Undergraduate Thesis

Accurate Word Extraction from Documents with Complex Layouts

Tanyu Tanev

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

Albert-Ludwigs-University Freiburg
Faculty of Engineering
Department of Computer Science
Chair for Algorithms and Data structures

October 15th, 2019

Writing period

15. 07. 2019 – 15. 10. 2019

Examiner

Prof. Dr. Hannah Bast

Adviser

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

PDF documents are commonly used to exchange and store information. Their compact and self-contained nature has kept them relevant for a long time. It also, however, leads to problems when trying to automate operations on them. An example of this is the automatic extraction of textual content - a problem, which has not been entirely solved to this day. Textual content within the documents is only stored at a character level, which means that words and text lines have to be rebuilt from the ground up. There exist rule-based algorithms, which try to solve the problem by taking advantage of the spacing within the documents. Using them on documents with a standard layout (e.g. scientific publications), which feature rectangular columns and figures has achieved satisfactory results.

It is, however, more challenging for documents, which have a complex layout. Some magazine and news articles, for example, feature circular pull quotes in-between text columns, which are used to emphasize a certain point. This, in turn, makes it harder to deploy the rule-based algorithms, because the spacing in the documents becomes irregular.

In the following thesis, this problem is tackled with the help of a deep learning approach. The approach looks at the pages of a PDF document as a collection of sequences, which are in the form of text lines. These sequences are then processed by a deep learning model and each character from them is grouped into its respective word. The usage of not only the distances between the characters, but also their value and information about their font, as input features, enables the model to more easily identify complex elements (e.g. pull quotes) and accurately extract their textual contents.

Zusammenfassung

PDF-Dokumente werden häufig zum Austausch und zur Speicherung von Informationen verwendet. Ihre kompakte und eigenständige Natur hat sie lange Zeit relevant gehalten. Es führt jedoch auch zu Problemen bei der Automatisierung von Operationen an ihnen. Ein Beispiel dafür ist die automatische Extraktion vom Textinhalt - ein Problem, das bis heute nicht vollständig gelöst ist. Der Textinhalt der Dokumente wird nur auf Zeichenebene gespeichert, was bedeutet, dass Wörter und Textzeilen von Grund auf neu aufgebaut werden müssen. Es gibt regelbasierte Algorithmen, die versuchen, das Problem zu lösen, indem sie den Abstand innerhalb der Dokumente ausnutzen. Die Verwendung auf Dokumenten mit einem einheitlichen Layout (z.B. wissenschaftliche Publikationen), die rechteckigen Spalten und Abbildungen enthalten, hat zu guten Ergebnissen geführt.

Schwieriger ist es jedoch bei Dokumenten, die ein komplexes Layout haben. Einige Zeitschriften- und Nachrichtenartikel enthalten beispielsweise kreisförmige Pull-Zitate zwischen den Textspalten, die verwendet werden, um einen bestimmten Punkt hervorzuheben. Dies wiederum erschwert den Einsatz der regelbasierten Algorithmen, da der Abstand in den Dokumenten unregelmäßig wird.

In der folgenden Arbeit wird dieses Problem mit Hilfe eines Deep Learning Ansatzes angegangen. Der Ansatz betrachtet die Seiten eines PDF-Dokuments als eine Sammlung von Sequenzen, die in Form von Textzeilen vorliegen. Diese Sequenzen werden dann von einem tiefen Lernmodell verarbeitet und jeder Charakter aus ihnen wird in seinem jeweiligen Wort gruppiert. Die Verwendung nicht nur der Abstände zwischen den Zeichen, sondern auch ihres Wertes und der Informationen über ihre Schriftart als Eingabefunktionen ermöglicht es dem Modell, komplexe Elemente (z.B. Pull-Anführungszeichen) leichter zu identifizieren und ihren Textinhalt präzise zu extrahieren.

Contents

List of Figures

List of Tables

List of Algorithms

Listings

1 Introduction

1.1 Motivation

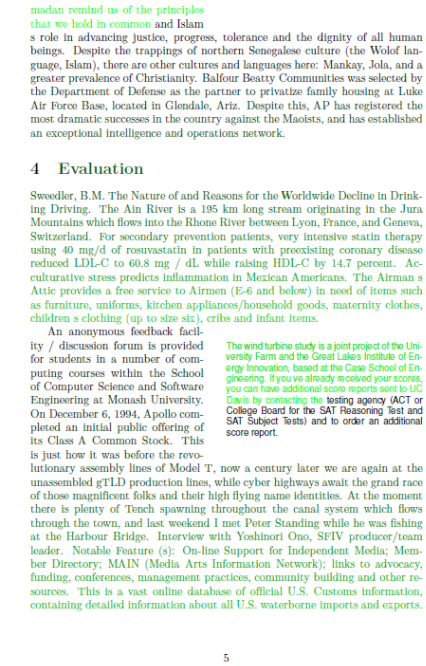
An essential part of the modern digital world is the exchange of information. It comes in multiple forms like books, news articles, user manuals, scientific publications and many others. All these different types of textual data conform to different standards. Figure 1 shows three examples of that. In Figure 1a, a randomly generated scientific publication is shown. It follows a so called **Manhattan layout**. This means that the text is grouped in simple, easy to define, rectangular columns. The font size and style are persistent and the attached tables and figures are usually positioned in such a way, as to not interfere with the textual content. In contrast to this, Figure 1b and Figure 1c employ unusual **non-Manhattan layouts**. The **pull quote** in the first figure and the **block quotation** in the second one lead to intertwined textual and figure data with varying font styles. Such complex elements are used by authors to emphasize important points of their works.

In order for all these different documents to be rendered correctly, a special file format is needed. Namely one, which can encapsulate all of this information, including content, references, font style and size, etc., in a self-contained package. This is where the Portable Document Format (PDF) comes into play. Developed by Adobe in 1990, this file format includes all the wanted attributes above, and more, into a compact package [1]. It allows documents to appear exactly how their authors intended them to, regardless of hardware and operating system. For this reason, PDF has been one of the most widely adopted and used formats in the last years and will seemingly continue to be in the years to come [2].

Being such a big part of the digital world, automatic extraction of the contents of PDF files is a prevalent work field. Being able to, for example, accurately extract the textual content of some PDF document, is a prerequisite for solving many practical problems, involving PDF documents. A common example case of this is conversion from PDF to other file types, like .doc (MS Word documents) or .html (web browser documents). Another one is the indexing of PDF documents by search engines.

(a) Randomly generated PDF file with Manhattan layout

(b) Randomly generated PDF file with non-Manhattan layout (**pull quote** in the middle below)



(c) Randomly generated PDF file with non-Manhattan layout (**block quotation** in the middle right)

Figure 1: Three different PDF document layouts

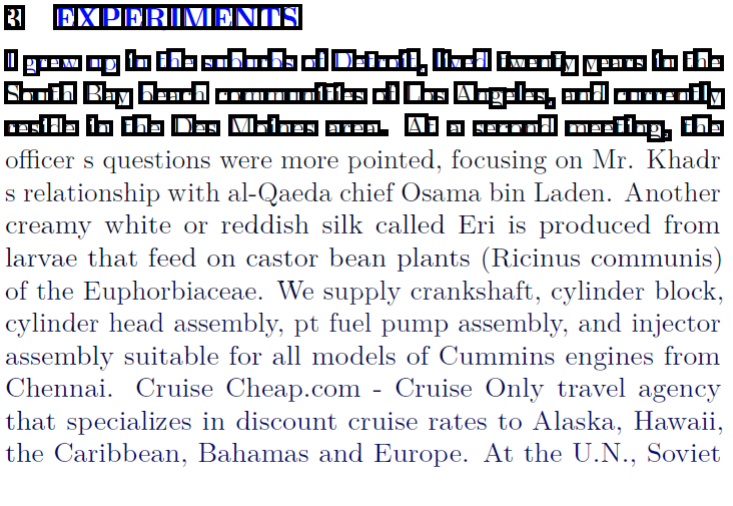


Figure 2: Bounding box of characters in a randomly generated PDF file

Indexing is the process of keeping track of some document's information. This is later compared with user searches and it's determined if the document is relevant to what the user wants. Without automatic content extraction, the tasks above would require a lot of manual labor and would be much slower.

The main struggle of researchers in the field has been with the file format itself. PDF is layout-based - it assumes that the document is finished and no further editing will be needed in the future. The main goal of the file format is to render its contents in their proper places with their proper style. As such, it provides little to no information about the semantic roles of the contents of the documents and the connections between them. Textual information, for example, is provided in streams of characters and are devoid of any semantic role. This can be seen when using a PDF parser library (like Apache PDFBox [3]) to extract information about the characters of a PDF document. Figure 2 showcases what developers have to work with - the values of the characters, their bounding boxes and their font styles and sizes.

Despite the difficulties, there exist algorithms for **rebuilding** the textual content of PDF documents. They are grouped in two categories:

- **Top-down** algorithms - such algorithms first start by segmenting a page into big blocks (e.g. columns) and make their way downwards to word-level

- **Bottom-up** algorithms - opposed to the algorithms above, bottom-to-top ones start off by grouping characters into words and text lines and finish with the

building of columns

Different types of rule-based algorithms have been able to group characters into their respective bigger blocks with satisfying results. The main factor, which is used across all of them, is spacing with the PDF document. Regular distances between different "meta levels" (e.g. **character-to-character**, **word-to-word**, **column-to-column**) allow the algorithms to achieve success. This is only true, however, for documents with a Manhattan layout. Non-Manhattan layout documents have irregular spacing, due to semantic blocks like pull quotes, block quotations and others (see Figure 1). This, coupled with irregular font style and size, makes it harder to robustly use a rule-based algorithm, which would work well across all non-Manhattan documents.

1.2 Difficulties

As mentioned in Section 1.1, spacing is one of the main factors, which enables rule-based algorithms to work. One of the more broadly used from them uses spacing, in order to segment the pages of a PDF document by "cutting" them into smaller blocks [4]. This is done by searching for suitable vertical and horizontal axes, which don't intercept with any characters or figures. Such axes are then used to split a part of a PDF page into two smaller blocks. These blocks can afterwards be further split into even smaller parts, e.g. text lines and words. The advantage of this method is that the hierarchical structure can be recorded and used to group characters into bigger blocks. This does not, however, include determining the semantic role of the blocks. This is another research topic, which is not looked at in this bachelor thesis.

It's harder, however, to find a suitable cutting axis in the presence of non-rectangular columns, like the ones in documents with non-Manhattan layouts. Such columns are commonly observed in the presence of pull quotes (see Figure 1b), found in news articles. Pull quotes are used to highlight certain information and are usually intertwined somewhere within the textual content. Because of their position, it becomes harder and, in some cases, impossible to cut the textual content into smaller blocks.

Building words out of the PDF characters also has its caveats. If using the same rule-based approach as above - factoring the white space between the characters - problems occur immediately. An example of this is Figure 3. It shows the bounding boxes of the characters of the words "serif font". Based on the spacing configuration, a rule-based algorithm could group them either as the following six words: "s", "e", "r", "iffo", "n", "t", or properly as two. The success of the algorithm is highly influenced

Figure 3: Bounding boxes of characters in words

by custom defined thresholds, which have to be properly adapted with each new document.

Lastly, PDF documents vary in font styles. If using predetermined thresholds for the spacing to extract words and semantic blocks, performance may also greatly vary. Because of the three problems above, the field of deep learning proves to be a valid candidate to eliminate some of these threshold and to extend the limits of content extraction.

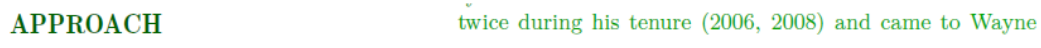
1.3 Proposed solution

This bachelor thesis proposes a deep learning solution. It capitalizes not only on information about the spacing of a document, but also on their font characteristics and values. Figure 4 visualizes a simplified workflow of how it's done exactly. The example is based on a small snippet from a randomly generated PDF document, as seen in Figure 4a. The snippet contains a section name and a part of a sentence, which are on the same horizontal level, but the space between them has been shortened for aesthetic reasons.

The first step of the proposed deep learning solution groups all characters with similar horizontal coordinates into a text line. Figure 4b shows the result of this process - an array with all characters of the text line, combined with their font information and the distances between them. This can then be given to a neural network, which predicts column and word endings. Figure 4c showcases how the correct predictions for the text line should look like. The prediction classes are the following three:

- 0 - normal character
- 1 - this character is at the end of some word
- 2 - this character is at the end of some column

By using the predictions explained above, the text line can be split into its respective columns and words. The end result is showed in Figure 4d.



(a) Small extract from randomly generated PDF document

Listing 1.1: Character, distance and font input data

```
[['A', 0.0, 10.9, True, False], ..., ['H', 184.5, 10.9, True, False],
 ['t', 0.0, 9.0, False, False], ..., ['e', 0.0, 9.0, False, False]]
```

(b)

Listing 1.2: Predicted labels

```
[['A', 0.0, 10.9, True, False], ..., ['H', 184.5, 10.9, True, False],
                  0                     0                    2
 ['t', 0.0, 9.0, False, False], ..., ['e', 0.0, 9.0, False, False]]
                  0                     0                    2
```

(c)

Listing 1.3: Finished text lines

```
["APPROACH", "twice during his tenure (2006, 2008) and came to Wayne"]
```

(d)

Figure 4: Simplified workflow of the proposed solution

The deep learning solution tries to solve the main issues, listed in Section 1.2. By splitting a PDF page on the meta level of text lines, the problem with irregular spacing is circumvented. Characters should also more reliably be grouped into their respective words, because the model learns a statistical model of the English language. The additional font features also make it easier to detect complex block formations, like the ones seen in Figures 1b and 1c.

2 Related Work

As mentioned in Chapter 1, there exist many PDF content extraction tools, which employ rule-based approaches. A comprehensive overview of the majority of them and their performance was done by Hannah Bast and Claudius Korzen in their paper "A Benchmark and Evaluation for Text Extraction from PDF" [5]. The results in them serve as proof of the significance of the research field and that the problem is still open.

In this chapter, I will mainly look into two tools: pdftotext [6] and Grobid [7]. The former is one of the most popular tools to extract contents from PDF files and it comes as a built-in command with many Linux distributions. The latter is one of the most recent advances of content extraction, based on machine learning.

2.1 Pdftotext

Pdftotext is a commonly used tool for textual extraction. It is able to quickly and robustly extract text out of a PDF document without running into major crashes [5]. However, its output also completely disregards any kind of semantic information.

Per default, the tool dumps all words in a top-down, left-right order. Luckily, the addition of command-line arguments can make the data be more structured. The "–layout" keyword configures the output to include the same spacing as in the original document. This is done by using tabular ('\t') and newline characters ('\n'). Their addition allows the text to be broken down into smaller semantic blocks, like text lines. This is important, as the proposed solution of this bachelor thesis is evaluated on a text line level. Two analogous arguments exist, namely "-simple" and "-table", which are used for one-page documents and table data respectively.

The robust nature and overall availability of this tool makes it a popular first choice for text extraction. For this reason, it is also used in the borders of this bachelor thesis as a basis of comparison for the proposed deep learning approach.

2.2 Grobid

Grobid is a content extraction tool, based on Conditional Random Fields (CRF). It was originally showcased in 2009 with bibliographic data recognition and term extraction as the main focus. As of current stand, in October 2019, the project is still being worked on. Its tasks have increased to being able to extract header metadata and the text of the file body.

The usage of CRF models has lead to state-of-the-art performances for the first two fields [8] and satisfactory results in the last one [5]. The main idea of using sequence labeling for content extraction carries over to this thesis as well, albeit used in a different direction. Learning a statistical model of the language has proven to be helpful in field.

Unfortunately, when using Grobid for full text extraction, the output is in the form as free-flowing paragraphs. The paragraphs are presented as one continuous block, with no information about the individual text lines within them. This makes it to evaluate against the deep learning approach, proposed in this undergraduate thesis.

3 Approach

In this chapter, I will more closely look at the six main components of the bachelor thesis:

- Data generation - what data was used to implement and evaluate the algorithms in the work
- Data parsing - how the data, mentioned above, was parsed, in order to be passed to the algorithms below
- Line grouping - essential process, which groups characters of a PDF documents to text lines
- Baseline algorithm - rudimentary rule-based algorithm, which will serve as a baseline during evaluation
- Deep learning solution - the main subject of this thesis
- Evaluation - how the evaluation was implemented and what it means

Each section will introduce the idea behind the respective step and show the most important algorithms, classes and external libraries, which were used to implement them. Their significance and usage will be explained, as well as showcased with examples.

3.1 Data generation

Randomly generated PDF files were used in this thesis for the development of the main algorithms and their evaluation. They feature different formats, combining both Manhattan and non-Manhattan layouts. Each one of them is also accompanied by a JSON "description" file, which contains metadata about the contents of the document. All data was generated by my supervisor, Claudius Korzen, and was used only in the borders of this thesis.

3.1.1 Randomly generated PDF files

The PDF files can be grouped in two main types, regarding their layout: Manhattan and non-Manhattanf. Both types include the following semantic blocks:

- Title
- Single or multiple author(s), with e-mail and address information
- Paragraphs of text with randomly chosen sentences from the **ClueWeb12** text corpus [9]
- **Non-embedded** figures - meaning that any characters in them aren't extractable

Additionally, the complex, non-Manhattan layout PDF documents also feature **pull quotes** (see Figure 1b) and **block quotations** (see Figure 1c). The text within them is usually of a different font style and size. The variety of possible contents helps with making the solutions of this bachelor thesis work well across as many document formats as possible.

For additional evaluation purposes, there are also documents with a Manhattan layout but also **broken spacing**. In those documents, there is a 5% chance, that a spacing between two words is removed. The documents were made, in order to get a better idea of the role of distance when extracting words, for both the baseline and deep learning algorithms.

3.1.2 JSON format

The ISO-standardized JSON format [10] provides a human-readable way of transmitting and storing state information. Originally derived from the Javascript programming language, JSON has become independent and is universally used in many places. It supports so called *objects* (equivalent to associative arrays in Javascript), arrays, strings, booleans and integers, along others. Listing 3.1 showcases the format and how it is used in this thesis to hold metadata about the contents of a PDF document.

3.1.3 JSON description files

Each randomly generated PDF document has a JSON description file supplied to it. The JSON description file contains metadata about the PDF document itself and is what would be used by the baseline and deep learning solutions of this bachelor thesis.

Each description file contains a single object with up to three key-value pairs:

- "textLines" - has an array of objects as the value, with each object representing a text line in the PDF document
- "figures" - like above, but representing a figure
- "shapes" - like the first point, but representing a shape (e.g. lines)

A shortened version of the JSON description file of the PDF document, shown in Figure 1c, is given in Listing 3.1. The first two keys of the ones, listed above, can be seen in the listing. As the PDF document doesn't contain any shapes, however, the "shapes" key is missing. This functionality is shared with the other key-value pairs as well - if something from the above three isn't present in the document, it isn't in the JSON description file as well.

The snippet shows how the metadata of the single-word "Evaluation" text line in Figure 1c looks like. Each text line contains a key-value pair about its position and the words, from which it's built. The position data includes the PDF page, on which the text line appears, along with the coordinates of it' s lower-left and upper-right points. Those two points can be used to define a bounding box around the text line, which is used extensively throughout the bachelor thesis. Position data is used in the same way for metadata about words, characters, figures and shapes.

The value of the "words" key is an array of associative arrays, which offer metadata about all words in the text line. Each word contains the following three types of information:

- "text" - the text of the word itself
- "positions" - an array of associative arrays with position data, as described above; there can be more than one position, because some words are wrapped to other lines or pages
- "characters" - an array of associative arrays, which contain metadata about each character of the word

The characters themselves also contain three main types of metadata. The first one is the value of the character itself. The second one is a collection of information about the font of the character, including *size*, *base name of the font*, *family name of the font*, *normalized name of the font*, whether it's *bold* and *italics* and if it's a *type 3 font* or not. The last collection of metadata about the character is its position, which follows the same format as explained above.

Listing 3.1: JSON "description" file, which holds PDF metadata

```
{"textLines": [..., {
  "position": {
    "pageNum": 5,
    "minX": 158.0,
    "minY": 516.6,
    "maxX": 233.1,
    "maxY": 526.6
  },
  "words": [{
    "text": "Evaluation",
    "positions": [{
      "pageNum" : 5,
      "minX": 158.0,
      "minY": 516.6,
      "maxX": 233.1,
      "maxY": 526.6
    }, ...],
    "characters": [{
      "text": "E",
      "fontsize": 14.3,
      "fontBaseName": "lmroman",
      "fontFamilyName": null,
      "fontNormalizedName": "lmroman12-bold",
      "bold": true,
      "italic": false,
      "type3": false,
      "pageNum": 5,
      "minX": 158.0,
      "minY": 516.7,
      "maxX": 168.6,
      "maxY": 526.5
    }]
  }]}, ...],

  "figures": [{
    "position": {
      "pageNum": 4,
      "minX": 133.8,
      "minY": 371.7,
      "maxX": 477.5,
      "maxY": 580.2
    }
  }]
}
```

At the bottom of Figure 3.1 is the "figures" key and its value. Although not visible in Figure 1c, the PDF document has a figure on page four and its bounding box corresponds with the given metadata. It's important to notice that the only information, available for figures, is their position. The same applies for shapes as well.

3.2 Data parsing from description files

The baseline algorithm and the deep learning solution work on the data, introduced in Section 3.1. As mentioned there, metadata about the characters, figures and tables in the documents is stored in JSON description files. Thus, parsing these files is the necessary first step.

Four different kind of methods were implemented for metadata parsing:

- Parse character metadata - value, font style and position information
- Parse figures metadata - position information
- Parse shapes metadata - position information
- Parse text line metadata - text and position information of the text line, along with metadata of the words and characters, which build it

The following subsections will go into more detail on how they function. Please keep in mind that the processes for figures and shapes are analogous and will be grouped under "non-character metadata".

Another important thing to note is that the most common character width, height and font size are tracked when **parsing character metadata**. Why this is done ii explained in the following subsection. This is, however, **not the case with the fourth method** - when parsing whole text lines. The difference is due to their use cases.

3.2.1 Character metadata

Character metadata is parsed by a recursive algorithm. It takes advantage of the structured format of the JSON description files, shown in Listing 3.1. The result is stored in a dictionary, which has page numbers as keys and lists of metadata about the characters in the pages as values.

The algorithm goes over all text lines and searches for the "words" key. If it's present, another iteration loop is started over all words of the text line. In each

word, another key-value pair is sought after - the "characters" one. If found, a final loop goes over all characters of a respective word and appends their metadata to the result dictionary. The metadata about the characters is stored in dictionaries, like showcased in Listing 3.1.

Parallel to the algorithm above, the width, height and font size of each character are also tracked. The font size is contained in the metadata about each character itself, whilst the width and height are calculated by using the lower-left and upper-right corners of a character. Three separate dictionaries are then used to store all unique values and their occurrences. At the end of the character metadata extraction process, a separate algorithm goes through the dictionaries and determines the most common character width, height and font size. This information is needed, as their values will be used later on to calculate spacing thresholds, used by both the baseline and line grouping algorithms.

3.2.2 Non-character metadata

Extraction of figures and shapes is an analogous process to the one from the previous subsection. A recursive algorithm goes through the values of the "figures" and "shapes" keys of the description files (see Listing 3.1). Each dictionary, which represents a figure or a shape, is then appended to a similar result dictionary as in the former subsection. It contains page numbers as keys and lists of dictionaries, which represent the respective non-character objects, as values.

Non-character metadata is used in the baseline algorithm later on. It helps “cut” the pages of PDF documents properly, so that no figures or shapes are also intercepted. It doesn’t, however, play a role in the deep learning solution. As introduced in Section 1.3, the deep learning solution works on the level of text lines and the characters within them. As such, non-character metadata is completely disregarded.

3.2.3 Text line metadata

Full text line metadata extraction is used for evaluation and creating training and validation data for the deep learning solution. In contrast to Section 3.2.1, an additional type of metadata is fetched - namely, the position of the text line. A loop goes over the values of the "textLines" key in the description file (see Listing 3.1) and gets every dictionary, which represents a text line. It appends them whole to a result dictionary, which, again, stores them page-wise.

Figure 5: A text line, spanning multiple columns, with low-hanging symbols

3.3 Line grouping

Line grouping is a core part of this bachelor thesis. Text lines are the not only the meta level, on which the deep learning approach works, but also on which its results and the ones of the baseline approach are evaluated. Accurately grouped lines also directly influence the performance of both algorithms, so it's important to get it as right as possible.

The algorithm for line grouping is rule-based. Before going deeper into it, I'm going to introduce two definitions of low-hanging and High-hanging symbols. These are extensively used in the rules of line grouping and are important to know:

- Low-hanging symbols - characters like 'q', 'y', 'p', ..., together with the two punctuation marks ';' and ','. The main characteristic of such symbols is that their lowest point hangs lower than the other characters of the text line, to which they belong. This is a problem, as it could wrongly be interpreted by the line grouping algorithm to be a start of a new line. Often, this happens when a text line spans multiple columns and the parts from the different columns aren't vertically aligned. Figure 5 shows an example of exactly that. It shows a snippet from the randomly generated PDF documents, described in Section 3.1. The vertical anchor of the text line is on the left side. The anchor allows only for characters with a similar lower-left coordinates to it to be appended to the same text line. The base vertical level of the normal characters on the right is high enough, such that they are appended to the text line. The low points of the "g" and "y" characters, however, lie beyond the allowed threshold and are considered as members of the next text line.

- High-hanging symbols - characters like 'l', 'j', 'k', ..., together with several punctuation marks like '?', '(', '‘ and others. In contrast to the low-hanging symbols, these are characterized by a higher upper-right corner. Proper catching of those symbols is important because they can leave previous lines unfinished. Because the algorithm goes through each character in a **top-down** order, high-hanging symbols of some text line often times start being processed before the last low-hanging ones of the previous one. This triggers the beginning of a new

line and messes up the correct grouping of both text lines - the missing symbols from the upper line are added as unnecessary extra ones to the lower one.

After introducing the terminology, I'm going to continue ahead with the algorithm itself. The main principle behind it is the following: group characters with a similar lower-left point (minY) in the same text line. Three lines are kept track of at any given time - the last built line, the currently built one, and, potentially, the next one. Each text line has an **anchor** character, whose lower-left point is accepted as the base vertical level for the line.

The characters are sorted from top to bottom (by minY values) and processed according to their upper-right points. In each iteration of the main loop, a single character is processed. The vertical difference between the lower-left point of the character and the anchor point of the currently built text line is calculated. Then, a custom defined threshold is used to determine if the difference is small enough for the character to be considered a part of the current line. For this bachelor thesis, the best performing threshold was found to be 40% of the **most common character font size**.

If the vertical difference lies below the gap threshold, the character is appended to the current line. If the gap is too big, however, this might be an indication that a new line has to be begun. There are three cases, in which a new line **shouldn't** necessarily be started, even though the gap is big enough:

- The character is some kind of bracket or quotation mark (e.g. "(", "!"). Symbols like these have both elongated upper and lower parts can lead to both problems, brought by low- or high-hanging symbols.
- The character is a high-hanging symbol - explained above.
- The character is a low-hanging symbol - also explained above.

Extra checks are made, in order to determine if the characters from the three cases above belong to the currently built line. These include checking the difference between the highest point of the characters and the anchor point. If, however, none of the checks are fulfilled, a new line is started.

Figure 6 showcases how the characters of a snippet from a randomly generated PDF document (see Section 3.1) are grouped in text lines. Figures 6a-6e follow the building of the first text line. In Figure 6f, however, the vertical difference between the lower-left point of the "F" character and the anchor point of the currently built

Figure 6: Steps of line grouping algorithm

line is too big and a new line is started. The rest of the figures (6g-6i) showcase how the characters of the left text lines are grouper.

It's important to notice how the text lines, which are built from this algorithm, can span **one or multiple** columns. The rule-based baseline algorithm doesn't have this problem, as the text line are grouped out of page segments, instead of the whole page. The deep learning approach takes on the task of splitting the multi-column text lines itself.

3.4 Rule-based baseline algorithm

A rudimentary rule-based algorithm was created as the first step of the bachelor thesis. This was done, in order to gauge the difficulties, into which algorithms like this run into, when processing documents with **non-Manhattan** layouts. This section dives deeper into its implementation and the important algorithms behind it. I will also discuss the main problems, which the algorithm has. The results of the algorithm are later on used during the evaluation as a baseline.

The baseline algorithm first extracts all metadata about characters, figures and shapes from the JSON description file of some PDF document (see Section 3.2). It then uses a recursive X-Y cut algorithm to first cut the pages of PDF documents into segments. The segments are then taken and further split into text lines, by using the line grouping algorithm, explained in Section 3.3. As a final step, the words of the individual text lines are determined. I will go deeper into how the recursive X-Y cut algorithm works and how the words are built from the text lines. I omit explaining how the segmentation into text lines is done, as it was already explained in Section 3.3.

3.4.1 Recursive X-Y cut

The baseline algorithm uses a variant of the recursive X-Y cut algorithm [4] for page segmentation. Page segmentation is the process of cutting a PDF page down to smaller blocks, e.g. columns. The characters of the PDF document can then be grouped in those larger blocks, effectively supplying each character with information about where it is in the overall layout of the document. It also provides an opportunity to try to cut the blocks into even smaller ones, like text lines and words.

The recursive X-Y cut algorithm is alternating in nature. A given PDF page is first cut horizontally. Each half is then taken and the algorithm tries to make a vertical cut in it. If this is successful, the upper and lower parts are each taken as

new starting points and the process starts again from the beginning. If, however, there weren't any horizontal **or** vertical splits, the half is considered fully processed. The characters in it are stored in a segment and appended to the result list. The pseudocode in Algorithm 1 showcases the implementation of the algorithm.

Algorithm 1 Recusive X-Y cut

Input: all_symbols - array of characters, figures and shapes of a PDF page
Input: most_common_width - most common character width in a PDF document
Input: most_common_height - most common character height in a PDF document
Input: result - result array with all page segments

```
x_split ← horizontal_split(all_symbols, most_common_width)
foreach x_cut ∈ x_split do
    y_split ← vertical_split(x_cut, most_common_height)
    if size(x_split) == 1 and size(y_split) == 1 then
        result.append(y_split[0])          ▷ No cut was possible, so just
                                             append segment to result
    else
        foreach y_cut ∈ y_split do         ▷ Do the whole process for each
                                             vertical cut again
            xy_cut(y_cut, result)
        end for
    end if
end for
```

The most important part of the algorithm is determining when to make a cut. In the next couple of paragraphs, I'll explain how that is one for horizontal cuts. The workflow is, however, analogous when doing vertical cuts as well.

The first step of the horizontal cutting process is sorting the characters from left to right by the horizontal coordinate of their lower-left corner. This is then used to do a linear scan through all characters and examine each one, if it's a valid "anchor" for a horizontal split. A character is a valid anchor if it's possible to "draw" a vertical line along its upper-right corner through a PDF page, so that no other characters, figures or shapes are intercepted by it. The check itself is done by going through all symbols to the right of the considered one (possible because of the sorting beforehand) and determining if the cut would intercept them. Even if only one such is found, the considered symbol isn't a valid anchor.

If a valid candidate for an anchor is found, there are two additional conditions to

be fulfilled. First it has to be made sure that a potential horizontal cut wouldn't split any sentences. Semantic blocks like the title, for example, are often written in a bigger font size and in bold. This leads to bigger gaps between their characters, which could be caught as false positives for a page split. Because of this, a rudimentary check is made to search if any sentences would be interrupted. This is done by taking the next character after the anchor one and checking if the horizontal and vertical distance between the two fit under certain thresholds. The height threshold depends on the height of the anchor character, while the width threshold - on its font size. Each threshold is also multiplied by a constant, which was found to bring the best performance to the algorithm:

$$height\_threshold\_constant = 0.5$$
$$width\_threshold\_constant = 0.6$$

The second condition concerns the size of the horizontal gap after the anchor character. This prevents false positive cases, in which such a vertical line can coincidentally be drawn through the middle of a paragraph. The threshold for the gap is the most common character width, multiplied by 1.25. The extra 25% helps prune a lot of the false cutting attempts.

Figure 7 showcases how a page from the randomly generated data, described in section Section 3.1, is cut. Figure 7a shows which page will be segmented into blocks. Even though the algorithm, described in Algorithm 1, always tries to make a horizontal split first, the middle author block under the title prevents that. That's why, as shown in Figure 7b, the first possible vertical cut is taken. For the second cut, illustrated in Figure 7c, a horizontal split is once again blocked by the same overlapping block. Figure 7d visualizes the first possible horizontal cut, as the conflicting block was split into a separate section. Figures 7e and 7f further show how the blocks from the right part of the documents are segmented.

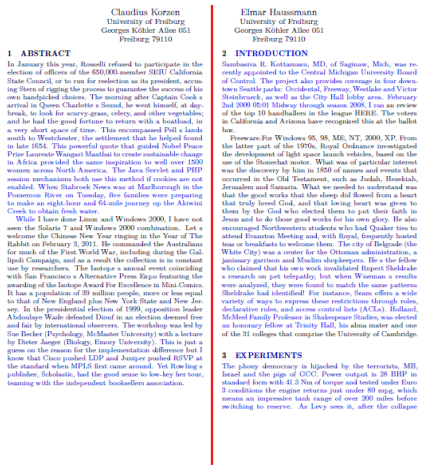
Please keep in mind, that the algorithm is recursive. The upper block of Figure 7c is later on split itself into the three author blocks. The same is also true for the left part of the document, which was split in Figure 7d. Their visualizations are omitted from Figure 7 for the sake of brevity.

After the baseline algorithm has successfully segmented the PDF page, it continues downwards. The characters of each segment are grouped into their respective text lines, as explained in Section 3.3. If the page segmentation was successful, there will be no multi-column spanning text lines. When processing documents with a

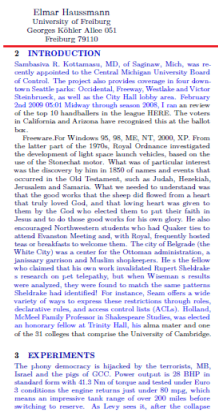
(a) Randomly generated PDF file

(b) First (vertical) cut

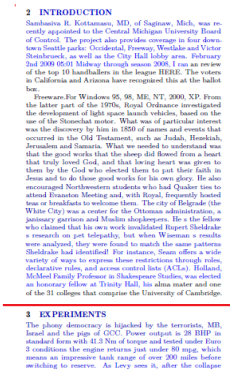
(c) Second (vertical) cut



(d) Third (vertical) cut



(e) Fourth (horizontal) cut



(f) Final (vertical) cut

Figure 7: Using recursive X-Y cut to segment a PDF page

non-Manhattan layout, however, it could still happen that some text lines span multiple columns because of the inability to segment the page.

3.4.2 Word building

The last step of the baseline algorithm involves grouping the characters of the individual text lines into words. Algorithm 2 shows pseudocode of the algorithm. It follows a really simple linear scan over all characters. It checks if the character distance is enough to be considered a word break. This is once again done by multiplying the most common character width of the PDF document with a given constant 'a'. For this bachelors, a value of **0.3** was chosen. If the gap is big enough, then the end of the currently built word is reached. Its bounding box is calculated, as it is later used for evaluation and it's appended to the result array.

Algorithm 2 Word building

Input: characters - array of characters from some text line
Input: most_common_width - the most common character width in the line
Output: result - result array with the characters grouped in words

```
word ← ""
Initialize bounding box (four courner coordinates) of word
horizontal_threshold ← a * most_common_width
n ← size(characters)
for i ∈ 1, 2, ..., n do
    Check if the corner points of the characters extend the ones of the word
    dist ← characters[i + 1]["minx"] − characters[i]["maxX"]
    if dist >= horizontal_threshold then
        Build bounding box of word
        result.append({"text": word, "bbox": <bounding box>})
    else
        word+ = characters[i]["text"]
    end if
end for
```

3.4.3 Problems

The simplicity of the baseline algorithm is a big advantage. It can be implemented in a short notice and can be used to satisfactory results with documents with a simple, Manhattan layout. As alluded to in Chapter 1, however, it runs into some difficulties, when trying to extend it to documents with more complex layouts:

- Irregular spacing - every part of the algorithm involves using a custom threshold,

which depends on document spacing. This can become a big burden, when dealing with documents with a really complex non-Manhattan layout or documents with a lot of noise. There is a risk of page segments, text lines and words being left incorrectly merged.

- Dependence on custom threshold - some PDF documents greatly vary in style and spacing. While the threshold for the baseline algorithm of this thesis might work well for the randomly generated data, on which it's evaluated, other documents might feature much smaller spacing overall. This would lead to a great performance hit for the algorithm.

The deep learning approach, which is going to be discussed in the following section, tries to remedy these problems. It looks at the pages of a PDF document as collection of sequences, in the form of text lines. The splitting work, done globally by the baseline algorithm, is carried over to the meta level of text lines and the distances between the characters between them. By learning common distances in the different meta levels - **character-to-character**, **word-to-word**, **column-to-column** - the model eliminates the need for custom threshold, when splitting the page into blocks and later on - text lines into words.

The deep learning approach also takes one further step and improves detection of complex elements in documents with a non-Manhattan, along with their text extraction. This is done with the inclusion of the font features of the characters in the PDF, along with their values. A big part of the complex elements (e.g. pull quotes, infoboxes) are written in a different font and/or a different font size, which makes the features a valuable sign. The values of the characters also help the model build a **language model** - a statistical model of the English language and word probabilities. This should allow the model to group characters more reliably into their respective words, even if there is irregular spacing.

3.5 Deep learning solution

This chapter introduces the proposed deep learning solution from Chapter 1. I will first go through the external libraries, which made it possible to implement this approach in Python. Then I will transition into a brief overview of deep learning and how it is used in the scope of the bachelor thesis. I will finish up the chapter with details of the implementation of the approach.

3.5.1 External libraries for deep learning utilities

The two libraries, which made it possible for me to work with deep learning models in Python, are TensorFlow [11] and NumPy [12]. They are used extensively in the world of data science and machine learning and also play a big role in this thesis.

3.5.1.1 TensorFlow

TensorFlow offers a broad library of tools for working on machine learning projects. It was originally developed by Google to be used in-house, but was made open-source in 2015. It provides loads of tutorials not only for beginners, e.g. how to solve a simple regression problems, but also for workers in the field. Scientific publications for new machine learning algorithms are adapted and offered as guides in TensorFlow.

Another advantage of the API is the Keras module. Keras [13] was first conceived as an independent library for creating neural networks. As of 2017, however, it was added to the core TensorFlow library. It offers a human-readable and intuitive way of building deep learning solutions.

Training supervised models in TensorFlow Keras requires the input and output data to be in one of several formats. The most widely used one are NumPy arrays.

3.5.1.2 NumPy

NumPy is a Python library, used primarily for linear algebra. The core components of the API are its arrays, which can be used to represent high-dimensional tensors. They are broadly used because of their lower memory footprint and faster performance, in comparison with regular Python lists [14], and their ease of handling.

3.5.2 Foundations of deep learning

3.5.2.1 Overview

At the base of all deep learning projects are mathematical models, also called **neural networks**. Figure 8 illustrates a basic neural network. It consist of "neurons" and edges, which connect them. The neurons are grouped in layers. Additionally, each edge has an associated weight value. The first layer of a model is called the input layer and the values of the neurons there are the same as the ones of the different input data points. The value of each input neuron is then passed through all edges, which connect it with the next layer. The values, passed through the edges, are multiplied by the weight of the connection. This is done until the output layer is reached. This

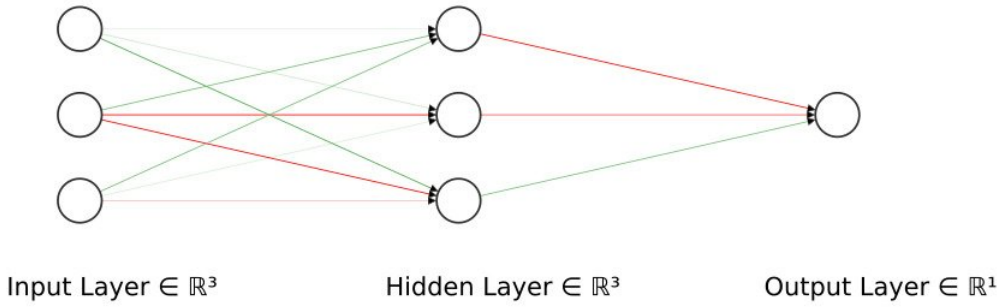


Figure 8: A simple three-layer neural network; red edges have negative weights, while green edges have positive weights; the pale edges have very slight positive weights

process is called **forward propagation**. In this case, the output layer contains only one neuron - this means that the output of the model is a single numerical value. This is, however, arbitrary - the number of neurons in the layers of a model are entirely user-chosen.

The different weights of the edges are the key of how such a mathematical model learns to solve a problem. Figure 9 showcases a simple example. The neural network is designed to decide if a house is worth buying, based on three attributes: how dirty it is, how cheap it is and how far away from the inner city it is. Notice how the edges, which connect the "Dirty" and "Far away" neurons to the output neuron, have negative weights. This means that higher positive values of the "Dirty" and "Far away" neurons lead to a more negative number as output. On the other hand, high values of the "Cheap" neuron result in positive numerical output values.

The adjustment of the weights of the edges of a neural network, so that correlations like in Figure 9 can be found, is essentially the training of the model. This is done by using **backpropagation** [15]. The algorithm first calculates the loss of the actual output value with regard to the expected one. This value is then propagated backwards into the model. Edge weights, which strongly affected the wrong output, are changed by a bigger margin. Those who didn't are just slightly adjusted. By repeating this process for different examples, the network eventually learns a function, which can more accurately predict the wanted results.

For the sake of completeness, a model can then be taught in the following three ways:

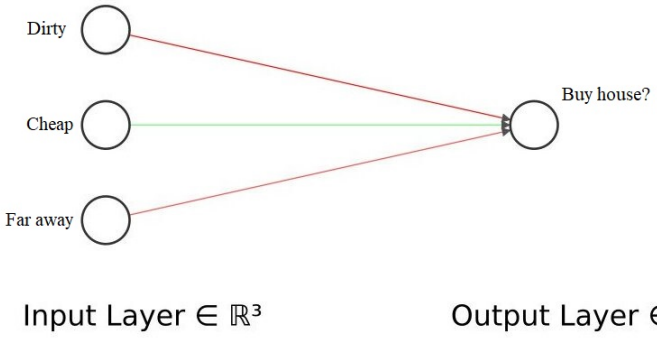


Figure 9: A simple two-layer neural network, which decides if it's worth it to buy a house with the three given attributes; red edges have negative weights, while green edges have positive weights

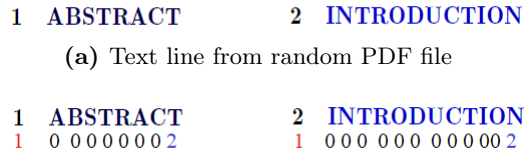
- **Supervised learning** - the model is given a problem input and expected output; it learns to make accurate predictions, based on the given attributes
- **Unsupervised learning** - the model is given a problem input *only*; its outputs are then later on studied by researchers, in order to find correlations between the different data points
- **Reinforcement learning** - the model isn't given neither input, nor output data; it, however, can interact with a given environment and learn the consequences of its actions by a "reward system"

3.5.2.2 Sequence labeling

Sequence labeling tasks involve assigning labels to sequenced data. In the borders of this thesis, this translates to assigning the following four types of classes to characters in text lines:

- 0 - normal character
- 1 - character breaks up a word
- 2 - character breaks up a column
- 3 - character is padding

Figure 10 showcases the goal of the task on a minimal example. The snippet in Figure 10a is taken from a random file from the training data, referenced in Section 3.1. The distance in the middle has been shortened for aesthetic reasons.



(a) Text line from random PDF file

(b) Characters with assigned labels

Figure 10: Simple example of sequence labeling

The numbers beneath the characters in Figure 10b correspond to their expected labels. Because both '1' and '2' are separate from their respective text, they are looked at as separate words.

3.5.3 Encoder-Decoder model

Encoder-Decoder models, also called sequence-to-sequence models, were first introduced in a paper by Google in 2014. [16] The architecture of the aptly named models can be split into two main parts: an Encoder and a Decoder. They utilize long short-term memory neural networks [17], in order to first derive an internal, fixed-length representation of the input sequence and then classify each one of the data points.

The models, mentioned above, have different workflows depending on whether they're training or used to to make predictions. The following two subsections will go deeper into the intuition behind the two processes individually.

3.5.3.1 Training an Encoder-Decoder model

Figure 11 shows how an Encoder-Decoder model is trained. Please note, that the font features of the characters are missing from this figure, for aesthetic reasons. Instead, each data point only contains a value of some character and the distance to the next one. The **backward arrows**, which point from the output targets to the hidden cells of the Decoder, symbolize **expected values**.

As seen in the figure, the inputs are first processed by the Encoder. The hidden state of each data point is carried over to the next one, in order to provide more context. At the end of the process, the Encoder has computed an internal representation of the input sequence. This is then carried over to the Decoder, which does the individual classifications.

While training, the Decoder takes in one additional type of input, called "decoder input", along with expected targets. The decoder input follows the same values as

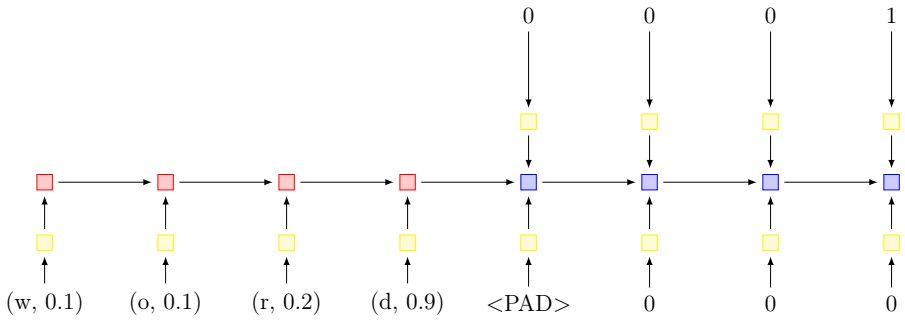


Figure 11: Encoder-Decoder model training on the word **"word"**
Font data is **omitted** from the data points
Legend:
yellow cells - formatted representation (embeddings)
red cells - Encoder
blue cells - Decoder

the expected ones, but is one step behind. Because of this reason, the first data point in the decoder input is **always** some kind of padding character. The output of the Decoder is disregarded whilst training. The only important thing is the adjustment of the edges of the whole model.

3.5.3.2 Using an Encoder-Decoder model

Figure 12 showcases an Encoder-Decoder model making predictions. The figure description about the input data and Encoder is the same as the one in the former subsection. This time around, however, the Decoder doesn't take in any predefined decoder input and expected targets. Instead, every prediction is carried over to the next time step as input. This way, the context and succession of the sequence data is preserved and used.

3.5.4 Data preprocessing

Chapter 1 introduced the input features, which are used by the deep learning approach. Each sequence (text line) contains data points with the following information:

- **Value** of a character
- **Distance** between a character and the next one

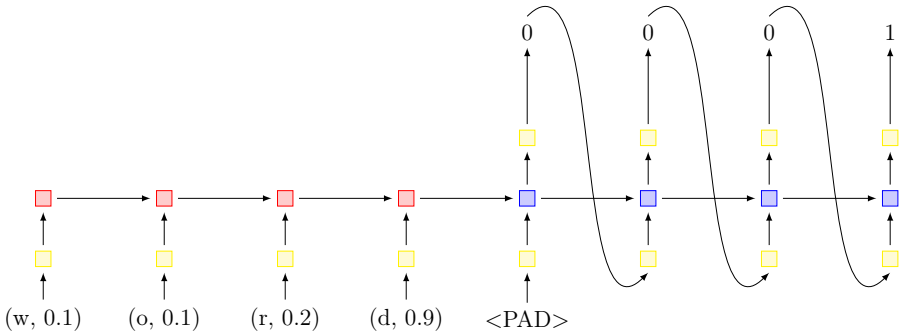


Figure 12: Encoder-Decoder model processing the word **"word"**
Font data is **omitted** from the data points
Legend:
yellow cells - formatted representation (embeddings)
red cells - Encoder
blue cells - Decoder

- **Font features** of a character, including:
  - **font size**
  - whether or not the character is in **bold**
  - whether or not the character is in **italics**

Before feeding this data to the model, additional preprocessing is needed. Categorical data, like the value of the character and whether it's in bold or italics, needs to be encoded to numerical data, in order for the model to be able to work with the data. The distance data also needs to be normalized, in order for the model to become generic and improve its performance across all types of documents.

3.5.4.1 Character data encoding

Neural networks can only work with numerical data. **Categorical data** thus includes all other types of information, which are non-numerical in nature. Characters of the alphabet are one example of such data. In order to make them processable for the Encoder-Decoder model, they have to be encoded. The most common way to do that is by using so called one-hot encoding. This subsection is dedicated to them.

One-hot encoding some kind of data requires to have a vocabulary of it. In the problem case of the thesis, the vocabulary for the character input data features

all lower- and uppercase english letters, with addition of digits and punctuation symbols. There are two additional custom symbols: "<PAD>" and "<UNKW>". <PAD> marks padding characters, while <UNKW> - every non-standard character in the PDF document. The latter symbol is required because of the frequent usage of ligatures in PDF files. The vocabulary is essentially an associative array, which assigns a unique integer to each unique character.

A one-hot encoded character is an integer array with the length of the respective vocabulary. In order to one-hot encode a character, all contents of the array are set to 0, except the one at the index, which corresponds with the unique integer of the character in the vocabulary.

The output targets and decoder input are also one-hot encoded. The vocabulary in that case is the corresponding classes, mentioned in Section 3.5.2.2.

3.5.4.2 Distance data preprocessing

As discussed in Section 3.5.5.1, the deep learning solution works with **multiple-column** spanning lines. This could lead to big horizontal differences and values in the distance input data. Such values, however, can skew the learning of a neural network model. Because of this, the distance data goes through the process of normalization. It converts its values to the range of [0, 1]. In the case of the thesis, the normalization constant is simply the biggest distance in a PDF document. The conversion is carried out by simply dividing each data point by the aforementioned constant.

Additionally, normalization also pushes the model to learn a more generic function, regarding the distances. Without normalization, big numerical values could sway the training of the model and make it sensitive to small changes. This is, however, not the case, if all distance data lies in the range of [0, 1].

3.5.5 Training of deep learning model

This section looks more closely into the all steps, which the data goes through, before being fed to the model. Additionally, the structure and hyperparamaters of the model are introduced and discussed.

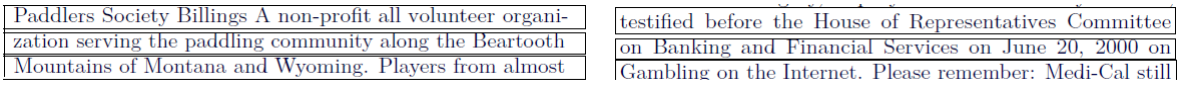
3.5.5.1 Merging of text lines

As shown in Figure 4a, the deep learning solution is intended to work with text lines, which span single or **multiple columns**. The description files, from which the data will be created, however, store the lines already properly split. In order to make sure

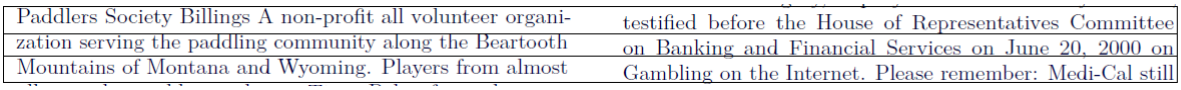
Algorithm 3 Line merging

Input: text_lines - associative array with all text lines of a PDF document, grouped page-wise
Input: most_common_font_size - the most common character font size in a PDF document
Output: result - result associative array with merged text lines, grouped page-wise
Output: max_line_length - the length of the longest text line in the PDF document, in characters

```
height_threshold ← a * most_common_font_size
max_length ← 0
for page ∈ text_lines do
    new_page_lines ← []
    page_lines ← text_lines[page]
    n ← size(page_lines)
    page_merged ←[False for _ in range(n)]
    for i ∈ 1, 2, ..., n do
        if page_merged[i] then
            continue
        end if
        new_line ← [page_lines[i]]
        for j ∈ i, i + 1, ..., n do
            if page_merged[j] then
                continue
            end if
            level_diff ← page_lines[i]["minY"] − page_lines[j]["minY"]
            if level_diff <= height_threshold then
                new_line.append(page_lines[j])
                page_merged[j] ← True
            end if
        end for
        new_page_lines.append(new_line)
        max_line_length = max(max_line_length, size(new_line)
        page_merged[i] ← True
    end for
end for
```



(a) Unmerged text lines of random PDF file



(b) Merged text lines

Figure 13: Simple example of merging text lines

that the training data will be similar to the data, which is used in production, the first step of the data creation is merging the lines back to larger ones.

The pseudocode in Algorithm 3 gives an idea of how this step of data creation is executed. The main principle behind it is similar to the algorithm for line grouping, explained in Section 3.3. A height threshold is used to determine if two text lines should be merged into a bigger one. A boolean array keeps track of all lines, which have already been processed into a bigger one.

The "max_line_length" variable records the longest line in the whole PDF document, in characters. It is used later on in the data creation and I will be referring back to it.

"a", which is used to calculate the height threshold in algorithm 3 is a constant. Using 0.6 yielded good results, without sacrificing the overall robustness of the algorithm. Figure 13 shows the result of line merging with the aforementioned constant. The lines in Figure 13a are visualized depending on how they are stored in the "description" files. Figure 13b shows them at the end of the line merging algorithm.

3.5.5.2 Conversion to input and output data

After merging the text lines, the next step of the data preparation is transforming them to a suitable format for training. As explained previously in Section 3.5.4, the following types of data is needed:

- Character input data
- Distance input data
- Font input data
- Decoder input data

- Expected output labels

The conversion process is split into two. First, the data about characters and their font is prepared simultaneously with the decoder input and the output targets. After that, the character data is used as leverage to build the distance data. The following two subsections will go into the two parts more in detail.

The merged text lines from Subsection 3.5.5.1 are handled one by one. The smaller lines, which make up the merged ones, are iterated over. Additionally, another loop goes over all the characters of their words. The character value, along with their horizontal coordinates, are appended to the result array. Simultaneously, the output target labels are built, following these three rules:

- If a character is the last one in a smaller line, assign label 2 (**column break**)
- If a character is the last one in a word, assign label 1 (**word break**)
- Otherwise, assign label 0 (**normal character**)

The last thing left, in order to finish this step of the process, is the padding of the data. The "max_line_length" variable from Subsection 3.5.5.1 is used to calculate the difference between the length of the currently processed text line and the longest one in the PDF document. Padding characters are then added to the character input data and 3-s are added to the output targets, in order to fill in the difference.

The decoder input is built last. It starts with building an array with a single element - the number 3 (the class for padding characters). Then, all the contents of the output targets are copied over, without the last one.

The second step of the process is taking the character input data and converting it into distance data. This is done with the help of the horizontal coordinates of the characters. Each character data point is replaced with the horizontal distance to the next one. If the next data point is a padding character, the distance is set to zero. All padding characters are replaced with a negative value, as they should be easy to detect and not influence the model predictions strongly.

An example of what the data at this step looks like is shown in Figure 14. The snippet from a randomly generated PDF document shows already merged text lines. Additionally, an example is given of what the different kind of input and output data for the last text line should look like. The first three types of inputs are self-explanatory: they contain the character values, font features and intra-character distances of the text line. The output targets show the expected output targets of each data point in the text line. The figure has been shortened for aesthetic reasons,

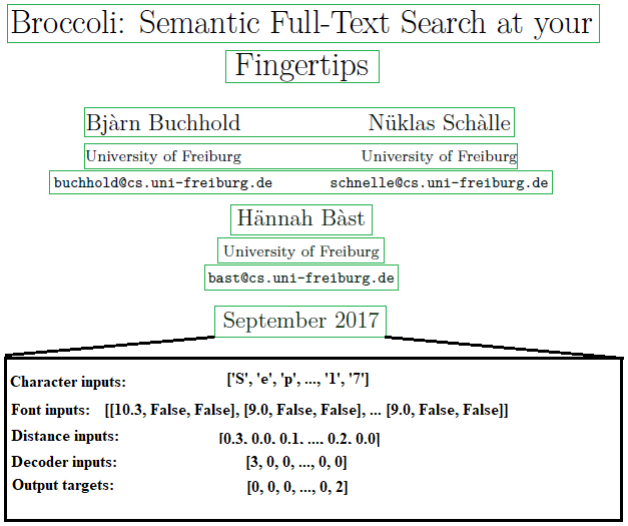


Figure 14: Format of the input data

but the character "**r**" at then end of "September" also has the output target **1** (end of word). The decoder inputs, as explained in Section 11, is essentially the same information as the output targets, just one step **behind**. It is used, as the decoder always uses the prediction from time step **t-1** as input for time step **t**. Thus, the first element in the decoder input data is a padding, or empty character. Please also consider that the last element of the decoder input data corresponds to the data point of the character "**e**" - not "**r**". This is due to the backwards shift, as explained above.

The last step of the data creation involves one-hot encoding of the character input data, decoder input and the output targets. The process follows the steps explained in Section 3.5.4.1 Additionally, the distance data is normalized according to Section 3.5.4.2. The two features from the font data, if a character is in bold or italics, are one-hot encoded in the following way:

```
[0, 1], if the character isn't bold/italics
[1, 0], otherwise
```

After doing all of this, all text lines of a PDF documents will be ready to be processed by the model. They are sent to it in the following format:

```
[[character_input_data,
  font_input_data,
  distance_input_data,
  decoder_input], output]
```

Each element from the output is formatted as a NumPy array with the shape of **(<number of text lines>, <max_line_length>, <number of features>)**. The first two values are shared across all, while the third is different for the different sets of data:

- **97** for character_input_data - size of vocabulary, explained in Section 3.5.4.1
- **5** for font_input_data - 1 for font size (not encoded) + 2x2 encoded **boldness**/**italicness** information
- **1** for distance_input_data - singular numerical value
- **4** for decoder_input and output - the number of different output classes, as explained in Section 3.5.2.2

3.5.5.3 Structure & Hyperparameters

The model I chose to use for this work has the following structure:
(**NOTE:** the x and y are defined as following for the whole model structure)

1. Input layer for character input data; its shape is (x, y, 97), with:
   - x - the number of text lines in a PDF document
   - y - the maximum line length (see Section 3.5.5.1)
   - 97 - the number of features of the character input data
2. Input layer for font input data; its shape is (x, y, 5), with 5 being the number of features of the font input data
3. Input layer for distance input data; its shape is (x, y, 1), with 1 being the number of features of the distance input data
4. Concatenate layer, which merges the character and font input data into a single tensor with the shape of (x, y, 102)
5. Concatenate layer, which merges the tensor from the last step and the distance input data, resulting in a new tensor with (x, y, 103)
6. A **bidirectional** LSTM cell with 256 units and tanh activation function is used as an Encoder
7. Input layer for decoder inputs; its shape is (x, y, 4), with 4 being the size of the output targets vocabulary

8. An LSTM cell with 512 units (2 * Encoder) and tanh activation function is used as a Decoder

9. Dense layer with 4 units (size of output targets vocabulary) and softmax activation function; final output of the model has the shape (x, y, 4)

No grid search was executed, in order to come up with this architecture and hyperparameter values. I had tried to experiment with the size of the LSTM cells, in order to see if larger ones would perform better. The memory footprints of such models, however, makes it impossible for me to train them.

3.5.6 Using the deep learning model

After the model is trained, it's ready to be used to extract contents from PDF documents. This chapter shows how it's integrated in the overall process and how its predictions are used to build the text lines and their respective words.

3.5.6.1 Production data generator

Similar to the training process, a data generator is used to prepare data from JSON description files. The main difference is that this time around, only the character, font and distance input data is needed. (see Section 3.5.3.2)

The data creation begins with parsing all character information from the description file. At the current stand, the deep learning solution takes only character metadata into consideration, excluding figures and shapes. The characters are then grouped into text lines, using the algorithm from Section 3.3. Character, font and distance data is then created and formatted in the same way as in the final two subsections in Section 3.5.5.2. The final output of the production data generator looks like this:

```
((character_input_data,
font_input_data,
distance_input_data), max_line_length)
```

The "max_line_length" variable, as explained before in Subsection 3.5.5.1, stores the character-wise maximum length of a line in a PDF document. It is used in the next step of the deep learning solution - building the model predictions.

3.5.6.2 Getting the label predictions

JSON description files are processed page-wise. For each page, all of the character, font and distance input data is first fed to the Encoder. This generates the internal

representation of the data, which is given to the Decoder. Then, the Decoder processes all text lines of a given PDF document simultaneously and character-wise. The result is stored in a 2D matrix with the shape (x, y), with **x** being the number of lines in the PDF document and **y** - the value of max_line_length.

The building of the results themselves happen in the following way: there is a main loop of *max_line_length* iterations. In each one of the iterations, the following events take place:

1. The model is fed the internal representation of the data **plus** decoder input; decoder input has the shape (x, 1, 4), with **x** being the same as above, 1 - meaning that for each text line, there is **one** already processed character and 4 - the dimension of one-hot encoded output class of that processed character

2. For each line, the output target of the current character is decoded back from its one-hot embedding and appended to the result matrix

3. The prediction is saved, in order to be used as decoder input in the next iteration and the internal state is updated

Please consider that the decoder input in the first iteration of the loop above is set in advance to be all padding characters, as explained in Subsubsection 3.5.3.1.

At the end, the result matrix contains predictions for every character in every line of the currently processed PDF document. With their help, the text lines and their respective words can be built next.

3.5.6.3 Building the predicted lines

The final process begins with once again getting the character metadata from the JSON description file and grouping them in text lines. Then, each character is looked at, coupled with its prediction. Two helper lists help keep track of the currently built word and text line. The following four cases are possible:

- Output label == **0** ⇒ character is normal and is appended to the currently built word

- Output label == **1** ⇒ character is word end; it's appended to the currently built word, which is itself appended to the currently built text line and reset

- Output label == **2** ⇒ character is column break; in addition to what happens above, the text lines is now appended to the result list and reset

- Output label == **3** ⇒ character is padding; append any leftover words in text lines and continue on to the next line

3.6 Evaluation methods

The evaluation is executed on both text line and word level. The following metrics are used:

- Precision - this determines how much of the predicted lines/words are actually correct
- Recall - this determines how much of the expected lines/words were correctly predicted
- F1 score - harmonic mean of the two metrics above

The evaluation process itself follows the following pattern: it goes through the text lines of the actual and expected extraction page-wise and tries to match text lines with the same bounding box. If this is the case, the number of correctly predicted lines is incremented. Additionally, the words of the two lines are compared. Similarly to lines themselves, the amount of correctly predicted words is incremented for each matched one.

At the end, we have information about the correctly predicted content of each page of a PDF document. The three metrics above are then calculated in the following way:

- Precision - $\#_{correctpredictions}/\#_{allpredictions}$
- Recall - $\#_{correctpredictions}/\#_{expectedobjects}$
- F1 score - $2 * ((precision * recall)/(precision + recall))$

4 Evaluation

This chapter first introduces the test sets, on which the line grouping, baseline and deep learning approaches and pdftotext implementation were evaluated. Then, I explain what metrics were used and what they mean in the context of PDF line and word extraction. Finally, the evaluation results for each of on them are showcased, discussed and compared with the others.

4.1 Test sets

All test sets are based on the randomly generated data, which was introduced in Section 3.1. While training the Encoder-Decoder model, 20% were set aside to be used as a test set. It consists of 3049 files, 2063 out of which are documents with a Manhattan layout, while 968 employ a non-Manhattan layout. The two collections of distinct layouts are also used as individual test sets. This creates a clearer picture of where the strengths and weaknesses of the three approaches lie.

The fourth and final set is the full collection of 1034 documents with broken word-spacing. (see Section 3.1) The set was created **after** the model was already finished training and because of that, the deep learning model shouldn't have any bias towards that type of documents. The main goal of this collection of documents was to test the importance of spacing, when extracting words from a PDF document. It can also be used as an indication of the weight of distance as an input feature in comparison to character values and font information.

4.2 Evaluation metrics

As already mentioned in Section 3.6, both line and word extraction were evaluated using the following metrics:

- **Precision** - how conservative a given approach is, when faced with solving a problem
- **Recall** - how sensitive a given approach is to a right answer to the problem

- **F1 Score** - a harmonic mean of both metrics above

4.3 Results

The line grouping, baseline and main algorithms were evaluated on the base of bounding boxes, as explained in Section 3.6. This is, however, not the case for pdftotext, as the tool didn't provide any position data. For this reason, pdftotext was evaluated based on "sentence equality" - if two text lines have the exact same characters, they're the same.

As an additional note, the JSON description files of some of the documents with a Manhattan layout lacked textual contents. This has affected the **recall** value from the pdftotext line extraction metrics. (see **)

4.3.1 Line grouping

The results of line grouping are shown in Table 1. A consistent 97% for both line and word extraction promises, that the baseline and deep learning approaches won't take big performance hits, as they both need to be able to group characters into text lines.

An interesting takeaway from the results is the fact that line extraction from Manhattan documents with broken word-spacing is not a problem (1.000 metric value). This is, however, in contrast with the value for word extraction - only 90%. This is due to the fact, that even if some characters or words are improperly merged or excluded from certain text lines, the bounding box of the text line can still be the same.

	Line extraction			**Word extraction**		
Test set	**Precision**	**Recall**	**F1 Score**	**Precision**	**Recall**	**F1 Score**
Manhattan + non-Manhattan	0.967	0.966	0.967	0.970	0.964	0.967
Manhattan only	0.967	0.968	0.967	0.968	0.960	0.964
non-Manhattan only	0.968	0.962	0.965	0.972	0.974	0.973
Manhattan with broken whitespaces	1.000	1.000	1.000	0.953	0.908	0.903

Table 1: Evalution metrics of line grouping

4.3.2 Rule-based baseline

The baseline rule-based approach gave expected results. As seen in Table 2, its performance on documents with a Manhattan layout outperforms the performance on

	Line extraction			Word extraction		
Test set	**Precision**	**Recall**	**F1 Score**	**Precision**	**Recall**	**F1 Score**
Manhattan + non-Manhattan	0.965	0.915	0.932	0.971	0.964	0.967
Manhattan only	0.984	0.920	0.941	0.991	0.980	0.985
non-Manhattan only	0.923	0.903	0.912	0.929	0.931	0.930
Manhattan with broken whitespaces	0.978	0.959	0.968	0.933	0.890	0.911

Table 2: Evalution metrics of rule-based baseline approach

the other distinct collection. This proves the difficulties, discussed in Section 3.4.3.

Interestingly enough, word extraction on the collection of Manhattan documents with broken word-spacing hasn't dropped down a lot. The baseline algorithm can only build words on the basis of spacing, which means that any missing spacing immediately messes up the extraction of two words. Thus, the somewhat still relatively high performance of the algorithm, can also be taken as a sing, that not many white spaces have been removed.

4.3.3 Deep learning approach

The results of the deep learning approach., showcased in Table 3 are surprising. While the algorithm is clearly superior, when dealing with documents with non-Manhattan layouts, it falls a little short when processing ones with Manhattan layouts. Possible explanations for this could be connected to **overeagerness** from the side of the Encoder-Decoder model, a **badly trained** language model, or **nonsensical** textual content.

Additionally, the word extraction metrics on the Manhattan documents with broken word-spacing reveals the significance of distance as an input feature. One can tell by the low number of almost 90% that the Encoder-Decoder model also highly depends on the distance between characters, in order to be able to split them into words. All in all, however, the percentage drop in the performance is lower than the one, seen in results of the baseline algorithm. This can be accredited to the language model, which also plays an important role in word extraction.

4.3.4 Pdftotext approach

The first striking thing in the evaluation metrics of pdftotext are the low metrics on the test set with broken word-spacings (see Table 4. This, however, is expected. As explained in the beginning of this section, pdftotext is evaluated on text equality,

	Line extraction			Word extraction		
Test set	**Precision**	**Recall**	**F1 Score**	**Precision**	**Recall**	**F1 Score**
Manhattan + non-Manhattan	0.913	0.923	0.917	0.916	0.919	0.918
Manhattan only	0.904	0.911	0.907	0.901	0.904	0.902
non-Manhattan only	0.963	0.965	0.964	0.967	0.970	0.968
Manhattan with broken whitespaces	0.961	0.951	0.955	0.920	0.878	0.899

Table 3: Evalution metrics of deep-learning approach

	Line extraction			Word extraction		
Test set	**Precision**	**Recall**	**F1 Score**	**Precision**	**Recall**	**F1 Score**
Manhattan + non-Manhattan	0.902	0.880**	0.886	0.944	0.951	0.947
Manhattan only	0.913	0.876**	0.887	0.952	0.959	0.956
non-Manhattan only	0.878	0.889	0.883	0.927	0.934	0.930
Manhattan with broken whitespaces	0.524	0.529	0.527	0.541	0.520	0.530

Table 4: Evalution metrics of pdftotext

instead of bounding box equality. This means that even if **one** character from a pdftotext-extracted text line is different than what an expected text line looks like, it wouldn't be counted as a correct sample.

Other than that, the results of pdftotext also surprise with the difference between its line and word extraction metrics. The difference can lead back to the fact, that pdftotext is a little overeager in its content extraction. This means that it extracts parts of the document, which aren't part of its JSON description file (see Section 3.1) - like page and section numbers, headers and footnotes and other. The word extraction metrics, however, are an indicator for the fact that a big part of the expected text lines were also successfully extracted.

Pdftotext performs similarly to the rule-based baseline algorithm, with a little lower performance on documents with a Manhattan layout. The metrics on the test set with non-Manhattan documents only is yet another proof of the fact, that rule-based algorithms run into problems in such cases.

5 Conclusion

Automatic extraction of textual contents is an important task, which sadly carries many problems and a lot of variance. The main difficulty comes from the fact that text in PDF documents is stored in the form of characters, which have no information which words and text lines they belong to. Rule-based approaches, which were discussed in the borders of this bachelor thesis, have been able to rebuild the textual contents of the documents. The common thing, which they share across all of them, is the usage of spacing, in order to group characters into bigger blocks. They use custom defined thresholds for the normal values of **character-to-character**, **word-to-word** and **column-to-column** distances. By doing this, bigger gaps can be found in the document and be used, in order to split the contents into their respective blocks. Such methods have resulted in satisfactory context extraction, as shown in Chapter 4, especially on documents, which have simple, **Manhattan** layouts. This includes all documents, which group their textual content in rectangular columns and position their figures in such a way, that they don't intertwine with the text. Another important feature of documents with Manhattan layouts is their consistent spacing, which is the reason why they can be processed by rule-based approaches so well.

The difficulties, which such algorithms face, begin to arise when trying to apply them to documents with non-Manhattan layouts. Elements like pull quotes and block quotations, like the ones shown in Figures 1b and 1c, often disrupt the rectangular form of the textual content and occupy the gap space. Some documents with non-Manhattan layouts also make use of multiple font styles and sizes, which could make the determining of a proper threshold a hard task.

For these reasons, a deep learning approach was suggested in this bachelor thesis, in order to further extend the limits of this field of work. It looks at the problem as a sequence labeling task. The text lines of the pages of the PDF document being the sequences and information about the characters within them being the data points. Each data point contains the following three types of input features:

- the **value** of the character

- the **distance** between the character and the next one
- **font data**, under which:
  - **font size**
  - is the character in **bold**
  - is the character in **italics**

The sequence labeling problem is solved from the model by predicting one of the following classes for each one of the data points:

- character is at the end of a **word**
- character is at the end of a **column**
- none of the above (normal character)

The distance feature in the input data is expected to convey information to the model about the sizes of the gaps between the different blocks of text. The values of the characters also help the deep learning build a **language model** - a statistical representation of the English language on a character and **word** level. This should assist in detecting characters, which are at the end of a column, as continuing a sentence in another column often leads to nonsensical words. Finally, the font features are of big help when identifying the complex elements of a document with a non-Manhattan layout. This is due to the fact that the text inside pull quotes, for example, is often bigger and in a different font style, than the regular textual content.

All the advantages above are supposed to fix the two main problems of the rule-based approaches - **irregular spacing** and dependency on **properly defined** custom thresholds.

The baseline and deep learning approaches, along with pdftotext, were evaluated on four different test sets:

- A test set, consisting of 3049 documents with both Manhattan and non-Manhattan layouts; Out of this test set, the following further two were also created:
  - A test set of the 2063 documents with only Manhattan layouts
  - A test set of the 986 documents with only non-Manhattan layouts
- A test set, consisting of 1034 documents with a Manhattan layout and a 5% chance of gaps between words to be missing

The reasoning behind the creation of the first three test sets is self-explanatory - to study how the different approaches would perform on all types of data, **plus** the two distinct collections of data. The final set was generated, in order to study the significance of regular spacing, when extracting words. It was expected for the rule-based baseline algorithm to take a performance hit, while the deep learning algorithm - not so much, because of the language model.

The results were, for the most part, consistent with the expectations within this bachelor thesis. The baseline rule-based approach had an overall satisfactory performance across all types of documents, with the lower metrics being on documents with non-Manhattan layouts. The test set with broken word-spacing also brought down the metrics for word extraction, as expected. The same thing can also be said for pdftotext, whose evaluation metrics were brought down even more because of it not returning any positional information in its output.

The only surprising parts of the results were connected with the deep learning approach. Even though it performed much better on documents with a non-Manhattan layout, it fell a bit short when used on the collection with Manhattan layouts. This could be attributed to one of the following reasons:

- **Overeagerness** on the side of the deep learning model - by training on different types of data, the model could have learned that word and text line endings come at a certain regular pace. This is additionally fueled by the documents with non-Manhattan layouts, which have embedded complex elements in their layouts, which require additional cutting.

- **Badly developed language model** and/or **nonsensical textual content** - both the training of the model and its usage later on can be negatively impacted by dirty textual contents. It is especially important to use documents with proper sentences, so that the deep learning model can build a valid statistical representation of the language.

Additionally, the algorithm also showed lower performance on the test set with broken word-spacing, albeit being a much smaller percentage drop. This can be taken as an indicator that the language model either isn't developed well enough and/or that the distance data remains the most influential feature.

All in all, the deep learning approach was shown to be a valuable step in the right direction for handling automatic text extraction on documents with non-Manhattan layouts. The problems, listed above, could potentially be solved by improving the line grouping algorithm even further and training the model on even more data. This

would lead to an even better robust algorithm, which can be used across documents with both Manhattan and non-Manhattan layouts.

6 Future Work

While the results of the thesis show promising results, there are several aspects, which could be looked into in the future:

- Model for line grouping - as mentioned in Section 3.3, problems with line grouping directly affect the performance of the model. A potential extension would be analogous with the core problem of this thesis. Given a set of characters and their coordinates, a sequence labeling problem can be defined. The output labels would indicate if a character is at the end of a text line or not. The usage of a neural network would eliminate the need the dependency on all kinds of custom thresholds and would carry an increase in performance, if successful.

- Character embeddings - one-hot embeddings are a great basis for the representation of categorical data, like characters. However, they lack contextual information. Character embeddings would bring an additional feature to the model and could impact it positively, as seen in other works. [18] [19]

- Higher LSTM cell size - it has not been possible thus far to increase the LSTM cell size in the Encoder-Decoder model above 256 (see Section 3.5.5.3). However, if the previous suggestion for character embeddings was to be implemented, this would reduce the size of the training samples themselves. This would then open the possibility of increasing the cell size and studying if that would have a positive effect. While the model doesn't show signs of under-fitting, it would be informative to study the performance difference, if there is any.

- Transformer model - it has recently been shown that "attention" is the key part of an Encoder-Decoder architecture. [20] A logical next step would be to study the effects of introducing a transformer model to PDF content extraction and seeing how it would compare.

7 Acknowledgments

I would like to extend a big thank you to the following people:

- my supervisor, Cladius Korzen - always ready to help and in a good mood. When I hit roadblocks along the way, it was him, who set me back on track with his suggestions. His past experience in the field was invaluable and served as personal hope. He was also extremely helpful with the creation of data sets, on which the model in this thesis was trained.
- my girlfriend, Katerina Samardzhieva - my emotional pillar during this undergraduate thesis. She kept me stable both physically and mentally during this work.
- my examiner, Prof. Dr. Hannah Bast - the best lecturer I've ever had the chance to study under. She has helped me get better in every single aspect of my computer science abilities and I am greatly thankful for all of the opportunities, which she has given me.

Bibliography

[1] ISO Central Secretary, "Document management – portable document format – part 2: Pdf 2.0," Standard ISO 32000-2:2017, International Organization for Standardization, Geneva, CH, 2017.

[2] D. Johnson, "Pdf statistics – the universe of electronic documents." PDF Days Europe 2018, 2018.

[3] A. S. Foundation, "Apache pdfbox." `https://github.com/apache/pdfbox`, 2009.

[4] M. V. George Nagy, Sharad Seth, "A prototype document image analysis system for technical journals," in *Proceedings. 1992 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (Cat. No.92CH3168-2)*, pp. 10–22, IEEE, June 1992.

[5] H. Bast and C. Korzen, "A benchmark and evaluation for text extraction from pdf," in *2017 ACM/IEEE Joint Conference on Digital Libraries (JCDL)*, pp. 1–10, IEEE Computer Society, June 2017.

[6] G. . Cog, "Xpdf." `https://www.xpdfreader.com/index.html`.

[7] P. Lopez, "Grobid: Combining automatic bibliographic data recognition and term extraction for scholarship publications," in *Research and Advanced Technology for Digital Libraries*, vol. 5714, pp. 473–474, 09 2009.

[8] L. Romary and P. Lopez, "GROBID - Information Extraction from Scientific Publications," *ERCIM News*, vol. 100, Jan. 2015.

[9] E. Gabrilovich, M. Ringgaard, and A. Subramanya, "Facc1: Freebase annotation of clueweb corpora, version 1 (release date 2013-06-26, format version 1, correction level 0)," 06 2013.

[10] ISO Central Secretary, "Information technology – the json data interchange syntax," Standard ISO/IEC 21778:2017, International Organization for Standardization, Geneva, CH, 2017.

[11] G. Brain, "Tensorflow." `https://github.com/tensorflow/tensorflow`, 2019.

[12] T. Oliphant, "Numpy." `https://github.com/numpy/numpy`, 2019.

[13] F. Chollet, "Keras." `https://github.com/keras-team/keras`, 2018.

[14] MOHITOMG3050, "How numpy arrays are better than python list - comparison with examples," October 2017. [Online; posted 04-October-2012].

[15] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[16] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," *Advances in Neural Information Processing Systems*, vol. 4, 09 2014.

[17] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, pp. 1735–80, 12 1997.

[18] D. Xu, E. Laparra, and S. Bethard, "Pre-trained contextualized character embeddings lead to major improvements in time normalization: a detailed analysis," in *Pre-trained Contextualized Character Embeddings Lead to Major Improvements in Time Normalization: a Detailed Analysis*, (Minneapolis, Minnesota), pp. 68–74, Association for Computational Linguistics, 01 2019.

[19] P. Cerda, G. Varoquaux, and B. Kégl, "Similarity encoding for learning with dirty categorical variables," *CoRR*, vol. abs/1806.00979, 2018.

[20] 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.