A PDF Wrapper System for Table Processing Purpose

Roya Rastan
School of CSE
University of New South Wales
Sydney, Australia
rrastan@cse.unsw.edu.au

Hye-Young Paik
School of CSE
University of New South Wales
Sydney, Australia
hpaik@cse.unsw.edu.au

John Shepherd
School of CSE
University of New South Wales
Sydney, Australia
jas@cse.unsw.edu.au

ABSTRACT

Tables are a widely-used structure for data presentation and summarisation in documents but are not yet well-utilised computationally because of the difficulty of extracting their structure and data automatically. Many pre-processing or wrapper tools have been developed to convert different document types to more structured data models. However, it is not straightforward for the tools to extract he document objects at the right level of granularity because it often depends on the intended applications.

In this paper, we propose a PDF document wrapper system that is specifically targeted at table processing applications. We review the PDF specifications and identify particular challenges from the table processing point of view. We specify a table-oriented document model containing the required atomic elements for table extraction and understanding applications, and present implementation of the wrapper that produces the model. Our evaluation showed that using our wrapper modules resulted in better table locating and segmenting, compared to a similar system. The wrapper was able to detect important features such as page columns, bullets and numbering in all measures, recording over 90% accuracy.

Keywords

Table processing, PDF Document, logical Document model, Wrapper System

1. INTRODUCTION

Tables in documents are a rich and widely-available source of inter-related data. It would be useful if their contents could be automatically extracted and manipulated by computers. However, the task of automatic extraction of tables still remains a challenging and difficult one, and a large set of these potentially useful data sources is still manually handled.

One of the better understood reasons for this difficulty is attributed to the nature of aesthetic-focused and free-style presentation of tabular data, especially in business documents.

However, another key inhibitor is the fact that the tables are embedded in various document formats including semi-structured or unstructured ones. The vast majority of them are a sub collection of Microsoft (e.g.,.doc, .docx, .ppt, .pptx), PDF or HTML. Each type of document contains different internal representations (e.g., plain text, XML, binary) with different approaches for rendering.

Therefore, many systems aiming to analyse text in documents incorporate a software component designed to give access to the low-level document objects. These type of software components, referred to as "text extraction tools", tend to be specific to their intended input document type (e.g., XPDF for PDF files).

For table processing systems, it is common to use an existing text extraction tools and process the output to build a *pre-processor* for further analysis to recognise or extract tables. There are a few pre-processors in the area, but they tend to ignore some features present in the low-level objects which could help more effective table processing [25].

Recently, there has been a shift to developing application-oriented (or goal-oriented) pre-processing systems [7] that selectively choose and process objects that are deemed *relevant* for the downstream application.

We refer to these kind of systems as "*application-oriented wrappers*". In fact, wrapping in the Information Extraction field is generally known as the process of extracting data containing information relevant to a specific application, and organizing the extracted data into a machine-readable format.

This paper focuses on processing PDF documents as the input type, as PDF is the most common standard for *print-oriented* formats in textual documents. A PDF document is described by a *content stream* which is a sequence of graphical and textual objects. Those objects are located at precise positions inside the document pages and are, in most cases, untagged. Such print-oriented nature of PDF documents raises many issues that make wrapping from PDF documents challenging [20].

We present the design and implementation of a PDF wrapper named `PDF2TableDoc`, an application-oriented wrapper whose purpose is to enrich the capability of table processing systems. For instance, tables can be described in different levels: physical, logical and abstract [18]. Our wrapper helps capture the relevant atomic elements in documents so that it is possible to express every level of description for a table.

The rest of the paper is structured as follows. In Section 2,

we give an overview of table extraction and understanding tasks and the PDF standards. Section 3 presents our observations on the characteristics of the PDF documents which make table processing challenging. Related work and systems are discussed in Section 4.

Based on the observation and analysis, we propose a document model that can capture the essential elements required by a table processing system in Section 5. The model is a structured XML document which can be used in composition with any table processing system.

In Sections 6–7, we present the design and implementation of `PDF2TableDoc` that extracts the relevant PDF document elements and maps them to our model, followed by evaluation results in Section 8 and conclusion in Section 9.

2. PRELIMINARIES

2.1 Table Extraction and Understanding

Tables in documents can be broadly classified into two categories: text tables where all table parts are composed of text elements, and image tables where some or all table parts contain an image. Our work and discussions focus on working with text tables.

Tables can also be described in different levels. The location of the regions containing parts of tables (e.g. coordinates of a line in a document page) in a document is defined through the `Physical` structure, while `Logical` structure describes the types of these regions and how they form a table (e.g. lines as the logical elements forming columns). The most abstract level of table description is the reading order and the relationships within table cells which Wang [29] defines as `Abstract` tables. Ways to accessing these information varies for different input formats.

In our earlier work, we have proposed an end-to-end table processing system named TEXUS which implements table extraction and understanding functionality for PDF documents [23]. TEXUS is a coherent sequence of tasks. The first half of the tasks consists of locating and segmenting tables to extract tables, the second half of the tasks consists of functional and structural analysis that form table understanding.

The standard definition of the tasks involved in end-to-end table processing led us to determine the specifications of the main elements that constitute the physical, logical and abstract descriptions of a table [23]. This, in turn, allows us to determine the right levels of granularity in the document objects that we need to acquire.

These objects may have different representations in various document formats and be presented as atomic elements or can be formed by grouping elements. In PDF documents, those representations are manifested through `Formatting` (e.g. font and styling specifications), `Layout` (e.g. spanned cell) or `Content` (e.g. leading or terminating characters: tab or colon) features [24].

To be able to effectively analyse these features with a view to help table extraction and understanding tasks later on, we specify `Document Converting` task at the beginning of the pipeline. This task is to handle the wrapping of required document elements (in our case PDF documents) specifically for table analysis. The conceptual design of TEXUS is shown in Figure 1.

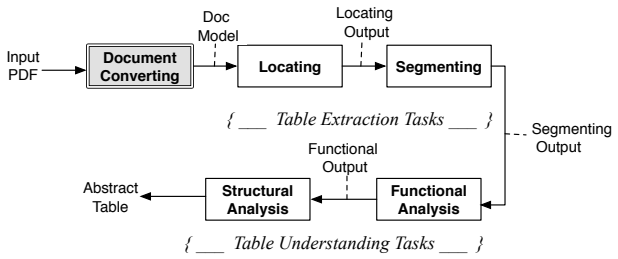


Figure 1: An End-to-end Table Processing Pipeline

2.2 Portable Document Format

Portable Document Format (PDF) is a document format which is independent of the operating system and can be viewed on any computer. It has played an important role in the information storage and transmission in many domains.

PDF is based on the Postscript page-descriptive language which describes the pages of a document using objects (numeric, boolean, string, name, array, dictionary, stream) [10]. The text content of a PDF document is described through *Text String* objects. Every text string (a character or sequence of characters) is described by its font attributes through a *text state* function. Text state considers each text string as a *Bounding Box (BBOX)* with certain positioning in the page and its related glyph. While a character is an abstract symbol, a glyph is a specific graphical rendering of a character specification. Glyphs are organised into fonts. Therefore it is the font that defines the glyphs for a particular character set.

There are some metric and positioning parameters for glyphs in PDF documents. The glyph bounding box is the smallest rectangle (oriented with the axes of the glyph coordinate system) that just encloses the entire glyph shape. The bounding box is expressed in terms of its left, bottom, right, and top coordinates relative to the glyph origin in the glyph coordinate system. A glyph's width (formally its horizontal displacement) is the amount of space it occupies along the baseline of a line of the text that is written horizontally. The glyph coordinate system gives the space in which an individual character's glyph is defined. The glyph origin is the point (0, 0) in the glyph coordinate system. For each BBOX in a PDF file, state array contains four numbers giving the coordinates of the left, bottom, right and top edges. Figure 2 shows a gylph with its metrics and positioning on a page.

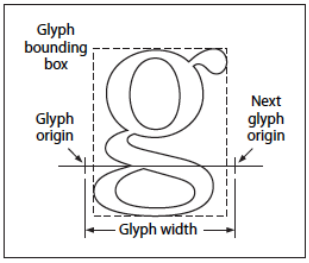


Figure 2: Gylph metrics and positioning

3. WRAPPING PDF FOR TABLE PROCESSING

To process text tables, text extracting tools for PDF such as XPDF[1] or PDFBOX[2] are commonly used to obtain the low-level document objects. Then, based on the output, the wrappers work further to segment the elements into regions containing text with similar attributes.

The information obtained with the help of these tools can be divided into two categories: the text content and the text style. The text content refers to the text strings; The text style includes the corresponding text attributes: the font, the size, line spacing and color, etc; The text streams extracted from PDF files may correspond to various objects: a character, a partial word, a complete word, a line, etc. In addition, the order of these text streams does not always correlated with the reading order. A word reconstruction and a reading order re-sorting steps are necessary in order to correctly extract the coherent text from a PDF file.

Although PDF has become the common standard for the distribution of electronic documents, the print-oriented nature of PDF leads to serious drawbacks for table processing.

Here, we discuss the main issues and requirements in building a PDF wrapper to enhance table processing results.

Lack of structural information.

Since PDF was originally designed for the final presentation of a document, most PDF documents are untagged. They contain very little or no explicit structural information about the content.

According to the literature, this creates difficulties in table processing applications as follows:

- Table boundary detection [16, 22]: Since there is little or no markup for tables in most PDF documents (in comparison with HTML and XML), table region identification and boundary detection is not straightforward. To successfully extract data from tables in such documents, detecting the table boundary is a crucial step.

- page column vs. table column [8, 5]: One of the issues in table detection in multi-column PDF documents is differentiating a page column from a table column. Since page columns are not stated explicitly in PDF, various indications (e.g. graphical lines, white space) have to be detected and analysed prior to table spotting to properly recognise page columns in a document.

- Multi-line cell detection [4, 6]: Since the text elements are described based on their positions on the page and there are no tags for higher level elements (e.g. table cells), finding the logical relations between texts is another challenge, especially when multiple text lines must logically form a single table cell.

Implicit information about formatting and styling.

In most of the cases formatting and styling could convey information about the logical relationships between the texts. For example, the font style and size for headers are normally different to paragraphs. Bullets are used to show hierarchies. There are aspects in table processing that could benefit from having explicit access to this kind of information.

- Hierarchical header detection: In some tables, the hierarchy of row headers is shown with paragraph styling such as bullets or numbering or indentation [24]. Having access to this information is necessary for better table understanding and tabular data relation extraction [26].

- Formatted headers: In some tables, the hierarchy of table headers is marked by differences in the font sizes and styles [1]. Again, these features should be preserved by the pre-processor for further processing.

Rendering order problem.

The main advantage of PDF documents is to preserve the original document presentation. As a user who views the document, the final PDF layout and content is more important than the process through which the page is produced.

Many different PDF generating systems can generate the same document appearances and rendered effect on the screen. However, they may follow different rendering order. A given PDF document might be structured to write out text a word/character at a time, or render specific types of text (e.g., footnotes) first. Very often, the rendering order is totally different from the 'reading order'. For multi-column text, the document is sometimes rendered across the columns by hopping the inter-column gutter.

Although the lack of standard rendering order does not affect the PDF document displaying and reading, it heavily impacts the performance of document structure analysis and understanding. Specifically, it affects the table locating task which relies on relative positions and sequence of text objects in the page [15].

In `PDF2TableDoc`, we aim to resolve these issues. For example, we incorporate purposely designed algorithms to detect page columns, logically related text lines, etc. This enhances the performance of the table locating and segmenting tasks. We perform detailed analysis of the styling features to accurately recognise them for future use in the processing pipeline. This enhances the performance of the functional and structural analysis tasks. The design and implementation of the `PDF2TableDoc` is underpinned by our own document model suitable for table processing which is introduced in Section 5.

4. RELATED WORK

As mentioned before, with the help of many commercial or open source text extraction tools, the page objects can be extracted from PDF sources directly. Most of PDF understanding systems obtain the low-level document objects using these tools first. Some of the well known text extraction tools that provide the low-level document objects are XPDF library, the Apache PDFBox library, PDFlib Text Extraction Toolkit (TET) [3], PDF2Text [4].

The major effort in pre-processing in wrappers goes to segmenting the text elements in a PDF page into regions containing text with similar attributes [25]. These attributes are

[1]http://www.foolabs.com/xpdf
[2]http://pdfbox.apache.org/
[3]https://www.pdflib.com/products/tet/
[4]http://www.pdf2text.com

used as key features with predefined heuristic-based methods for tagging the various segments.

There are three main approaches to analyse the PDF document structure: top-down, bottom-up, or hybrid of the previous two[19]. The top-down approach is suitable for the cases that we know the PDF document structure and the page model in advance. Having the whole PDF document page, this approach segments it into smaller objects based on paragraph breaks or inter-column gutters in an iterative manner. The segmenting procedure stops when some criterion is satisfied. [9].

If there is no predefined page template , a bottom-up method is adopted. Bottom-up algorithms usually start from the low-level document objects (e.g., the characters) and merge them into larger objects such as words,lines or zones [2].

Various techniques are used in the literature to understand document content and structures. Since most of the text extraction tools provide low-level objects information, the bottom-up methods are mostly applied. As mentioned in [9] most of the systems following the bottom-up approach, go through three main steps :

- Step 1: Accessing to the low-level objects in the PDF. For this purpose, various PDF extraction tools are available to be used for extracting the PDF document atomic objects [28, 14].
- Step 2: Merging low-level object to form compound object such as creating words, lines, paragraphs and other layout structures by relying on geometric and formatting attributes [21, 3, 30, 12].
- Step 3: Detecting logical objects in the document. Usually the logical objects (such as tables, lists, scientific charts, etc.) are identified based on the simple and compound objects extracted in previous steps [17, 13].

In this paper, we follow the similar first two main steps to process a text extraction tool output. However, at every step, the guidelines for the decision making in object extraction and segmenting are informed by our need to enhance the table extraction/understanding capabilities. The final output of the wrapper then is mapped to our own document model which is suitable for table processing.

5. TABLE-ORIENTED DOCUMENT MODEL

In this section, we introduce our document model which becomes the target schema of `PDF2TableDoc` in terms of extracting *table processing document elements*.

As described before, a PDF document is represented by a series of objects and their associated structure information. Therefore, to capture the necessary elements to facilitate further table processing, we focus not only on the text content of a PDF document but also on formatting and layout features.

Figure 3 shows the hierarchical structure of the elements in our proposed document model. *Text Chunk*s are the basic elements of a PDF document. A Text Chunk is an atomic object (i.e. textual element such as a character, a word or a sentence), which is totally contained within a document page. The graphical representation of a Text Chunk on the page layout takes up a certain room delimited by the Text Chunk bounding box.

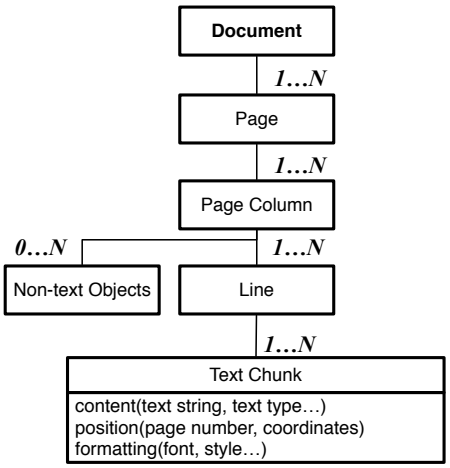


Figure 3: Elements of table-oriented document model

In order to formally define the Text Chunk for our model, we rely on the *Text String* Object introduced in the PDF reference document [11]. We also consider and define the features that highly applicable to our table processing pipeline (such as font attribute and text styling features). To illustrate the elements of our document model, we use the sample document shown in Fig. 4. For example, we have highlighted some of the Text Chunks as Ch_i).

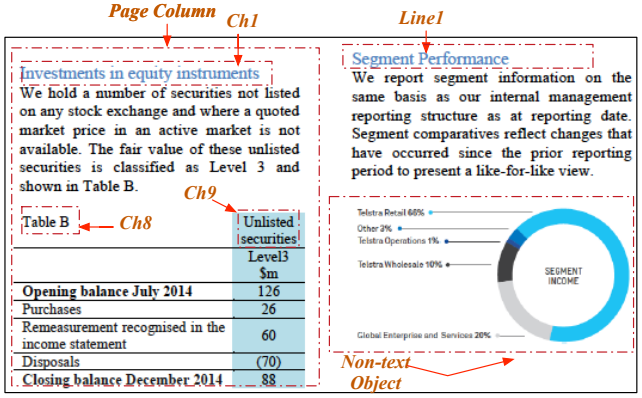


Figure 4: A sample of a document page

Text Chunk: A `Text Chunk` Ch is an instance of a *Text String* with certain bounding box and font specifications. We define a A `Text Chunk` as a tuple $\langle C, S, F\rangle$, where:

- C contains the information about the content of the `Text Chunk`.

  $C = \langle Ts, Tt, Tc\rangle$, Ts shows the text string, Tt represent the generic type of the string (e.g. Numeric, Alphabetic, Alphanumeric, Date, Percent, ...) and Tc is to show the corresponding ontology/semantic concept for the text content, if applicable.

- S represents the spatial attributes of the `Text Chunk`.

  $S = \langle Top, Left, Height, Width, P\rangle$ to show the bounding box of the Text Chunk by determining its distance from the top and left of the page P and also the height and width of the Text Chunk.

- F represents the formatting attributes of the `Text Chunk`.

  $F = \langle Font, Style \rangle$, where $Font = \langle Family, Color, Size, Face \rangle$ is to convey the font information and $Style$ describes the layout styling of the `Text Chunk` (e.g. Bullet and Numbering).

Line: A `Line` L_j is a sequence of horizontally consecutive `Text Chunk(s)` $L_j = \langle Ch_1, Ch_2, ..., Ch_n \rangle$, terminated by an End-of-Line delimiter. The coordinates of the bounding boxes for Ch_1 and Ch_n are used to determine whether the `Text Chunks` are horizontally aligned.

Non-Text Object: Any element, such as an image, that is not composed of text is treated as part of a `Non-Text Object`. A $NTxO_j$ has a bounding box on the page. Since we are only interested in text tables, we do not retrieve information about such elements from PDF document, except the bounding box.

Page Column: A `Page Column` PaC_j is an ordered sequence of one or more `Line`s and zero or more `Non-text objects`. Page column boundaries can be detected by noting the resetting of the Top values of subsequent document elements.

Page: A `Page` P_j is an ordered sequence of `Page Column`s $P_j = \langle PaC_1, PaC_2, ..., PaC_k \rangle$, which terminates to an End-of-Page marker.

Document: A `Document` Doc is an ordered list of `Pages`, $Doc = \langle P_1, P_2, ..., P_q \rangle$.

In above definition, we assume that each table is contained within a `Page` and do not span multiple pages.

6. DESIGN OVERVIEW OF THE WRAPPER

The PDF document content stream lists all of the page objects, such as text, image, and path. Therefore in order to discover the logical components of a PDF document we should analyse and interpret the layouts and attributes of the page objects so as to correctly break or group them into different logical components.

Our design of `PDF2TableDoc` starts from a text extraction tool which provides the low-level information (characters, words, coordinates, etc.). Since we would like to build our own structure information for table processing, this tool gives enough information as our starting point.

According to the comparison done in [25], XPDF is the most accurate and fastest PDF source viewer between the six evaluated tools. Therefore, we develop our system based on this utility.

We designed `PDF2TableDoc`, which is the implementation of the "Document Converting Component" in the TEXUS pipeline (see Figure 1), as a web service which receives a PDF document as input and provides the all elements of our document model. There are two sub modules operating inside `PDF2TableDoc`: `PDFtoXML` and `TableDocWrapper`. The PDF is passed to the `PDFtoXML` module which is implemented using XPDF as the core. XPDF utility reads the document characters along with their state function. `PDFtoXML` then merges characters to detect *Words* and *Text Chunks*. The list of text chunks will be passed to the `TableDocWrapper` module to identify more elements relevant to our document model. Figure5 shows the design of `PDF2TableDoc` as our document converting component.

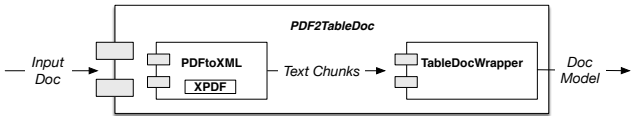


Figure 5: Document Converting Component:`PDF2TableDoc`

We represent the output of the document converting task in XML format. We chose XML because, if it is utilized correctly, it can be applied to create identical representations of the original documents and also can provide better searchability and flexibility when analysing documents.

7. IMPLEMENTATION

In this section, we explain the steps undertaken in each sub module of `PDF2TableDoc`.

7.1 PDFtoXML

After receiving the document information at individual character level from XPDF, we take the following main steps to present the document at the Text Chunk level.

7.1.1 Word Detection

We rely on the BBOX information for every character in a PDF file, to form *words*. Algorithm 1 describes the main steps in detecting words from characters. AddChar() takes inputs, text state, x y coordinates (Top, Left), width and height of the character, the character (in Unicode) and the length of the character (in Unicode). First, we actualise the coordinates based on the text state parameters (character space and horizontal spacing) to be able to compare the character coordinates with the current string (Line 2). Then, we check if the text direction of the character is the same as the direction of current word. If they are not in the same direction we consider the newly seen character as the start a new word (Line 3-5). Otherwise, we check if the character is horizontally close enough to the current word to be considered as part of that word. Since the character is presented in Unicode its length may be more than 1 byte, therefore we consider the average width and length of the Unicode encoding of the character (Line 7-9) and then we calculate the horizontal distance between the character and the right most coordinates of the current word string (Line 12). If we see a whitespace character or the distance is bigger than a predefined parameter (α) we start a new word (Line 13-15) otherwise, we add the character to the current word string (Line 17). The reason we consider α is that sometimes the horizontal distance between characters is less than a whitespace. e.g. Ex.1 in Figure 8

7.1.2 Text Chunk Detection

The main aim of this step is to merge words represented as text strings based on the intersections between coordinates and the similarity in font attributes. we consider four positioning of horizontally consequent words in the page to be merged as text chunks.

- str_1 is `vertically lower than` str_2 (Figure 6-a): $(str_2.y_{Min} \geq str_1.y_{Min}$ & $str_2.y_{Min} \leq str_1.y_{Max})$.

- str_1 is `vertically higher than` str_2 (Figure 6-b): $(str_2.y_{Max} \geq str_1.y_{Min}$ & $str_2.y_{Max} \leq str_1.y_Max)$

Algorithm 1 addChar(textState state,x_1, y_1, w_1, h_1, Unicode char, int charLen)

```
1: n = curStr.len
2: actualise coordinates (state)
3: if (char.getDirection != curStr.dir) then
4:    endString()
5:    beginString(state, NULL);
6:    return;
7: end if
8: if  (charLen ≠ 0) then
9:    w1 /= charLen
10:   h1/= charLen
11: end if
12: for (i=0, i ≤ charLen, ++i) do
13:    gap = x1 + i * w1 − curStr.xMax
14:    if (char[i] = ' ') || (gap > α) then
15:       endString()
16:       beginString(state, NULL)
17:       return;
18:    else
19:       curStr.appendChar(state, x1+i×w1, y1+i×h1, w1, h1,
          char[i])
20:    end if
21: end for
```

Algorithm 2 $formTextChunks(Str_1, Str_2)$

```
1: while (Exists(str1) & (str2 = str1.Nextstring)) do
2:    height= str1.yMax − str1.yMin
3:    horSpace = str2.xMin − str1.xMax
4:    if (!addLineBreak) &(str1.xMin − str2.xMin < α) then
5:       vertSpace = str2.yMin − str1.yMax
6:    end if
7:    if (str2 is vertically lower than str1) then
8:       vertOverlap = str1.yMax − str2.yMin
9:    else if str2 is vertically higher than str1 then
10:      vertOverlap = str2.yMax − str1.yMin
11:   else if  str2 covers str1 entirely then
12:      vertOverlap = str1.yMax − str1.yMin
13:   else if  str2 is entirely contained by str1 then
14:      vertOverlap = str2.yMax − str2.yMin
15:   else
16:      vertOverlap = 0
17:   end if
18:   if ((vertOverlap > 0.5 × height) & (horSpace < β) & (
      0 < vertSpace < 0.5 × height ) & (str1.dir = str2.dir) &
      (str1.font = str2.font)) then
19:      str1.append(str2)
20:   end if
21: end while
```

- str_1 is `entirely contained by` str_2 (Figure 6-c): $(str_2.y_{Min} \geq str_1.y_{Min}$ & $str_2.y_{Max} \leq str_1.y_{Max})$
- str_1 `entirely covers` str_2 (Figure 6-d): $(str_2.y_{Min} \leq str_1.y_{Min}$ & $str_2.y_{Max} \geq str_1.y_{Max})$

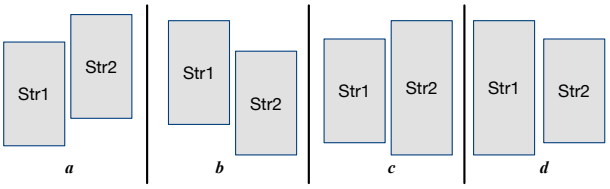


Figure 6: vertical positioning of horizontal consequent words

Algorithm 2 lists the details of *formTextChunks()* which takes two strings and performs a merge if one of the positioning checking rules matches.

7.2 TableDocWrapper

XPDF reads characters in the raw order and the `PDFtoXML` module merges them to form text chunks preserving the raw order. However, text sequence error is still a common problem in the existing text extraction tools. This means the extracted text chunks follow a different sequence from its original appearance in PDF documents. If such a text sequence error happens in the table contents, it could generate wrong results during the table detection, such as segmenting a single table into several pieces, wrong column or row information, and omitting cells or missing a whole table.

In this section, we review the steps after receiving the output of `PDFtoXML` to transform it into our target document model. The main task of `TableDocWrapper` module is to go through the text chunks produced bu `PDFtoXML` and analyse/detect the relevant elements of the target document model.

7.2.1 Related Text Merging

One of the common cases in tables is the existence of *Multi-Line Cells*. In which multiple text chunks form a logically single table cell. These text chunks however may appear in different lines Since we have the text chunks in raw order, we have a chance to detect these cases before sorting the text chunks based on the coordinates in the page.

We consider each text chunk as a rectangular object represented by its four coordinates attributes and font specifications. Every vertically consequent text chunk which is in the close top distance from the page is compared to see whether there is any horizontal intersection between them or not.

Figure 7 shows different scenarios of horizontal intersections between text chunks and how the merged text chunks are actualised.

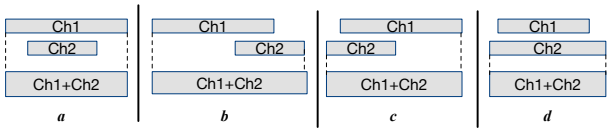


Figure 7: Different interval intersection to merge text chunks

In order to make the merge more accurate, we check if the text chunks share the same font and also the same alignment. Algorithm 3 shows the detail of this merging. It receives two text chunks (the last seen text chunk lCh and a newly seen text chunk nCh), α as a predefined threshold parameter for calculating the vertical closeness of the text chunks and β a threshold for calculating the horizontal intersection between the text chunks. We set some variables for the coordinates and font information of nCh and lCh in Line 1 to Line 9. Then in Line 10 we check whether the lCh and nCh are in the acceptable vertical distance to be considered as potential cases to be merged. If so we check their horizontal intersection between them and their font and alignment compatibility in Line 11. Provided all checking is satisfied, text chunk will be merged and the attributes (coordinates and font) will be updated for the new merged text chunk (Line 12-13).

Algorithm 3 *mergeTextChunks*(Text chunk nCh,Text Chunk lCh, α, β)

```
1: ttop= nCh.Top()
2: t_l top =lCh.Top()
3: t_l height = lCh.Height()
4: tleft= nCh.Left()
5: twidth= nCh.width()
6: t_l left =lCh.left()
7: tfont= nCh.font()
8: t_l width =lCh.width()
9: t_l font =lCh.font()
10: if (t_l top < ttop) & (ttop ≤ (t_l top + t_l height + α) then
11:    if isIntervalContains(t_l eft, t_w idth, tl_l eft, tl_w idth, β) &
       (tfont =t_l font) & (talignment=t_l alignment) then
12:       lCh.setValue(lCh.text() + nCh.text())
13:       updateNodeAttributes(lCh, nCh)
14:    end if
15: end if
```

7.2.2 Page Column Detection

As the next step, we mark page columns. Since the text chunks are created in the raw order, the texts in a page column are supposed to appear sequentially (in reading order). The start of a new page column can be detected by a large decrease (bigger than a predefined threshold α) in the `Top` value in the following text chunks. Algorithm 4) shows the detail of page column detection. For all text chunks in the page, we compare the top difference between two consequent text chunks and if the decrease in the distance between them is more than α we add a new column to the page (Line 8-12). Otherwise the attributes of the page column get updated based on the newly seen text chunks. The page column gets the top attribute from the first text chunk in the column boundary, the left most text chunk form the left attribute and the height of the page column is the coming from the differences of the top between the first and the last text chunk in the page column boundary.

Algorithm 4 *detectPageColumns*(Page P, α)

```
1: formerTextChunkTop = 0
2: currentColumn = newPageColumn(P, "pageColumn")
3: ArrayList Columns = new ArrayList<pageColumn>()
4: Columns.add (currentColumn)
5: chNumber= P.Chunks.Size()
6: for (i = 0, i < chNumber, ++i) do
7:    if (P.Chunks[i].top < formerTextChunkTop) & (former-
      TextChunkTop - P.Chunks[i].top >α) then
8:       currentColumn = newPageColumn(P, "pageColumn")
9:       Columns.add (currentColumn)
10:      currentColumn.setAttribute()
11:      formerTextChunkTop = P.Chunks[i].Top()
12:   else
13:      currentColumn.updateAttribute()
14:   end if
15: end for
```

7.2.3 Line Detection

After detecting the page columns, we mark lines in each column. The main idea of the line formation is to consider the text chunks in the same top distance in each page column belong to the same line. Therefore, Algorithm 5 receives a page column and a predefined threshold parameter (β) to check the top distance of the text chunks in the page column. Since we have text chunks in raw order and also we have merged some text chunks in `Related Text Merging` (introduced new coordinates for the merged text chunks) we need to sort the text chunks. Line 4 and 5 sorts all text chunks in the page column first based on the `Top` and then `Left` attributes. Starting from the first text chunk in the page column if the vertical distance of the following text chunk and the former text chunk is bigger than a threshold, it indicates the start of a new line (Line 7-13). As we create a new line we update its coordinate attributes by considering its contained text chunks (Line 15).

Algorithm 5 *buildLinesInColumn*(Page Column $column, \beta$)

```
1: formerTextChunkTop = -1
2: ArrayList Lines = new ArrayList<Line>();
3: chNumber= column.Chunks.Size()
4: for (i = 0, i < chNumber, ++i) do
5:    sortChunks(column.Chunks[i],Top, Left)
6: end for
7: for (i = 0, i < chNumber, ++i) do
8:    if (column.Chunks[i].Top() > (formerTextChunkTop + β)
      then
9:       currentLine = new Line(null, "line")
10:      currentLine.setAttributes()
11:      currentLine.append (column.Chunks[i])
12:      Lines.add (currentLine)
13:      formerTextChunkTop = column.Chunks[i].Top()
14:   else
15:      currentLine.updateAttributes()
16:   end if
17: end for
18: for (Line n:Lines) do
19:    column.add(n)
20: end for
```

7.2.4 Bullet Detection

Most of the current PDF wrappers are unable to detect the detailed formatting of the text elements in the PDF file, such as bullets or numbered text elements. Since these formatting information can convey important hints about hierarchical structure of the content it is useful to properly recognise them in a table processing system [26, 24]. Algorithm 7 shows the bullet detection for an input text chunk. It gets the first character in the text chunk and checks whether its corresponding Hexadecimal number belongs to a list of bullet numbers (Line 4-7).

Algorithm 6 shows the processing of bulleted and numbered text chunks. It receives a page column and go through each text chunks in each line of the page column to detect the numbered and bulleted text and add the style attribute to the text chunk (Line 3-6). The pattern for a numbered text is described in Line 1.

Algorithm 6 *processBulletAndNumbered*(Page Column $column$)

```
1: numberedTextPattern = '∧[(]?([a−zA−Z]|[0−9]+)[.\\− :])'
2: Pattern p = Pattern.compile(numberedTextPattern)
3: for each line ∈ pagecolumn do
4:    for each textChunk ∈ line do
5:       if (textChunk!= null)&(startsWithBullet(textChunk)
         then
6:          textChunk.att.add('style', 'Bulleted')
7:       end if
8:    end for
9: end for
```

Algorithm 7 *startWithBullet*(String *text*)

```
1: if (text = null) || (text.length() ≤ 0) then
2:     return false
3: end if
4: char c = text.charAt(0)
5: String s = Integer.toHexString(c | 0 × 10000)
6: if (s.equals('12022') || s.equals('100b7') || s.equals('100a1')
   then
7:     return true
8: end if
```

8. EVALUATION

For the evaluation of PDF2TableDoc, we used the well-known corpus in the community which is introduced in the 2013 table competition at ICDAR[5]. It contains 67 PDF documents with 156 tables. The ground-truth is provided for locating and segmenting tasks with the corpus.

We present two scenarios for the evaluation in the following subsections.

8.1 PDFtoXML

First, we compare the functionality of our core module PDFtoXML with a similar, open system PDFtoHTML[6]. PDFtoHTML is a utility which is also developed based on XPDF and can be configured to create XML output and list text chunks from PDF documents along with their coordinates.

To compare the performance of the two utilities we run our TableDocWrapper module with the outputs from PDFtoXML, and also with the outputs from PDFtoHTML. In each run, we then feed the output of TableDocWrapper to our table extraction system to locate and segment tables. We then compare the locating and segmenting results of each run with the ground-truth.

Here, we are able to compare the performance of PDFtoXML and PDFtoHTML on their ability to accurately extract the text chunks, as more accurate extraction of text chunks leads to more accurate extraction of the final document model by TableDocWrapper.

Table 1 shows the evaluation results. We have compared the performance of two runs by utilising two measures:

- *Completeness* and *Purity*, measured for the whole document set (total number of tables 156). A table is considered as *complete*ly extracted if it includes all cells in the table region; and a detected table is called *pure* if it does not include any cells which are not in the table region. A correctly detected table is therefore both complete and pure [27].

- *Recall* and *Precision*, measured specifically for the unsuccessful cases and reported in a cell-by-cell comparison with the ground-truth.

Our own detailed study of the results showed that the main improvements obtained by PDF2XML is from how the gap analysis between characters is done during the word detection phase. PDFtoHTML seems to simply rely on a whitespace character to decide whether to merge characters or not. We check the size of the gap as well as the existence of the whitespace character. If the size is more than a threshold we do not merge characters into one words (Line 11-15 in Algorithm 1).

[5]http://www.tamirhassan.com/dataset

[6]https://sourceforge.net/projects/pdftohtml

Table 1: PDFtoXML and PDFtoHTML performance comparison

Utility	Per-document average			Whole DocSet #Tables=156	
	Recall	Prec.	F-meas.	Complete	Pure
PDFtoXML	0.9971	0.9729	0.9848	142	148
PDFtoHTML	0.9644	0.9569	0.9606	138	130

Figure 8 shows some tables that PDFtoHTML fails to detect as separate text chunks and merge them as one text chunk because of the small horizontal distance between them (marked by dotted lines).

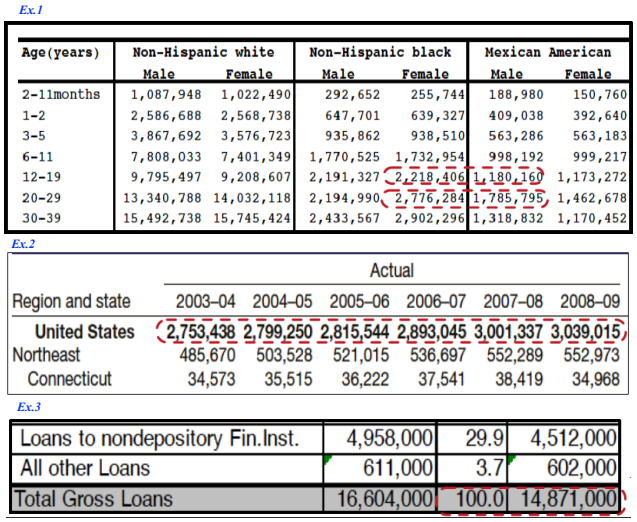


Figure 8: Examples where PDF2HTML failed due to insufficient gap analysis

8.2 TableDocWrapper

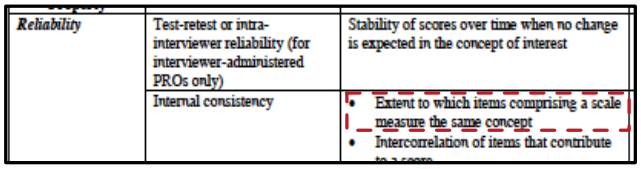
In order to evaluate the performance of the TableDocWrapper module, we categorise the documents in the corpus based on the key elements according to the document model (e.g., page columns, bullets, numbering) and then investigate the accuracy of the output at the detected element level. We compare the result with the ground truth documents reviewed and validated by two experts. Table 2 reports on the accuracy of *page column* and *bullet/numbering* detection as well as *related text merge*. The results are calculated per document and then the overall performance is shown by the average on the whole documents. Table 2 shows the results.

Table 2: TableDocWrapper Performance Evaluation

Element Type	Recall	Prec.	F-meas.
Merged text	0.91	0.95	0.93
Bullet and Numbering	1	1	1
Page Column	1	0.92	0.96

The system has 100% accuracy in detecting bullets and numbering in the documents. Particularly, it successfully

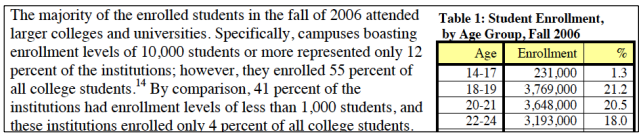
recognised a bullet point applied on a merged text chunk. Figure 9 shows one example of this.



Reliability	Test-retest or intra-interviewer reliability (for interviewer-administered PROs only)	Stability of scores over time when no change is expected in the concept of interest
	Internal consistency	• Extent to which items comprising a scale measure the same concept • Intercorrelation of items that contribute

Figure 9: Correct bullet detection for merged text

The system generally performed well in detecting page columns. However in some cases, such as floating tables, the system reports false positive page columns. As shown in Figure 10, the floating table on the right caused the system to treat the page as a two-column page.



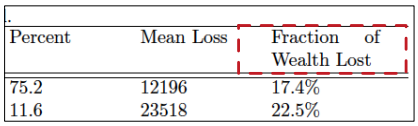
The majority of the enrolled students in the fall of 2006 attended larger colleges and universities. Specifically, campuses boasting enrollment levels of 10,000 students or more represented only 12 percent of the institutions; however, they enrolled 55 percent of all college students.[14] By comparison, 41 percent of the institutions had enrollment levels of less than 1,000 students, and these institutions enrolled only 4 percent of all college students.

Table 1: Student Enrollment, by Age Group, Fall 2006

Age	Enrollment	%
14-17	231,000	1.3
18-19	3,769,000	21.2
20-21	3,648,000	20.5
22-24	3,193,000	18.0

Figure 10: Incorrectly detected page column

For related text merging, the system achieved over 90% performance in all three measures. However, cases with irregular styling and spacing between text chunks result in poor performance. As can be seen in Figure 11, there is a vertical spacing between "Fraction of" and "Wealth Lost" which was slightly larger than the *vertical closeness threshold* and the chunks were not merged.



Percent	Mean Loss	Fraction of Wealth Lost
75.2	12196	17.4%
11.6	23518	22.5%

Figure 11: Failed to correctly merge related text

9. CONCLUSION

In this paper we presented `PDF2TableDoc` a PDF wrapper designed for table processing applications. We reviewed the specific challenges and issues for wrapping PDF with the focus on table extraction and understanding. We then presented the document model that captures (i) logical groups of objects such as page columns and lines, (ii) more complete text chunk metadata such as styling features (e.g., bullets, numbering) and (iii) merged text chunks for better recognition of table cells. We also presented the implementation of the `PDF2TableDoc` to produce the model. Our evaluation showed that using the `PDFtoXML` module resulted in better table locating and segmenting, compared to using `PDFtoHTML`. The `TableDocWrapper` module was able to detect important features such as page columns, bullets and numbering in all measures, recording over 90% accuracy.

This PDF wrapping approach underpinned by our model helps focus the task of the pre-processor on producing features that are most useful for table processing.

REFERENCES

[1] Norah Alrayes and Wo-Shun Luk. *Automatic transformation of multi-dimensional web tables into data cubes*. Springer, 2012.

[2] Anjo Anjewierden. Aidas: Incremental logical structure discovery in pdf documents. In *icdar*, page 0374. IEEE, 2001.

[3] Bertrand Coüasnon and Aurélie Lemaitre. Recognition of tables and forms. In *Handbook of Document Image Processing and Recognition*, pages 647–677. Springer, 2014.

[4] David W Embley, Daniel Lopresti, and George Nagy. Notes on contemporary table recognition. In *Document Analysis Systems VII*, pages 164–175. Springer, 2006.

[5] Jing Fang, Liangcai Gao, Kun Bai, Ruiheng Qiu, Xin Tao, and Zhi Tang. A table detection method for multipage pdf documents via visual seperators and tabular structures. In *Document Analysis and Recognition (ICDAR)*, pages 779–783. IEEE, 2011.

[6] Jing Fang, Prasenjit Mitra, Zhi Tang, and C Lee Giles. Table header detection and classification. In *AAAI*, 2012.

[7] Bettina Fazzinga, Sergio Flesca, Andrea Tagarelli, Salvatore Garruzzo, and Elio Masciari. A wrapper generation system for pdf documents. In *Proceedings of the 2008 ACM symposium on Applied computing*, pages 442–446. ACM, 2008.

[8] Tamir Hassan and Robert Baumgartner. Intelligent text extraction from pdf documents. In *Computational Intelligence for Modelling, Control and Automation, 2005 and International Conference on Intelligent Agents, Web Technologies and Internet Commerce, International Conference on*, volume 2, pages 2–6. IEEE, 2005.

[9] Jianying Hu and Ying Liu. Analysis of documents born digital. *Handbook of Document Image Processing and Recognition*, pages 775–804, 2014.

[10] Adobe Systems Incorporated. *PDF Reference*.

[11] Adobe Systems Incorporated. Pdf reference. Technical Report Version 1.7, Novemeber 2006.

[12] Piyushee Jha and George Nagy. Wang notation tool: Layout independent representation of tables. In *Pattern Recognition, 2008. ICPR 2008. 19th International Conference on*, pages 1–4. IEEE, 2008.

[13] Shah Khusro, Asima Latif, and Irfan Ullah. On methods and tools of table detection, extraction and annotation in pdf documents. *Journal of Information Science*, page 0165551514551903, 2014.

[14] Ying Liu, Kun Bai, and Liangcai Gao. An efficient pre-processing method to identify logical components from pdf documents. In *Advances in Knowledge Discovery and Data Mining*, pages 500–511. Springer, 2011.

[15] Ying Liu, Kun Bai, Prasenjit Mitra, and C Lee Giles. Improving the table boundary detection in pdfs by fixing the sequence error of the sparse lines. In *10th International Conference on Document Analysis and Recognition (ICDAR'09)*, pages 1006–1010. IEEE, 2009.

[16] Ying Liu, Prasenjit Mitra, and C Lee Giles. Identifying table boundaries in digital documents via sparse line detection. In *Proceedings of the 17th ACM*

Conference on Information and Knowledge Management, pages 1311–1320. ACM, 2008.

[17] Ying Liu, Prasenjit Mitra, C Lee Giles, and Kun Bai. Automatic extraction of table metadata from digital documents. In *Proceedings of the 6th ACM/IEEE-CS joint conference on Digital libraries*, pages 339–340. ACM, 2006.

[18] Vanessa Long. *An Agent-Based Approach to Table Recognition and Interpretation*. PhD thesis, Macquarie University Sydney, Australia, 2010.

[19] Simone Marinai. Introduction to document analysis and recognition. In *Machine learning in document analysis and recognition*, pages 1–20. Springer, 2008.

[20] Rosmayati Mohemad, Abdul Razak Hamdan, Zulaiha Ali Othman, and Noor MaizuraMohamad Noor. Automatic document structure analysis of structured pdf files. *International Journal of New Computer Architectures and their Applications (IJNCAA)*, 1(2):404–411, 2011.

[21] Anssi Nurminen. *Algorithmic extraction of data in tables in PDF documents*. PhD thesis, TAMPERE University Of Technology, 2013.

[22] Ermelinda Oro and Massimo Ruffolo. PDF-TREX: An approach for recognizing and extracting tables from pdf documents. In *10th International Conference on Document Analysis and Recognition (ICDAR'09)*, pages 906–910. IEEE, 2009.

[23] Roya Rastan, Hye-Young Paik, and John Shepherd. Texus: A task-based approach for table extraction and understanding. In *Proceedings of the 2015 ACM Symposium on Document Engineering*, pages 25–34. ACM, 2015.

[24] Roya Rastan, Hye-Young Paik, John Shepherd, and Armin Haller. Automated table understanding using stub patterns. In *Database Systems for Advanced Applications*. Springer, 2016.

[25] Nicholas Robinson. A comparison of utilities for converting from postscript or portable document format to text. Technical report, 2001.

[26] Sachin Seth and George Nagy. Segmenting tables via indexing of value cells by table headers. In *Document Analysis and Recognition (ICDAR), 2013 12th International Conference on*, pages 887–891. IEEE, 2013.

[27] Ana Costa E Silva. New metrics for evaluating performance in document analysis tasks_application to the table case. In *Document Analysis and Recognition, 2007. ICDAR 2007. Ninth International Conference on*, volume 1, pages 481–485. IEEE, 2007.

[28] Ralph Sommerer. Presentable document format: Improved on-demand pdf to html conversion. Technical report, Technical Report MSR-TR-2004-119, Microsoft Research (MSR), 2004.

[29] Xinxin Wang. *Tabular abstraction, editing, and formatting*. PhD thesis, University of Waterloo, 1996.

[30] Yalin Wang, Ihsin T Phillips, and Robert M Haralick. Table structure understanding and its performance evaluation. *Pattern Recognition*, 37(7):1479–1497, 2004.