A Benchmark and Evaluation for Text Extraction from PDF

Hannah Bast
University of Freiburg
79110 Freiburg, Germany
bast@cs.uni-freiburg.de

Claudius Korzen
University of Freiburg
79110 Freiburg, Germany
korzen@cs.uni-freiburg.de

ABSTRACT

Extracting the body text from a PDF document is an important but surprisingly difficult task. The reason is that PDF is a layout-based format which specifies the fonts and positions of the individual characters rather than the semantic units of the text (e.g., words or paragraphs) and their role in the document (e.g., body text or caption). There is an abundance of extraction tools, but their quality and the range of their functionality are hard to determine.

In this paper, we show how to construct a high-quality benchmark of principally arbitrary size from parallel TeX and PDF data. We construct such a benchmark of 12,098 scientific articles from *arXiv.org* and make it publicly available. We establish a set of criteria for a clean and independent assessment of the semantic abilities of a given extraction tool. We provide an extensive evaluation of 14 state-of-the-art tools for text extraction from PDF on our benchmark according to our criteria. We include our own method, *Icecite*, which significantly outperforms all other tools, but is still not perfect. We outline the remaining steps necessary to finally make text extraction from PDF a "solved problem".

KEYWORDS

Text Extraction, PDF, Benchmark, Evaluation

ACM Reference format:
Hannah Bast and Claudius Korzen. 2017. A Benchmark and Evaluation for Text Extraction from PDF. In *Proceedings of Joint Conference On Digital Libraries, Toronto, Ontario, Canada, June 2017 (JCDL'17),* 10 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

PDF continues to be one of the most popular electronic document formats. Google alone currently indexes over 3 billion PDF documents, more than for any other document format except HTML. Unfortunately, PDF is a layout-based format: it specifies the positions and fonts of the individual characters, of which the text is composed; see Figure 1 for an example. Many applications require instead information about the semantic building blocks of the text (e.g., the words and the division into paragraphs and sections) and their semantic roles (e.g, whether a piece of text is part of the body text or of a footnote or of a caption). This semantic information is usually[1] not provided as part of the PDF.

[1]PDF documents can be *tagged* with semantic information, but such tags are rarely provided, and almost never on the level needed for typical applications.

JCDL'17, Toronto, Ontario, Canada

DOI: 10.1145/nnnnnnn.nnnnnnn

1.1 Kinds of semantic information

In the following, we briefly describe the kind of semantic information that we investigate in this paper.

Word identification. This is crucial for applications like search: a word that has not been identified correctly will not be found. Word identification in a PDF is non-trivial and challenging for a number of reasons. The spacing between letters can vary from

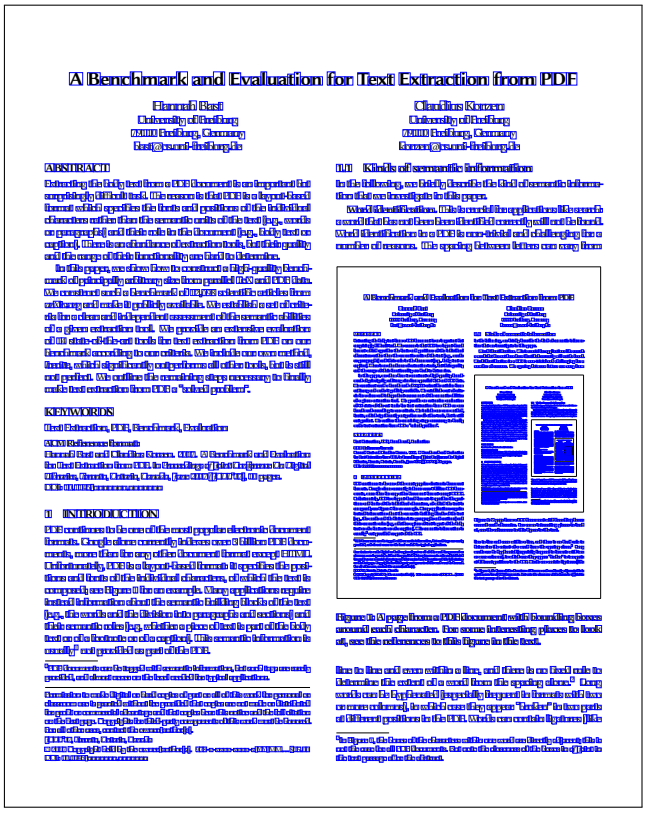

Figure 1: A page from a PDF document with bounding boxes around each character. For some interesting places to look at, see the references to this figure in the text.

line to line and even within a line, and there is no fixed rule to determine the extent of a word from the spacing alone.[2] Long words can he hyphenated (especially frequent in formats with two or more columns), in which case they appear "broken" in two parts at different positions in the PDF. Words can contain ligatures (like

[2]In Figure 1, the boxes of the characters within one word are directly adjacent; this is not the case for all PDF documents. But note the closeness of the boxes in *of Joint* in the text passage after the abstract.

ffi or *Th*, see Figure 1), which are one character in the PDF but actually translate to multiple characters in the text. Words can also contain characters with diacritics (like à or ã), which are often two characters in the PDF but translate to a single character in the text.

Word order. Determining the correct reading order of the words is crucial for *reflow* applications, where the text is cast in a different format (with different font or page sizes). Reflow is important for e-book readers or small devices, or when one simply wants or needs the text in raw text format. Word order can also be important in search, when proximity information is needed. The order of the words within a line are easy to derive from the positions of the words in the PDF. However, the order between lines is much less clear. For example, PDFs with a two-column layout of the text often contain the lines in an order interleaving between the two columns. If text is output in that order — as indeed done by simple extraction tools — it is, of course, quite unreadable.

Paragraph boundaries. Deriving the beginning and end of a paragraph is again crucial for reflow applications or when reading the text in plain text format.[3] This task is even more challenging than word identification and word order. Text from the same paragraph can be interrupted by a formula or a figure, but still belong to the same paragraph; for example, this is the case for the paragraph interrupted by Figure 1 in Figure 1. Similarly, text from the same paragraph may end at the bottom of one page or column and continue on the next page or column. But these same interruptions can also mark a real break between two paragraphs.

Semantic roles. The text elements in a PDF play different semantic roles. For the purpose of this paper, we distinguish between 16 roles, including: title, body text, formulas, figures; a complete list is given in Section 3.2. For reflow applications, it is particular important to distinguish the body text from the rest. For a targeted search application, it might also be useful to know whether a particular word occurs in the body text or in the caption of a figure.

1.2 Existing tools

A large number of tools for text extraction from PDF exist. A Google query for *text extraction from PDF* provides countless hits with tools or pages recommending tools for this task. The variety is confusing and there does not seem to be a clear winner. Most tools do not specify for which of the aspects above they are actually useful. All of the tools do word identification and consider word order (they wouldn't be of much use if they didn't). Only the more sophisticated tools provide paragraph boundaries and semantic roles.

So far, there has been no rigorous benchmark for this problem and no comprehensive evaluation of existing systems. This is surprising, given the practical importance of the problem, but it also hints at the complexity. Bringing some clarity and order into this jungle has been the main motivation behind this paper.

1.3 Contributions

This paper is about an extensive evaluation of existing PDF extraction tools, and about the non-trivial task of constructing a benchmark and developing meaningful criteria for carrying out such an evaluation. We consider the following as our main contributions.

- We describe how to construct a high-quality benchmark of principally arbitrary size from parallel TeX and PDF data (that is, for each TeX file, the PDF produced from it). The main component of this construction is a special-purpose TeX parser that can identify the logical text blocks of a document.
- Using this mechanism, we construct a benchmark of 12,098 scientific articles from *arXiv*. The articles were selected to represent a variety of topics and creation times (and thus formats) as wide as possible. The benchmark and all our code is publicly available under https://github.com/ckorzen/arxiv-benchmark.
- We establish a set of criteria that allows for a clean assessment of a given extraction tool with respect to the aspects described in Section 1.1. Establishing and measuring these criteria independently turned out to be a challenging problem; see Section 4.3.
- We provide an extensive evaluation of 14 state-of-the-art tools for text extraction from PDF on our benchmark according to our criteria. For each tool, we provide a concise description of its main mechanism and of its strengths and weaknesses. We include our own method, *Icecite*, which significantly outperforms all other systems, but is still not perfect.
- We discuss the remaining steps necessary to build a fully satisfactory tool for text extraction from PDF.

2 RELATED WORK

There are some related datasets which are used in various fields of document analysis in order to train machine learning models or to evaluate the quality of obtained results. The typical use cases are (1) dividing document pages into columns and blocks, known as *page segmentation*, (2) identifying the reading order of blocks in a page, (3) identifying the semantic roles of blocks, known as *block classification*, (4) extracting specific blocks, known as *metadata* or *information extraction*, (5) extracting metadata from reference strings, known as *reference extraction*.

We distinguish the datasets into three groups, each of them differing in the granularity of the provided data. First, *datasets with metadata only*, which usually provide data like titles, authors, abstracts or citations of a specific set of scientific articles. Second, *datasets with unstructured full texts*, which additionally provide the full texts of articles, but with no or only little semantic markup. Third, *datasets with structured full texts*, which provide the full texts enriched with semantic markups that identify text blocks with their semantic roles and their positions in the outline hierarchy.

2.1 Datasets with metadata only

The *DBLP* dataset [19] provides bibliographic metadata (title, author(s), publication year, journal, volume, etc.) of about 3.7 million computer science articles. The data are given as records in a single XML file, where each record includes the metadata of a single article. Most of the records also include an external link that points to a PDF of the related article or to a page where the PDF can be found. The data are highly accurate because the final step in the data curation pipeline is manual.

The *Cora Information Extraction* dataset [22] is split into two subsets. The first subset includes titles, authors, affiliations and authors extracted from the headers of 935 computer science articles. The second subset includes titles, authors, journals and volumes

[3] For example, a wrong paragraph break often breaks a sentence apart.

extracted from 500 citation strings. Both subsets were extracted from PDF files by several machine learning techniques.

The *UMass Citation Field Extraction Dataset* [1] consists of 1,829 manually labeled citation strings which originate from 1,200 articles from *arXiv.org*. It gives both coarse-grained labels (like authors, title, venue, date, etc.) and fine-grained labels (e.g., booktitle, address, volume, etc. of a venue) for each citation string.

The *Marmot* datasets [14] provide not only bibliographic metadata but also (1) tables extracted from 2,000 PDF pages and (2) 9,500 formulas along with their bounding boxes, characters and graphics extracted from 194 PDF files using Conditional Random Fields.

Lipinski et al. [20] compiled a dataset consisting of metadata (title, authors, abstract and publication year) of 1,153 random articles from *arXiv.org*. The dataset was used to evaluate the performance of seven PDF extraction tools, with respect to their accuracies on extracting the metadata from scientific articles.

2.2 Datasets with unstructured full texts

The *CiteSeerX* dataset [6] provides full bibliographic metadata and the full texts of approximately 6 million scientific articles, extracted from PDF files using Support Vector Machines. The data are given as XML files, where each XML file belongs to a single scientific article. They include specific markups in order to distinguish different blocks like titles, abstracts, authors, venues, references, etc. However, the full texts themselves are given as unstructured continuous texts and do not allow to distinguish between any blocks or to identify the outline hierarchy.

2.3 Datasets with structured full texts

The *Grotoap* dataset [27] consists of 113 articles taken from the Directory of Open Access Journals (DOAJ). For each article, it provides the hierarchical structure of pages, blocks, lines, words and characters in XML files. Basically, the data were extracted from PDF files using Hidden Markov Models and corrected by human experts afterwards in order to get full and reliable data. Obviously, this approach does not scale to larger collections.

The follow-up dataset, *Grotoap2* [28], consists of 13,210 articles taken from the Open Access Subset of PubMed Central [30] and provides the hierarchical structures of scientific articles in XML files as before. The data are extracted by a series of supervised and unsupervised machine learning algorithms, see [29]. Again, the extraction process is followed by a manual review step, but limited to a small random sample of articles, in order to identify common problems and to develop heuristics to correct them.

The *ACL Anthology Reference Corpus* [4] provides (1) metadata like the title, the author(s), the publication venue and the publication year, (2) the full texts, broken down into hierarchical blocks and (3) the parsed references of 22,878 articles from the *ACL Anthology*[4]. The data acquisition is split into two steps. In the first step, characters, words, lines, blocks, etc. are extracted from PDF files using an OCR software (*Nuance Omnipage*). In the second step, the extracted data are post-processed by the tool *ParsCit* (which we will evaluate in Section 4) in order to obtain semantic information.

PubMed Central [30] and *BioMed Central* [25] are the most extensive datasets in this group. They provide full bibliographic metadata, the hierarchical structures of full texts (with sections, headings and paragraphs), figures and tables. However, these data are publicly available only for a small subset of their archived articles. Usually, the data are either served by the publishers or extracted directly from the PDF files, followed by an extensive manual review process in order to correct any extraction errors. However, the details of the underlying extraction techniques are neither published nor publicly accessible. Further, the articles of an archive often originate from a well-defined set of publishers and thus exhibit a homogeneous structure which greatly facilitates the extraction process and the ability to provide extensive data.

Most of the datasets introduced in Sections 2.2 and 2.3 were derived directly from PDF files. Hence, without manual reviewing, the problems outlined in Section 1.1 are inevitably solved imperfectly and are indeed a frequent source of errors. In contrast, TeX is a markup language that provides semantic information like word boundaries, paragraph boundaries and semantic roles explicitly. Thus, TeX files (with the PDFs built from them) are much more suitable to create high-quality benchmarks. Those benchmarks are eligible for applications based not only on TeX-born PDF files, but also on all digitally-born PDFs (e.g., created by *Microsoft Word*) and even on image-based PDFs, as long as they were processed by any OCR software that identified the characters, their bounding boxes and their fonts accurately. The reason is that the listed PDF types do not show any type specific differences in the structure of their logical text blocks.

3 OUR BENCHMARK GENERATION

This section is about the generation of a PDF extraction benchmark from TeX files of scientific articles, divided into the following three steps: (1) parse TeX files syntactically in order to identify and model the hierarchies of their TeX elements, see Section 3.1; (2) identify the logical text blocks (LTBs) from TeX elements using rules, see Section 3.2; (3) serialize the LTBs to files, see Section 3.3.

3.1 Parsing TeX files

TeX[5] is a language that allows to build statements using macros. Given a TeX file, the goal of this step is to model these statements by a *syntax tree* representing the hierarchies of its TeX elements. For an illustration, see the TeX snippet given in Figure 2 (a). We want to compute the syntax tree given in Figure 2 (b), representing the hierarchies of TeX elements after the expansion of macro calls.

We proceed in three steps. First, we introduce a grammar that describes the basic syntax of the TeX language. Second, based on this grammar, we generate a parser in order to build a syntax tree that models the hierarchies of the TeX elements *before* macro calls were expanded. Third, we search the syntax tree for macro calls in order to expand them recursively.

3.1.1 The TeX grammar. In this section, we give a slightly simplified version of our grammar that describes the syntax of the basic TeX elements, in EBNF notation. In fact, the original grammar is a bit more extensive to handle (1) several special syntax cases of widely used plain TeX commands like `$`, `$$` or `\def\tex{TeX}`;

[4] http://aclweb.org/anthology/

[5] To be precise, there is a difference between *plain TeX* and *LaTeX*. However, we use the term *TeX* in a generic sense for both types.

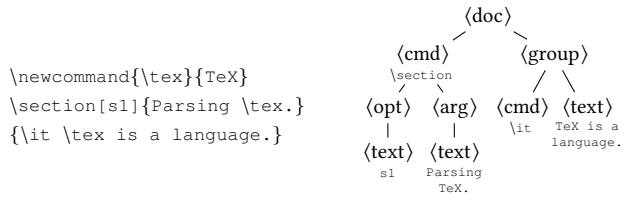

Figure 2: (a) A simple TeX snippet with typical TeX elements. Boilerplate commands like \documentclass{...} or \begin{document} were omitted for reasons of brevity. (b) The syntax tree that represents the hierarchies of the elements in (a), without the macro definition in the first line.

(2) any number of whitespaces and newlines within the elements or (3) the *starred variants* of commands like `\section*{...}` or `\begin{figure*}`. However, the following grammar does not lack any significant features and is detailed enough to illustrate the most important aspects of the TeX language:

⟨*doc*⟩ ::= (⟨*element*⟩)* (1)
⟨*element*⟩ ::= ⟨*group*⟩ | ⟨*cmd*⟩ | ⟨*marker*⟩ | ⟨*comment*⟩ | ⟨*text*⟩ (2)
⟨*group*⟩ ::= '{' (⟨*element*⟩)* '}' (3)
⟨*cmd*⟩ ::= ⟨*break-cmd*⟩ | ⟨*ctrl-cmd*⟩ | ⟨*symb-cmd*⟩ (4)
⟨*break-cmd*⟩ ::= ('\n' | '\r\n')+ (5)
⟨*ctrl-cmd*⟩ ::= '\' (⟨*letter*⟩)+ (⟨*arg*⟩ | ⟨*opt*⟩)* (6)
⟨*symb-cmd*⟩ ::= '\' ⟨*non-letter*⟩ [⟨*arg*⟩ | ⟨*opt*⟩ | ⟨*letter*⟩] (7)
⟨*arg*⟩ ::= '{' (⟨*element*⟩)* '}' (8)
⟨*opt*⟩ ::= '[' (⟨*element*⟩)* ']' (9)
⟨*marker*⟩ ::= '#' ⟨*digit*⟩ (10)
⟨*comment*⟩ ::= '%' (⟨*element*⟩)* ⟨*break-cmd*⟩ (11)
⟨*text*⟩ ::= (⟨*char*⟩ | ⟨*whitespace*⟩)+ (12)
⟨*char*⟩ ::= ⟨*letter*⟩ | ⟨*digit*⟩ | ⟨*non-letter*⟩ (13)
⟨*whitespace*⟩ ::= ' ' | '\t' (14)
⟨*letter*⟩ ::= ['A'-'Z', 'a'-'z'] (15)
⟨*digit*⟩ ::= ['0'-'9'] (16)
⟨*non-letter*⟩ ::= [^'A'-'Z', 'a'-'z', '0'-'9'] (17)

The grammar consists of 17 production rules, where the non-terminal ⟨*doc*⟩ is the start symbol and may expand to any number of TeX elements, see rule (1). A TeX element is either given by a *group*, a *command*, a *marker*, a *comment* or a *text phrase*, see rule (2).

On commands, we distinguish between *break commands*, *control commands*, and *symbol commands*, see rule (4). A break command describes any kind of a line break. A control command describes a command that follows the "regular" command syntax with potential argument groups and option groups, like `\today`, `\section{...}` or `\begin{table}[h]`. A symbol command describes a command that is mainly but not exclusively used to encode a special character, like `\#`, `\\[10pt]`, `\"a` or `\"{a}`.

Further, a marker is a placeholder for an argument group in a macro definition and a comment is a piece of text which we will exclude from further processing.

3.1.2 The generation of the TeX parser. Given the grammar introduced above, the next step is to generate a parser that builds the syntax tree. We use *JavaCC*, a parser generator that creates *LL*(k) parsers from given *LL*(k) grammars. In general, an *LL*(k) parser is a top-down parser that reads input sequences from left to right in order to find *leftmost derivations* in a grammar, starting at the start symbol. At any time, an *LL*(k) parser looks at k *lookahead symbols* in the input sequence to decide which production rule to apply, where k is as large as a production rule can be chosen unambiguously. In our case, $k = 2$, because for the sequence '\' (a backslash) of length 1 the parser needs to look at one more symbol to decide which kind of command (⟨*break-cmd*⟩, ⟨*ctrl-cmd*⟩ or ⟨*symb-cmd*⟩) is denoted by the sequence.

JavaCC allows to associate each production rule with so-called *parser actions*, which are in fact Java code snippets that are executed when the production rule was derived. They consume series of *tokens*, which can be seen as associations between substrings in the input sequence and the production rules. We use this mechanism to construct the syntax tree and a *macro dictionary*.

In principle, the constructed syntax tree reflects the hierarchy given by the grammar introduced above. It is a rooted and ordered tree, where each node correlates to one of the following production rules: ⟨*doc*⟩, ⟨*group*⟩, ⟨*cmd*⟩, ⟨*arg*⟩, ⟨*opt*⟩, ⟨*marker*⟩ or ⟨*text*⟩. The DFS order of nodes correlates to the order of the related elements in the TeX file. The ⟨*doc*⟩, ⟨*group*⟩, ⟨*arg*⟩ and ⟨*opt*⟩ nodes may have any number of child nodes representing the enclosed elements. ⟨*arg*⟩ and ⟨*opt*⟩ nodes exist only beneath ⟨*cmd*⟩ nodes; ⟨*text*⟩ and ⟨*marker*⟩ nodes do not have any child nodes.

The macro dictionary is a dictionary that holds all macro definitions. Whenever we identify a macro definition (like `\newcommand{\tex}{TeX}`), we insert it with the macro name (`\tex`) as the key and the syntax tree that represents the replacement (`{TeX}`), called *replacement tree*, as the value.

3.1.3 The expansion of macro calls. Given the syntax tree and the macro dictionary, the last step of the parsing process is to expand the macro calls in the syntax tree recursively. We traverse the syntax tree in DFS order to identify macro calls by looking up the name of each command in the macro dictionary. If a macro call was found, each marker in the associated replacement tree is replaced by the related argument group of the macro call. Afterwards, the subtree in the syntax tree representing the macro call is replaced by the resulting replacement tree. This process is done in a recursive fashion in order to identify and expand nested macro calls.

3.2 Identifying logical text blocks

Given a syntax tree with expanded macro calls, the next step is to identify the LTBs with one of the following 16 semantic roles: title, author, affiliation, date, abstract, heading, paragraph of the body text, formula, figure, table, caption, listing-item, footnote, acknowledgements, references and appendix.

Our procedure is rule-based and is sketched in the algorithm below. For the sake of brevity, a Python-like syntax is used. However, the original code is written in Java. The procedure accepts a syntax tree and a dictionary of rules, where each rule defines features for a specific TeX command that give details about how to handle the command on identifying the LTBs. The output is a list of LTBs, where each LTB has the attributes *level* (an integer representing its level in the outline hierarchy, which defaults to 0), *text* (its textual

Algorithm: The procedure of identifying LTBs using rules

Input:

`tree` A syntax tree.
`rules` A dictionary of rules.

Output:

List of LTBs (the identified logical text blocks).

```
1  def identify_blocks(tree, rules):
2    level = 0                # The hierarchy level.
3    itr = dfs_iterator(tree)          # DFS order.
4    stack = [LTB(level=level)]  # The active LTBs.
5    finished = []             # The finished LTBs.
6    for element in itr:
7      if type(element) is Text:
8        stack[-1].text += element.text
9      if type(element) is Command:
10       rule = rules.get(element)
11       if rule is None:
12         element.args = []  # Do not visit args.
13         element.opts = []  # Do not visit opts.
14         continue
15       if rule.hierarchy_level > 0:
16         level = rule.hierarchy_level
17       if rule.starts_ltb > 1:
18         finished.append(stack.pop())
19       if rule.starts_ltb > 0:
20         stack.push(LTB(level=level))
21       if rule.semantic_role is not None:
22         stack[-1].role = rule.semantic_role
23       if rule.text_phrase is not None:
24         stack[-1].text += rule.text_phrase
25         if rule.end_command is not None:
26           itr.skip_to(rule.end_command)
27       for i in len(element.args):
28         if i not in rule.args_to_visit:
29           element.args[i] = None
30       if rule.ends_ltb:
31         finished.append(stack.pop())
32   # Remove remaining blocks from stack.
33   while len(stack) > 0:
34     finished.append(stack.pop())
35   return finished
```

content, which defaults to the empty string) and *role* (its semantic role, which defaults to "body text").

The basic idea is to traverse the syntax tree in DFS order (see line 3) and to have a stack of *active* LTBs, initialized with a single, empty LTB (see line 4) and a list of *finished* LTBs (see line 5). On visiting a node, one or more of the following actions may be triggered, depending on the type of the related TeX element:

(A1) Push a new LTB to the stack.
(A2) Append a text phrase to the topmost LTB in the stack.
(A3) Set the semantic role of the topmost LTB in the stack.
(A4) Set the hierarchy level for LTBs to be created subsequently.
(A5) Pop the topmost LTB and add it to list of finished blocks.
(A6) Skip to a given node in the syntax tree.

In case of a text, action (A2) is triggered, see line 8. In case of a command, the triggered action(s) depend on the related rule, see lines 9-31. Details about the rules are given in Section 3.2.1. If there is no such rule for a command, the complete subtree defined by the command is ignored (the arguments and options are removed, such that they are not visited by the iterator, see lines 11-14). In case of a group, option or argument, no special action is triggered and the algorithm continues with the next node in DFS order. Once the traversal of the tree is completed, all remaining LTBs in the stack are popped and are added to the list of finished LTBs, see lines 33-34. Finally, the list of finished LTBs is returned, see line 35.

3.2.1 The rules. The rules are given as a dictionary of `Rule` objects, where each `Rule` gives the following seven features for a referred command:

Hierarchy level (`hierarchy_level`): A digit between 1 and 5. It denotes the level of the section in the outline hierarchy, in case of the command defines a section heading. A higher value means a deeper level. Triggers action (A4) if a value is given, see lines 15-16.

Starts new LTB? (`starts_ltb`): A digit; either 1 or 2, where 1 means: The command introduces a new LTB (and (A1) is triggered, see lines 19-20); 2 means: The command ends the LTB and introduces a new one (and (A5) and (A1) are triggered, see lines 17-20).

Semantic role (`semantic_role`): The semantic role that is induced by the command. If set, action (A3) is triggered, see lines 21-22.

Text phrase (`text_phrase`): A text phrase to append to the current LTB. It is used (1) to define the text phrase that is in fact encoded by the command (e.g., a special character); or (2) to define a placeholder for an LTB for which (a) it is unclear from the TeX file how it is visualized in the PDF file (like a citation produced by e.g. the command `\cite{...}`) or (b) there are no standardized ways to serialize it to plain text properly, which is the case for tables, figures and formulas. Placeholders are ignored in the evaluation, see Section 4.3.2 for details. If set, (A2) is triggered (lines 23-24).

End command (`end_command`): The command that denotes the end of the TeX environment (e.g., `\end{table}`), in case of the command introduces one (e.g., `\begin{table}`). This property is needed to skip to the end of the environment, in case of the environment should be replaced by a text phrase, see lines 25-26.

Arguments to visit (`args_to_visit`): List of indices of argument groups to examine. This feature is used to decide whether an argument of a command is relevant to the identification of LTBs or not. For example, the argument `{Introduction}` in the command `\section{Introduction}` is relevant, because it contains textual content of an LTB. In contrast, the argument `{5pt}` in the command `\vspace{5pt}` is not relevant, as it does not affect any properties of an LTB. All arguments, which are not covered by this list, are ignored (are cleared, see lines 27-29).

Ends current LTB? (`ends_ltb`): A boolean that indicates whether the command ends the current LTB. Triggers action (A5) if the value is set to true, see lines 30-31.

Overall, our dictionary contains about 1200 rules. Figure 3 gives an excerpt with the values of four concrete `Rule` objects. The complete rules are given at https://github.com/ckorzen/arxiv-benchmark.

3.3 Serializing logical text blocks

Given the list of identified LTBs, the last step is to serialize them to files, optionally filtered by given semantic roles. Our benchmark

```
rules["\section"] = Rule (
    starts_ltb = 2,
    semantic_role = "heading",
    hierarchy_level = 1,
    ends_ltb = true,
    args_to_visit = [0],
)
```

```
rules["\footnote"] = Rule (
    starts_ltb = 1,
    semantic_role = "footnote",
    ends_ltb = true,
    args_to_visit = [0],
)
```

```
rules["\n\n"] = Rule (
    starts_ltb = 2,
    ends_ltb = true,
)
```

```
rules["\%"] = Rule (
    text = %,
)
```

Figure 3: The initialization and indexing of four concrete `Rule` objects for the commands \section, \footnote, \n\n and \%. Each rule is indexed by the name of the referred command and gives features on how to handle the command. For example, the feature `starts_ltb` in the rule for command \n\n is 2 (denoting that the command ends the previous LTB and starts a new one), because in TeX files, paragraphs are separated by blank lines and we want to identify each paragraph as a single LTB. For more details about the meaning of the individual features, see Section 3.2.1.

generator provides the following output formats: plain text, XML and JSON. In case of plain text, the textual contents of the selected LTBs are joined in a flat way, separated by blank lines and keeping their order in the TeX file. In case of XML or JSON, the texts of the LTBs are enriched with descriptive markups, giving their semantic roles and reflecting their order in the TeX file and their outline hierarchies.

3.4 Common pitfalls

In this section, we describe two TeX-specific pitfalls, which can lead to a faulty ground truth if not considered appropriately.

First, there may be some LTBs, which are present in the PDF file but not directly deducible from the TeX file – either because (1) they are not defined in the TeX file but in some supplementary `sty`- or `cls`-files of included packages or (2) they are only defined at compile time, e.g. because of conditional macros consisting of `\if` and `\else` commands. Related examples are page headers, page footers, page numbers or section numberings. All of them won't be extracted by our benchmark generator.

Second, authors occasionally misuse or ignore convenient TeX commands. A common example is the "hard coding" of section headers (e.g., the use of `{\large \bf Introduction}` instead of `\section{Introduction}`) or citations (e.g., '`[2]`' instead of `\cite{foo}`). Our rule-based approach is not flexible enough to handle those cases. It means that, for example, sections like references or appendices may be identified as part of the body text mistakenly.

3.5 Usage

As seen in Section 3.3, our benchmark generator provides built-in options in order to produce various kinds of benchmarks, with individual compositions of LTBs and various output formats. Thus, it is applicable to a wide variety of other applications or evaluations related to document analysis and metadata extraction. The code of our benchmark generator is publicly available and can be found under the link given above. There you will find detailed instructions and examples on how to use and how to customize the generator to personal needs.

4 EVALUATION OF CURRENT TOOLS

In this section, we evaluate and compare 14 state-of-the-art tools for text extraction from PDF files. In Section 4.1, we introduce the evaluated tools, each with a concise description of its main mechanism, strengths and weaknesses. In Section 4.2, we describe our benchmark, which was constructed using the method described in Section 3. In Section 4.3, we describe our evaluation methods, in particular, the criteria we use to assess and compare the semantic abilities of the tools. Section 4.4 provides the evaluation results.

4.1 The PDF extraction tools

We have evaluated the following 14 tools. An overview and comparison of their feature sets is given in Table 1.

pdftotext [12] is probably the most familiar PDF extraction tool. It converts any PDF files to plain text files rapidly, but does not make any effort to identify paragraph boundaries or semantic roles or only the body text.

pdftohtml [18] converts a given PDF file to XML or HTML, broken down into text lines. It does not identify paragraphs or semantic roles, extracts characters with diacritics as two characters and does not merge hyphenated words.

pdftoxml [11] converts a given PDF file to XML, broken down into "blocks" (which do not correlate to paragraphs), text lines and words. Ligatures, diacritics and hyphenated words are not handled.

PdfBox [2] is a widespread PDF library by Apache that is able to convert a given PDF file to plain text. It does not identify paragraph boundaries or semantic roles, but handles ligatures and characters with diacritics. Hyphenated words are not merged.

pdf2xml [26] uses *Apache Tika* (which uses *PdfBox* under the hood) and *pdftotext* to extract text from a given PDF file. In a postprocessing step, the tool combines the result of both tools in order to improve the identification of word boundaries.

ParsCit [15] does not actually extract text from a PDF file but processes the results of third-party tools (like *pdftotext*) to extract the body text and parse reference strings. Its abilities therefore depend on the utilized third-party tool. In our evaluation, we use *pdftotext*.

LA-PdfText [5] is a tool that focuses on PDF files of scientific articles and extracts LTBs based on (user-defined) rules, which must be defined for each different article layout [23]. However, there are some default rules, which we use in the evaluation.

PdfMiner [24] is a tool that is able to analyze the structure of a given PDF file and converts it to plain text, XML or HTML, broken down into paragraphs, lines and characters. Ligatures, characters with diacritics and hyphenated words are not handled properly.

System	PA	OR	RO	LI	DI	HY	FORMAT
pdftotext [12]	–	✓[1]	–	✓	✓	✓	txt
pdftohtml [18]	–	✓	–	✓	–	–	xml, html
pdftoxml [11]	–	✓	–	–	–	–	xml
PdfBox [2]	–	✓	–	✓	✓	–	txt
pdf2xml [26]	✓	✓[1]	–	✓	–	✓	xml, html
ParsCit [15]	–[2]	–[2]	✓	–[2]	–[2]	–[2]	xml
LA-PdfText [5]	–	✓[1]	✓[3]	✓	–	–	txt
PdfMiner [24]	✓	✓[1]	–	–	–	–	txt, xml, html
pdfXtk [13]	–	✓	–	✓	–	–	xml, html
pdf-extract [31]	–	✓[1]	–	✓	–	–	xml
pdfx [7]	–	✓	✓	✓	✓	✓	xml
PDFExtract [3]	✓	✓	✓	✓	✓	✓	xml
Grobid [21]	–	✓	✓	✓	✓	✓	xml
Icecite [17]	✓	✓	✓	✓	✓	✓	txt, xml, json

Table 1: Overview of the features of 14 PDF extraction tools, broken down into: PA: identification of paragraph boundaries; OR: identification of the reading order; RO: identification of semantic roles; LI: translation of ligatures; DI: extraction of characters with diacritics as single characters; HY: merging of hyphenated words. If a feature is fully provided by a tool, it is denoted by a "✓". A number next to an entry points to one of the following constraints: (1) lines from different text columns are mixed sometimes; (2) depends on the used 3rd-party tool; (3) depends on the used rules. The last column FORMAT gives the available output format(s).

pdfXtk [13] is built upon *PdfBox* and converts a given PDF file to XML or HTML, broken down into "blocks" (which do not correlate to paragraphs), lines, words and characters. Characters with diacritics and hyphenated words are not handled properly.

pdf-extract [31] converts PDF files to XML, broken down into "regions" (which do not correlate to paragraphs) and text lines. Its only semantic ability is to distinguish reference sections from non-reference sections and to split them into individual references.

pdfx [7] is a rule-based tool that analyzes fonts and layout specifics in order to construct a geometrical model of a PDF file and to identify the title, sections, tables, etc. from it [8]. The sections are broken down into "regions", which do not correlate to paragraphs.

PDFExtract [3] is one of the most powerful tools. It converts PDFs of scientific articles to XML and is able to identify the semantic roles *title*, *abstract*, *headings* and *paragraphs*. It handles ligatures, characters with diacritics, and hyphenated words.

Grobid [21] is another powerful tool that breaks down PDFs into several LTBs, like title, abstract, sections (but not paragraphs), etc. using Conditional Random Fields. Further it is able to handle ligatures, characters with diacritics, and hyphenated words.

Icecite [17] is our own tool, which extracts LTBs from scientific articles, with a focus on paragraphs of the body text. In principle, it is based on a rule-based approach that analyzes the distances, positions and fonts of characters, words and text lines. Another focus is the precise extraction of words, including an accurate handling of ligatures, diacritics and hyphenated words.

There are some other related tools, which were not included in the evaluation, because (1) they are commercial tools (like *JPedal*[6] or *PDFlib TET*[7]); (2) their methods and feature sets are very similar to an already included tool (e.g. *iText*[8], which is similar to *PdfBox*); or (3) they are described in a scientific article, but there are no executables provided [10] or they are not available anymore [16].

4.2 The Benchmark

Our benchmark consists of 12,098 scientific articles, taken from *arXiv.org* [9], a digital library that hosts about 1.2 million scientific articles (on topics like physics, mathematics, computer science, biology, finance and statistics), indexed by month, beginning from August 1991. For most of them, *arXiv* provides both, a PDF file and the related TeX source file(s).

From each month, we selected 1% of the articles randomly, resulting in 12,098 articles. This sample yields a good variety of topics, creation times and thus formats of the articles from *arXiv*[9]. We also tried larger sample sizes, but experienced only minimal variances in our evaluation results (± 0.5%).

For each article, the benchmark contains a ground truth file and the related PDF file. Each ground truth file was generated via the benchmark generator described in Section 3 and contains the *title*, the *section headings* and the *body text paragraphs* of a particular article in plain text format. The PDF files we use are not those provided by *arXiv*, due to occasional (contentual) mismatches with the corresponding TeX files, but we regenerated them from the provided TeX files.

4.3 Evaluation methods

For each tool, the PDF files of the benchmark were processed in batches. We have chosen reasonable input parameters in order to get output files that reflect, as much as possible, the structure of the ground truth files. The exact parameters for each tool can be found under https://github.com/ckorzen/arxiv-benchmark.

For the tools with XML output, we translated the output to plain text by identifying the relevant text parts. If semantic roles were provided, we only selected those parts that are also present in the ground truth files. If texts were broken down into any kind of blocks (like paragraphs, columns, or sections), we have separated them by blank lines (like in the ground truth files).

The main purpose of the evaluation was to assess each tool by comparing its output files with the ground truth files using a set of easily interpretable and independent criteria. This was harder than expected, especially the "independent" part. In the following, we first define our evaluation criteria and then explain how we compute them (which turned out be non-trivial).

4.3.1 Establishing the evaluation criteria. We are looking for easily interpretable and independent criteria that assess the quality of an output file with respect to four aspects: (1) paragraph boundaries, (2) distinction of body text and non-body text, (3) reading

[6] http://www.idrsolutions.com/jpedal
[7] http://www.pdflib.com/products/tet
[8] http://www.itextpdf.com/
[9] https://arxiv.org/help/stats/2016_by_area/index/

order, and (4) word boundaries. Independence here means that it should be possible, in principle, to perform well for any subset of criteria but poorly for the others. We eventually came up with three groups of criteria, that measure the *differences* between an output file and the related ground truth file.

- **Newline differences** capture the quality of the detection of paragraph boundaries and are broken down into:
  - NL^+: the number of *spurious* newlines in the output file.
  - NL^-: the number of *missing* newlines in the output file.
- **Paragraph differences** capture the quality of the distinction between body and non-body text and of the reading order. They are broken down into:
  - P^+: the number of *spurious* paragraphs in the output file.
  - P^-: the number of *missing* paragraphs in the output file.
  - $P^{\uparrow\downarrow}$: the number of *rearranged* paragraphs in the output file.
- **Word differences** capture the quality of the recognition of individual words and their boundaries and are broken down into:
  - W^+: the number of *spurious* words in the output file.
  - W^-: the number of *missing* words in the output file.
  - $W^{\sim}$: the number of *misspelled* words in the output file.

These criteria are indeed easily interpretable and independent. For example, a tool can perform well with respect to $W^{\sim}$, if it handles ligatures and hyphenated words properly; but poorly with respect to NL^+ and NL^-, if it does not identify any paragraph boundaries.

4.3.2 Measuring the evaluation criteria. The evaluation criteria introduced above are easily interpretable, but measuring them is non-trivial. In particular, for a given output file O and ground truth file G, there are multiple ways to assign values to these criteria. An example for this is given in Figures 4 and 5. We address this problem by computing an assignment that minimizes

$$Z = (NL^+ + NL^-) + (W^+ + W^- + W^{\sim}) + c \cdot (P^+ + P^- + P^{\uparrow\downarrow}),$$

where $c \geq 1$ is a (constant) penalty score, introduced to increase the weight for paragraph differences compared to newline- and word differences. In the evaluation, we use $c = 5$.

In the following, we describe our heuristic algorithm *doc-diff*, that finds, in most cases, an optimal assignment to the evaluation criteria with minimal Z. Let w_O resp. w_G be the list of words per paragraph in O resp. G, transformed to lower case and without any punctuation marks. For Figure 4, w_O is given by [[*text*, *extraction*, *pdf*], [*a*, *benchmark*, *and*], [*evaluation*, *for*]] and w_G is given by [[*a*, *benchmark*, *and*, *evaluation*, *for*, *text*, *extraction*, *from*, *pdf*]], where each list at index i contains the words of paragraph i.

The approach of *doc-diff* is to compare w_O and w_G wordwise and to classify the differences into the following type of *phrases*:

- **Common phrase** (= [word$_1$, ..., word$_i$]): a sequence of i consecutive words which are common to w_O and w_G.
- **Differing phrase** (~ [word$_1$, ..., word$_j$], [word$_1$, ..., word$_k$]): a sequence of j *spurious* words, which occur in w_O but not in w_G; and of k *missing* words, which occur in w_G but not in w_O.
- **Rearranged phrase** ($\uparrow\downarrow$ [word$_1$, ..., word$_m$], [word$_1$, ..., word$_n$]): a sequence of m words in w_O and n words in w_G, which are (almost) equal ($m \approx n$), but their positions in w_O and w_G do not correlate.

The phrases are computed in two rounds. In the first round, the common and differing phrases are computed by an algorithm called *word-diff*, which works similar to the Unix *diff* command, but based on words instead of lines. The phrases are computed per paragraph and know the related paragraph numbers in w_O and w_G. For the example above, *word-diff* computes the phrases p_1: (~ [*text*, *extraction*, *pdf*], []); p_2: (= [*a*, *benchmark*, *and*]); p_3: (= [*evaluation*, *for*]) and p_4: (~ [], [*text*, *extraction*, *from*, *pdf*]).

output file O	ground truth file G
Text Extraction PDF. <BLANKLINE> *A Benchmark and* <BLANKLINE> *Evaluation for*	*A Benchmark and Evaluation for Text Extraction from PDF.*

Figure 4: An excerpt of an output file O with three paragraphs and the related ground truth file G with a single paragraph.

Assignment 1: P^+: 3, P^-: 1	Assignment 2: P^+: 1, NL^+: 1, W^- 4	Assignment 3: $P^{\uparrow\downarrow}$: 1, NL^+: 2, W^-: 1
Text Extraction PDF. <BLANKLINE> *A Benchmark and* <BLANKLINE> *Evaluation for* *A Benchmark and Evaluation for Text Extraction from PDF.*	*Text Extraction PDF.* <BLANKLINE> *A Benchmark and* <BLANKLINE> *Evaluation for Text Extraction from PDF.*	<BLANKLINE> *A Benchmark and* <BLANKLINE> *Evaluation for* <BLANKLINE> *Text Extraction from PDF.*

Figure 5: Three different assignments to the evaluation criteria from Section 4.3.1 in order to assess O against G from Figure 4, with related visualizations.

In the second round, the rearranged phrases are computed by an algorithm called *rearr-diff*, which is a local alignment algorithm and works similar to the Smith-Waterman algorithm, but based on words instead of characters. In principle, *rearr-diff* looks at the differing phrases, identifies *similar word regions* between spurious and missing words, wraps them into rearranged phrases and associates the rearranged phrases with the related differing phrases. For the phrases p_1,..., p_4 in the example above, *rearr-diff* identifies a similar word region between the spurious words of phrase p_1 and the missing words of phrase p_4 and creates the rearranged phrase p_5: ($\uparrow\downarrow$ [*text*, *extraction*, *pdf*], [*text*, *extraction*, *from*, *pdf*]). Initially, all computed rearranged phrases are seen as preliminary phrases and could be refused while assigning values to the evaluation criteria, see below.

Given the phrases, the next step is to assign concrete values to the evaluation criteria. *Doc-diff* proceeds again in two rounds, in which each phrase p_i is seen as a standalone unit with individual evaluation criteria W_i^+, W_i^-, $W_i^{\sim}$, P_i^+, etc. (called *phrase criteria*) and an individual score Z_i that scores the phrase criteria equivalently to Z.

In the first round, *doc-diff* examines the rearranged and differing phrases in order to assign the values for word- and paragraph differences. For each phrase p_i, *doc-diff* simulates various type-dependent evaluation scenarios, where each scenario S_j is again given by individual evaluation criteria $W_{S_j}^+$, $W_{S_j}^-$, $W_{S_j}^{\sim}$, $P_{S_j}^+$, etc.

(called *scenario criteria*) and a score Z_{S_j} that scores the scenario criteria equivalently to the score Z. In case p_i is a rearranged phrase, the scenarios are:

S_1: $\boldsymbol{P}^{\uparrow\downarrow} = 1$; plus the differences resulting from *doc-diff*$(w_O^{p_i}, w_G^{p_i})$
S_2: $\boldsymbol{P}^{+} = 1$ (if $m > 0$); $\boldsymbol{P}^{-} = 1$ (if $n > 0$).
S_3: $\boldsymbol{W}^{\sim} = min(m, n)$; $\boldsymbol{W}^{+} = m - min(m, n)$; $\boldsymbol{W}^{-} = n - min(m, n)$

where $w_O^{p_i}$ resp. $w_G^{p_i}$ is the list of related words in p_i from w_O resp. w_G, $m = |w_O^{p_i}|$ and $n = |w_G^{p_i}|$. To clarify, scenario S_1 defines the evaluation criteria that would result when p_i would indeed be rearranged, S_2 the criteria that would result when p_i would be assessed only by paragraph differences, and S_3 the criteria that would result when p_i would be assessed only by word differences. If S_1 is the scenario with the minimal score, p_i will be accepted as rearranged phrase and the related scenario criteria will be added to the phrase criteria of p_i. Otherwise, p_i is refused. For example, for phrase p_5, the scenario criteria of S_1 are: $P^{\uparrow\downarrow} = 1$, $W^{-} = 1$; and of S_2: $P^{+} = 1$, $P^{-} = 1$ and of S_3: $W^{\sim} = 3$, $W^{+} = 0$, $W^{-} = 1$. The related evaluation scores are given by $Z_{S_1} = c + 1$; $Z_{S_2} = 2c$ and $Z_{S_3} = 4$. Thus, p_5 is accepted as a rearranged phrase only if $c \leq 3$. Otherwise, p_5 is refused.

In case p_i is a differing phrase, the simulated scenarios are S_2 and S_3, where m resp. n is given by the number of spurious resp. missing words in p_i which are not a member of an accepted rearranged phrase. There is a special scenario S_4, where none of the evaluation criteria are affected, if the spurious words consist of at least one placeholder (see Section 3.2.1 for details about the concept of placeholders). The criteria of the scenario with the minimal score Z_{S_i} are added to the phrase criteria of p_i. In case of a tie, the criteria of the scenario which comes first in the introduced order are chosen. For p_1 in the example above, the scenario criteria depend on whether p_5 is accepted or not. If p_5 is accepted, there are no scenario criteria given, because $m = 0$ and $n = 0$. If p_5 is refused, $m = 3$ and $n = 0$ and the criteria of S_2 are: $P^{+} = 1$, $P^{-} = 0$ and of S_3: $W^{\sim} = 0$; $W^{+} = 3$, $W^{-} = 0$. The related scenario scores are given by $Z_{S_2} = c$ and $Z_{S_3} = 3$, meaning that the scenario criteria of S_2 are added to the phrase criteria if $c \leq 3$ and of S_3 otherwise.

In the second round, *doc-diff* iterates over the phrases in order to assign the values for the newline differences. For each phrase p_i, *doc-diff* analyzes the paragraph numbers of p_i and p_{i-1} in order to identify paragraph breaks in w_O and w_G. If there is a paragraph break in w_O but not in w_G, an NL^{+} is added to the phrase criteria of p_i. Analogously, if there is a paragraph break in w_G but not in w_O, an NL^{-} is added. For the example above, a NL^{+} is added to the phrase criteria of p_3, because there is a paragraph break between p_2 and p_3 in w_O, but not in w_G.

At the end, the final assignment results from the union of all computed phrase criteria. For example, if $c \leq 3$, the final assignment would be: $P^{\uparrow\downarrow} = 1$, $NL^{+} = 2$; $W^{-} = 1$ (which corresponds to assignment 3 in Figure 5) and $W^{+} = 3$, $W^{-} = 4$, $NL^{+} = 2$ if $c > 3$.

4.4 Evaluation results

Table 2 gives an overview of the evaluation results for each of the evaluated PDF extraction tools, broken down by the evaluation criteria computed by the *doc-diff* algorithm explained above.

For most of the tools, either NL^{+}, NL^{-} or both are pretty high. Low values in both criteria are only achieved by those tools, which indeed identify paragraph boundaries, in particular *Icecite*. The comparatively large NL^{-} value for *PDFExtract* is caused by the fact that it does not consider isolated formulas as single paragraphs. *PdfMiner* has problems with identifying the correct paragraph boundaries if paragraphs were split by page breaks, column breaks or LTBs like figures, tables or captions.

The same is true for the criteria P^{+} and P^{-}: low values in both criteria are only achieved by the more sophisticated tools, which are able to identify the semantic roles of LTBs (like *Parscit*, *pdfx*, *PDFExtract*, *Grobid* and *Icecite*). In particular, tools like *pdftotext* and *PdfBox* show low P^{-} values, but high P^{+} values, because they extract full texts without considering semantic roles. The large P^{-} value of *LA-PdfText* is due to the fact that we used the default rules (see Section 4.1), which resulted in a lot of missing LTBs.

In principle, all tools are able to identify the correct reading order of words. However, some tools have problems with two-column articles, as illustrated by the large values in the $P^{\uparrow\downarrow}$ criteria for *pdf2xml* and *pdf-extract*.

In the criteria W^{+} and W^{-}, *pdf-extract* has problems with the correct extraction of subscripts and superscripts. In many cases, the tool extracted them as separate text lines, as they did not share the same baseline with the belonging text line. Finally, the value for $W^{\sim}$ is large for those tools, which do not translate ligatures into multiple characters and/or do not extract characters with diacritics as single characters and/or merge hyphenated words (like *pdftoxml*, *PdfMiner* or *pdf-extract*).

Only *Icecite* yields satisfactory results in all criteria (close to the optimum among the evaluated tools). However, *Icecite* is work in progress and not perfect yet either:

- Our rule-based approach on identifying LTBs, which is not flexible enough to handle each single anomaly in the structures of scientific articles properly.
- Characters (in particular ligatures and special characters) which are printed in so called *Type-3 fonts*, where the characters are in fact not of textual nature but are *drawn* into the PDF and therefore are not identifiable as text.
- Compound words with mandatory hyphens (like *sugar-free*) which seem to be hyphenated words because they are split at the mandatory hyphen across two text lines. In most cases, Icecite handles them as *normal* hyphenated words and removes the hyphen mistakenly (merges *sugar-free* to *sugarfree*).

The second and third issues are well known problems, which were also observed in most other tools. In particular, the second is a general issue of PDF, which needs more sophisticated methods to solve (OCR-based or learning-based).

5 CONCLUSION

We have presented an evaluation on the semantic abilities of 14 PDF extraction tools, based on a high-quality benchmark, which we have constructed from parallel TeX and PDF data. We found that our own PDF extraction tool, *Icecite*, significantly outperforms other tools with respect to (1) paragraph boundaries, (2) body text paragraphs, (3) reading order, and (4) word boundaries. However, it is still not perfect due to the limits of its rule-based approach. We are confident that a learning-based approach can fix the open problems.

System	Features	NL^+	NL^-	P^+	P^-	$P^{\uparrow\downarrow}$	W^+	W^-	$W^\sim$	ERR	T∅
pdftotext [12]	– O – L D H	14 (16%)	44 (53%)	60 (29%)	2.3 (0.6%)	1.4 (1.9%)	24 (0.7%)	2.4 (0.1%)	41 (1.2%)	2	**0.3**
pdftohtml [18]	– O – L – –	3.6 (4.3%)	70 (84%)	9.2 (31%)	4.2 (3.2%)	0.1 (0.1%)	16 (0.5%)	1.6 (0.0%)	95 (2.9%)	**0**	2.2
pdftoxml [11]	– O – – – –	33 (40%)	20 (25%)	80 (31%)	1.8 (0.5%)	0.1 (0.1%)	21 (0.6%)	1.6 (0.0%)	154 (4.7%)	1	0.7
PdfBox [2]	– O – L D –	**3.0 (3.6%)**	70 (85%)	7.6 (27%)	**0.9 (0.2%)**	**0.0 (0.1%)**	17 (0.5%)	**1.5 (0.0%)**	53 (1.6%)	2	8.8
pdf2xml [26]	P O – L – H	33 (40%)	39 (48%)	44 (21%)	40 (30%)	7.8 (9.5%)	8.6 (0.3%)	3.6 (0.1%)	34 (0.9%)	1444	37
ParsCit [15]	– – R – – –	15 (18%)	39 (47%)	10 (10%)	14 (6.4%)	1.3 (1.8%)	16 (0.5%)	2.3 (0.1%)	37 (1.1%)	1	6.8
LA-PdfText [5]	– O R L – –	5.5 (6.4%)	23 (28%)	**4.8 (3.1%)**	52 (73%)	2.9 (5.9%)	**5.7 (0.1%)**	6.1 (0.1%)	26 (0.6%)	324	24
PdfMiner [24]	P O – – – –	32 (38%)	18 (21%)	84 (30%)	3.6 (1.0%)	1.4 (2.1%)	34 (1.0%)	2.6 (0.1%)	110 (3.3%)	23	16
pdfXtk [13]	– O – L – –	7.9 (9.7%)	68 (84%)	12 (29%)	4.5 (3.5%)	0.1 (0.1%)	59 (1.8%)	6.1 (0.2%)	95 (3.0%)	739	22
pdf-extract [31]	– O – L – –	95 (114%)	53 (64%)	99 (32%)	8.4 (3.1%)	4.1 (7.7%)	74 (2.1%)	41 (1.2%)	149 (4.2%)	72	34
pdfx [7]	– O R L D H	6.6 (8.8%)	32 (42%)	9.4 (9.6%)	19 (27%)	0.3 (0.4%)	35 (1.1%)	2.2 (0.1%)	55 (1.7%)	812	70
PDFExtract [3]	P O R L D H	9.5 (11%)	33 (40%)	28 (21%)	22 (25%)	0.8 (0.9%)	12 (0.4%)	2.8 (0.1%)	61 (1.8%)	176	46
Grobid [21]	– O R L D H	9.5 (11%)	30 (36%)	7.5 (6.7%)	11 (15%)	**0.0 (0.0%)**	14 (0.4%)	1.6 (0.0%)	63 (1.9%)	29	42
Icecite [17]	P O R L D H	3.4 (4.0%)	**10 (13%)**	6.2 (4.2%)	7.7 (5.5%)	0.1 (0.1%)	10 (0.3%)	1.7 (0.1%)	**21 (0.6%)**	34	41

Table 2: Summary of the evaluation results of 14 PDF extraction tools. The second column gives a summary of Table 1, for convenience. The evaluation results are given in columns 3-10, broken down into the criteria NL^+: the number of spurious newlines; NL^-: the number of missing newlines; P^+: the number of spurious paragraphs; P^-: the number of missing paragraphs; $P^{\uparrow\downarrow}$: the number of reordered paragraphs; W^+: the number of spurious words; W^-: the number of missing words; $W^\sim$: the number of misspelled words. For each criterion, its absolute value and a percentage is given, which is computed as follows: for NL^+ and NL^-, it is the absolute value divided by the number of newlines in the ground truth; for the other criteria, it is the number of affected words relative to the number of words in the ground truth files. The best values in each criteria are printed in blue and bold, the two worst values in red. The column ERR gives the aggregated number of PDF files where (a) the extraction process resulted in an error or (b) the runtime of the extraction process exceeded the timeout of five minutes. The column T∅ gives the average time needed to process a single PDF file, in seconds.

REFERENCES

[1] S. Anzaroot and A. McCallum. A New Dataset for Fine-Grained Citation Field Extraction. In *ICML Workshop (PEER)*, 2013.
[2] Apache. PdfBox. https://pdfbox.apache.org/, 2017.
[3] Ø. R. Berg. PDFExtract. https://github.com/oyvindberg/PDFExtract/, 2011.
[4] S. Bird, R. Dale, B. J. Dorr, B. R. Gibson, M. T. Joseph, M. Kan, D. Lee, B. Powley, D. R. Radev, and Y. F. Tan. The ACL Anthology Reference Corpus: A Reference Dataset for Bibliographic Research in Computational Linguistics. In *LREC*, 2008.
[5] G. Burns. LA-PdfText. https://github.com/GullyAPCBurns/lapdftext, 2013.
[6] C. Caragea, J. Wu, A. M. Ciobanu, K. Williams, J. P. F. Ramírez, H. Chen, Z. Wu, and C. L. Giles. CiteSeerX : A Scholarly Big Dataset. In *ECIR*, 2014.
[7] A. Constantin, S. Pettifer, and A. Voronkov. pdfx. http://pdfx.cs.man.ac.uk/, 2011.
[8] A. Constantin, S. Pettifer, and A. Voronkov. PDFX: Fully-automated PDF-to-XML Conversion of Scientific Literature. In *DocEng*, 2013.
[9] Cornell University. arXiv.org e-Print archive. https://arxiv.org/, 2017.
[10] N. V. Cuong, M. K. Chandrasekaran, M. Kan, and W. S. Lee. Scholarly Document Information Extraction using Extensible Features for Efficient Higher Order Semi-CRFs. In *JCDL*, 2015.
[11] H. Dejean and E. Giguet. pdftoxml. https://sourceforge.net/projects/pdf2xml/, 2016.
[12] FooLabs. Xpdf: A PDF Viewer for X. http://www.foolabs.com/xpdf, 2014.
[13] T. Hassan. pdfXtk. https://github.com/tamirhassan/pdfxtk, 2013.
[14] Institute of Computer Science and Technology of Peking University. Marmot Datasets. http://www.icst.pku.edu.cn/cpdp/data/marmot_data.htm, 2016.
[15] M.-Y. Kan. ParsCit. https://github.com/knmnyn/ParsCit, 2016.
[16] S. Klampfl, M. Granitzer, K. Jack, and R. Kern. Unsupervised Document Structure Analysis of Digital Scientific Articles. *JCDL*, 2014.
[17] C. Korzen. Icecite. https://github.com/ckorzen/icecite, 2017.
[18] M. Kruk. pdftohtml. https://sourceforge.net/projects/pdftohtml/, 2013.
[19] M. Ley. DBLP - Some Lessons Learned. *PVLDB*, 2009.
[20] M. Lipinski, K. Yao, C. Breitinger, J. Beel, and B. Gipp. Evaluation of Header Metadata Extraction Approaches and Tools for Scientific PDF Documents. In *JCDL*, 2013.
[21] P. Lopez. Grobid. https://github.com/kermitt2/grobid, 2017.
[22] A. McCallum, K. Nigam, J. Rennie, and K. Seymore. Automating the Construction of Internet Portals with Machine Learning. *Inf. Retr.*, 2000.
[23] C. Ramakrishnan, A. Patnia, E. H. Hovy, and G. A. P. C. Burns. Layout-Aware Text Extraction from Full-Text PDF of Scientific Articles. *Source Code for Biology and Medicine*, 2012.
[24] Y. Shinyama. PdfMiner. https://github.com/euske/pdfminer, 2016.
[25] Springer Nature. BioMed Central. https://www.biomedcentral.com/, 2017.
[26] J. Tiedemann. pdf2xml. https://bitbucket.org/tiedemann/pdf2xml/, 2016.
[27] D. Tkaczyk, A. Czeczko, K. Rusek, L. Bolikowski, and R. Bogacewicz. GROTOAP: Ground Truth for Open Access Publications. In *JCDL*, 2012.
[28] D. Tkaczyk, P. Szostek, and L. Bolikowski. GROTOAP2 - The Methodology of Creating a Large Ground Truth Dataset of Scientific Articles. *D-Lib Magazine*, 2014.
[29] D. Tkaczyk, P. Szostek, P. J. Dendek, M. Fedoryszak, and L. Bolikowski. CERMINE - Automatic Extraction of Metadata and References from Scientific Literature. In *DAS*, 2014.
[30] U.S. National Institutes of Health's National Library of Medicine. PubMed Central. https://www.ncbi.nlm.nih.gov/pmc/, 2017.
[31] K. J. Ward. pdf-extract. https://github.com/CrossRef/pdfextract/, 2015.