Bachelor's Thesis

Segmentation Of Layout-Based Documents

Elias Kempf

Examiner: Prof. Dr. Hannah Bast
Advisers: Claudius Korzen

University of Freiburg
Faculty of Engineering
Department of Computer Science
Chair for Algorithms and Data Structures

September 14th, 2021

Writing Period

14. 06. 2021 – 14. 09. 2021

Examiner

Prof. Dr. Hannah Bast

Advisers

Claudius Korzen

Declaration

I hereby declare that I am the sole author and composer of my thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work.
I hereby also declare that my Thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

_____ _____

Place, Date Signature

Abstract

The Portable Document Format (PDF) continues to be one of the most prominent file formats for exchanging electronic documents. Thus, applications that extract high-quality text from PDF documents are also high in demand. However, PDF is a layout-based format that only stores characters and their positions rather than the plain text itself. PDF also does not store any whitespaces, and characters can be stored in an arbitrary order. Therefore, the task of extracting text in the correct reading order is unexpectedly hard to solve. In this thesis, we address an important subtask of text extraction, which is text block extraction. This task includes detecting text blocks and sorting them by reading order. In a separate step, we also split the extracted text blocks into individual words.

We propose a top-down page segmentation algorithm based on the recursive XY-cut algorithm for text block detection. Additionally, we evaluate different strategies for detecting reading order, including learning-based approaches. On average, our application detected around 51% of the expected text blocks perfectly, about 13% were split too often, and about 15% were not split enough. For reading order, we achieved an average normalized Kendall-τ-correlation [1] between expected and detected reading order of 0.873.

Contents

1 Introduction **1**

2 Related Work **5**

2.1 Rule-based approaches 5

2.1.1 Top-down approaches 6

2.1.2 Bottom-up approaches 11

2.2 Learning-based approaches 12

2.2.1 Visual segmentation using object detection 12

2.2.2 Pre-trained models for document understanding 13

2.2.3 Pre-trained models for reading order detection 15

3 Background **17**

3.1 Layout-based documents 17

3.1.1 Document 17

3.1.2 Page 18

3.1.3 Text block 18

3.1.4 Characters and Glyphs 19

3.1.5 Figures and Shapes 19

3.1.6 Bounding Box 20

3.2 Page segmentation 21

3.2.1 Cut 21

3.2.2 Representing XY-trees 22

3.2.3 Visualizing segmentation 23

4 Approach **25**

4.1 Problem Definition 25

4.2 Data preparation 26

4.3 Page segmentation algorithm 27

4.3.1 Computing valid cuts 30

4.3.2 Choosing the best cut 31

4.3.3 Splitting the page 33

4.4 Text block post-processing 34

4.5 Improving reading order 35

4.5.1 Motivation 36

4.5.2 Rule-based approaches 38

4.5.3 Score-based approaches 39

4.5.4 Context-based approaches 43

5 Experiments **47**

5.1 Datasets 47

5.2 Model training 51

5.3 Results 52

5.3.1 Methodology 52

5.3.2 Discussion 59

6 Conclusion **65**

7 Acknowledgments **67**

Bibliography **71**

List of Figures

1 Segmentation and XY-trees 7
2 Recursive XY-cut algorithm 10

3 Text blocks 19
4 Bounding boxes 21
5 Cuts and their validity 23

6 Problem definition 26
7 PdfAct output 27
8 Basic page segmentation 28
9 Computation of valid cuts 32
10 Valid cuts to choose from 32
11 Text block post-processing 35
12 Reading order detection 37
13 Fully linear logistic regressor 42
14 Fully linear context model 44
15 Transformer context model 45

16 Training and validation dataset structure 49
17 Labels from ground truth 50
18 Computation of evaluation metrics 58
19 Missing text blocks in the ground truth 60
20 Limits of the XY-cut method 63

List of Tables

1 Text block detection 59
2 Reading order detection 61

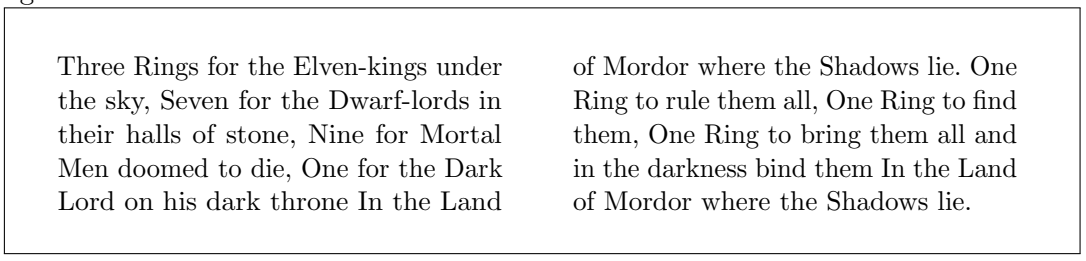
List of Algorithms

1 Page segmentation algorithm 29
2 Recursive XY-cut algorithm 29

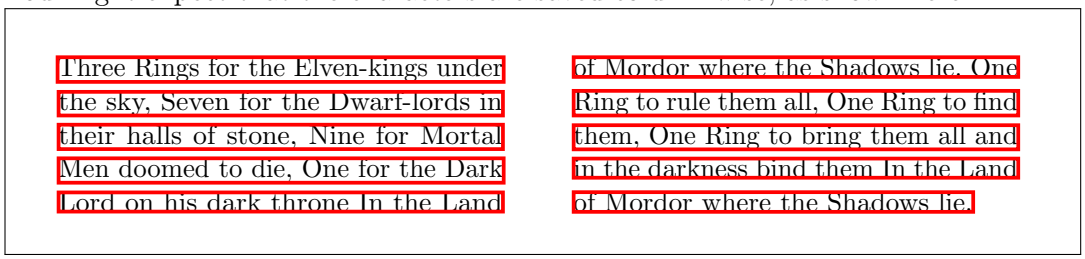
1 Introduction

PDF is one of the most widely used file formats to store and exchange textual information. It is operating-system independent and convenient to use. PDFs come with an increasing number of features like digital forms or contracts to an extent where they can replace regular paper documents completely. This fact combined with its usability explains the sheer amount of PDF-Viewers and similar tools available today that allow working with PDFs easily. Features like text search, copying, or extracting text from PDFs have become standard. However, these features do not always work as well as we would expect. Sometimes we may be unable to find some words using the search even though they exist within the document.

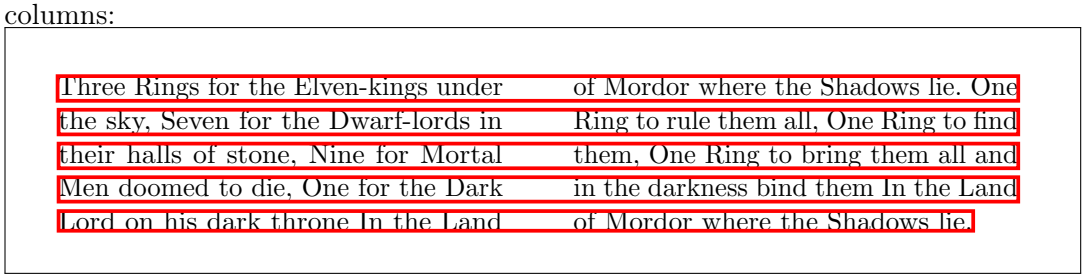
We can understand how such mistakes can happen when looking at how PDF works. PDF is a layout-based format, meaning that it does not store text line-by-line or word-by-word but character-by-character. To complicate matters further, characters do not even have to be stored in the natural reading order of the document. PDF also does not store whitespace characters. Spacing is created by adjusting the positions of characters belonging to different words. Thus, there is no trivial way to detect word borders. For instance, let us consider a two-column layout, as seen in the following figure:



You might expect that the characters are saved column-wise, as shown here:



However, PDF could also store the characters from left to right and therefore across columns:



An application that provides a text search feature needs a list of all the words in the document. Detecting the reading order is also necessary to process search queries consisting of multiple words. For example, without knowing the correct reading order, we may not get any results when searching for "under the sky" in our example page. Though, we might obtain results for "under of Mordor". Finally, PDFs also come in an almost arbitrary amount of different layouts. Overall, this makes the extraction of words and reading order from PDFs a surprisingly difficult task.

In this thesis, we propose a page segmentation algorithm based on the recursive XY-cut algorithm. Page segmentation is the process of segmenting the characters of a layout-based document back into semantic units like words, lines, text blocks, or columns. In this work, we will segment characters into text blocks. In addition to detecting these text blocks, we also sort them by reading order. Reading order will be a primary focus of this work, as this is a common weakness of many existing tools [2]. In particular, we will evaluate multiple rule-based and learning-based approaches for detecting reading order.

The rest of this thesis is structured as follows: In Chapter 2, we will review publications relevant to our work. In Chapter 3, we will discuss necessary background

information about layout-based documents and page segmentation and introduce the terminology used in this thesis. Chapter 4 presents our approach, mainly the core concepts and structure of our page segmentation algorithm. Our datasets, details on the learning-based aspects of this work, and evaluation results are discussed in Chapter 5. Lastly, Chapter 6 concludes this thesis.

2 Related Work

Approaches for page segmentation have been the subject of extensive research for over 30 years. In this chapter, we want to discuss different techniques that are employed to segment layout-based documents. In particular, we will discuss different rule-based and learning-based segmentation approaches.

2.1 Rule-based approaches

They are two main types of rule-based segmentation: top-down approaches and bottom-up approaches. Top-down approaches start by dividing a complete page into smaller blocks. Namely, they start segmenting at the *top* and go *down* by dividing further and further. These divisions are chosen based on information considering the whole page. For example, this can be information about free space between units or font types and font sizes. A different approach for page segmentation is the bottom-up approach. These methods begin by considering the smallest units of the page (e.g., characters). From that point on, they start to aggregate these small units into larger units (e.g., words or text blocks) using only local information about the initial units.

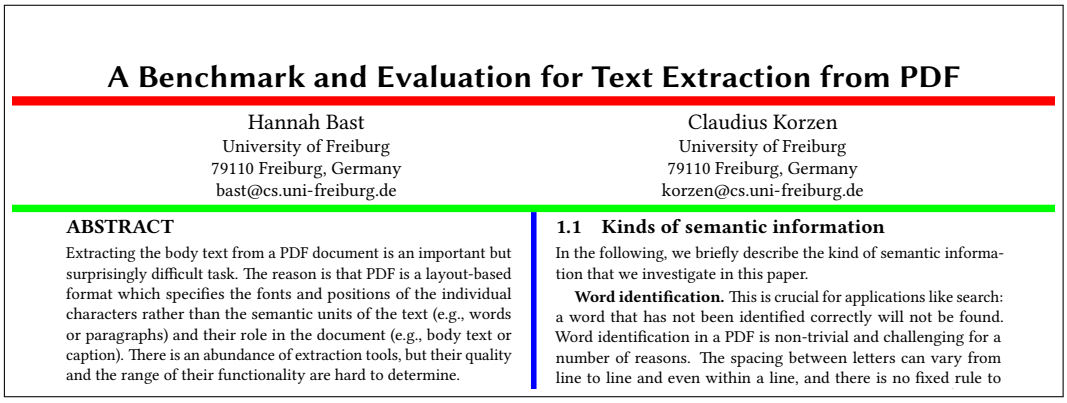
2.1.1 Top-down approaches

Almost 30 years ago, in 1992, Nagy et al. published a paper about layout analysis on scientific articles [3]. In their paper, among other things, they present a data structure called XY-tree. They use it to describe the layout of a page by segmenting it into rectangular blocks. These blocks can represent words, text lines, columns, etc., depending on how a page is segmented. The chosen granularity of the subdivisions can vary depending on the task that is solved. Nagy et al. segment each page down to paragraph-level.

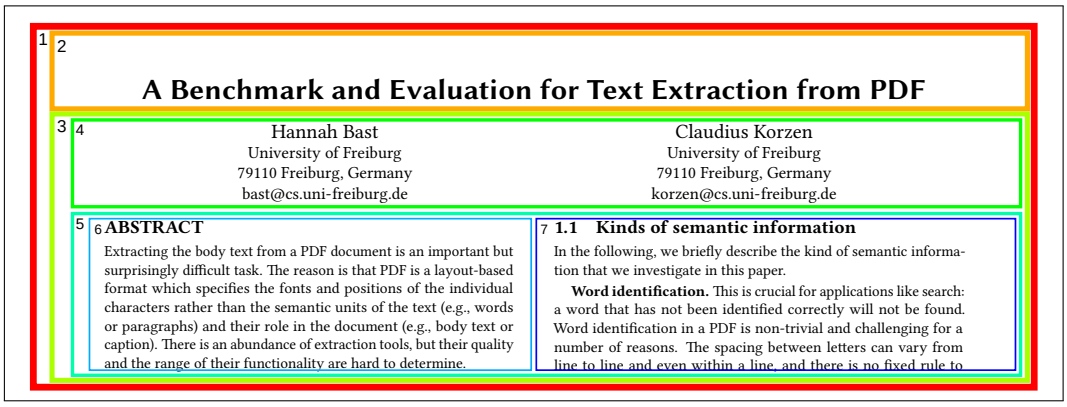
Nagy et al. create their segmentation by successively applying horizontal or vertical subdivisions (also called *cuts*) to the page. An XY-tree represents the resulting hierarchy of these cuts. Hence the name XY-tree as only cuts through the x- or the y-axis are allowed. Also, the hierarchy of cuts forms a tree structure. Each node of an XY-tree corresponds to a rectangular block on the page. If a block is cut, the two resulting subblocks will be child nodes of the original node in the XY-tree. Figure 1 shows cuts that divide a page into nested rectangles and an XY-tree representing this segmentation.

To obtain an accurate description of a document's layout this way, we have to choose these cuts carefully. That is, we need to divide the page using the correct cuts. However, before choosing the correct cuts, we first need to compute where we can cut at all. This step is essential for meaningful page segmentation. For example, it does not make sense to choose a cut that crosses the text on the page. Nagy et al. solve this problem by scanning the document horizontally and vertically in a raster scanning pattern. They then identify all rows and columns of the scanning raster that contain only white pixels. After we have computed the possible cuts, we can start segmenting the document.

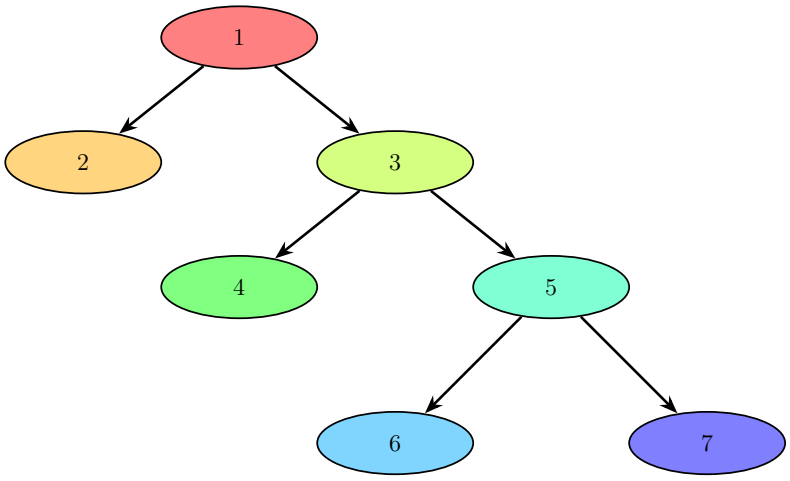
There are multiple approaches to construct an XY-tree from a given page layout. Nagy et al. use block grammars for this purpose. A block grammar is a set of rules on how to subdivide a given block further. For example, a title page is usually



(a) Horizontal and vertical subdivisions applied to a page



(b) Resulting subdivision of the page into nested rectangles



(c) XY-tree representing the subdivision

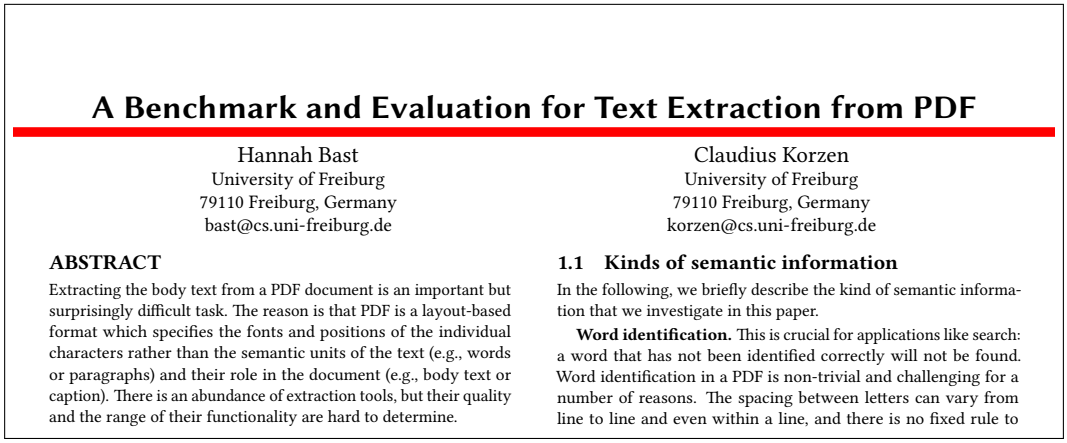
Figure 1: Segmentation and XY-trees. Figure (a) shows a simple segmentation consisting of only three cuts, (b) shows the corresponding division into nested rectangles, and (c) shows the corresponding XY-tree.

split into a headline, authors, and the text body. A corresponding block grammar could specify the expected positions of these units and indicate which cuts to choose. Each of the resulting blocks then has its own block grammar defining how to divide further. However, these rules are always heuristic, meaning what holds true for one document might not hold for another. Usually, these rules are also not known from the start but have to be inferred indirectly from the layout specification. This can be done by analyzing geometric properties of a specific layout type like heading, line, and paragraph spacing. It is also possible to *learn* such grammars using a machine-learning approach as shown by Shilman et al. [4] in 2005.
Nagy et al. use their segmentation results to associate blocks of plain text with their original positions on the page. They do not need more information about the reading order for their purpose. Thus, no further reading order detection is performed.
Overall the approach of Nagy et al. still has weaknesses. Their block grammars require *a priori* knowledge about the expected layout to split pages correctly. When scanning documents, they are also susceptible to skew (i.e., misalignment of the page while scanning). Too much skew can prevent possible horizontal or vertical subdivisions. Lastly, when increasing the scanning resolution, their approach becomes quite time-intensive because they consider all of the scanned pixels.

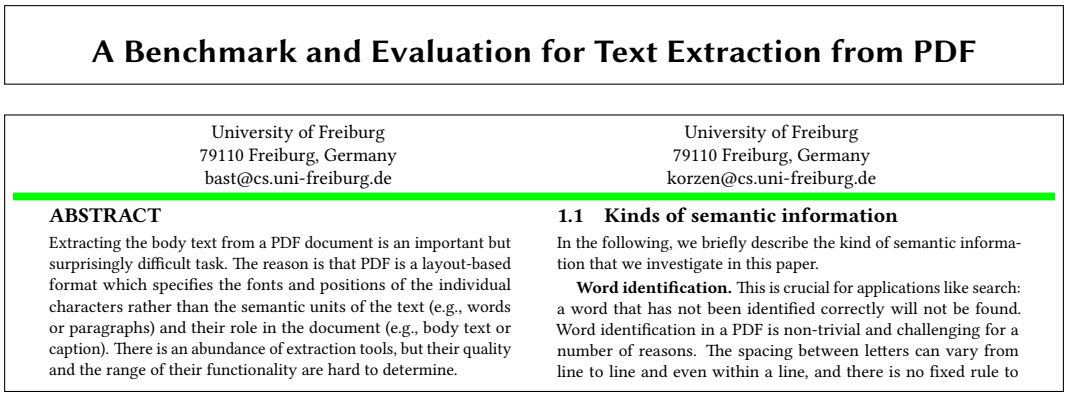
Three years later, in 1995, Ha et al. proposed a refinement of the XY-tree approach [5]. The method of Nagy et al. computed possible cuts by looking at every pixel of the scanned document. Even at a lower scanning resolution, this process can become quite time-intensive. Ha et al. present means to reduce the computational complexity of this problem. They achieve this by considering bounding boxes of connected components (e.g., symbols like letters, digits, punctuation, etc.) instead of individual pixels.
They apply their technique to the *recursive XY-cut* algorithm. This algorithm constructs an XY-tree by choosing a cut through the page and recursively splitting the resulting subpages. In each recursion step, Ha et al. split the page along all cuts in a specific direction that exceed a certain size threshold. The direction of the

considered cuts alternates between recursion steps. Ha et al. mention that their XY-tree allows a sequential ordering of the detected blocks. However, no further details on how this order can be inferred are provided. Figure 2 shows an example execution of the recursive XY-cut algorithm up to a depth of 3.
The approach of Ha et al. improves upon some of the weaknesses the method of Nagy et al. had. In particular, they improved efficiency and reduced skew dependency by using skew correction. However, their approach almost blindly divides a page wherever possible until a certain threshold. This leaves much room for improvement in the choice of cuts.
For our approach, we use a slightly modified version of the XY-tree data structure. That is, our algorithm only constructs *binary* XY-trees, whereas Nagy et al. and Ha et al. allow multiple cuts within a single recursion step. We make use of the techniques proposed in Ha's paper by using a recursive XY-cut structure that computes valid cuts using bounding boxes. We explain the full details in Chapter 4.

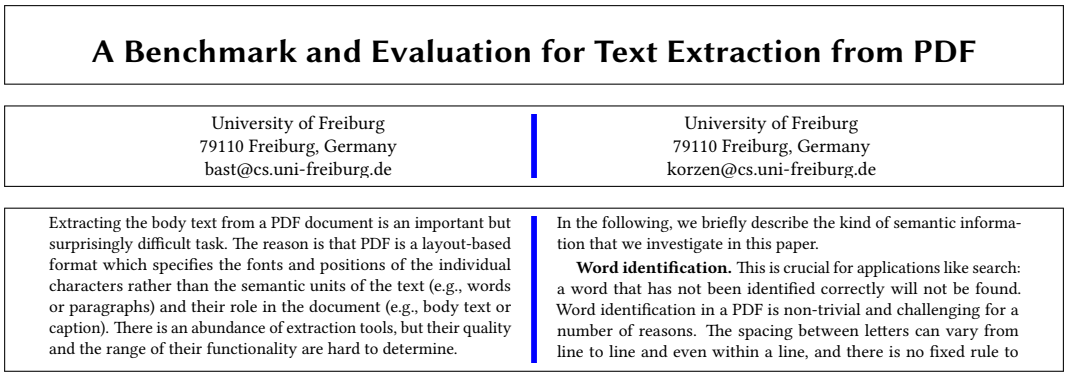
An XY-cut approach specifically targeted at detecting reading order was proposed by Meunier in 2005 [6]. As suggested by Ha et al., Meunier implemented his XY-cut algorithm on bounding boxes. His approach also divides each subpage until no cuts above a certain size threshold are possible. Thus, the method of Meunier only divides each page into rectangular blocks while not targeting proper text block detection. He also proposes to choose cuts by maximizing a score function that assesses the resulting blocks and their order. His score function favors arranging the created blocks into columns, as he found this to be most suited for scientific articles. He achieves this by rewarding the cumulative height of the resulting columns. Meunier employs a dynamic programming approach (i.e., a combination of recursion and memoization) for efficiently maximizing his score function.
On an evaluation set of 800 document pages, Meunier's approach ordered 98% of blocks correctly. Unfortunately, no more precise evaluation results are presented. Nevertheless, Meunier's algorithm seems to perform well even though it only considers geometric features (e.g., distances between lines, heights of columns, etc.). His results



(a) Whole page on depth 1, next cut (represented by the red line) is between title and authors.



(b) Subpages on depth 2, next cut (represented by the green line) is between authors and columns.



(c) Subpages on depth 3, next cuts (represented by the blue lines) are between authors and between columns.

Figure 2: Recursive XY-cut algorithm. The figure shows subpages across three recursion steps. In each subpage, we marked where the subpage will be split in the next step.

also agree with our evaluation, in a sense that we also achieved decent results in reading order detection using only geometric features (see Section 5.3.2).

2.1.2 Bottom-up approaches

Another method for page segmentation is the so-called *bottom-up* approach. In contrast to our previous examples, a bottom-up approach would start combining characters into words, words into lines, lines into text blocks, and so forth.
O'Gorman proposed a noteworthy representative of this segmentation approach in 1993, called the *docstrum* (short for *Document Spectrum*) [7]. He used a nearest-neighbors clustering [8] to aggregate page components. Nearest neighbor clustering works by computing distances between all components on a page and combining components that are close together. O'Gorman's method focused primarily on reassembling text blocks and did not specifically sort the detected blocks by reading order.
Bottom-up approaches are often better applicable to various layout types than top-down approaches. This is because layouts usually differ more on the global scale of headlines, columns, etc., than locally on a character scale. However, many bottom-up algorithms, including docstrum, suffered from their quadratic complexity in time and space due to computing distances for all unit pairs on a page. In 1997, Simon et al. showed that complexity can be reduced to linear when using heuristics and path compression [9]. They achieved this using an approach based on Kruskal's algorithm [10] and a special distance-metric between unit pairs. They represented the final layout hierarchy as a minimum-spanning tree.

2.2 Learning-based approaches

There are many different learning-based approaches to page segmentation. The most common approaches we will discuss in this section are: (1) visual segmentation approaches using object detection on document images, and (2) semantic segmentation approaches leveraging textual and layout information.

2.2.1 Visual segmentation using object detection

Convolutional neural networks (CNNs) have long been used for computer vision tasks like image classification. They have been improved and extended many times. A particular noteworthy extension of the classic CNN is the region-based CNN (R-CNN) [11]. R-CNNs excel at object detection tasks. In contrast to regular CNNs, they do not simply classify an image, but they propose bounding boxes of objects detected in the image. For each detected object, the R-CNN then predicts a label (e.g., person, cat, bicycle, etc.).

While not being designed for page segmentation, CNNs and, in particular, R-CNNs can also be applied to document images. Yi et al. showed which modifications were necessary to reliably detect semantic units like headings, paragraphs, figures, tables, etc., in document images [12]. For their work, they partially redesigned the R-CNN, its training strategy, and its network structure. In particular, they scaled down the network size and adjusted the region-proposal step to better suit document images, which are far less noisy than natural images and rarely contain overlapping objects. For page segmentation, their modified R-CNN model significantly outperformed all the other non-modified R-CNN models. In particular, when detecting text lines, formulas, figures, and tables, they achieved a mean average precision of 81.5%. Interestingly, their model excelled at detecting text lines at an average precision of 95.3% whereas the model struggled noticeably with figures at an average precision of only 66.8%.

Even though R-CNNs are exceptionally suited for object detection tasks and have been

shown to work well on document images, they all come with a disadvantage. That is, they all work exclusively on document *images*. When converting a digital-born PDF into an image, potentially important information is lost. This includes information such as font names and font sizes. For example, we could use font information to distinguish headlines from paragraphs, footnotes from body text, etc. Positional information on individual characters, while still implicitly contained in the document image, also gets blurred.

A recent blog post by Team Konfuzio, published February 2021, demonstrates an another example of page segmentation on document images [13]. In their blog post, they describe their approach for automatic text summarization in PDFs. Automatic text summarization is a process where text is extracted from a PDF first and then the key aspects of the said text are summarized. Thus, the first step of their work is closely related to what we want to achieve: page segmentation. But as we will see, their approach is quite different. They work on a purely visual level using images of the considered documents. This allows them to use the visual segmentation techniques described above but also introduces potential loss of important information. Their pipeline consists of three main steps. First, they employ a computer vision approach using faster R-CNNs [14] for object detection. This step divides the document into semantic units. Team Konfuzio used the following semantic units: title, text, table, list, and figure. Second, they convert all elements that classify as text into actual text. This conversion is performed by using optical character recognition (OCR). In the third step, they summarize the extracted text using a transformer model [15]. We do not provide further detail as this step is not relevant to our work.

2.2.2 Pre-trained models for document understanding

In 2019, Xu et al. presented a paper on a pre-trained deep learning model for document image understanding, called LayoutLM [16]. Document image understanding describes various tasks that all aim to extract and structure information from scanned document

images. These tasks include *form understanding*, *receipt understanding*, and *document image classification*. Form understanding is the task of extracting and structuring textual contents of forms (e.g., extracting key-value pairs like *name*, age, etc. together with their respective values). Receipt understanding involves using a scanned receipt image to fill out predefined semantic slots (e.g., company, address, date, and total). Document image classification aims to predict a class label (e.g., letter, form, email, etc.) for each document image. Up until this point, most other approaches only focused on processing textual information. That is, they neglected layout information which is crucial for document image understanding. The model of Xu et al. jointly processes both text and layout information at the same time. Until this point, these two types of information have not been modeled simultaneously within a single framework. Using their model, they achieve new state-of-the-art results in the previously mentioned document image understanding tasks. In particular, they managed to outperform two of the previous pretrained state-of-the-art natural language processing models: BERT [17] and RoBERTa [18]. In form understanding, they achieved an F1 score of 0.7927, improving on the previous best of 0.656. In receipt understanding, they achieved an F1 score of 0.9524. In document image classification, they achieved a classification accuracy of 94.42%. Despite their strong results, their approach does not directly consider reading order. Reading order is only implicitly modeled as part of the general document understanding task. Nevertheless, the model of Xu et al. exemplifies how to use extracted text for deeper document understanding. Even though they do not work directly on extracted characters like our approach does, we can also apply the idea of combining semantic information with layout information to our page segmentation algorithm. That is, we leverage information on semantic roles of text blocks to improve our reading order detection. The details of this step are explained in Section 4.5.

2.2.3 Pre-trained models for reading order detection

Recently, in August 2021, Wang et al. proposed an extension of the LayoutLM model [16], the LayoutReader [19]. In their publication, they focused specifically on reading order detection. Only a few previous approaches for detecting reading order took advantage of more advanced deep learning techniques. The ones that did were mostly trained on specifically created in-house datasets and thus not easily transferable to other tasks. Wang et al. made two major contributions to improve this situation. First, they present their dataset for reading order detection that consists of 500,000 document pages. They created their dataset by leveraging the reading order information embedded in the XML metadata of WORD documents. In addition to making their dataset publicly available, they provide a pre-trained deep learning model to solve the reading order detection task. Like LayoutLM, they trained their model on text and layout information. To solve the reading order prediction, they employ a sequence-to-sequence (seq2seq) approach [20]. They use sequences of words on a page together with their bounding boxes as an input for their model. In the encoding step, they feed the input sequence into LayoutLM while using a self-attention mask. However, they modify the sequence generation step to predict indices in the input sequence. The predicted indices represent the detected reading order of the input sequence. Wang et al. managed to achieve an average page-level BLEU score [21] of 0.9819 and an average relative distance of 1.75. Compared to their other baselines, LayoutReader thus achieves state-of-the-art results in reading order detection.

3 Background

In this chapter, we want to discuss the key concepts and terminology used throughout this work. Most of these definitions are motivated by PDF not being a text-based but a layout-based format. PDF stores individual elements like glyphs, figures, and shapes together with their respective attributes. For each of these concepts, we will (1) give an intuitive but likely informal description and (2) a precise formal definition for each term.

3.1 Layout-based documents

3.1.1 Document

To represent a multi-page document, we will use a list of pages. We intuitively order this list by page numbers in ascending order. Formally, let $P = \{P_1, \ldots, P_k\}$ be a set of k pages ($k > 0$). A document D consisting of these pages is a totally ordered set $(P, <_P)$, where $<_P$ represents the natural order induced by the page numbers. Or formally, for all $P_i, P_j \in P$

$$P_i <_P P_j \iff P_i.\text{page\_number} < P_j.\text{page\_number}.$$

3.1.2 Page

Every page of a layout-based document has its specific contents (e.g., text organized in paragraphs, tables, figures, etc.). The height and width of the page can also convey important information about the layout. Lastly, we use page numbers to distinguish pages, as we will mainly work with multi-page documents. We define a page to be a class with the following attributes: (1) a list of glyphs (see Section 3.1.4), (2) a list of figures and shapes (see Section 3.1.5), (3) height, (4) width, and (5) a page number. When we talk about glyphs, figures, or shapes on a page, we also want to know their position. Therefore, we use a coordinate system to describe these positions on a page. The origin of this coordinate system is the lower-left corner of each page, x-coordinates increase from left to right and y-coordinates from bottom to top. The unit of measurement is the typographic *point* (pt). 1 pt is equal to $\frac{1}{72}$ inch.

3.1.3 Text block

A text block is a set of words that are adjacent in a document's layout. Sections or paragraphs can consist of multiple text blocks. Text blocks differ from paragraphs in that they can also contain non-body text like headings or author information. Text blocks also have semantic roles within a document. Throughout this work, we use the following semantic roles: title, author (includes names, affiliations, and email addresses), heading, paragraph, abstract, caption, date, footnote, formula, marginal (e.g., page headers, footers, or page numbers), reference, table, table-of-contents, and other (used if no suitable role can be predicted). Figure 3 shows text blocks and their semantic roles on an example page.

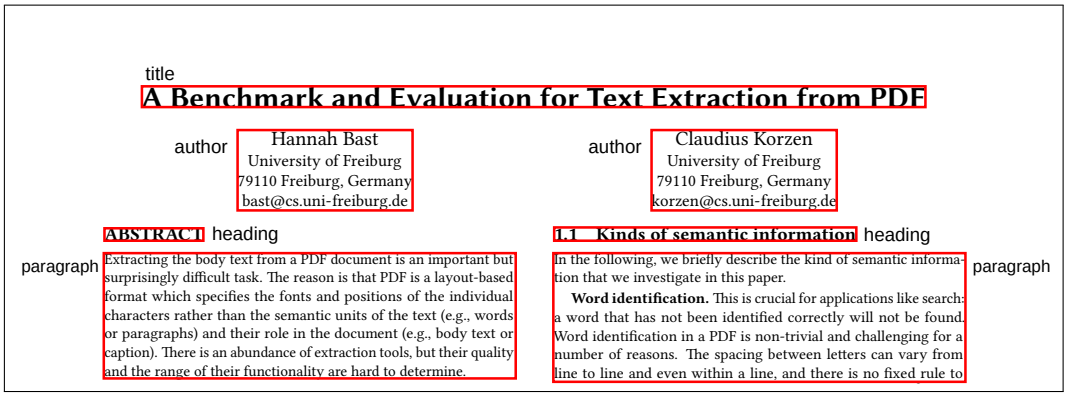


Figure 3: Text blocks. A page split into text blocks, each with its semantic role.

3.1.4 Characters and Glyphs

The term character refers to a symbol, most often a letter or a digit. Every character in a layout-based document has a visual representation within the document. This representation depends on additional attributes like font, font size, font color, etc., associated with the character. We call this visual representation of a character a glyph. In addition to their visual attributes, every glyph has a bounding box (see Section 3.1.6).

More precisely, we consider the following attributes of a glyph: (1) an underlying character, (2) a bounding box (see Section 3.1.6), (3) a font name, (4) a font size, and (5) additional font specifiers for **bold** and *italic*. Font color is another visual attribute of a glyph which we, however not consider in this work.

3.1.5 Figures and Shapes

Figures and shapes are visual units within a layout-based document. Figures mostly correspond to images, graphics, etc., within a document. On the other hand, shapes are small visual elements that make up compound images or graphics (e.g., lines, circles, or curves). For figures and shapes, we only consider the bounding box as an attribute (see Section 3.1.6).

Note that figures and shapes do not directly influence the textual content of a PDF. However, they do affect the layout. Yet, we only need to know that they exist and where they are on the page. That is the reason we do not consider additional attributes figures and shapes could possess (e.g., a figure's content or a shape's color).

3.1.6 Bounding Box

The bounding box of a glyph, a figure, or a shape is the smallest rectangle that completely encloses said object. More specifically, it is the smallest rectangle that contains every pixel of the graphical representation of the object. Formally, let O be one of glyph, figure, or shape. We can uniquely define a rectangle by its lower left and its upper right corner. Thus, the bounding box of O can be defined as two points (x_1, y_1) and (x_2, y_2) such that

$$x_1 \leq x \leq x_2 \text{ and } y_1 \leq y \leq y_2$$

for each pixel (x, y) belonging to O where x_1 and y_1 are maximal and x_2 and y_2 are minimal with this property.

This definition canonically extends to compound objects like words, lines, text blocks, or tables. More specifically, the bounding box of a word is the smallest rectangle that contains the bounding box of every glyph belonging to the word. Analogously, the bounding box of a line contains the bounding boxes of its words, the bounding box of a table contains the bounding boxes of all glyphs, figures, or shapes in the table, and so forth. Note that a bounding box also describes the position of the object it encloses. Figure 4 shows the bounding boxes of words on a page.

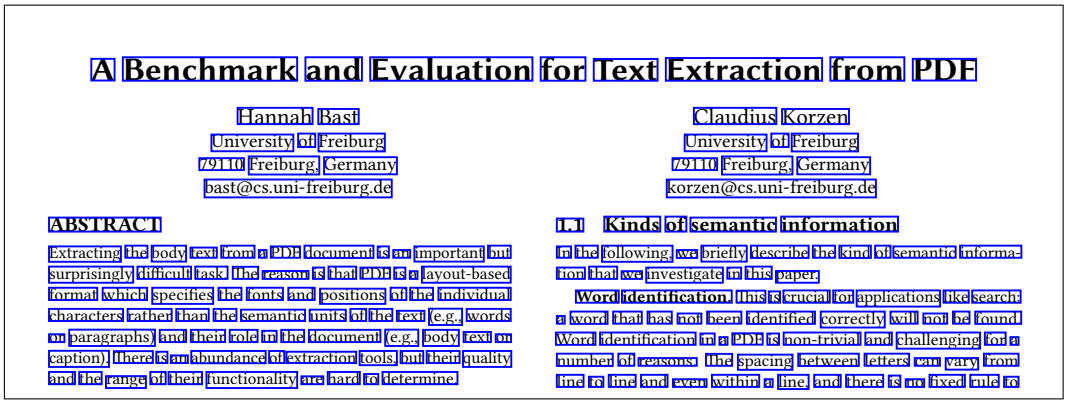


Figure 4: Bounding boxes. The figure shows bounding boxes of words on a page.

3.2 Page segmentation

3.2.1 Cut

A cut specifies the location and the direction (horizontal or vertical) of a subdivision on a page. A cut consists of an interval $[a, b] \subset \mathbb{R}$, specifying the position on the page we cut through, and a flag, specifying if the interval is on the x- or the y-axis. We use $\mathbf{Y}$ for horizontal and $\mathbf{X}$ for vertical cuts. Therefore, from now on, we will also refer to horizontal cuts as y-cuts (as they cut through the y-axis of the page) and vertical cuts as x-cuts (as they cut through the x-axis of the page). Formally, a cut C is an element of $Cuts = \{[a, b] \subset \mathbb{R} \mid a \leq b\} \times \{\mathbf{X}, \mathbf{Y}\}$.

Later on, we also want to compare cuts by their *size*. We define the size of a cut to be the length of its corresponding interval. Note that this definition is independent of the direction of a cut. Formally, we express the size as a function

$$\begin{aligned} size: \quad Cuts \quad &\rightarrow \quad \mathbb{R}^+ \\ ([a, b], \mathbf{dir}) \quad &\mapsto \quad b - a. \end{aligned}$$

To divide a page using cuts, we need to compute the set of *valid* cuts. A cut is valid on a page if no glyphs, figures, or shapes on the page overlap with the range of the cut. Each valid cut divides a page into two subpages. In which order we should read

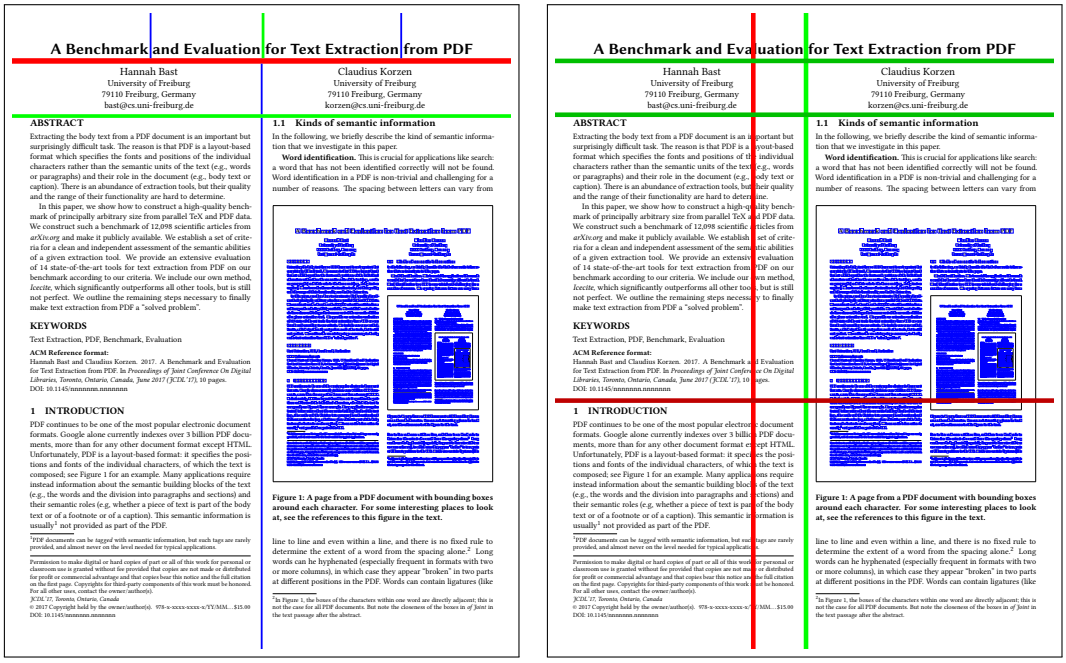
these subpages is defined by the direction of a cut. Here, we need to distinguish between x-cuts and y-cuts. An x-cut divides the page vertically into a left and a right subpage. We define the reading order according to our left-to-right writing system as left first, then right. Analogously, y-cuts divide the page horizontally, and we define the upper subpage comes first, then the lower one, as it is common in our writing system. Note that when using other writing systems like right-to-left (e.g., Arabic, Hebrew) or vertical (e.g., Chinese, Japanese), we need to adjust this definition accordingly. Figure 5a gives an example of cuts dividing a page.
We will now precisely define when a cut is valid on a given page. Let $C = ([a, b], \mathbf{dir})$ be a cut and P a page. Given a bounding box $B = (x_1, y_1), (x_2, y_2)$, we say

$$B \text{ does not overlap } C \iff \begin{cases} x_2 \leq a \text{ or } b \leq x_1, & \text{for } \mathbf{x}\text{-cuts}, \\ y_2 \leq a \text{ or } b \leq y_1, & \text{for } \mathbf{y}\text{-cuts}. \end{cases}$$

We now define that a cut C is valid on page P, iff no bounding box $(x_1, y_1), (x_2, y_2)$ of any object on P overlaps with C. Note that if a cut is valid or not is determined only by the bounding boxes of page elements. However, if a cut makes sense or not on a specific page has to be decided by a cut-choosing strategy (see Section 4.3.2). Figure 5b shows some examples of valid and invalid cuts on a page.

3.2.2 Representing XY-trees

In this work, we use nested lists to represent XY-trees. Each inner node of an XY-tree is a list consisting of three elements: (1) the cut chosen on the subpage represented by the current node, (2) the left subtree of the current node, and (3) the right subtree of the current node. Leaf nodes are empty lists. Note that nodes of the XY-tree still correspond to blocks on the current page. The root node represents the blocks corresponding to the whole page. The left and right child nodes of the root represent the subblocks that emerge when applying the cut associated with the root to the



(a) Cuts dividing a page into subpages **(b)** Valid and invalid cuts on a page

Figure 5: Cuts and their validity. (a) shows a page divided into smaller subpages by cuts, and (b) shows three valid cuts (green) and two invalid cuts (red) that either collide with the glyphs or the figure on the page.

whole page. In general, the child nodes of a node represent subblocks of their parent block.

3.2.3 Visualizing segmentation

In this work, we often use figures to visualize a segmentation. Depending on what the figure shows, we may visualize the chosen cuts, the resulting nested rectangles, or the detected text blocks. When visualizing chosen cuts or nested rectangles, we try to visually distinguish different recursion levels. We do this in two ways: (1) we color-code objects with respect to increasing recursion depth from red to green to blue (red is the top-level and blue is the deepest visualized recursion level), and (2) we decrease line or border width with increasing recursion depth (object on deeper

recursion levels appear thinner). For detected text blocks, we also the described coloring scheme to visualize the detected reading order. When recursion depth or the order of the cuts or text blocks are particularly important within a figure, we will also enumerate cuts accordingly. Figure 5a shows such a visualization leveraging color and width to visualize recursion depth. Note that despite also using colors, Figure 5b shows different valid and invalid cuts on the *same* recursion level. In that example, colors visualize cut validity and not recursion depth.

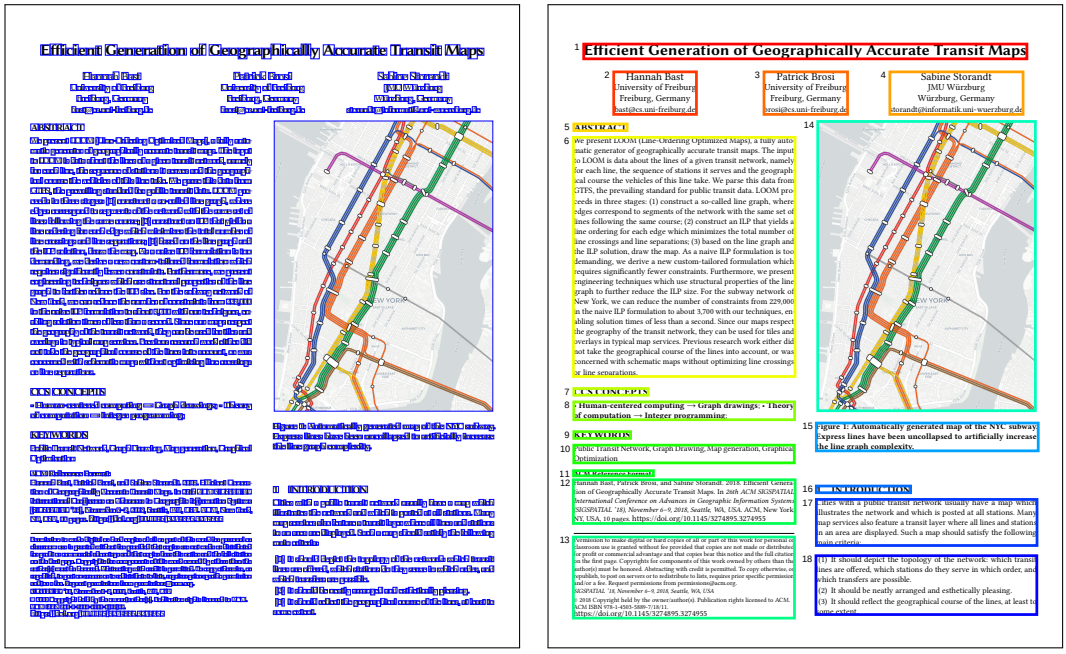
4 Approach

In this chapter, we will discuss our approach and the basic structure of our page segmentation algorithm. For this, we will give a precise definition of the problem we want to solve first. After that, we will briefly talk about the necessary data preparation and then describe the general structure of the algorithm. In particular, we will discuss the functionality of the three major subroutines the page segmentation algorithm is built on. We will also explain how we post-process the detected text blocks. Lastly, we propose a method for improving reading order detection and discuss the different strategies we used for this.

4.1 Problem Definition

Intuitively the problem we want to solve consists of two parts: For a given document, (1) we want to detect text blocks using page segmentation and then (2) sort the detected text blocks by reading order. We illustrate an input and a corresponding output in Figure 6.

Let us now be more precise: the input for our algorithm is a list of pages that represents the document we want to segment. Each page comes with the glyphs, figures, and shapes it contains (we explained the properties of these objects in detail in Section 3.1). The resulting output is a division of the given document into text blocks sorted by reading order.



(a) Input for the algorithm **(b)** Output of the algorithm

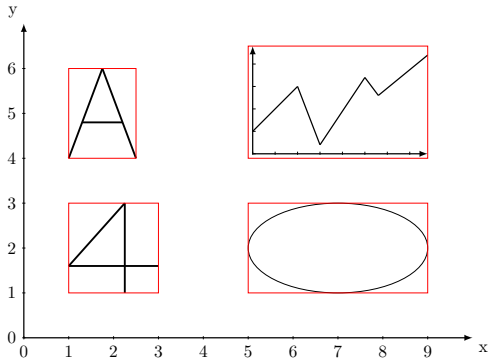
Figure 6: Problem definition. Figure (a) shows a page and the bounding boxes of all the glyphs, figures, and shapes we get as an input for the algorithm. (b) shows the output of the algorithm, that is the detected text blocks sorted by reading order.

4.2 Data preparation

To properly prepare the input for our algorithm, we need to extract the glyphs, figures, and shapes from the given PDF. For this step, we use a PDF extraction tool called *PdfAct* [22] developed by Korzen in 2017. PdfAct provides the extracted page elements together with the dimensions of each page in a JSON file. Figure 7 shows a simple example page and the corresponding output from PdfAct. We omitted some additional glyph attributes from the example for better readability (e.g., font names, specifiers like bold or italic).

We then use a self-implemented parser that converts the information provided by PdfAct into instantiations of our predefined classes (see Section 3.1). Note that all glyphs, figures, and shapes belong to a page. Thus, we can represent the whole

document by a list of pages. This preparation step yields the desired input format for our algorithm.



(a) Example page with bounding boxes (coordinate grid for illustration only)

```
{
    "glyphs": [
      {
        "char": "A",
        "font size": "11pt",
        "bounding box": [1, 4, 2.5, 6],
        "page": 1
      },
      {
        "char": "4",
        "font size": "11pt",
        "bounding box": [1, 1, 3, 3],
        "page": 1
      }],
    "figures": [
      { "bounding box": [5, 4, 9, 6.5], "page": 1 }],
    "shapes": [
      { "bounding box": [5, 1, 9, 3], "page": 1 }],
    "pages": [
      { "number": 1, "width": 10, "height": 7 }]
}
```

(b) Simplified output of PdfAct

Figure 7: PdfAct output. Figure (a) shows an example page with two glyphs, one figure, and one shape. (b) shows the simplified output of PdfAct.

4.3 Page segmentation algorithm

In the previous section, we discussed how to extract a list of pages from a given PDF. In this step, we will talk about how the algorithm segments pages to detect text blocks. We segment each page as follows:

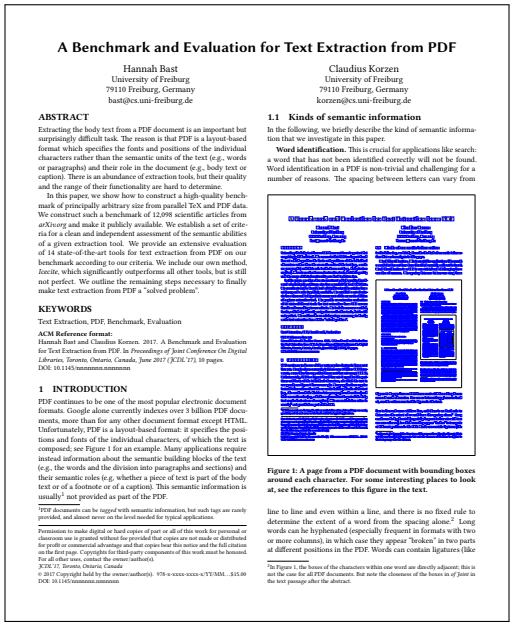
(1) start with the rectangle representing the whole page,

(2) compute all valid x- and y-cuts on the page,

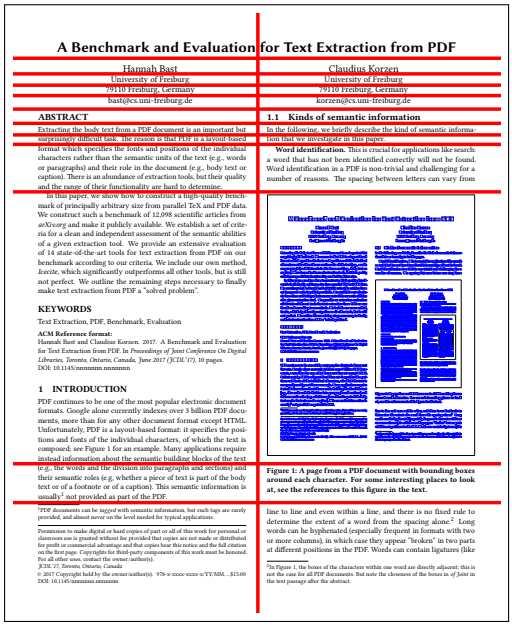
(3) choose one of the valid cuts,

(4) split the page along the chosen cut (this step generates two subpages)

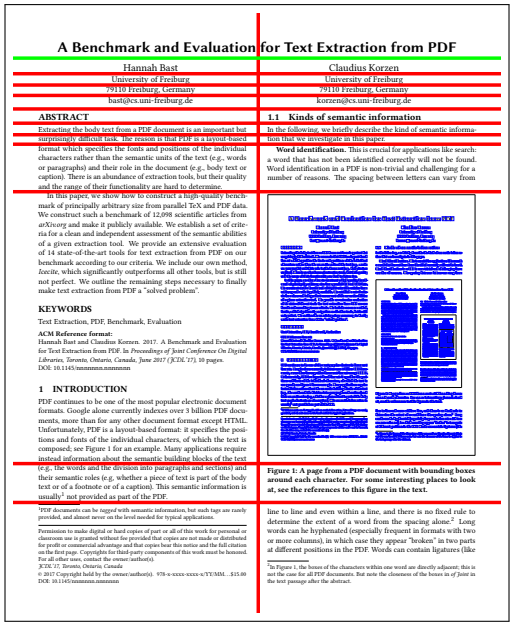
(5) construct the XY-tree by using the chosen cut as a root node and computing the left and right subtrees by recursing on the two subpages.



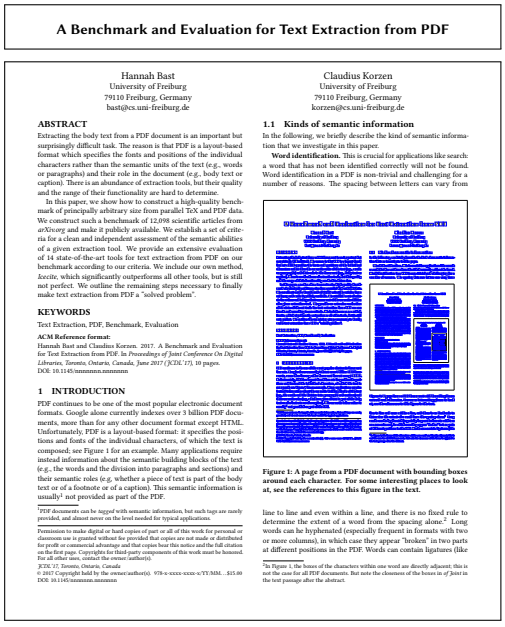
(a) Starting point of the algorithm



(b) Computing valid cuts on the current page



(c) Choosing the best valid cut



(d) Split page along the chosen cut and recurse on subpages

Figure 8: Basic page segmentation. (a) shows the block representing the whole page, (b) shows the computed valid cuts on the page, (c) indicates the best cut we use to split the page, (d) shows the resulting subdivision (i.e., the subpages we recurse on).

We illustrated steps (1)-(4) in Figure 8. The design of the algorithm, especially the page segmentation using XY-trees, is influenced by the already mentioned paper on XY-trees by Nagy et al. [3]. Our algorithm also employs a recursive XY-cut approach using bounding boxes as described by Ha et al. [5]. We will present the basic structure of the algorithm in pseudo-code.

Algorithm 1 Page segmentation algorithm

Input: list of pages $p_1, \ldots, p_n$
Output: list of XY-trees $t_1, \ldots, t_n$ (where t_i is the tree for page p_i)
1: Initialize $t_1, \ldots, t_n$ as empty trees
2: **foreach** page p_i **do**
3: $\quad t_i \leftarrow$ ***recursive-xy***(p_i)
4: **end for**
5: **return** $t_1, \ldots, t_n$

where ***recursive-xy*** is the recursive XY-cut algorithm that builds an XY-tree in a nested list representation (see Section 3.2.2) for each page of our document. ***recursive-xy*** is defined as:

Algorithm 2 Recursive XY-cut algorithm

Input: a page p
Output: an XY-tree t
1: $cuts \leftarrow$ ***valid-cuts***(p) ▷ Compute valid cuts
2: *best-cut* $\leftarrow$ ***choose-best-cut***$(cuts)$ ▷ Choose the best cut
3: **if** *best-cut* **is null then**
4: $\quad$ **return** list() ▷ A leaf node is created in the base case of the recursion
5: **end if**
6: $p_1, p_2 \leftarrow$ ***split-page***$(p, best\text{-}cut)$ ▷ Split page into two subpages
7: **return** list(*best-cut*, ***recursive-xy***(p_1),
8: $\quad$ ***recursive-xy***(p_2)) ▷ Build nested list representation of the XY-tree

In line 1 of the recursive XY-cut algorithm, we compute all valid cuts on the current subpage (see Section 4.3.1). In line 2, we make a cut choice, meaning we decide where to split the current subpage (see Section 4.3.2). Lines 3-5 cover the base case for our

recursion. When no valid cuts to choose as the best cut remain, we stop recursing. In line 6, we use the chosen cut to create two new subpages from our current subpage (see Section 4.3.3). Lastly, in lines 7 and 8, we create the nested list representation of our XY-tree by using the chosen cut as a node and recursively computing its two subtrees.

4.3.1 Computing valid cuts

For a given subpage, our goal is to compute all valid x- and y-cuts on this subpage. Note that one subpage corresponds to one recursion step. We can intuitively understand this step as determining where we can draw a vertical or horizontal line through the given subpage without crossing any objects on the subpage. Each line that we can draw in this fashion corresponds to a valid cut on the subpage.
We will now discuss how to compute valid cuts algorithmically. First, we will split the computation of all valid cuts into the computation of x-cuts and y-cuts respectively. Let us consider an example of how to determine all the valid x-cuts on a given page. The given page will contain a list of objects that each has a bounding box. We will consider a page with four objects:

$$\{o_1 : (0,1),(2,3)\}, \{o_2 : (1,0),(3,1)\}, \{o_3 : (4,2),(6,3)\}, \{o_4 : (5,0),(6,2)\},$$

see Figure 9a for a visualization of these objects. For the computation of valid x-cuts, it is sufficient to consider the x-coordinates of the bounding boxes. Therefore we will simplify the bounding boxes to the interval they occupy on the x-axis:

$$\{o_1 : [0,2]\}, \{o_2 : [1,3]\}, \{o_3 : [4,6]\}, \{o_4 : [5,6]\}.$$

This yields a list of intervals representing sections of the x-axis that are blocked by objects on the subpage. We illustrated these additional blocked sections in Figure 9b.

We now compute to the union of these intervals which yields

$$[0, 2] \cup [1, 3] \cup [4, 6] \cup [5, 6] = [0, 3] \cup [4, 6].$$

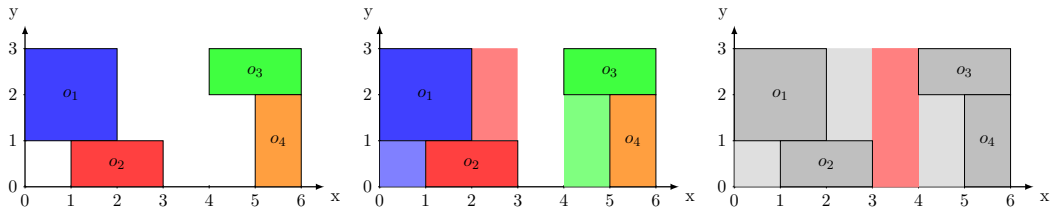
Note that none of these resulting intervals will overlap. We now computed all intervals we can not cut through vertically without crossing a bounding box of one or more objects. So our final step is to compute the gaps of these intervals. For this, we sort intervals by their starting point. Then, for each pair of consecutive intervals $[a_i, b_i]$ and $[a_{i+1}, b_{i+1}]$ we consider the gap $[b_i, a_{i+1}]$ between them. Our example yields exactly one such gap, as seen in Figure 9c. This interval represents the first possible valid x-cut on our example page.

In this fashion, we can compute all intervals of valid x-cuts on a given subpage. When computing valid y-cuts, we only need to switch out x- for y-coordinates. After computing both x- and y-cuts, we pair these intervals with the corresponding flag (i.e., **X** or **Y**) indicating an x- or a y-cut. This process results in a list of all valid cuts we need to consider for a given subpage.

Our initial goal is to detect text blocks. Thus, we are only interested in cuts that split between text blocks and not through them. We can preemptively filter valid cuts by using a threshold $m \geq 0$ and only considering cuts whose interval has a length of at least m. In practice, our algorithm computes the value of m in every recursion step. The computed value of m is the maximum of the average font size on the current subpage (multiplied by a constant) and a heuristic value that estimates line spacing. This mostly prevents us from splitting text blocks more than necessary (e.g., between lines of text).

4.3.2 Choosing the best cut

The previous step yields the set of all valid cuts on a given subpage. As a next step, we need to choose the best cut from this set. We then use the chosen cut in the



(a) Example objects **(b)** Blocked vertical space **(c)** Free space for a valid cut

Figure 9: Computation of valid cuts. (a) shows the bounding boxes of the objects on the page. (b) also shows the space these objects block vertically, the blocked space is colored in a slightly more transparent color as the object that blocks it. (c) shows the remaining free space (or the gap) in red.

next step of the algorithm. This step is crucial for detecting text blocks accurately. The choices made here directly influence the structure of the XY-tree. Thus, a poor decision in this step can potentially change the whole perceived layout of a document. Therefore, making good choices here is especially important. Our goal is to always choose the *best cut* from a layout perspective. That is, the best cut should have two basic properties: (1) it should be consistent with text block boundaries (e.g., it should split between and not through text blocks); (2) it respects the natural reading order (e.g., we can read everything on one side of the cut first and then continue on the other side; see Section 3.2.1). In some cases, the best cut can be ambiguous. For example, let us consider the cuts shown in Figure 10.

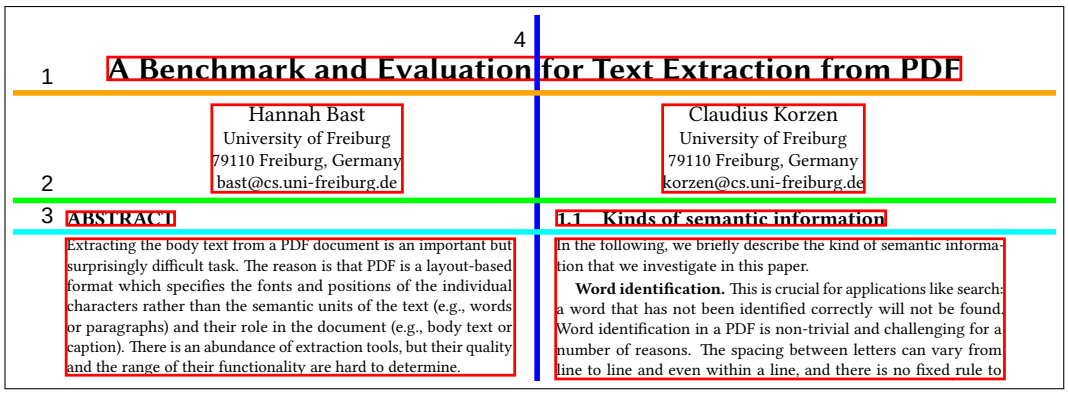


Figure 10: Valid cuts to choose from. The figure shows four examples of valid cuts on an a page. Bounding boxes of text blocks are for illustration only.

Here, cuts 1 and 2 possess both properties (1) and (2) and thus are both possible candidates for the best cut. The other cuts are also valid but do not have both of these properties. Cut 3 splits between text blocks but does not respect reading order, as we would have to read everything above it first and then everything below it. Cut 4 also disregards reading order. Additionally, it splits through the text block containing the title.

Now we have established a basic understanding of what the best cut is. Though we still need a strategy on how to choose the best cut from the set of valid cuts. There are many possible ways to try choosing the best cut. For text block detection, we used a simple rule-based strategy based on cut size, the weighted-largest cut strategy (see Section 4.5.2). We will also discuss different cut-choosing strategies specifically for reading order detection in Section 4.5.

4.3.3 Splitting the page

Unlike the previous subroutine, applying a chosen cut to a given subpage is pretty straightforward. We solve this by creating two new subpages that represent the two new subpages that emerge when applying the chosen cut to the current subpage. In the next step, we assign all page elements of the current subpage to one of the two new subpages.

We will now represent this step formally. Let $C = ([a, b], \mathbf{dir})$ be a cut and let O be the set of all objects (i.e., glyphs, figures, and shapes) on the page

$$O = \{o_i : (x_1^i, y_1^i), (x_2^i, y_2^i) \mid 1 \leq i \leq n\},$$

where $(x_1^i, y_1^i), (x_2^i, y_2^i)$ is the bounding box of o_i. We now need two consider two different cases depending on if C is an x- or a y-cut:

1. If C is an x-cut the first subpage contains the following set of objects

$$O_1 = \{o_i : (x_1^i, y_1^i), (x_2^i, y_2^i) \mid x_2^i \leq a\},$$

similarly the second subpage has the following objects

$$O_2 = \{o_i : (x_1^i, y_1^i), (x_2^i, y_2^i) \mid x_1^i \geq b\}.$$

2. If C is a y-cut the subpages will look as follows

$$O_1 = \{o_i : (x_1^i, y_1^i), (x_2^i, y_2^i) \mid y_2^i \leq a\},$$

and

$$O_2 = \{o_i : (x_1^i, y_1^i), (x_2^i, y_2^i) \mid y_1^i \geq b\}.$$

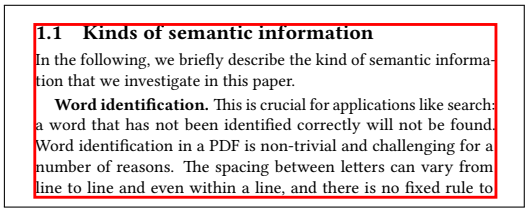
From the way we compute valid cuts on a page, we know that each element of O has to be either in O_1 or O_2 if C is valid.

4.4 Text block post-processing

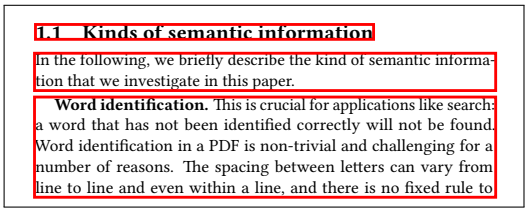
Detecting text blocks using only cut size and a size threshold (see Section 4.3.1) usually works reasonably well. However, there are situations where we would want to apply some post-processing to the detected text blocks. For instance, consider the text block shown in Figure 11a. In this example, we did neither separate the heading from the paragraphs nor the two paragraphs from each other. These cases cannot always be reliably detected using only line spacing. Yet, both cases are clearly identifiable when looking at the text block. The heading has a different font type than the paragraph and the second paragraph begins with an indentation. After we have detected our text blocks, we can check them for headings and paragraph indentations that have not been properly split. If needed, we split the detected text

blocks again. Figure 11b shows the post-processing result of the text block from the previous example.

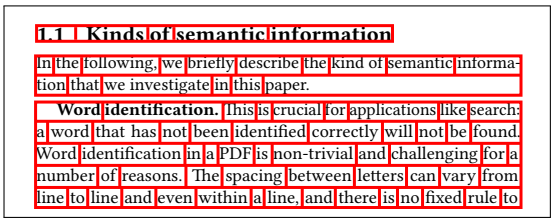
In a next step, we will split all detected text blocks down into words, as we will need the information about the contained words later (see Section 4.5). To split a text block into words, we can simply segment each text block again using a smaller size threshold (see Section 4.3.1). We choose the threshold $m = 0.5$ pt which is usually low enough to allow cuts between lines of text and between words within a line. Figure 11c shows the post-processed text blocks from our example split down into words.



(a) Detected text block



(b) Post-processed text blocks



(c) Post-processed text blocks split down into words

Figure 11: Text block post-processing. (a) shows a raw detected text block, (b) shows the post-processed version of the text blocks where the heading and the paragraph break are correctly split, (c) shows the post-processed text blocks split into words.

4.5 Improving reading order

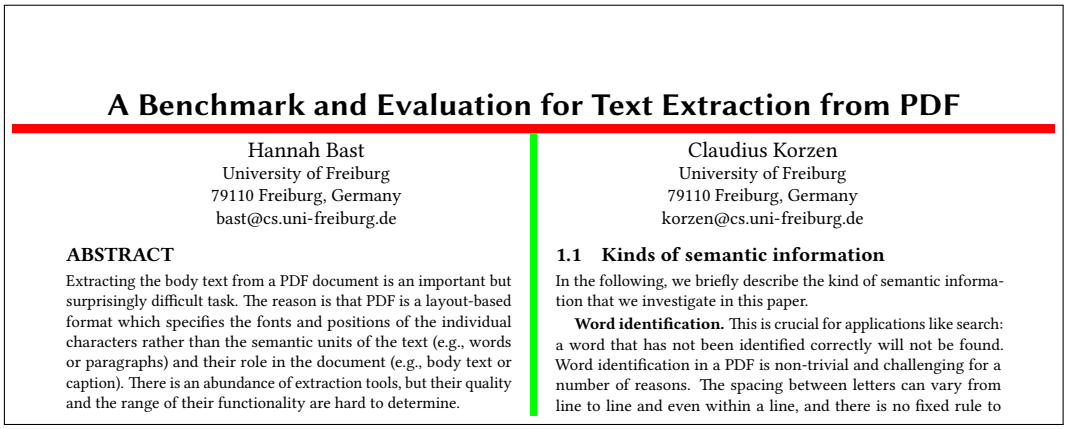
In this section, we will motivate and discuss the strategies we investigated for improving reading order detection.

4.5.1 Motivation

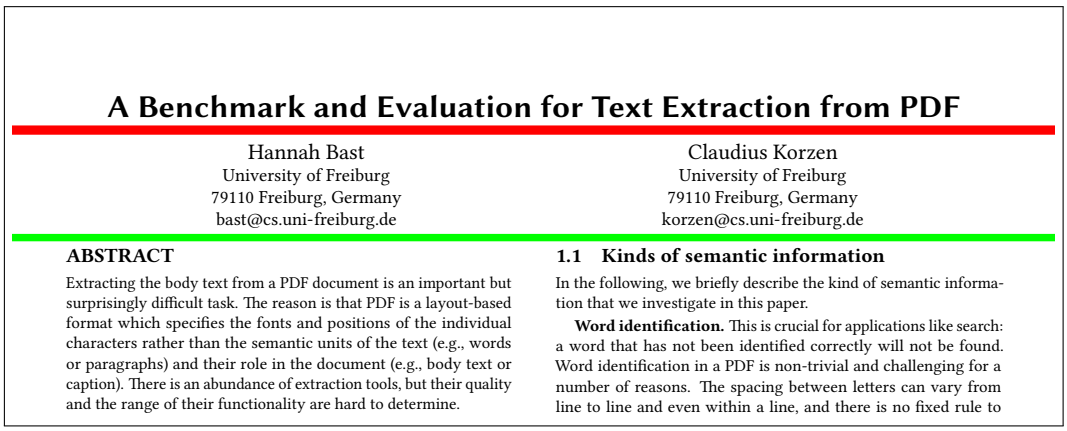
When we segment a document to detect text blocks, we infer a preliminary reading order from the chosen cuts (see Section 3.2.1). However, we detect text blocks using only a rule-based strategy based on cut size. Thus, the preliminary reading order is only based on cut size as well. This can lead to problems with the detected reading order. For instance, let us consider the example shown in Figure 12. When only trying to detect text blocks, it does not matter if we split between authors and columns or treat the authors as part of the columns. In both cases, we can still detect all text blocks correctly. However, when treating the authors as part of the columns, the preliminary reading order is incorrect. In such a situation, it is difficult to reliably choose the best cut with respect to text block detection *and* reading order when only considering cut size.

That is why we propose detecting reading order in a separate step from text block detection. While detecting reading order, we will then know the bounding boxes of the detected text blocks already and can focus solely on ordering them. In addition to the information about their bounding boxes, we will also use information about the semantic roles of the detected text blocks, as seen in Figure 12c. PDF does not provide these semantic roles. Therefore, we have to compute them ourselves. To solve this task, we use another tool developed by Korzen at the chair of Algorithms and Data Structures at the University of Freiburg. This tool is still under development and not yet published. It provides functionality to predict semantic roles of text blocks using a machine-learning model. The model uses features like bounding boxes, contained text, font names, font size, etc., to make its predictions. Korzen's tool achieves a classification accuracy of 92.7%. Further information and evaluation results can be found under `http://ad-research.cs.uni-freiburg.de:17002`.

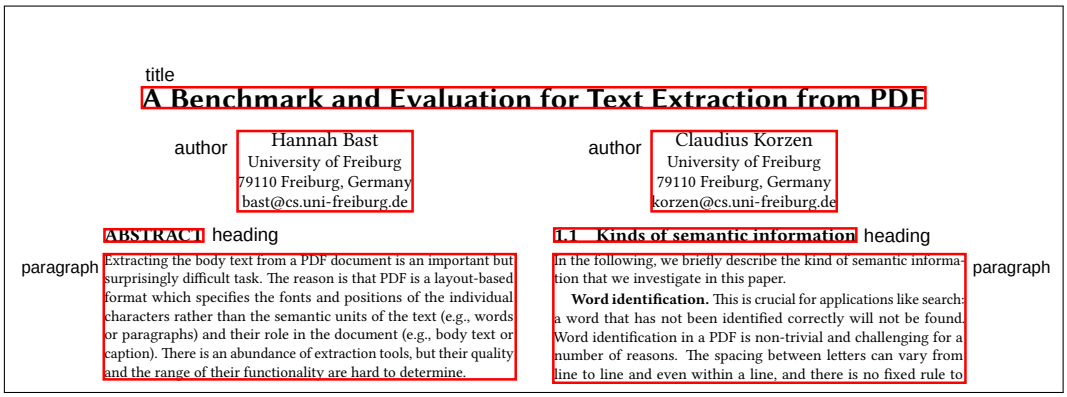
In the following, we will discuss the three types of strategies we used for reading order detection: (1) rule-based approaches, (2) score-based approaches, and (3) context-based approaches.



(a) Split between the two columns and treat authors as part of the columns



(b) Split between authors and columns to treat authors separately



(c) Page with information about text blocks and their semantic roles

Figure 12: Reading order detection. (a) and (b) exemplify a situation where geometric information does not suffice to reliably detect reading order, and (c) shows the information on text blocks and their semantic roles that we can use for better decisions.

4.5.2 Rule-based approaches

This category consists of the simplest (i.e., also the weakest) cut-choosing strategies. While being extremely basic, they are convenient to implement and use.

Largest cut

Layout-based documents often use distance for the visual distinction of their individual parts. For example, the distance between the title and the following body of text will be larger than the distance between two lines within a paragraph. We can make use of this by choosing cuts according to their size. This leads us to the *largest-cut* strategy. This strategy iterates over all valid cuts and chooses the one with the largest size. However, when considering the previous example from Figure 12, the largest-cut strategy would treat the authors as part of the columns. This is because the distance between columns is larger than the distance between the authors and the headings below.

Weighted-largest cut

While the previous section showed that distance is an important indicator when choosing cuts, it is not perfect. Specifically, we saw that the size of the vertical cut between the columns was too large compared to the size of the horizontal cut below the authors. This is quite common in PDF and other layout-based formats. Horizontal distances can often be larger than vertical distances. In such cases, even when comparing cuts of a similar semantic level (e.g., like the cut between columns and the cut below the authors), x-cuts will be larger than y-cuts.
Thus, we can generalize our largest-cut strategy to the *weighted-largest-cut* strategy. This approach works almost exactly like largest-cut, but it will prefer y-cuts by a given scaling parameter $r \geq 1$. It is easy to see that weighted-largest-cut generalizes

largest-cut. For $r = 1$ both strategies are the same. How to choose the value of r depends on the documents we want to segment. When manually experimenting with this value, we still used a much smaller dataset of about 20 PDFs, consisting mainly of scientific articles using a two-column layout. Using this dataset, we found a r-value of 2.5 worked best for our documents. Note, the optimal value of r can vary, and what worked well on our small dataset does not need to work well in general. Nevertheless, using the weighted-largest-cut strategy with $r = 2.5$ yielded the overall best results of all rule-based approaches we tested.
In our previous example from Figure 12, the weighted-largest cut strategy would split between the authors and the columns. This is because, while the distance between columns is larger than the distance between authors and the headings, it is not 2.5 times larger. Thus, the strategy prefers the smaller y-cut below the authors. However, there are still many layouts where choosing cuts only by size will not suffice.

4.5.3 Score-based approaches

The second category we will discuss are the score-based approaches. This strategy type is more complex than the rule-based approaches due to the way we will compute scores. However, they come with a significant advantage. While our rule-based approaches only focus on cut size, score-based approaches can be generalized to work with multiple properties. All score-based approaches employ the same working principle. We compute a score for each cut and choose the cut that maximizes this score. That is, a higher score signals a *better* cut to choose.

Parameter cut

Before we discuss this strategy, we need to define the term *parameter*. A parameter p is a function from the set of valid cuts to the interval $[0, 1]$. We use parameters to assess cut properties, where larger parameter values are better. Let us look at an

example of such a parameter. We consider the parameter p_{size} that assigns each cut its size relative to the size of the largest cut in the same direction. Let $C = ([a, b], \mathbf{dir}_C)$ be a cut and $C_{max} = ([c, d], \mathbf{dir}_{C_{max}})$ the largest valid cut with $\mathbf{dir}_C = \mathbf{dir}_{C_{max}}$. Then we compute p_{size} as follows:

$$p_{size}(C) = \frac{b-a}{d-c} = \frac{size(C)}{size(C_{max})}.$$

The larger the size of a cut the higher its p_{size} value is, the largest cut will have the maximum value of 1.

We now consider a fixed set of parameters $\{p_1, \ldots, p_n\}$ that we want to use for cut-choosing. For example, this set can include parameters for cut properties like size, position, direction, fonts and font sizes on both sides of the cut, etc. The parameter-cut strategy then uses a *score-aggregation* function to combine the parameter values into a single score. We say a function $f : [0, 1]^n \to [0, 1]$ is a score-aggregation function iff it satisfies the following two properties:

1) For all $x_1, \ldots, x_n, y_1, \ldots, y_n \in [0, 1]$ the implication

$$\bigwedge_{i=1}^{n} x_i \leq y_i \implies f(x_1, \ldots, x_n) \leq f(y_1, \ldots, y_n)$$

   holds (i.e., f is monotonically increasing in every variable).

2) For all $x_1, \ldots, x_n \in [0, 1]$

$$\bigwedge_{i=1}^{n} x_i = 1 \iff f(x_1, \ldots, x_n) = 1.$$

A comprehensible example of a score-aggregation function is the arithmetic mean $\overline{x}$ of the input vector

$$\begin{array}{rccc} \overline{x} & : & [0,1]^n & \to & [0,1] \\ & & (x_1, \ldots, x_n) & \mapsto & \frac{1}{n}\sum_{i=1}^{n} x_i. \end{array}$$

Our definition also allows for *weighted* score-aggregation functions. For example, let $w_1, \ldots, w_n$ be positive weights with $\sum_i w_i = 1$. Then the weighted arithmetic mean

$$\begin{array}{rccc} \overline{x}_w & : & [0,1]^n & \to & [0,1] \\ & & (x_1, \ldots, x_n) & \mapsto & \sum_{i=1}^{n} w_i x_i \end{array}$$

is a score-aggregation function. The arithmetic mean $\overline{x}$ from the previous example is a special case of the weighted arithmetic mean with $w_1 = \cdots = w_n = \frac{1}{n}$.

Given a score-aggregation function f, we can now assign scores to cuts. We compute these scores by evaluating the expression $f(p_1(C), \ldots, p_n(C))$ for each cut C. When choosing cuts using this strategy, we choose the cut C that maximizes $f(p_1(C), \ldots, p_n(C))$. The adjustable set of parameters and the choice of the score-aggregation function make this strategy extremely flexible. However, it turns out properly tuning these values is difficult. For example, simple weight adjustments can impact seemingly unrelated parameters and thus lead to a lower overall cut-choosing quality.

We used only two parameters with this strategy, as including more than two led to difficult to predict behaviour. The two parameters include the already mentioned p_{size} and a position-based parameter p_{pos}. p_{pos} assess how close a cut is to the natural reading order. That is, it prefers x-cuts close to the left margin and y-cuts close to the top of the page. Let W be the width, H be the height of the current page, and $C = ([a, b], \mathbf{dir})$ a cut. Then p_{pos} is computed as follows:

$$p_{pos}(C) = \begin{cases} 1 - \frac{a}{W}, & \text{for } \mathbf{x}\text{-cuts}, \\ \frac{a}{H}, & \text{for } \mathbf{y}\text{-cuts}. \end{cases}$$

For score-aggregation, we used the arithmetic mean of p_{size} and p_{pos}.

LogisticRegressor

While the previous strategy already has most of the advantages that score-based approaches offer, it has one major drawback. Incorporating new properties is time-consuming due to the manual definition of parameters and weights. However, deriving a parameter from a given property and weighting its importance is a suitable task for a neural network. This leads to our first learning-based strategy, a simple logistic regressor. Our LogisticRegressor model uses a fully linear architecture. In our default configuration, it uses three hidden layers with sizes 256, 256, and 64. We visualized the default configuration in Figure 13. The model takes tensors representing different cut properties as an input features (see Section 5.2 for details on the used properties). We refer to the size of the feature tensors by F. The model then outputs a single score for each cut in the input. If needed we can map the output scores to the interval $[0, 1]$ (e.g., using a Sigmoid function). When using this strategy, we only need to translate all desired cut properties into a feature tensor. The network then takes care of weighting and aggregating these individual values into one final score. For details on how we trained our models, refer to Section 5.2.

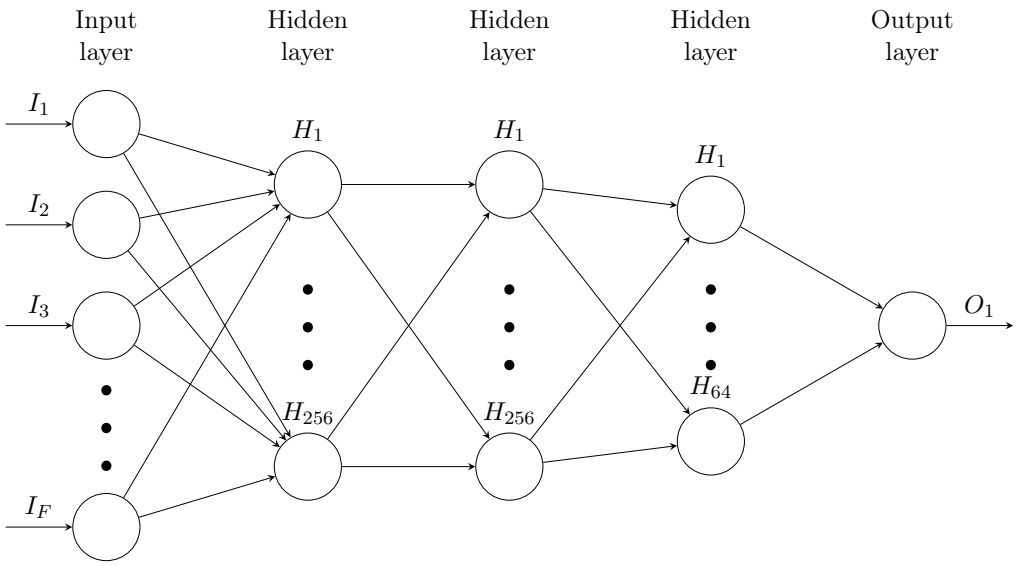


Figure 13: Fully linear logistic regressor

4.5.4 Context-based approaches

Our third and last category of cut-choosing strategies are the context-based approaches. This type of strategy is the most sophisticated but also the most complex of the ones we use in this work. They all share the same principle of comparing cuts against each other directly instead of rating them individually. The implementation of these strategies is also more complex than previous approaches. In particular, we implement each strategy by a deep neural network with a specific architecture. They all share the same input and output shape. The input shape is (S, F) where S is the maximum number of cut samples we look at, and F is the size of the feature representation of a cut (see Chapter 5.2 for details on the used features). When choosing cuts with a context model, we translate all valid cuts in a recursion step into a tensor of shape (S, F). Each row of this tensor corresponds to a valid cut. If there are less than S valid cuts, we pad the input with zeroes. The output is a tensor of length S. Each element of the output tensor corresponds to a probability that the corresponding cut in the input is the best cut. This is analogous to a multi-class classification problem where we have S classes that correspond to indices in the input sequence. We choose the best cut by choosing the cut at the index corresponding to the class with the highest probability. In the following paragraphs, we will present the different context-based strategies or, more precisely, the architecture of the underlying models we used. For details on how we trained our models, refer to Section 5.2.

BatchClassifier

The BatchClassifier model is the simplest of our context-based models. It employs a fully linear architecture. The first layer "flattens" the (S, F) input tensor into a tensor of length $S \cdot F$. This step is necessary as we cannot directly feed our two-dimensional input into a linear layer. Following that, the model uses multiple linear layers, each including a ReLU activation function. The last linear layer produces the desired

output tensor of length S. Our default BatchClassifier model uses three hidden layers with sizes 256, 256, and 64, respectively. Figure 14 shows the architecture.

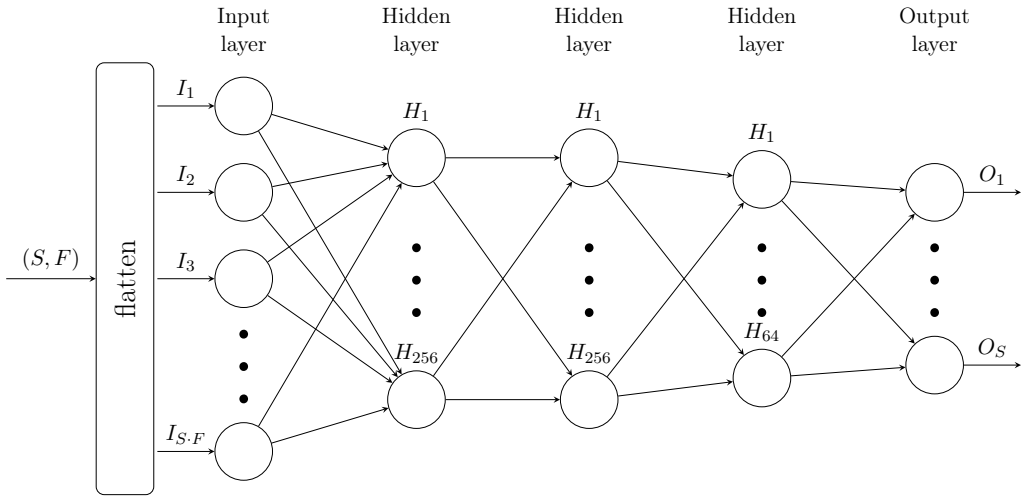


Figure 14: Fully linear context model

Transformer

The Transformer model is the most experimental context model we use. It uses a transformer encoder [15] for feature learning followed by a linear classifier for decoding. The transformer encoder does not change the shape of the input, but only alters its values. The linear decoder then transforms the (S, F) shape to an output with the desired S shape. Our default Transformer model uses a four-layered transformer encoder with five attention heads. Its linear decoder shares the exact architecture of the BatchClassifier. Figure 15 shows the architecture.

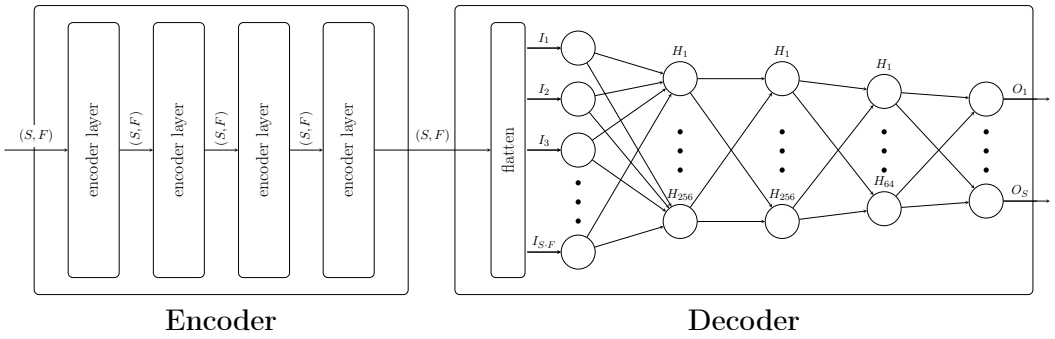


Figure 15: Transformer context model

5 Experiments

In this chapter, we will discuss three main topics: (1) the creation and structure of our datasets, (2) the training methodology we used to train our models, and (3) the evaluation results of our different strategies.

5.1 Datasets

Before we explain how we created our datasets, we will discuss their purposes and their desired structure. We use three different datasets: (1) a training dataset for training our models to choose the best cuts for detecting reading order, (2) a validation dataset for checkpointing our models during training, and (3) an evaluation dataset for assessing the capabilities of our algorithm. To create our datasets, we used about 35,500 randomly selected documents from *arXiv*. arXiv is an open-access repository for scientific publications. In addition to a PDF version, arXiv usually provides the respective source material, usually the underlying TEX code, of each article.
As a first step, we generate a ground truth for each of the selected documents. Each ground truth contains the expected text blocks for the respective document, sorted by natural reading order. To create these ground truths, we make use of the technique described by Bast and Korzen [2]. They devised a method for generating high-quality text extraction benchmarks from PDF files with available TEX data. Using their method, we can compute the expected text blocks, the semantic roles of the text blocks, and the reading order for all selected documents.

In the next step, we divide the set of ground truths into training, validation, and evaluation datasets. We used about 85% of the documents for training, 10% for validation, and 5% for evaluation. For the evaluation dataset, the structure of the ground truths already fits perfectly, as we later want to evaluate how well our algorithm detects and orders text blocks. However, for training and validation, we need datasets that can be used to learn choosing cuts. That is why we need a different structure for these datasets.

We selected a TSV file structure for these datasets. Our goal is to learn how to choose cuts for detecting the reading order of the detected text blocks. Therefore, for each document and each page, we will train on the recursion steps of the page segmentation algorithm where cuts between text blocks are possible. For each recursion step, our TSV file contains a line of meta-information with the following information: (1) the name of the current PDF, (2) the current page number, (3) the absolute width and height of the page, (4) the bounding box of the current subpage, (5) the current recursion depth, and (6) the direction of the chosen cut in the recursion level above (if present). After this meta-information line, we list all valid cuts in the current recursion step. For each cut, we provide (1) the respective interval, (2) the direction of the cut, (3) the semantic roles of blocks left or above the cut, (4) the semantic roles of blocks right or below the cut, and (6) a binary label (i.e., 1 or 0) that indicates if the current cut is the best cut. Note that for cuts that split through text blocks, we do not provide semantic roles, as such cuts are not interesting for detecting reading order. We illustrate the structure in Figure 16.

We can compute most of the information needed to create these datasets easily. For instance, computing valid cuts is already part of our page segmentation algorithm (see Section 4.3.1), and semantic roles are provided in the ground truth. The only missing part is the labeling of the cuts. For the labeling, we need to compute the best cut. To solve this problem, we can use the expected text blocks from the ground truth.

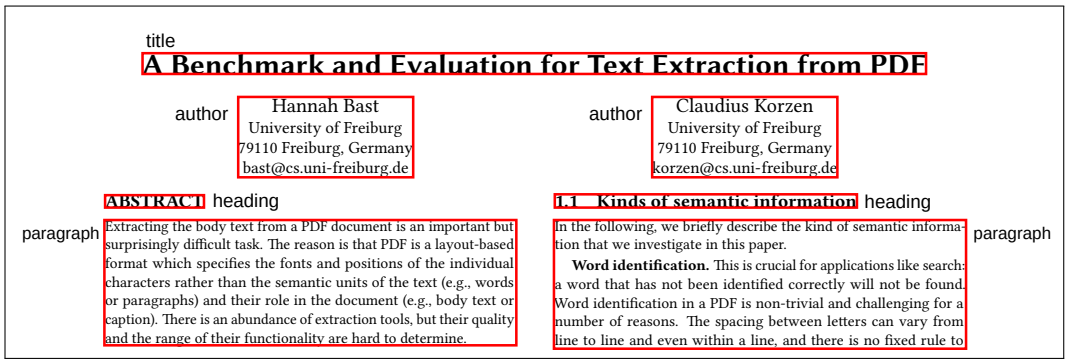
To better understand how we can achieve this, let us consider the example in Figure

# name	page num	width,height	subpage	depth	direction
example.pdf	42	612,796	0,0,360,640	2	**X**
# cut	left/upper semantic roles		right/lower semantic roles		label
([530, 550], **Y**)	heading		paragraph		1
([270, 290], **Y**)	paragraph		paragraph		0
([170, 175], **X**)	-		-		0
example.pdf	43	612,796	0,0,612,796	1	-
([720, 740], **Y**)	marginal		heading,paragraph		0
([680, 685], **Y**)	-		-		0
([420, 430], **Y**)	table,paragraph		formula, caption		0
([296, 316], **X**)	heading,table,caption		marginal,paragraph,formula		1

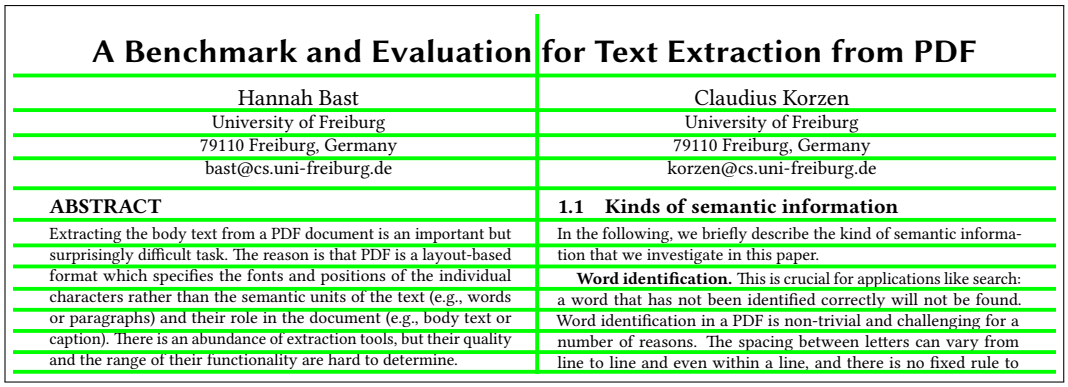
Figure 16: Training and validation dataset structure. Lines starting with # are intended for clarification only and are not part of the dataset.

17. The figure shows the ground truth and all valid cuts for a page. We know that the best cut will not split through the detected text blocks, as we are only interested in ordering them. Thus, we can already assign the label "0" to all cuts that split through text blocks. In Figure 17c, we marked all valid cuts from our previous example that cross a text block in gray. Furthermore, we also label all cuts that conflict with the reading order of the ground truth with "0" as well. We illustrated this in Figure 17c such that all cuts that violate reading order are marked in blue. Most often, only a single cut will now remain unlabeled. This cut is then our wanted best cut. If more than one cut remains, the best cut is ambiguous in this situation. This case occurs in our example as we still have two valid green cuts left. Both of these cuts split between text blocks and respect reading order. By convention, for x-cuts, we will choose the left-most, and for y-cuts, the upper-most cut, as the best cut. We now label the best cut with "1" and all other potentially remaining cuts with "0".

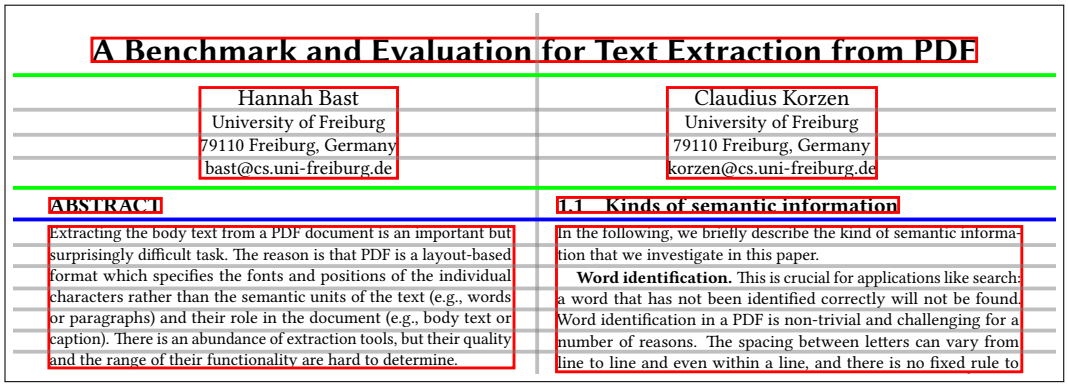
After generating the training and validation dataset from our set of ground truths, the resulting TSV files consist of roughly 4,275,000 recursion steps and contain about 77,000,000 individual cuts.



(a) Ground truth



(b) Valid cuts



(c) Labeling valid cuts using the ground truth

Figure 17: Labels from ground truth. (a) shows the ground truth of the page, (b) shows all valid cuts on the page, and (c) shows how we distinguish cuts using the ground truth: gray cuts cut through text blocks, blue cuts violate reading order, and green cuts are candidates for the best cut.

5.2 Model training

Now that we have discussed how we created our datasets, we will explain how and with what features we trained our models. In particular, we want to train our models for reading order detection. To detect reading order, we segment each page again. However, pages will now contain the previously detected text blocks, instead of characters, figures, and shapes. This allows us to use information about the semantic roles of text blocks (see Section 4.5) as an additional input for our models.

Let us now discuss the exact features we use to represent valid cuts. The geometric features we use include (1) relative position on the page, (2) direction, (3) size relative to the largest valid cut with the same direction, (4) the relative bounding box of the current subpage, and (5) the aspect ratio of the page. Additional non-geometric features are (6) the current recursion depth, (7) the direction of the cut in the recursion level above, and (8) a flag indicating if we currently are on the title page. Lastly, we use the (9) semantic roles of the text blocks adjacent to the cut. In particular, we distinguish on which side of the cut the respective semantic roles are located.

Now that we have discussed features, we will talk about how we trained our models. The score-based model takes a batch of cuts as input and predicts a score for each one. We optimized the score-based model on a binary labeling task using binary cross-entropy loss. That is, we trained the model to predict a score of 1 for cuts labeled "1" and 0 otherwise. The context-based models take all valid cuts of a single recursion step as input and predict a single "class" label. In this case, classes correspond to indices in the input sequence of cuts. For instance, if the model predicts class 2 for a given input, we would choose the cut with index 2 in the input. In training, we optimized the context-based models on a multi-class classification task using cross-entropy loss. We trained the models to predict the class corresponding to the index of the best cut for each recursion step in our dataset.

We trained all models on our training dataset for 100 epochs using early stopping with patience of 10 epochs. We used the Adam optimizer with a learning rate of

10^{-4} and a batch size of 64. We obtained our hyperparameters by performing a small manual grid search on learning rate and batch size. For checkpointing our models, we used the cut-choosing accuracy on the validation dataset. The cut-choosing accuracy is the quotient of the number of recursion steps where we chose the best cut correctly divided by the total number of considered recursion steps.

5.3 Results

In this section, we will discuss evaluation results. In particular, we will present our evaluation methodology together with the gathered results.

5.3.1 Methodology

To evaluate the capabilities of our strategies, we need to compare their results against a known ground truth. For this purpose, we use the ground truth generated from TEX data discussed in Section 5.1. In particular, the relevant information we will use is the list of expected text blocks and their respective reading order.
We will now formalize our evaluation process. First of all, we describe the output of our algorithm and the structure of the ground truth mathematically. Note that we will be evaluating on a per-page basis. Therefore, for each page of a document, we define two sets: (1) the set of expected text blocks R_G (i.e., the ground truth), and (2) the set of text blocks R_A detected by the algorithm (i.e., the set that we want to evaluate). To accord for the reading order, we will equip both sets with a strict total order $(R_G, <_G)$ and $(R_A, <_A)$ that corresponds to the respective reading order. Now we can describe our evaluation more precisely. First, we want to compare the elements of R_G and R_A. We compare text blocks by comparing their bounding boxes. In the best case, we would have $R_G \subset R_A$ (i.e., the algorithm detected all expected text blocks) but also $R_A \subset R_G$ (i.e., the algorithm did not detect any "wrong" text blocks).

For the best case follows that $R_A = R_G$. Secondly, we want to compare the reading orders against each other. While it is difficult to compare orders on different base sets, we can easily compare the induced orders $(R_G \cap R_A, <_G)$ and $(R_G \cap R_A, <_A)$. Ideally, the orders $<_G$ and $<_A$ should coincide on the subset $R_G \cap R_A$.

Now that we have formalized the objects we want to evaluate, we can define our evaluation metrics. Let $(R_G, <_G)$ be a ground truth and $(R_A, <_A)$ a result of our algorithm. We will use six different metrics: $B_G^=$, $B_A^=$, B_G^+, B_A^-, τ_n, and τ_n^f. The first four of these metrics assess block detection, that is they compare R_A against R_G. The fifth and sixth metric measures similarity of expected and detected reading order. $B_G^=$ is the number of expected blocks that were correctly detected relative to the total number of expected blocks. Formally,

$$B_G^= = \frac{|R_G \cap R_A|}{|R_G|}.$$

$B_A^=$ is the number of detected blocks that were actually expected relative to the total number of detected blocks. Formally,

$$B_A^= = \frac{|R_G \cap R_A|}{|R_A|}.$$

To properly define B_G^+ and B_A^-, we will introduce some additional notation. For two text blocks r_1, r_2, we will use the notation $r_1 \cap r_2 \neq \emptyset$ to express that the bounding boxes of r_1 and r_2 overlap. This notation makes sense because every bounding box can be canonically seen as a closed and connected subset of $\mathbb{R}^2$ which contains all points that lie within the bounding box. Two bounding boxes overlap if and only if the corresponding subsets of $\mathbb{R}^2$ intersect. B_G^+ is the number of expected blocks that have been split too much by the algorithm relative to the total number of expected blocks. We say an expected block is *split too much* if it overlaps with two or more detected blocks. Formally,

$$B_G^+ = \frac{\left|\left\{r \in R_G \;\middle|\; |\{r' \in R_A \mid r \cap r' \neq \emptyset\}| \geq 2\right\}\right|}{|R_G|}.$$

Similarly, B_A^- is the number of detected blocks that have not been split enough relative to the total number of detected blocks. We say a detected block is *not split enough* if it overlaps with two or more expected blocks. Formally,

$$B_A^- = \frac{\left|\left\{r \in R_A \;\middle|\; |\{r' \in R_G \mid r \cap r' \neq \emptyset\}| \geq 2\right\}\right|}{|R_A|}.$$

When detecting text blocks, we want to maximize both $B_G^=$ and $B_A^=$ while keeping B_G^+ and B_A^- as low as possible. Our fifth metric τ_n is the normalized Kendall-τ-correlation [1] of the detected order $(R_G \cap R_A, <_A)$ compared to the expected order $(R_G \cap R_A, <_G)$. To compute the non-normalized value of τ, we need to count the number of concordant pairs nc and the number of discordant pairs nd of the two orders. A concordant pair is a pair of distinct elements $r_1, r_2 \in R_G \cap R_A$ where $r_1 <_G r_2$ and $r_1 <_A r_2$. A discordant pair is a pair of distinct elements $r_1, r_2 \in R_G \cap R_A$ where $r_1 <_G r_2$ but $r_2 <_A r_1$. The last value we need to compute τ is the total number of ordered pairs np. For a totally ordered set of cardinality n, the total number of ordered pairs is $np = \frac{n \cdot (n-1)}{2}$. Because we defined our orders on sets, they cannot contain duplicates. Thus, all ordered pairs are either concordant or discordant and therefore $np = nc + nd$. Finally to obtain τ, we take the difference of nc and nd and divide it by np. This yields the τ-correlation coefficient between -1 and 1. To obtain τ_n, we normalize τ by adding 1 to it and dividing it by 2. τ_n now specifies if $<_G$ and $<_A$ are more positively correlated ($\tau_n > 0.5$), uncorrelated ($\tau_n = 0.5$), or more negatively correlated ($\tau_n < 0.5$). Formally,

$$\begin{aligned}
nc &= |\{(r_1, r_2) \in (R_G \cap R_A)^2 \mid r_1 <_G r_2, r_1 <_A r_2\}|, \\
nd &= |\{(r_1, r_2) \in (R_G \cap R_A)^2 \mid r_1 <_G r_2, r_2 <_A r_1\}|, \\
\tau &= \frac{nc - nd}{np} = \frac{nc - nd}{nc + nd}, \\
\tau_n &= \frac{\tau + 1}{2}.
\end{aligned}$$

Our last metric τ_n^f will be computed exactly like τ_n, except during computation we will disregard all blocks from R_G whose semantic role is one of: *table*, *caption*, or *marginal*. We will discuss our reasons for considering this metric and its interpretation in Section 5.3.2.

To better understand how to compute and interpret these metrics, we will consider some example evaluations. Figure 18a shows the ground truth R_G for our two examples. Its base set consists of seven text blocks. We will refer to each text block by using the letter next to it. Thus, $R_G = \{A, B, C, D, E, F, G\}$. The numbers next to the text blocks induce the required total order on R_G. Therefore, the total order for our ground truth is $A <_G B <_G C <_G D <_G E <_G F <_G G$. Now let us compare $(R_G, <_G)$ against our first example result from Figure 18b. The base set of this example consists of the same seven text blocks, $R_A^1 = \{A, B, C, D, E, F, G\}$. We can describe its total order by $A <_A^1 B <_A^1 D <_A^1 E <_A^1 C <_A^1 F <_A^1 G$. Now we can calculate all of our metrics. As we have already seen, we have $R_G = R_A^1$ in this case and thus

$$B_G^= = \frac{|R_G \cap R_A^1|}{|R_G|} = \frac{|R_G|}{|R_G|} = 1,$$

and

$$B_A^= = \frac{|R_G \cap R_A^1|}{|R_A^1|} = \frac{|R_A^1|}{|R_A^1|} = 1.$$

Furthermore, we obtain $B_G^+ = B_A^- = 0$ because each block only overlaps with exactly one other block, that is itself. The $B_G^=$ and $B_A^=$ values tell us that the algorithm not only managed to find all text blocks it was supposed to find but it also did not detect any wrong blocks. Additionally, the values of B_G^+ and B_A^- indicate that no blocks were split too much or too less. In total, this is a perfect result for text block detection. However, we do have some mistakes with reading order in this example. According to $<_A^1$ blocks D and E come before block C which contradicts with $<_G$. Thus, we have two discordant pairs

$$nd = |\{(r_1, r_2) \in (R_G \cap R_A^1)^2 \mid r_1 <_G r_2, r_2 <_A^1 r_1\}| = |\{(C, D), (C, E)\}| = 2,$$

and because $|R_G \cap R_A| = 7$, we obtain $np = \frac{|R_G \cap R_A^1| \cdot (|R_G \cap R_A^1| - 1)}{2} = \frac{7 \cdot 6}{2} = 21$. In total, we can conclude

$$nc = np - nd = 21 - 2 = 19.$$

With this we can compute the value of τ and thus the value of τ_n

$$\tau = \frac{nc - nd}{np} = \frac{19 - 2}{21} = \frac{17}{21} \approx 0.81,$$
$$\tau_n = \frac{\tau + 1}{2} \approx \frac{0.81 + 1}{2} \approx 0.91,$$

which tells us that while not being identical these orders are still strongly correlated. This result is in line with our expectation as only two pairs of blocks appeared in the wrong order.

As for our second example from Figure 18c, we have only six text blocks as our base set, $R_A^2 = \{A, B, C, H, I, J\}$. Their order is as follows $A <_A^2 B <_A^2 C <_A^2 H <_A^2 I <_A^2 J$. Lets again calculate our metrics scores. For $B_G^=$, we obtain

$$B_G^= = \frac{|R_G \cap R_A^2|}{|R_G|} = \frac{|\{A, B, C\}|}{|\{A, B, C, D, E, F, G\}|} = \frac{3}{7} \approx 0.43,$$

and for $B_A^=$ we get

$$B_A^= = \frac{|R_G \cap R_A^2|}{|R_A^2|} = \frac{|\{A, B, C\}|}{|\{A, B, C, H, I, J\}|} = \frac{3}{6} = 0.5.$$

These values tell us that the algorithm only managed to detect about 43% of the expected blocks correctly whereas 50% of the detected blocks were also expected. We can obtain more information about the type of mistakes when computing B_G^+ and

B_A^-. For B_G^+, we obtain

$$\begin{aligned} B_G^+ &= \frac{\left|\left\{r \in R_G \mid |\{r' \in R_A^2 \mid r \cap r' \neq \emptyset\}| \geq 2\right\}\right|}{|R_G|} \\ &= \frac{|\{G\}|}{|\{A, B, C, D, E, F, G\}|} \\ &= \frac{1}{7} \\ &\approx 0.14, \end{aligned}$$

and for B_A^- we get

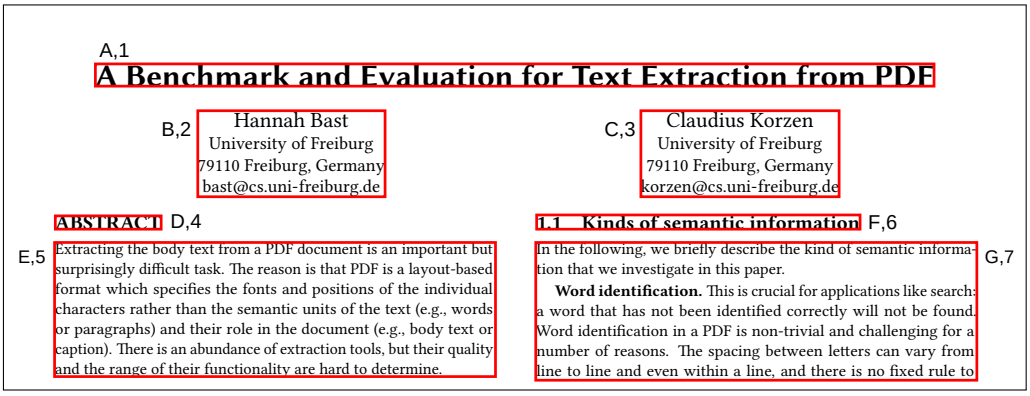
$$\begin{aligned} B_A^- &= \frac{\left|\left\{r \in R_A^2 \mid |\{r' \in R_G \mid r \cap r' \neq \emptyset\}| \geq 2\right\}\right|}{|R_A^2|} \\ &= \frac{|\{H, I\}|}{|\{A, B, C, H, I, J\}|} \\ &= \frac{2}{6} \\ &= 0.\overline{3}. \end{aligned}$$

This tells us that about 14% of expected blocks were split too much whereas $33.\overline{3}\%$ of detected blocks should have been split further. Before calculating τ_n, we need to take the intersection of R_A^2 and R_G. This step is necessary to compare $<_G$ and $<_A^2$ on the same base set. We obtain $R_G \cap R_A^2 = \{A, B, C\}$ and because $<_G{\restriction}\{A, B, C\} = <_A^2{\restriction}\{A, B, C\}$ we do not have any discordant pairs in this example. Therefore, we obtain

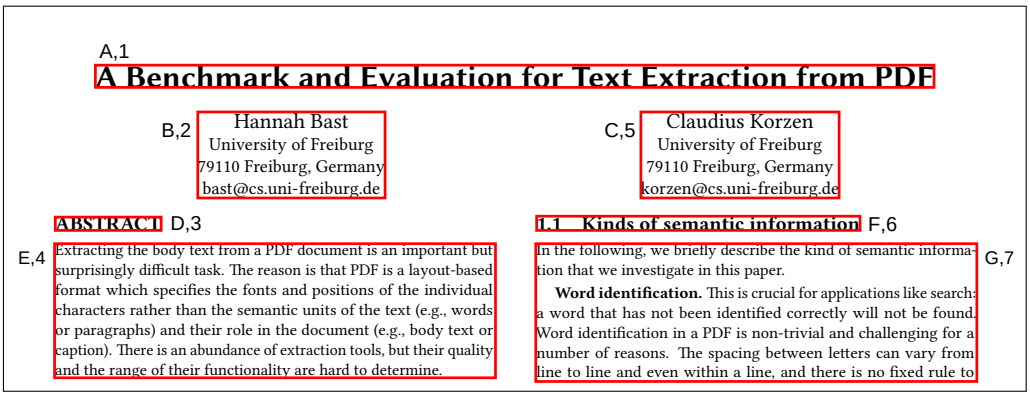
$$\tau = \frac{nc - nd}{np} = \frac{nc - nd}{nc + nd} = \frac{nc - 0}{nc + 0} = \frac{nc}{nc} = 1 = \frac{1 + 1}{2} = \tau_n,$$

which tells the orders are not only strongly correlated but identical. Thus, we know that all correctly detected text blocks appear in the correct reading order.

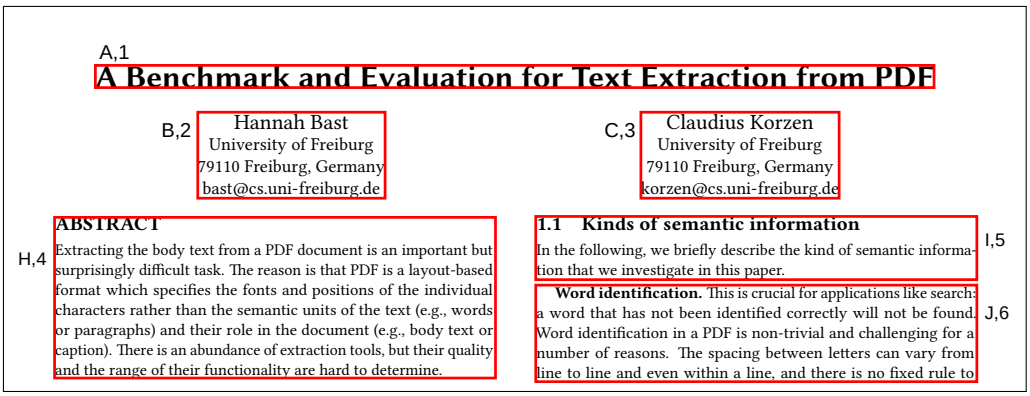
Note that we omitted the computation of τ_n^f in our examples, as it is equal to τ_n in both cases. We will see cases where τ_n^f and τ_n differ in the next section.



(a) Ground truth



(b) Segmentation result 1



(c) Segmentation result 2

Figure 18: Computation of evaluation metrics. (a) shows the ground truth for our example, (b) and (c) show different segmentation results. (b) detected all text blocks correctly but failed at the reading order while (c) failed at detecting text blocks correctly but not at reading order. We identify text blocks by the letter next to them, whereas the numbers next to them describe reading order.

As mentioned above, we compute these metrics on a per-page basis. To compute metrics for a document, we compute the metrics for each page first. After that, we can aggregate the computed values by using a reduction like arithmetic mean, median, etc.

5.3.2 Discussion

We evaluated text block detection and our reading order detection strategies on a set of roughly 1,750 documents from arXiv (see Section 5.1). We also compare the results of our algorithm to PdfAct [22], as PdfAct also provides text block and reading order detection. The values we report are obtained by taking the arithmetic mean once over the pages of each document and once overall documents.

Text block detection

Table 1 shows our results for text block detection. On average, our algorithm managed to detect 51.4% of expected blocks perfectly, whereas 46.7% of the detected blocks were also expected. 12.9% of expected blocks were split too much, and 14.7% of detected blocks were not split enough by our algorithm.
Compared to PdfAct, our algorithm performs worse with respect to all computed metrics. Especially for $B_G^=$, PdfAct outperforms our approach by over 15%. Similar differences can be seen in B_A^- where only around half as many detected blocks are split too less. For the other two metrics, the differences become smaller while remaining evident.

	$B_G^=$	$B_A^=$	B_G^+	B_A^-
Thesis	51.4%	46.7%	12.9%	14.7%
PdfAct	66.5%	54.3%	10.1%	7.5%

Table 1: Text block detection

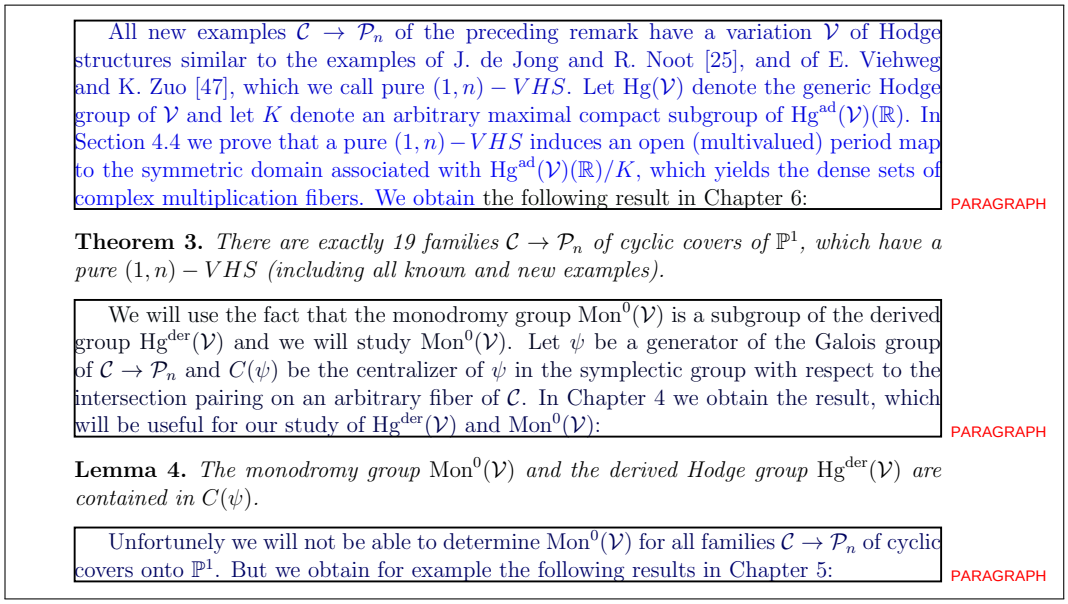


Figure 19: Missing text blocks in the ground truth. The figure shows an example from our ground truth that contains TEX environments the ground truth cannot recognize.

One point also protrudes from the results. For both PdfAct and our approach, $B_{\overline{G}}^{=}$ values are noticeably higher than the respective $B_{\overline{A}}^{=}$ values. This has to do with the way expected text blocks are computed in our ground truth. To recognize a block in the ground truth, we need to parse the TEX environment that generates it. For instance, this could be a *paragraph* or a *table* environment. However, many environments are either uncommonly used or even custom-defined. Such environments are not recognized in the ground truth, as seen in Figure 19. Thus, both PdfAct and our algorithm will sometimes detect more text blocks than expected, leading to lower $B_{\overline{A}}^{=}$ values.

Reading order detection

Table 2 shows our results for reading order detection. For both considered metrics, the score-based LogisticRegressor achieved the best results. All strategies achieved a strong correlation between expected and detected reading order with τ_n values

between 0.86 to 0.873. When filtering the expected text blocks, we achieve a nearly perfect correlation with τ_n^f values between 0.978 and 0.994. However, overall our evaluated strategies differed only marginally. That is, the difference for τ_n between our best strategy, the LogisticRegressor, and our worst strategy, the Transformer, is barely noticeable at 0.013. For τ_n^f, the difference increases slightly to 0.016.

When comparing our results to PdfAct, all strategies managed to achieve a small improvement in τ_n values over PdfAct. For τ_n^f, weighted-largest cut, parameter cut, and the Transformer performed slightly worse than PdfAct, whereas largest cut, the LogisticRegressor, and the BatchClassifier performed slightly better. The LogisticRegressor achieved the largest improvements with an increase of 0.014 in τ_n and an increase of 0.009 in τ_n^f over PdfAct. However, these results need to be seen in the context of the number of correctly detected text blocks, as only these are used to evaluate the detected reading order. As discussed before, PdfAct significantly outperforms our algorithm in text block detection. Therefore, we need to be careful when interpreting and comparing these values.

strategy	τ_n	τ_n^f
Largest cut	0.872	0.993
Weighted-largest cut	0.863	0.983
Parameter cut	0.865	0.984
LogisticRegressor	0.873	0.994
BatchClassifier	0.872	0.992
Transformer	0.860	0.978
PdfAct	0.859	0.985

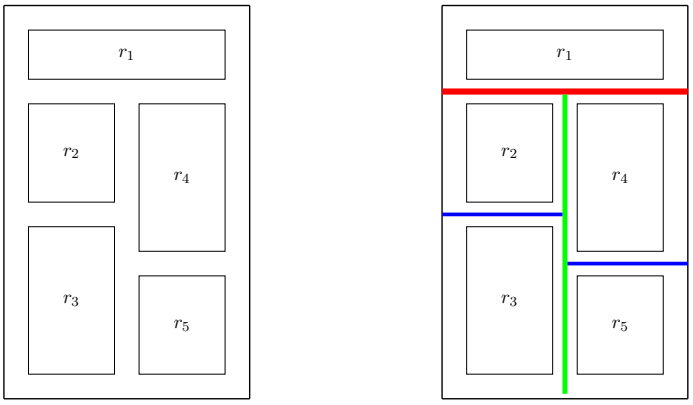
Table 2: Reading order detection

Two aspects stand out from the results. First, all strategies performed almost the same. In particular, our learning-based approaches did not manage to outperform our much simpler rule-based approaches. Second, for all strategies, the value of τ_n^f is much higher than the respective τ_n value. We will now discuss possible reasons for

these observations.

Why do all strategies perform nearly identical? There could be multiple reasons for this. The most obvious reason is that text blocks for which we need semantic information to correctly detect reading order (see the example described in Section 4.5) only make up a small part of the total number of text blocks in a document. For example, in the document from the example shown in Section 4.5, the two text blocks in question only correspond to roughly one percent of the total number of text blocks (≈ 200) in the document. Another reason could be that documents that contain such text blocks are under-represented in our evaluation dataset.
A much less apparent reason for the similar results is the nature of ordering blocks using the XY-cut method. Our strategies can only decide which cuts to choose and in which order. However, they can neither influence the set of valid cuts nor our definition of which side of the cut to read first (see Section 3.2.1). In some cases, this can massively restrict a strategy's impact on the detected reading order. For instance, consider the text blocks shown in Figure 20a. When we segment the page containing these text blocks again to detect reading order, we will always obtain the segmentation seen in Figure 20b. The reason for this is the arrangement of the text blocks. In each recursion step, there is only one valid cut. Therefore, it does not matter which strategy we use to sort these blocks, as all strategies have to choose the one available cut. In the end, every strategy produces the reading order $r_1 < r_2 < r_3 < r_4 < r_5$.

Why is τ_n^f higher than τ_n for all strategies? The reason for this is the additional filtering we apply before computing τ_n^f. Namely, we remove all blocks with semantic role table, caption, or marginal. To understand why these text blocks negatively impact detected reading order, we need to explain a few details on how our ground truth is generated. The reading order for each document in the ground truth comes from the respective document's TEX code. Let us now consider an example wherein the TEX code, a table is defined after a paragraph. TEX could choose to place the



(a) Detected text blocks

(b) Only possible segmentation

Figure 20: Limits of the XY-cut method. (a) shows detected text blocks on a page. (b) shows the only possible segmentation on this page.

table (and its caption) above the paragraph in the compiled PDF. This is due to TEX's way of positioning float environments. However, in our ground truth, the table and its caption will always come after the paragraph. Since this is the order, they appear in the TEX code. Thus, the reading order for tables and captions can sometimes be wrong. A similar case happens with marginal, which is also used for page numbering. In the ground truth, a text block containing a page number will always be one of the first blocks on a page with respect to reading order. However, in many documents, page numbers are at the bottom of the page. This leads to another case where the reading order detected by our algorithm will differ from the ground truth. To evaluate the impact of these mistakes on the detected reading order, we added τ_n^f as a metric. Our results show that this negative impact is very noticeable. Filtering out text blocks with potentially wrong reading order results in a nearly perfect correlation between expected and detected reading order.

6 Conclusion

This work aimed to extract text blocks from layout-based documents sorted by their natural reading order. Furthermore, we investigated if reading order detection can be improved by incorporating information on semantic roles. To solve this task, we devised an algorithm based on the recursive XY-cut algorithm to group the glyphs of a document into text blocks. The algorithm segments pages of layout-based documents by successively applying horizontal or vertical cuts. Afterward, we tested multiple strategies for sorting the detected text blocks by reading order. These strategies included simple rule-based approaches but also more complex learning-based approaches which incorporated semantic role information.

In summary, we managed to achieve decent results on text block detection and reading order detection. When filtering out potential mistakes from our ground truth, we even came close to a perfect result for reading order detection on correctly detected text blocks. However, we could not show any relevant improvements in reading order detection when using information about the semantic roles of the text blocks.

Our work leaves three main points for improvement in future work. The most important of these points is improving text block detection. Our evaluation showed that detecting text blocks based only on cut size does not yield satisfactory results. Thus, a more sophisticated approach that leverages font information and more reliably estimates line and column spacing could be used. Another aspect we did not consider in this work is resolving diacritic marks and hyphenation problems after splitting text blocks into words. However, these features are essential for proper text extraction.

Lastly, in our discussion, we explained the limits of the XY-cut method when it comes to reading order detection. We could bypass these limitations by using a learning-based reordering approach like described by Wang et al. [19]. Such an approach could also allow us to make better use of the information on the semantic roles of the detected text blocks.

We hope that our findings can help improve current page segmentation techniques and overall contribute to making text extraction from layout-based documents a "solved problem".

7 Acknowledgments

First and foremost, I would like to thank my adviser Claudius Korzen for many hours of exchange and discussion, his work on the datasets, and his extensive guidance and feedback. I also want to thank Hannah Bast for being my examiner and for the many helpful guidelines and tips on writing a thesis found on the chair's website. I thank Frank Dal-Ri and Matthias Hertel for helping me overcome the technical difficulties I encountered when training my models. Last but not least, I want to thank my friends and family who took the time to read my thesis write-up and suggested many helpful improvements.

Bibliography

[1] M. G. Kendall, "A new measure of rank correlation," *Biometrika*, vol. 30, no. 1-2, pp. 81–93, 1938. https://doi.org/10.1093/biomet/30.1-2.81.

[2] H. Bast and C. Korzen, "A benchmark and evaluation for text extraction from PDF," in *JCDL*, pp. 99–108, IEEE Computer Society, 2017. https://ieeexplore.ieee.org/document/7991564.

[3] G. Nagy, S. C. Seth, and M. Viswanathan, "A prototype document image analysis system for technical journals," *Computer*, vol. 25, no. 7, pp. 10–22, 1992. https://ieeexplore.ieee.org/document/144436.

[4] M. Shilman, P. Liang, and P. A. Viola, "Learning non-generative grammatical models for document analysis," in *ICCV*, pp. 962–969, IEEE Computer Society, 2005. https://ieeexplore.ieee.org/document/1544825.

[5] J. Ha, R. M. Haralick, and I. T. Phillips, "Recursive X-Y cut using bounding boxes of connected components," in *ICDAR*, pp. 952–955, IEEE Computer Society, 1995. https://ieeexplore.ieee.org/document/602059.

[6] J. Meunier, "Optimized XY-cut for determining a page reading order," in *ICDAR*, pp. 347–351, IEEE Computer Society, 2005. https://ieeexplore.ieee.org/document/1575567.

[7] L. O'Gorman, "The document spectrum for page layout analysis," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 15, no. 11, pp. 1162–1173, 1993. https://ieeexplore.ieee.org/document/244677.

[8] N. S. Altman, "An introduction to kernel and nearest-neighbor nonparametric regression," *The American Statistician*, vol. 46, no. 3, pp. 175–185, 1992. https://www.tandfonline.com/doi/abs/10.1080/00031305.1992.10475879.

[9] A. Simon, J. Pret, and A. P. Johnson, "A fast algorithm for bottom-up document layout analysis," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 19, no. 3, pp. 273–277, 1997. https://ieeexplore.ieee.org/document/584106.

[10] J. B. Kruskal, "On the shortest spanning subtree of a graph and the traveling salesman problem," *Proceedings of the American Mathematical Society*, vol. 7, no. 1, pp. 48–50, 1956. https://www.jstor.org/stable/2033241.

[11] R. B. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *CVPR*, pp. 580–587, IEEE Computer Society, 2014. https://ieeexplore.ieee.org/document/6909475.

[12] X. Yi, L. Gao, Y. Liao, X. Zhang, R. Liu, and Z. Jiang, "CNN based page object detection in document images," in *ICDAR*, pp. 230–235, IEEE, 2017. https://ieeexplore.ieee.org/document/8269977.

[13] Team Konfuzio, "Automatic text summarization in documents with faster R-CNN and PEGASUS." Blog post at https://konfuzio.com/en/automatic-text-summarization-in-pdf-files, accessed September 2021.

[14] S. Ren, K. He, R. B. Girshick, and J. Sun, "Faster R-CNN: Towards real-time object detection with region proposal networks," *CoRR*, vol. abs/1506.01497, 2015. https://arxiv.org/abs/1506.01497.

[15] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," *CoRR*, vol. abs/1706.03762, 2017. https://arxiv.org/abs/1706.03762.

[16] Y. Xu, M. Li, L. Cui, S. Huang, F. Wei, and M. Zhou, "LayoutLM: Pre-training of text and layout for document image understanding," *CoRR*, vol. abs/1912.13318, 2019. https://arxiv.org/abs/1912.13318.

[17] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," *CoRR*, vol. abs/1810.04805, 2018. https://arxiv.org/abs/1810.04805.

[18] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "RoBERTa: A robustly optimized BERT pretraining approach," *CoRR*, vol. abs/1907.11692, 2019. https://arxiv.org/abs/1907.11692.

[19] Z. Wang, Y. Xu, L. Cui, J. Shang, and F. Wei, "LayoutReader: Pre-training of text and layout for reading order detection," *CoRR*, vol. abs/2108.11591, 2021. https://arxiv.org/abs/2108.11591.

[20] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," *CoRR*, vol. abs/1409.3215, 2014. https://arxiv.org/abs/1409.3215.

[21] K. Papineni, S. Roukos, T. Ward, and W. Zhu, "Bleu: A method for automatic evaluation of machine translation," in *ACL*, pp. 311–318, ACL, 2002. https://aclanthology.org/P02-1040.

[22] C. Korzen, "PdfAct." Repository at https://github.com/ad-freiburg/pdfact, accessed September 2021.