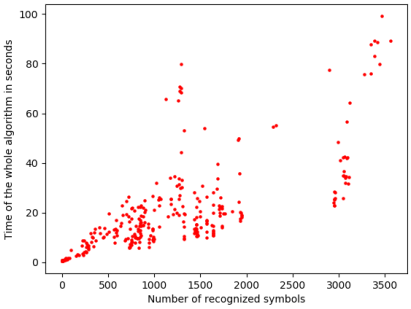


Figure 4.3: The overall time of execution with respect to the number of recognized symbols.

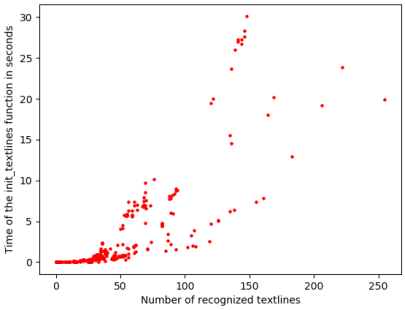


Figure 4.4: The time of execution of the `init_textlines` function with respect to the number of recognized textlines.

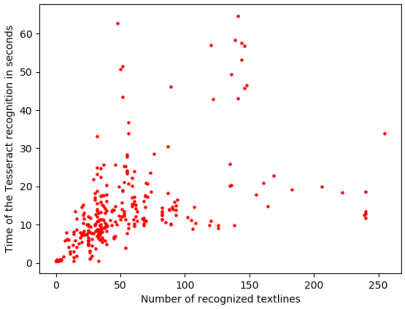


Figure 4.5: The time of execution of Tesseract recognition with respect to the number of recognized textlines.

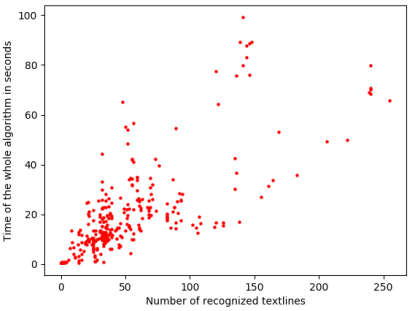


Figure 4.6: The overall time of execution with respect to the number of recognized textlines.

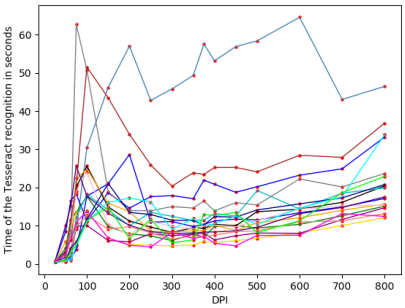


Figure 4.7: The time of Tesseract recognition with respect to the DPI of the given image (different lines = different images).

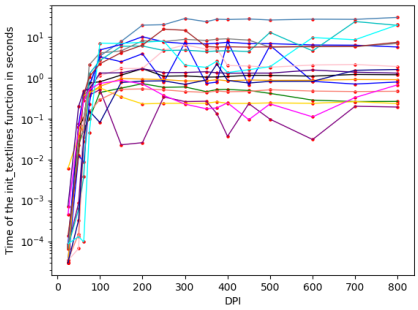


Figure 4.8: The time of execution of the `init_textlines` function with respect to the DPI of the given image (different lines = different images).

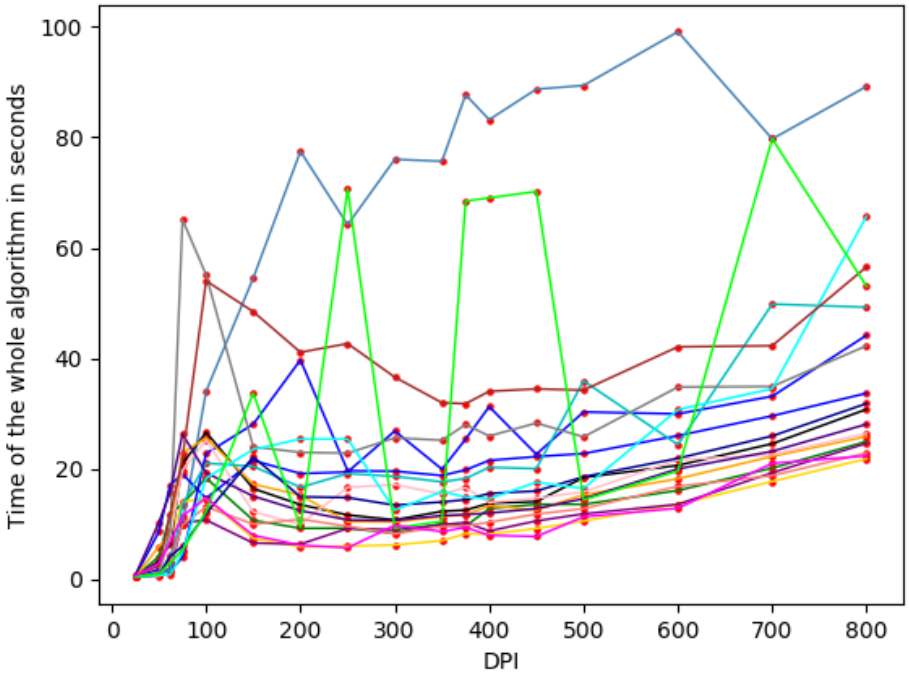


Figure 4.9: The overall time of execution with respect to the DPI of the given image (different lines = different images).

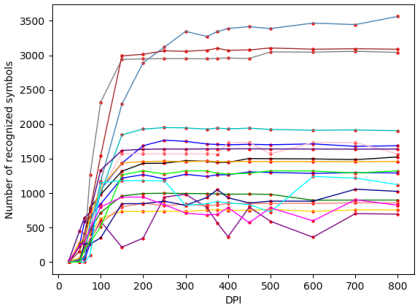


Figure 4.10: The number of recognized symbol with respect to the DPI of the given image (different lines = different images).

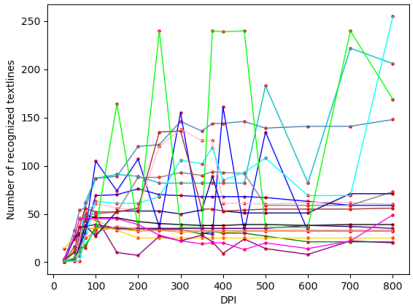


Figure 4.11: The number of recognized textlines with respect to the DPI of the given image (different lines = different images).

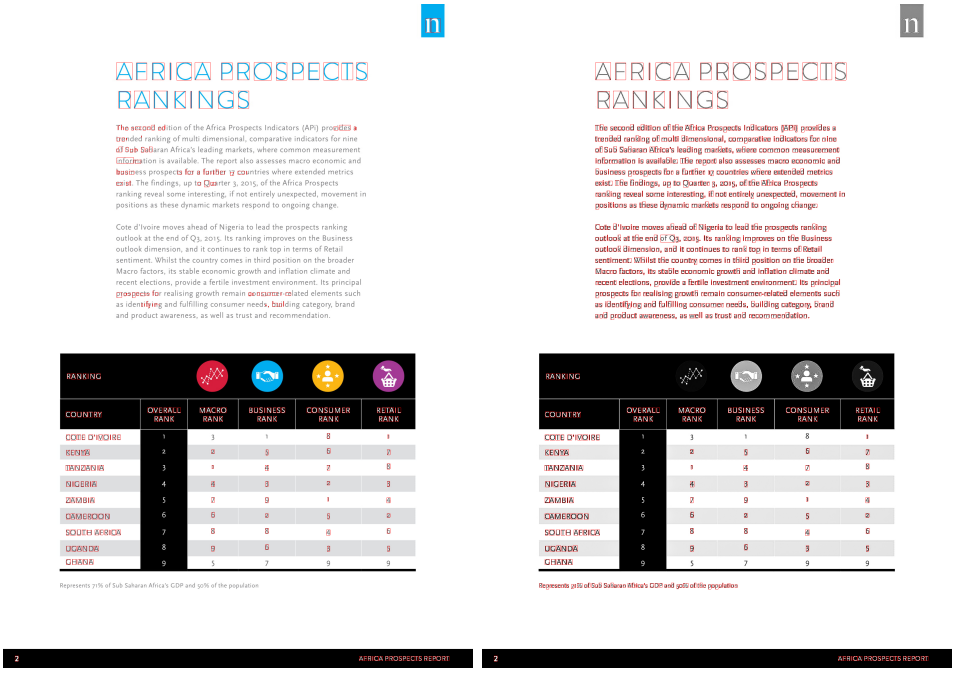


Figure 4.12: The effects of preprocessing on Tesseract recognition. Left: Tesseract does not recognize most of the symbols of the original image because of low contrast. Right: Upon applying a simple luma greyscale conversion and increasing contrast, Tesseract outputs notably better results.

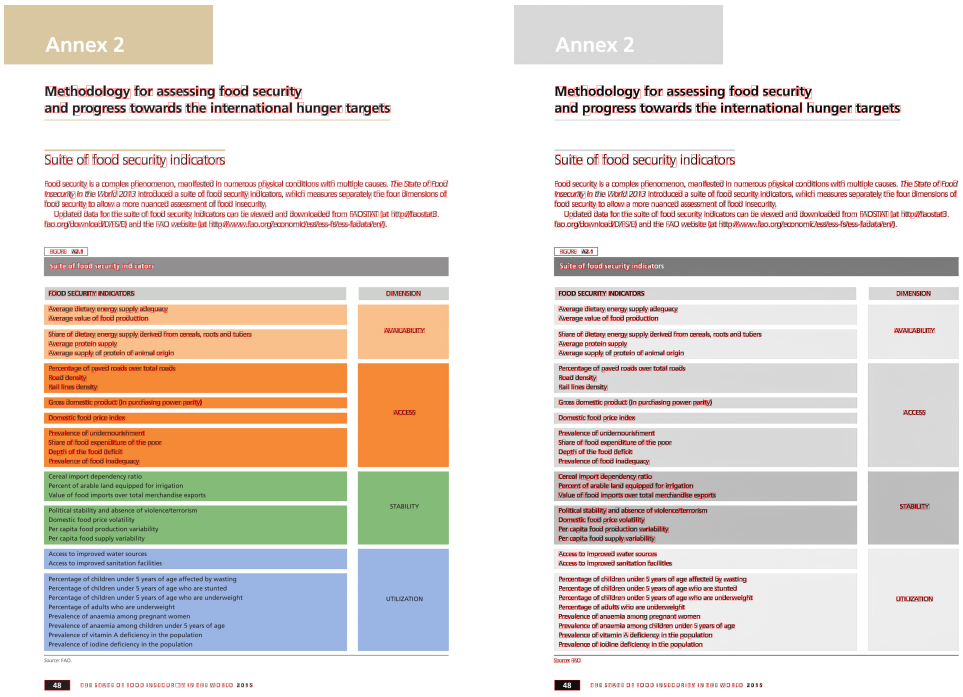
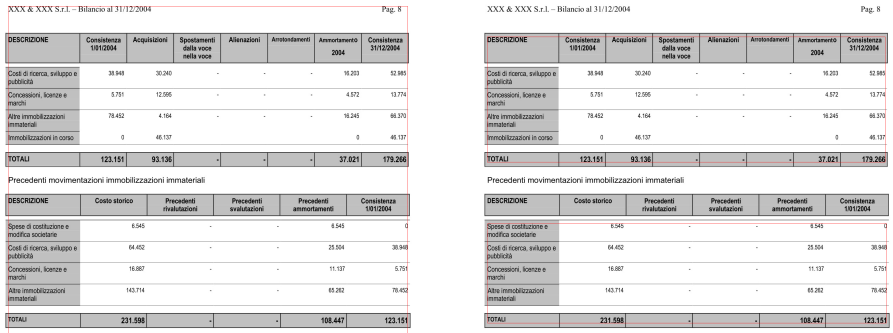


Figure 4.13: The effects of preprocessing on Tesseract recognition. Left: The lower part of the image is not recognized properly due to a dark background. Right: Tesseract recognizes all the symbols of the image once it is preprocessed by applying an adaptive greyscale conversion (by lowering blues, reds, yellows and greens).



XXX & XXX S.r.l. – Bilancio al 31/12/2004 Pag. 8

DESCRIZIONE	Consistenza 1/01/2004	Acquisizioni	Spostamenti dalla voce nella voce	Alienazioni	Arrotondamenti	Ammortamento 2004	Consistenza 31/12/2004
Costi di ricerca, sviluppo e pubblicità	38.948	30.240	-	-	-	16.203	52.985
Concessioni, licenze e marchi	5.751	12.595	-	-	-	4.572	13.774
Altre immobilizzazioni immateriali	78.452	4.164	-	-	-	16.245	66.370
Immobilizzazioni in corso	0	46.137				0	46.137
TOTALI	123.151	93.136	-	-	-	37.021	179.266

Precedenti movimentazioni immobilizzazioni immateriali

DESCRIZIONE	Costo storico	Precedenti rivalutazioni	Precedenti svalutazioni	Precedenti ammortamenti	Consistenza 1/01/2004
Spese di costituzione e modifica societarie	6.545	-	-	6.545	0
Costi di ricerca, sviluppo e pubblicità	64.452	-	-	25.504	38.948
Concessioni, licenze e marchi	16.887	-	-	11.137	5.751
Altre immobilizzazioni immateriali	143.714	-	-	65.262	78.452
TOTALI	231.598	-	-	108.447	123.151

Immobilizzazioni materiali

Sono state iscritte al costo di acquisto o di produzione, comprensivo degli oneri accessori o di diretta imputazione.

Le spese di manutenzione che non comportano un aumento della vita utile dei cespiti vengono imputate al Conto Economico.

Il valore così determinato è rettificato dagli ammortamenti stanziati nel corrente esercizio ed in quelli precedenti determinati sulla base di aliquote economico – tecniche determinate in relazione alle residue possibilità di utilizzo dei beni; le aliquote applicate non sono state modificate rispetto all'esercizio precedente. Nell'esercizio in cui il cespite viene acquisito l'ammortamento viene ridotto forfettariamente alla metà, nella convinzione che ciò rappresenti una ragionevole approssimazione della distribuzione temporale degli acquisti nel corso dell'esercizio.

XXX & XXX S.r.l. – Bilancio al 31/12/2004 Pag. 8

DESCRIZIONE | Consistenza 1/01/2004 | Acquisizioni | Spostamenti dalla voce nella voce | Alienazioni | Arrotondamenti | Ammortamento 2004 | Consistenza 31/12/2004
Costi di ricerca, sviluppo e pubblicità | 38.948 | 30.240 | - | - | - | 16.203 | 52.985
Concessioni, licenze e marchi | 5.751 | 12.595 | - | - | - | 4.572 | 13.774
Altre immobilizzazioni immateriali | 78.452 | 4.164 | - | - | - | 16.245 | 66.370
Immobilizzazioni in corso | 0 | 46.137 | | | | 0 | 46.137
TOTALI | 123.151 | 93.136 | - | - | - | 37.021 | 179.266

Precedenti movimentazioni immobilizzazioni immateriali

DESCRIZIONE | Costo storico | Precedenti rivalutazioni | Precedenti svalutazioni | Precedenti ammortamenti | Consistenza 1/01/2004
Spese di costituzione e modifica societarie | 6.545 | - | - | 6.545 | 0
Costi di ricerca, sviluppo e pubblicità | 64.452 | - | - | 25.504 | 38.948
Concessioni, licenze e marchi | 16.887 | - | - | 11.137 | 5.751
Altre immobilizzazioni immateriali | 143.714 | - | - | 65.262 | 78.452
TOTALI | 231.598 | - | - | 108.447 | 123.151

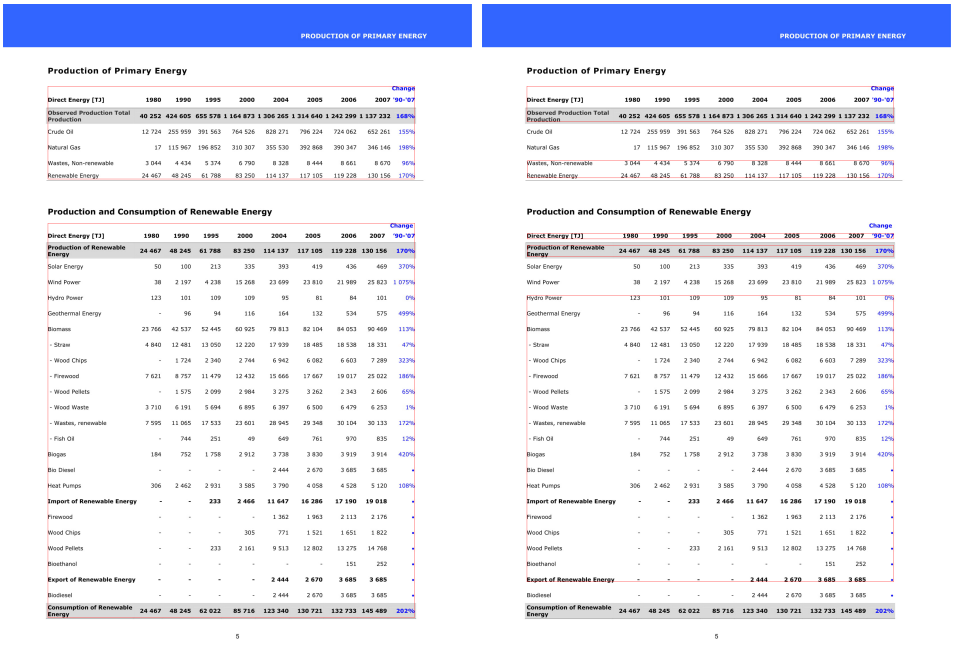
Immobilizzazioni materiali

Sono state iscritte al costo di acquisto o di produzione, comprensivo degli oneri accessori o di diretta imputazione.

Le spese di manutenzione che non comportano un aumento della vita utile dei cespiti vengono imputate al Conto Economico.

Il valore così determinato è rettificato dagli ammortamenti stanziati nel corrente esercizio ed in quelli precedenti determinati sulla base di aliquote economico – tecniche determinate in relazione alle residue possibilità di utilizzo dei beni; le aliquote applicate non sono state modificate rispetto all'esercizio precedente. Nell'esercizio in cui il cespite viene acquisito l'ammortamento viene ridotto forfettariamente alla metà, nella convinzione che ciò rappresenti una ragionevole approssimazione della distribuzione temporale degli acquisti nel corso dell'esercizio.

Figure 4.14: Comparison of TableFind algoritm (left) and our algoritm (right). Our algorithm renders better results in case tables are a part of a text document, while TableFind often results in over-segmentation.



PRODUCTION OF PRIMARY ENERGY

Production of Primary Energy

Direct Energy [TJ]	1980	1990	1995	2000	2004	2005	2006	2007	Change '90-'07
Observed Production Total Production	**40 252**	**424 605**	**655 578**	**1 164 873**	**1 306 265**	**1 314 640**	**1 242 299**	**1 137 232**	**168%**
Crude Oil	12 724	255 959	391 563	764 526	828 271	796 224	724 062	652 261	155%
Natural Gas	17	115 967	196 852	310 307	355 530	392 868	390 347	346 146	198%
Wastes, Non-renewable	3 044	4 434	5 374	6 790	8 328	8 444	8 661	8 670	96%
Renewable Energy	24 467	48 245	61 788	83 250	114 137	117 105	119 228	130 156	170%

Production and Consumption of Renewable Energy

Direct Energy [TJ]	1980	1990	1995	2000	2004	2005	2006	2007	Change '90-'07
Production of Renewable Energy	**24 467**	**48 245**	**61 788**	**83 250**	**114 137**	**117 105**	**119 228**	**130 156**	**170%**
Solar Energy	50	100	213	335	393	419	436	469	370%
Wind Power	38	2 197	4 238	15 268	23 699	23 810	21 989	25 823	1 075%
Hydro Power	123	101	109	109	95	81	84	101	0%
Geothermal Energy	-	96	94	116	164	132	534	575	499%
Biomass	23 766	42 537	52 445	60 925	79 813	82 104	84 053	90 469	113%
- Straw	4 840	12 481	13 050	12 220	17 939	18 485	18 538	18 331	47%
- Wood Chips	-	1 724	2 340	2 744	6 942	6 082	6 603	7 289	323%
- Firewood	7 621	8 757	11 479	12 432	15 666	17 667	19 017	25 022	186%
- Wood Pellets	-	1 575	2 099	2 984	3 275	3 262	2 343	2 606	65%
- Wood Waste	3 710	6 191	5 694	6 895	6 397	6 500	6 479	6 253	1%
- Wastes, renewable	7 595	11 065	17 533	23 601	28 945	29 348	30 104	30 133	172%
- Fish Oil	-	744	251	49	649	761	970	835	12%
Biogas	184	752	1 758	2 912	3 738	3 830	3 919	3 914	420%
Bio Diesel	-	-	-	-	2 444	2 670	3 685	3 685	•
Heat Pumps	306	2 462	2 931	3 585	3 790	4 058	4 528	5 120	108%
Import of Renewable Energy	**-**	**-**	**233**	**2 466**	**11 647**	**16 286**	**17 190**	**19 018**	**•**
Firewood	-	-	-	-	1 362	1 963	2 113	2 176	•
Wood Chips	-	-	-	305	771	1 521	1 651	1 822	•
Wood Pellets	-	-	233	2 161	9 513	12 802	13 275	14 768	•
Bioethanol	-	-	-	-	-	-	151	252	•
Export of Renewable Energy	**-**	**-**	**-**	**-**	**2 444**	**2 670**	**3 685**	**3 685**	**•**
Biodiesel	-	-	-	-	2 444	2 670	3 685	3 685	•
Consumption of Renewable Energy	**24 467**	**48 245**	**62 022**	**85 716**	**123 340**	**130 721**	**132 733**	**145 489**	**202%**

5

PRODUCTION OF PRIMARY ENERGY

Production of Primary Energy

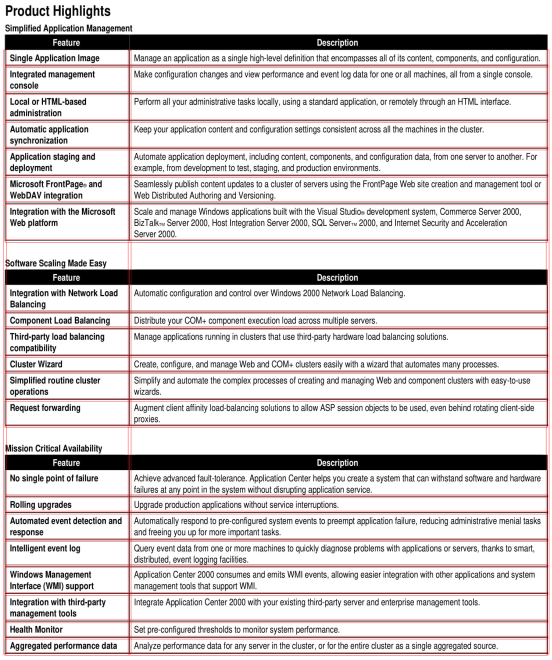
Direct Energy [TJ] 1980 1990 1995 2000 2004 2005 2006 2007 Change '90-'07
Observed Production Total Production 40 252 424 605 655 578 1 164 873 1 306 265 1 314 640 1 242 299 1 137 232 168%
Crude Oil 12 724 255 959 391 563 764 526 828 271 796 224 724 062 652 261 155%
Natural Gas 17 115 967 196 852 310 307 355 530 392 868 390 347 346 146 198%
Wastes, Non-renewable 3 044 4 434 5 374 6 790 8 328 8 444 8 661 8 670 96%
Renewable Energy 24 467 48 245 61 788 83 250 114 137 117 105 119 228 130 156 170%

Production and Consumption of Renewable Energy

Direct Energy [TJ] 1980 1990 1995 2000 2004 2005 2006 2007 Change '90-'07
Production of Renewable Energy 24 467 48 245 61 788 83 250 114 137 117 105 119 228 130 156 170%
Solar Energy 50 100 213 335 393 419 436 469 370%
Wind Power 38 2 197 4 238 15 268 23 699 23 810 21 989 25 823 1 075%
Hydro Power 123 101 109 109 95 81 84 101 0%
Geothermal Energy - 96 94 116 164 132 534 575 499%
Biomass 23 766 42 537 52 445 60 925 79 813 82 104 84 053 90 469 113%
- Straw 4 840 12 481 13 050 12 220 17 939 18 485 18 538 18 331 47%
- Wood Chips - 1 724 2 340 2 744 6 942 6 082 6 603 7 289 323%
- Firewood 7 621 8 757 11 479 12 432 15 666 17 667 19 017 25 022 186%
- Wood Pellets - 1 575 2 099 2 984 3 275 3 262 2 343 2 606 65%
- Wood Waste 3 710 6 191 5 694 6 895 6 397 6 500 6 479 6 253 1%
- Wastes, renewable 7 595 11 065 17 533 23 601 28 945 29 348 30 104 30 133 172%
- Fish Oil - 744 251 49 649 761 970 835 12%
Biogas 184 752 1 758 2 912 3 738 3 830 3 919 3 914 420%
Bio Diesel - - - - 2 444 2 670 3 685 3 685 •
Heat Pumps 306 2 462 2 931 3 585 3 790 4 058 4 528 5 120 108%
Import of Renewable Energy - - 233 2 466 11 647 16 286 17 190 19 018 •
Firewood - - - - 1 362 1 963 2 113 2 176 •
Wood Chips - - - 305 771 1 521 1 651 1 822 •
Wood Pellets - - 233 2 161 9 513 12 802 13 275 14 768 •
Bioethanol - - - - - - 151 252 •
Export of Renewable Energy - - - - 2 444 2 670 3 685 3 685 •
Biodiesel - - - - 2 444 2 670 3 685 3 685 •
Consumption of Renewable Energy 24 467 48 245 62 022 85 716 123 340 130 721 132 733 145 489 202%

5

Figure 4.15: Comparison of TableFind algoritm (left) and our algoritm (right). TableFind renders notably better results when it comes to full-page tables.



Product Highlights

Simplified Application Management

Feature	Description
Single Application Image	Manage an application as a single high-level definition that encompasses all of its content, components, and configuration.
Integrated management console	Make configuration changes and view performance and event log data for one or all machines, all from a single console.
Local or HTML-based administration	Perform all your administrative tasks locally, using a standard application, or remotely through an HTML interface.
Automatic application synchronization	Keep your application content and configuration settings consistent across all the machines in the cluster.
Application staging and deployment	Automate application deployment, including content, components, and configuration data, from one server to another. For example, from development to test, staging, and production environments.
Microsoft FrontPage® and WebDAV integration	Seamlessly publish content updates to a cluster of servers using the FrontPage Web site creation and management tool or Web Distributed Authoring and Versioning.
Integration with the Microsoft Web platform	Scale and manage Windows applications built with the Visual Studio® development system, Commerce Server 2000, BizTalk™ Server 2000, Host Integration Server 2000, SQL Server™ 2000, and Internet Security and Acceleration Server 2000.

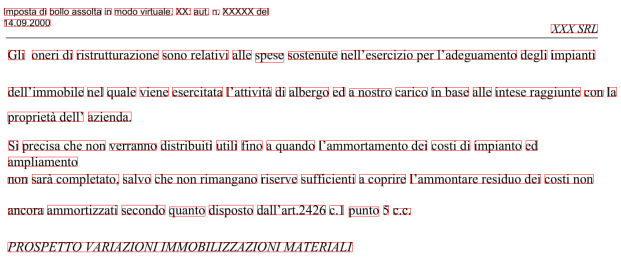
Software Scaling Made Easy

Feature	Description
Integration with Network Load Balancing	Automatic configuration and control over Windows 2000 Network Load Balancing.
Component Load Balancing	Distribute your COM+ component execution load across multiple servers.
Third-party load balancing compatibility	Manage applications running in clusters that use third-party hardware load balancing solutions.
Cluster Wizard	Create, configure, and manage Web and COM+ clusters easily with a wizard that automates many processes.
Simplified routine cluster operations	Simplify and automate the complex processes of creating and managing Web and component clusters with easy-to-use wizards.
Request forwarding	Augment client affinity load-balancing solutions to allow ASP session objects to be used, even behind rotating client-side proxies.

Mission Critical Availability

Feature	Description
No single point of failure	Achieve advanced fault-tolerance. Application Center helps you create a system that can withstand software and hardware failures at any point in the system without disrupting application service.
Rolling upgrades	Upgrade production applications without service interruptions.
Automated event detection and response	Automatically respond to pre-configured system events to preempt application failure, reducing administrative menial tasks and freeing you up for more important tasks.
Intelligent event log	Query event data from one or more machines to quickly diagnose problems with applications or servers, thanks to smart, distributed, event logging facilities.
Windows Management Interface (WMI) support	Application Center 2000 consumes and emits WMI events, allowing easier integration with other applications and system management tools that support WMI.
Integration with third-party management tools	Integrate Application Center 2000 with your existing third-party server and enterprise management tools.
Health Monitor	Set pre-configured thresholds to monitor system performance.
Aggregated performance data	Analyze performance data for any server in the cluster, or for the entire cluster as a single aggregated source.

Figure 4.16: Empty horizontal and vertical lines recognized by the Tesseract textline recognition. In our algorithm, we remove them and in the end, our algorithm determines there is no table present in the image. With the use of these lines, table recognition could significantly improve.



imposta di bollo assolta in modo virtuale) XXI aut. n. XXXXX del 14.09.2000

XXX SRL

Gli oneri di ristrutturazione sono relativi alle spese sostenute nell'esercizio per l'adeguamento degli impianti

dell'immobile nel quale viene esercitata l'attività di albergo ed a nostro carico in base alle intese raggiunte con la

proprietà dell' azienda.

Si precisa che non verranno distribuiti utili fino a quando l'ammortamento dei costi di impianto ed ampliamento
non sarà completato, salvo che non rimangano riserve sufficienti a coprire l'ammontare residuo dei costi non

ancora ammortizzati secondo quanto disposto dall'art.2426 c.1 punto 5 c.c.

PROSPETTO VARIAZIONI IMMOBILIZZAZIONI MATERIALI

Figure 4.17: Improperly recognized words in a text. The biggest issue is that multiple words are often merged into one.

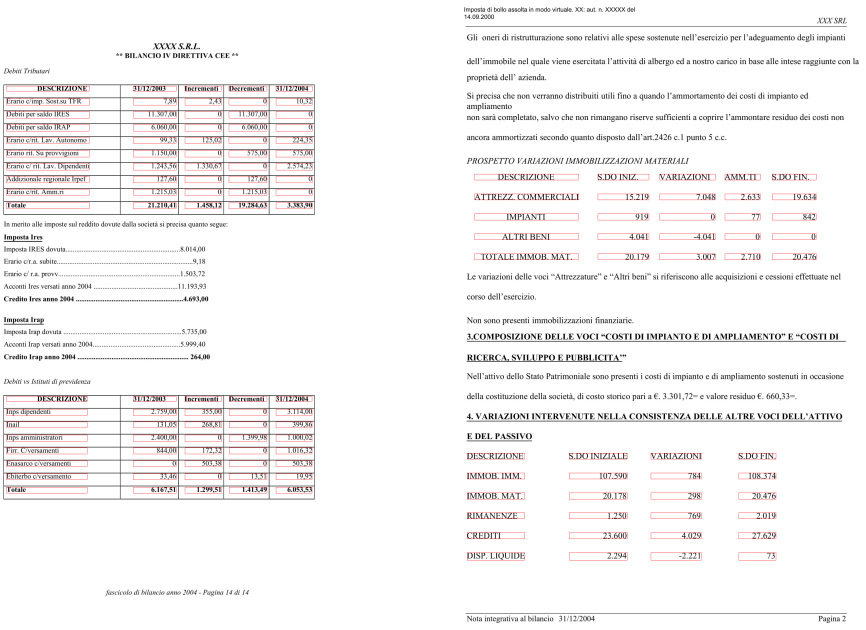
Contratto	Valore attuale rate non scadute	Durata del contratto (scadenza)	Onere finanziari o effettivo	Valore del bene nel bilancio 31.12.04 in caso di utilizzo del metodo finanziario	Ammortamenti virtuali del periodo	Fondo amm.to virtuale alla fine dell'esercizio
n.VA 713395 del 05/03/2003	8.313	07.03.06	503	23.832	5.958	8.937
n. VA 713376 del 23/02/2003	7.211	26.02.06	466	22.179	5.545	8.317

Figure 4.18: Improperly recognized multi-row cells.

VOCE	S.DO INIZ.	VARIAZ.	AMM.TI	S.DO FINALE
COSTI IMPIANTO	1.321	0	661	660
ONERI RISTRUTTURAZ.	106.269	12.216	10.771	107.714
TOTALE	107.590	12.216	11.432	108.374

VOCE	S.DO INIZ.	VARIAZ.	AMM.TI	S.DO FINALE
COSTI IMPIANTO	1.321	0	661	660
ONERI RISTRUTTURAZ.	106.269	12.216	10.771	107.714
TOTALE	107.590	12.216	11.432	108.374

Figure 4.19: Table split errors. Although the image contains only one table, two are recognized. Our algorithm assumes that the first table has 6 columns, while the second has only 5 which do not align properly. Left: recognized tables by cells; right: borders of the recognized tables.



XXXX S.R.L.
**** BILANCIO IV DIRETTIVA CEE ****

Debiti Tributari

DESCRIZIONE	31/12/2003	Incrementi	Decrementi	31/12/2004
Erario c/imp. Sost.su TFR	7,89	2,43	0	10,32
Debiti per saldo IRES	11.307,00	0	11.307,00	0
Debiti per saldo IRAP	6.060,00	0	6.060,00	0
Erario c/rit. Lav. Autonomo	99,13	125,02	0	224,35
Erario rit. Su provvigioni	1.150,00	0	575,00	575,00
Erario c/ rit. Lav. Dipendenti	1.243,56	1.330,67	0	2.574,23
Addizionale regionale Irpef	127,60	0	127,60	0
Erario c/rit. Amm.ri	1.215,03	0	1.215,03	0
Totale	**21.210,41**	**1.458,12**	**19.284,63**	**3.383,90**

In merito alle imposte sul reddito dovute dalla società si precisa quanto segue:

Imposta Ires

Imposta IRES dovuta.....8.014,00
Erario c/r.a. subite.....-9,18
Erario c/ r.a. provv.....-1.503,72
Acconti Ires versati anno 2004.....-11.193,93
Credito Ires anno 2004 .....4.693,00

Imposta Irap

Imposta Irap dovuta 5.735,00
Acconti Irap versati anno 2004.....-5.999,40
Credito Irap anno 2004 264,00

Debiti vs Istituti di previdenza

DESCRIZIONE	31/12/2003	Incrementi	Decrementi	31/12/2004
Inps dipendenti	2.759,00	355,00	0	3.114,00
Inail	131,05	268,81	0	399,86
Inps amministratori	2.400,00	0	1.399,98	1.000,02
Fin. C/versamenti	844,00	172,32	0	1.016,32
Enasarco c/versamenti	0	503,38	0	503,38
Ebiterbo c/versamento	33,46	0	13,51	19,95
Totale	**6.167,51**	**1.299,51**	**1.413,49**	**6.053,53**

fascicolo di bilancio anno 2004 - Pagina 14 di 14

Imposta di bollo assolta in modo virtuale. XX: aut. n. XXXXX del 14.08.2000

XXX SRL

Gli oneri di ristrutturazione sono relativi alle spese sostenute nell'esercizio per l'adeguamento degli impianti dell'immobile nel quale viene esercitata l'attività di albergo ed a nostro carico in base alle intese raggiunte con la proprietà dell' azienda.

Si precisa che non verranno distribuiti utili fino a quando l'ammortamento dei costi di impianto ed ampliamento non sarà completato, salvo che non rimangano riserve sufficienti a coprire l'ammontare residuo dei costi non ancora ammortizzati secondo quanto disposto dall'art.2426 c.1 punto 5 c.c.

PROSPETTO VARIAZIONI IMMOBILIZZAZIONI MATERIALI

DESCRIZIONE	S.DO INIZ.	VARIAZIONI	AMM.TI	S.DO FIN.
ATTREZZ. COMMERCIALI	15.219	7.048	2.633	19.634
IMPIANTI	919	0	77	842
ALTRI BENI	4.041	-4.041	0	0
TOTALE IMMOB. MAT.	20.179	3.007	2.710	20.476

Le variazioni delle voci "Attrezzature" e "Altri beni" si riferiscono alle acquisizioni e cessioni effettuate nel corso dell'esercizio.

Non sono presenti immobilizzazioni finanziarie.

3.COMPOSIZIONE DELLE VOCI "COSTI DI IMPIANTO E DI AMPLIAMENTO" E "COSTI DI RICERCA, SVILUPPO E PUBBLICITA'"

Nell'attivo dello Stato Patrimoniale sono presenti i costi di impianto e di ampliamento sostenuti in occasione della costituzione della società, di costo storico pari a €. 3.301,72= e valore residuo €. 660,33=.

4. VARIAZIONI INTERVENUTE NELLA CONSISTENZA DELLE ALTRE VOCI DELL'ATTIVO E DEL PASSIVO

DESCRIZIONE	S.DO INIZIALE	VARIAZIONI	S.DO FIN.
IMMOB. IMM.	107.590	784	108.374
IMMOB. MAT.	20.178	298	20.476
RIMANENZE	1.250	769	2.019
CREDITI	23.600	4.029	27.629
DISP. LIQUIDE	2.294	-2.221	73

Nota integrativa al bilancio 31/12/2004 Pagina 2

Figure 4.20: Correct table recognition (our implementation).

Conclusion

In this thesis, we explored and reviewed the existing approaches for optical character recognition, and focused on applying its results to the problem of table recognition.

As a main result of the thesis, we have developed a software package that combines the OCR functionality available in the Tesseract library with several methods of image preprocessing, and a newly developed heuristic-based algorithm that aims to improve the available possibilities of table extraction from scanned documents.

We have compared the results obtained from running this combination on a simple testing data set to the results from the TableFind algorithm of Tesseract. While both implementations required similar computational resources for processing the data, our implementation produced more accurate results for more complicated types of table layouts, especially on tables without available border separators, and in cases where the table layout depends e.g. on subtle differences in cell formatting. Additionally, we have assessed how the resolution and information content of the input image affects the time complexity and output quality of the algorithms.

Despite the improvements in table recognition capabilities, we have observed that the outcome of the recognition still mainly depends on the quality of the input image, and is mostly improved by correct preprocessing.

In the future, we plan to improve the main deficiencies of the whole pipeline, primarily the mentioned preprocessing and several open problems with table recognition summarized in section 4.6. From the observed results, we believe that after applying the preprocessing improvements, the table recognition system will produce results of excellent quality.

Bibliography

[AaKK10] Salem Saleh Al-amri, Dr. N. V. Kalyankar, and Dr. S. D. Khamitkar. "Linear and Non-linear Contrast Enhancement Image Mr". In: 2010.

[ABBa] ABBYY. *ABBYY FineReader*. `https://abbyy.technology/en:kb:images_resolution_size_ocr`. Accessed: 2018-12-03.

[ABBb] ABBYY. *ABBYY FineReader Software*. `https://www.abbyy.com/en-ee/finereader/`. Accessed: 2019-05-16.

[AR15] Laith Abdul-Rahaim. "Design Proposed Features Extraction Recognition System of Latin Handwritten Text Based on 3D-Discrete Multiwavelet Transform". In: *The Open Electrical & Electronic Engineering Journal* 3 (Apr. 2015), pp. 51–63. DOI: `10.12691/ajeee-3-2-5`.

[Ari] Manuel Aristarán. *Tabula*. `https://github.com/tabulapdf/tabula`. Accessed: 2019-05-16.

[BCM05] Antoni Buades, Bartomeu Coll, and J-M Morel. "A non-local algorithm for image denoising". In: *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*. Vol. 2. IEEE. 2005, pp. 60–65.

[BG12] Hetal Bhavsar and Amit Ganatra. "A comparative study of training algorithms for supervised machine learning". In: *International Journal of Soft Computing and Engineering (IJSCE)* 2.4 (2012), pp. 2231–2307.

[BGR07] Wojciech Bieniecki, Szymon Grabowski, and Wojciech Rozenberg. "Image preprocessing for improving ocr accuracy". In: *Perspective Technologies and Methods in MEMS Design, 2007. MEMSTECH 2007. International Conference on*. IEEE. 2007, pp. 75–80.

[BHR14] Anukriti Bansal, Gaurav Harit, and Sumantra Dutta Roy. "Table extraction from document images using fixed point model". In: *Proceedings of the 2014 Indian Conference on Computer Vision Graphics and Image Processing*. ACM. 2014, p. 67.

[Blo] DS Bloomberg. "Analysis of document skew". In: *Leptonica2002 [2] JONATHAN J. HULL," Document Image Skew Detection: Survey And Annotated Bibliography", Document Analysis Systems* ().

[Blo01] Dan Bloomberg. *Leptonica Library*. http://www.leptonica.com/. Accessed: 2018-12-04. 2001.

[Cat+98] Roldano Cattoni et al. "Geometric layout analysis techniques for document image understanding: a review". In: *ITC-irst Technical Report* 9703.09 (1998).

[Ces+02] Francesca Cesarini et al. "Trainable table location in document images". In: *Pattern Recognition, 2002. Proceedings. 16th International Conference on*. Vol. 3. IEEE. 2002, pp. 236–240.

[Cha+17] Arindam Chaudhuri et al. *Optical character recognition systems for different languages with soft computing*. Springer, 2017.

[CJP] Kalpit R Chandpa, Ashwini M Jani, and Ghanshyam I Prajapati. "Comparative Study of Linear and Non-linear Contrast Enhancement Techniques". In: *International Journal of Research and Scientific Innovation* 1 (), pp. 37–41.

[Com] Nuance Communications. *OmniPage*. https://www.nuance.com/print-capture-and-pdf-solutions/optical-character-recognition/omnipage.html. Accessed: 2019-05-16.

[Dem] Demofox. *Methods of non-adaptive interpolation*. https://blog.demofox.org/. Accessed: 2019-05-16.

[DM11] Vikas J Dongre and Vijay H Mankar. "A review of research on Devnagari character recognition". In: *arXiv preprint arXiv:1101.2491* (2011).

[Dor+15] R Dorothy et al. "Image enhancement by Histogram equalization". In: *Int. J. Nano. Corr. Sci. Engg* 2.4 (2015), pp. 21–30.

[DS14] Andreas Dengel and Faisal Shafait. "Analysis of the logical layout of documents". In: *Handbook of Document Image Processing and Recognition* (2014), pp. 177–222.

[Fad14] Shreyas Fadnavis. "Image Interpolation Techniques in Digital Image Processing : An Overview". In: 2014.

[Gmb] a9t9 software GmbH. *OCRSPace*. https://ocr.space/tablerecognition. Accessed: 2019-05-16.

[Gro] IRIS Group. *ReadIris*. https://www.irislink.com. Accessed: 2019-05-16.

[Had+04] Karim Hadjar et al. "Xed: a new tool for extracting hidden structures from electronic documents". In: *First International Workshop on Document Image Analysis for Libraries, 2004. Proceedings.* IEEE. 2004, pp. 212–224.

[Han13] Dianyuan Han. "Comparison of Commonly Used Image Interpolation Methods". In: *Proceedings of the 2nd International Conference on Computer Science and Electronics Engineering*. Atlantis Press, 2013. ISBN: 978-90-78677-61-1. DOI: 10.2991/iccsee.2013.391. URL: http://dx.doi.org/10.2991/iccsee.2013.391.

[Hu+99] Jianying Hu et al. "Medium-independent table detection". In: *Document Recognition and Retrieval VII*. Vol. 3967. International Society for Optics and Photonics. 1999, pp. 291–303.

[IC00] Itseez Intel Corporation Willow Garage. *OpenCV library*. https://opencv.org/about.html. Accessed: 2018-12-04. 2000.

[JR14] MAC Akmal Jahan and Roshan G Ragel. "Locating tables in scanned documents for reconstructing and republishing". In: *Information and Automation for Sustainability (ICIAfS), 2014 7th International Conference on*. IEEE. 2014, pp. 1–6.

[KB14] Gaurav Kumar and Pradeep Bhatia. "A Detailed Review of Feature Extraction in Image Processing Systems". In: Feb. 2014. DOI: 10.1109/ACCT.2014.74.

[KC12] Christopher Kanan and Garrison W. Cottrell. "Color-to-Grayscale: Does the Method Matter in Image Recognition?" In: *PLOS ONE* (2012). DOI: 10.1371/journal.pone.0029740. URL: https://doi.org/10.1371/journal.pone.0029740.

[KD98] Thomas Kieninger and Andreas Dengel. "The t-recs table recognition and analysis system". In: *International Workshop on Document Analysis Systems*. Springer. 1998, pp. 255–270.

[LKR13] William B Lund, Douglas J Kennard, and Eric K Ringger. "Combining multiple thresholding binarization values to improve OCR output". In: *Document Recognition and Retrieval XX*. Vol. 8658. International Society for Optics and Photonics. 2013, 86580R.

[LMG08] Ying Liu, Prasenjit Mitra, and C Lee Giles. "Identifying table boundaries in digital documents via sparse line detection". In: *Proceedings of the 17th ACM conference on Information and knowledge management*. ACM. 2008, pp. 1311–1320.

[LT03] Yue Lu and Chew Lim Tan. "A nearest-neighbor chain based approach to skew estimation in document images". In: *Pattern Recognition Letters* 24.14 (2003), pp. 2315–2323.

[Luc18] Guislain Luc. *Statistical Features*. https://heptadvices.eu/index.php/2018/01/18/handwriting-rec-features-extraction/. Accessed: 2018-05-14. 2018.

[MK11] C Mythili and V Kavitha. "Efficient technique for color image noise reduction". In: *The research bulletin of Jordan, ACM* 1.11 (2011), pp. 41–44.

[Mud+07] Nadira Muda et al. "Optical character recognition by using template matching (alphabet)". In: *National Conference on Software Engineering & Computer Systems*. 2007.

[Nin93] Li Ning. "An Implementation of OCR System Based on Skeleton Matching". In: 1993.

[NJ02] Andrew Y Ng and Michael I Jordan. "On discriminative vs. generative classifiers: A comparison of logistic regression and naive bayes". In: *Advances in neural information processing systems*. 2002, pp. 841–848.

[Pac] Hewlett Packard. *Tesseract About*. https://github.com/tesseract-ocr/tesseract/wiki/ReadMe. Accessed: 2018-12-04.

[Pac96] Hewlett Packard. *Tesseract Improve Quality*. https://github.com/tesseract-ocr/tesseract/wiki/ImproveQuality. 1996.

[Pan] Parul Pandey. *Binarization techniques*. https://towardsdatascience.com/image-segmentation-using-pythons-scikit-image-module-533a61ecc980. Accessed: 2019-05-14.

[PK04] Worapoj Peerawit and Asanee Kawtrakul. "Marginal noise removal from document images using edge density". In: *4th Information and Computer Engineering Postgraduate Workshop, Phuket, Thailand*. Citeseer. 2004.

[Rah+16] Shanto Rahman et al. "An adaptive gamma correction for image enhancement". In: *EURASIP Journal on Image and Video Processing* 2016.1 (2016), p. 35.

[Ros+14a] Daniel Rosner et al. "Image Skew Detection: A Comprehensive Study". In: *Proceedings of IWoCPS-3, The Third International Workshop On Cyber Physical Systems, Bucharest, Romania*. 2014.

[Ros+14b] Daniel Rosner et al. “Image Skew Detection: A Comprehensive Study”. In: *Proceedings of IWoCPS-3, The Third International Workshop On Cyber Physical Systems, Bucharest, Romania*. 2014.

[SB09] Faisal Shafait and Thomas M. Breuel. “A simple and effective approach for border noise removal from document images”. In: *2009 IEEE 13th International Multitopic Conference* (2009), pp. 1–5.

[Seb02] Fabrizio Sebastiani. “Machine learning in automated text categorization”. In: *ACM computing surveys (CSUR)* 34.1 (2002), pp. 1–47.

[Seg04] Mark R Segal. “Machine learning benchmarks and random forest regression”. In: (2004).

[SGM+13] P Suganya, S Gayathri, N Mohanapriya, et al. “Survey on Image Enhancement Techniques”. In: *International Journal of Computer Applications Technology and Research* 2.5 (2013), 623–meta.

[SK13] Ruby Singh and Ramandeep Kaur. “Improved skew detection and correction approach using Discrete Fourier algorithm”. In: *International Journal of soft computing and Engineering* 3.4 (2013), pp. 5–7.

[SKB08] Faisal Shafait, Daniel Keysers, and Thomas Breuel. “Performance evaluation and benchmarking of six-page segmentation algorithms”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 30.6 (2008), pp. 941–954.

[SKK16] Bhavesh Kumar Shukla, Gautam Kumar, and Ashish Kumar. “An approach for Skew Detection using Hough Transform”. In: *International Journal of Computer Applications* 136.9 (2016), pp. 20–23.

[Smi07] Ray Smith. “An overview of the Tesseract OCR engine”. In: *Ninth International Conference on Document Analysis and Recognition (ICDAR 2007)*. Vol. 2. IEEE. 2007, pp. 629–633.

[Smi09] Raymond W Smith. “Hybrid page layout analysis via tab-stop detection”. In: *2009 10th International Conference on Document Analysis and Recognition*. IEEE. 2009, pp. 241–245.

[SP00] Jaakko Sauvola and Matti Pietikäinen. “Adaptive document image binarization”. In: *Pattern recognition* 33.2 (2000), pp. 225–236.

[SS04] Mehmet Sezgin and Bülent Sankur. “Survey over image thresholding techniques and quantitative performance evaluation”. In: *J. Electronic Imaging* 13 (2004), pp. 146–168.

[SS10] Faisal Shafait and Ray Smith. “Table detection in heterogeneous documents”. In: *Proceedings of the 9th IAPR International Workshop on Document Analysis Systems*. ACM. 2010, pp. 65–72.

[VK15] Purna Vithlani and CK Kumbharana. “Structural and statistical feature extraction methods for character and digit recognition”. In: *International Journal of Computer Applications* 120.24 (2015), pp. 0975–8887.

[YKM05] Burcu Yildiz, Katharina Kaiser, and Silvia Miksch. “pdf2table: A method to extract table information from pdf files”. In: *IICAI*. 2005, pp. 1773–1785.

[You11] Jamileh Yousefi. “Image Binarization using Otsu Thresholding Algorithm”. In: (2011).

[Ĉad08] M Ĉadík. “Perceptual evaluation of color-to-grayscale image conversions”. In: *Computer Graphics Forum*. Vol. 27. 7. Wiley Online Library. 2008, pp. 1745–1754.

A. Software user guide

In this appendix, we provide a user guide for compiling and running the project.

The software has been tested on Windows 10 (MSCV 2017) and Ubuntu 18 (g++ 7.4). As we do not provide any binaries, the user must compile the project first.

To compile the project, execute the following steps:

Windows:

1. Clone GitHub directory from `https://github.com/todoval/Bachelor-s-Thesis.git`.
2. Download and setup cppan.
3. Run cppan in any directory.
4. Build the project using CMake (make sure you have installed MSVC 2017 and CMake 3.8).
5. Set the environment variable `TESSDATA_PREFIX` to `tabularOCR/tessdata directory`.

Linux:

1. Clone GitHub directory from `https://github.com/todoval/Bachelor-s-Thesis.git`.
2. Install the following libraries: libleptonica-dev, libtesseract-dev, libopencv-dev.
3. Build the project using CMake (make sure you have installed g++ 7.4 and CMake 3.8).
4. Set the environment variable `TESSDATA_PREFIX` to `tabularOCR/tessdata directory`.

Follow these steps to run a sample demo:

1. Take one of the pictures from our sample images directory at `https://drive.google.com/open?id=1cPPQcOH2AYUB7jHM8_6KwlQO5YNOrJdY`, for example `11-1.jpg`, and copy it to your compiled tabularOCR executable.

2. From this directory, run command:

   - Windows: `tabularOCR.exe 11-1.jpg`
   - Linux: `./tabularOCR 11-1.jpg`

   In the rest of this user guide, we will be using a unified command call `tabularOCR`.

3. The results can be found in a results directory (located where you run the command) as both `11-1.png` and `11-1.json`.

Note 1: The input image does not necessarily have to be in the same directory as the tabularOCR compiled binary. However, in such case, the path to the image must be provided.

It is possible to run the program with several options. The explained usage with complete list of options is as follows:

```
Usage: tabularOCR [-options] (filenames | directory name)
where options include:
    (-e | --enhance) (SIMPLE | GAMMA | EQUALIZATION)
        enhance the contrast of the image before processing
    (-g | --greyscale) (AVG | MIN | MAX | LUMA)
        set the image mode to greyscale before processing
    (-b | --binarize) (OTSU | SAUVOLA)
        binarize image before processing
    -p | --preprocess
        preprocess image with the default preprocessing options
        before processing
    -sk | --deskew
        deskew image before processing
    --output-json
        output the result in a json file
    --output-image
        output the result in an png file as a bounding box
        around each cell
```

Examples of usage:

- `tabularOCR -e EQUALIZATION --binarize OTSU 11-1.jpg`

  This command will perform the table extraction on a binarized `11-1.jpg` image (with the exact method of binarization being Otsu) with enhanced contrast (with the exact method of contrast enhancement being histogram equalization).

- `tabularOCR ./image_directory`

  This command will process all the images from image_directory and save their results into a new results/image_directory directory.

- `tabularOCR -output-image 11-1.jpg 7-1.jpg 5-1.jpg`

  This command will process the images `11-1.jpg`, `7-1.jpg`, `5-1.jpg` in the given order and save their results as `11-1.png`, `7-1.png` and `5-1.png` files, excluding JSON output.

The internal C++ structure of the project is as follows:

- `main.cpp` — used for calling the individual functions of all other files consecutively.

- `parser` — contains functions for parsing the input command line arguments, file manipulation and error handling.

- `preprocess` — contains the calls of individual preprocessing methods of Leptonica depending on the parser's output.

- `process` — the core of the program. Contains the Tesseract API calls and the implementation of table recognition algorithm including various heuristics.

- `utils` — contains several helper functions and structures used in the `process` file.