TrapRange: a Method to Extract Table Content in PDF Files

**THO Q LUONG @ SETA ASIA**

**HANOI, VIETNAM**

**Introduction:**

Table data structure is one of the most important data structure in document, especially when exporting data from systems, data is usually in table format.

Have several data file formats are often used to store data including csv, text, pdf. With first two formats, it is very easy to extract data content by openning files, loop through lines and split cells depdending on its cell separator, and ofcourse having may libraries that help you to do that work automatically. It's normal work. But with pdf file, it's not familiar format to read and process directly from InputStream because it is a complicated file format that can contain not only text data, font, content style, but also image, audio and video[1]. In this post, i describe my solution used to extract table data in pdf files. My solutions was implemented, experimented and adapted with pdf files having high density of table content. The advantages and disadvantage of my solution are also discussed.

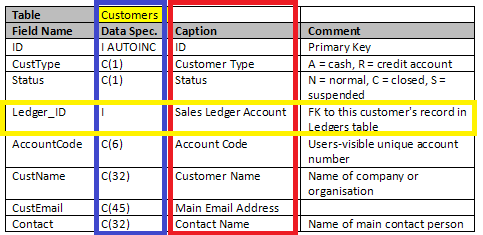
**What is document having high density of table content?**

Document having high density of table content is a document that most of it content is contained in a single table. Content that is not in this table is called "noisy content". My solution focuses on this kind of table because it aims to extract table content in pdf file that contains exported data from systems. Example: your customer exports their users data in a pdf file (lots of systems only support exporting data in this format) and you must import them into your app, ofcourse, these data is in table format. In our algorithm, we set threshold 20%, if percent of noisy line in every page of document  is less than or equal to the threshold then this document is high density of table content.

**How to recognize a table**

After some investigation i realized that:

1. **Column**: text content in cells of the same column are lied on a rectangular space that does not overlap with other rectangular spaces of another column. Example: see the following image, red rectangle and blue rectangle are separated spaces
2. **Row**: words are horizontal alignment are in the same row, this is just enough condition because a cell may be a multi-line cell and other cells in the same row may not. For example: the fourth cell in the yellow line has two lines, word "Ledgers table" and other words are not horizontal alignment but it is still considered is in the same line. In my solution, i simply assume that content in a cell only is single-line content. Different lines in a cell are different rows. So the yellow rectangle in my solution contains two rows: 1. {"Ledger\_ID" , "|" , "Sales Ledger Account" , "FK to this customer's record to"} 2.  {NULL , NULL , NULL , "Ledgers table"}



**ICEpdf API [2]**

Our work is based one the data returned by ICEpdf API, an opensource project. To extract text from a pdf file, ICEpdf API provides 4 classes:

* Document: contains information of entire pdf file, to load a pdf file we use method Document.setUrl(url: URL)
* PageText: represent for each page in pdf Document, we can archive a specific page content by passing the index of page in method: Document.getPageText(pageIdx: int)
* LineText: represents a group of text chunks that are horizontal alignment but text in the same line in document may not return in the same LineText object. Example, ICEpdf may return two or more LineText object although these objects seem to be data of the same line. LineText is make up WordText objects
* WordText: represents an individual word in the document. Archiving all WordText objects of a LineText via method LineText.getWords(). WordText object has method getBounds() that returns its poistion in page.

In our work, we process directly with text chunk by using WordText objects. For each text chunk in PDF file it returns a text element with following attributes:

* x: horizontal distance from the left of the page
* y: vertical distance from the top border of the page
* maxX: equals x + width of the text chunk
* maxY: equals y+ height of the text chunk

C:\Users\ThoLuong\AppData\Local\Temp\enhtmlclip\bound-sample.png

**Preprocessing:** classifying text chunks into lines

My first approach to classify text chunks into lines is based on algorithm DBSCAN (Density-based spatial clustering of applications with noise)[3]. But i found another approach[4] which is more simple and has the similar accuracy. The following describes algorithm of the second approach:

Algorithm 1: classifying text chunks into lines

for each WordText w in words of pdf document

begin

     Line line <- lines.getLastLine()

     if line <> NULL && ( w.y or w.maxY lies between line.y and line.maxY) then

          line.y <- min (line.y, w.y)

          line.maxY <- max(line.maxY, w.maxY)

          line.add( w )

     else

          newLine <- new Line()

          newLine.y = w.y

          newLine.maxY = w.maxY

          newLine.add( w )

          lines.add( newLine )

     end if

end

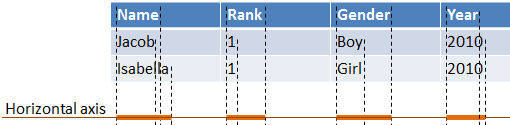
**Trap ranges**

After classifying wordText into lines, now with each line, if this line is row of data in the table, we need to identify cell for words. To do so, we focus on calculating ranges that contains all wordText of cells corresponding to each table column. We name these ranges are trap-ranges

Trap range has attributes:

* x: vertical distance from the left of page
* maxX: x + width of the range

To calculate attributes of trap-ranges, we loop through each line of page and project range of each word onto horizontal axis and union with existed ranges. After looping through all lines of page we will calculate trap-ranges and using them to identify cell data of the table. Example:



Algorithm 2: calculating trap-ranges for each pdf page:

trapRanges <- []

for each Line line in lines of page

begin

     for each WordText w in words of line

     begin

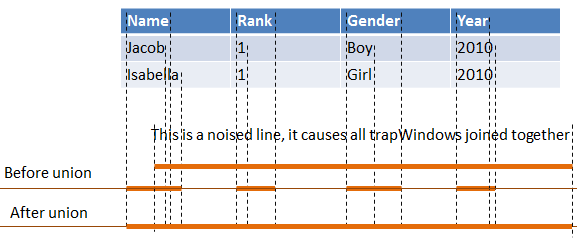
          trapRanges <- union(trapRanges , {w.x, w.maxX} ), {w.x, w.maxX} )

     end

end

**Avoid noisy data**

Algorithm 2 only works well without noisy line. Noisy lines are lines that don't belong to the table, when union them withtrapRanges , they may connect all trapRanges to single one then damage our result. Following is an example  that noisy line union with existed trapRanges:



So how to avoid noisy line in our union operation? As mentioned above, we uses threshold 20% and a blacklist to save lines that may be noisy line, after looping through all lines of page, we will re-union lines in black list t

o make the trap-ranges more accurately. Following is our algorithm to avoid noisy data when calculating trapRanges

Algorithm 3: calculating trap-ranges with noise

trapRanges <- []

blackLines <- []

maxNoisyLines <- 20 \* page.getNumberOfLines() / 100

for each Line line in lines of page

begin

     newTrapRanges = trapRanges;

     for each WordText w in words of line

     begin

          newTrapRanges <- union(newTrapRanges , {w.x, w.maxX} )

     end

     if newTrapRanges .size() < trapRanges.size() then

          if blackLines.size() < maxNoisyLines then

               blackLines.add(line)

          else

               throw exception: the document is not a "high desity of table content" document

          end if

     else

          trapRanges <- newTrapRanges

     end if

end

// reunion with lines in blacklist to make trapRanges more accurate

for each line in blackLines

     newTrapRanges = trapRanges;

     for each WordText w in words of line

     begin

          newTrapRanges <- union(newTrapRanges, {w.x, w.maxX})

     end

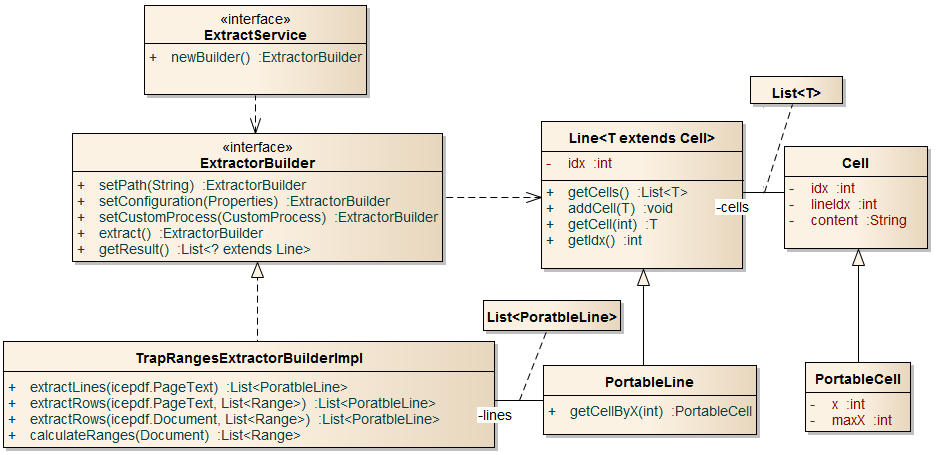
     if newTrapRanges.size == trapRanges.size then

          trapRanges <-newTrapRanges

     end if

end

**Design and implement**



The above is class diagram that describes main classes in our projects:

* **ExtractService** is an interface having method newBuilder() to create an implementation of ExtractorBuilder
* **Line**/**Cell**: in context of a page, it contains a line/words in pdf document; in context of a table it contains a row/cell in table
* **ExtractorBuilder** is the most important interface, it contains methods to initialize and extract table data from an input pdf file. Builder pattern was applied for this interface. Methods in this interface:
  + setPath: set filePath of pdf file
  + setConfiguration: set custom configurations for the builder
  + setCustomProcess: CustomProcess allows you to verify extracted table data
  + extract: do extraction
  + getResult: get result of the builder after extraction process. This method return list of rows of table
* **TrapRangesExtractorBuilderImpl**: this is a builder implementation using TrapRanges method. The class contains main methods:
  + calculateRanges: calculate trap-ranges using algorithm 3
  + extractLines: using algorithm 1 to classify text chunks into lines
  + extractRows: extract table data from a pdf page or pdf document

To extract table data from a pdf file, we use following code block:

ExtractService pdfExtractService = ExtractorServiceFactory.getExtractService("pdf");// may have other implementations for other document format, for example: csv, word...

List<Line> result = pdfExtractService.newBuilder()

                                   .setPath("seta-table.pdf")

                                   .setCustomProcess(null)

                                   .extract()

                                   .getResult();

// do some business with result

**Evaluation**

Reference:

1. <http://en.wikipedia.org/wiki/Portable_Document_Format>

2. <http://res.icesoft.org/docs/icepdf/latest/core/>

3. <http://en.wikipedia.org/wiki/DBSCAN>

4. <http://ieg.ifs.tuwien.ac.at/pub/yildiz_iicai_2005.pdf>