Extracting Text from PDF Files with Python: A Comprehensive Guide

A complete process to extract textual information from tables, images, and plain text from a PDF file

George Stavrakis · Follow

Published in Towards Data Science

17 min read · Sep 22, 2023

Photo by Giorgio Trovato on Unsplash

Introduction

In the age of Large Language Models (LLMs) and their wide-ranging applications, from simple text summarisation and translation to predicting stock performance based on sentiment and financial report topics, the importance of text data has never been greater.

There are many types of documents that share this kind of unstructured information, from web articles and blog posts to handwritten letters and poems. However, a significant portion of this text data is stored and transferred in PDF format. More specifically, it has been found that over 2 billion PDFs are opened in Outlook each year, while 73 million new PDF files are saved in Google Drive and email daily (2).

Developing, therefore, a more systematic way to process these documents and extract information from them would give us the ability to have an automated flow and better understand and utilise this vast volume of textual data. And for this task, of course, our best friend could be none other than Python.

However, before we start our process, we need to specify the different types of PDFs that are around these days, and more specifically, the three most frequently appearing:

1. **Programmatically generated PDFs**: These PDFs are created on a computer using either W3C technologies such as HTML, CSS, and Javascript or another software like Adobe Acrobat. This type of file can contain various components, such as images, text, and links, which are all searchable and easy to edit.

2. **Traditional scanned documents**: These PDFs are created from non-electronic mediums through a scanner machine or a mobile app. These files are nothing more than a collection of images stored together in a PDF file. Saying that, the elements appearing in these images, like the text, or links can't be selected or searched. Essentially, the PDF serves as a container for these images.

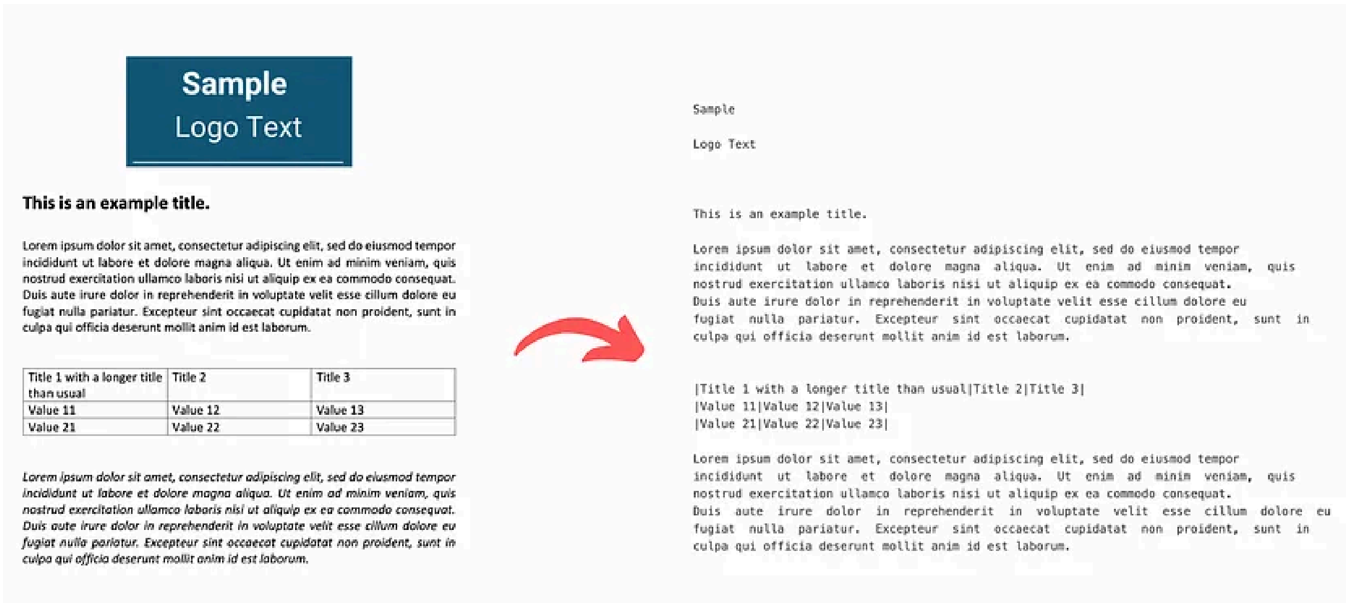
3. **Scanned documents with OCR:** In this case, Optical Character Recognition (OCR) software is employed after scanning the document to identify the text within each image in the file, converting it into searchable and editable text. Then the software adds a layer with the actual text to the image, and that way you can select it as a separate component when browsing the file. (3)

Even though nowadays more and more machines have OCR systems installed in them that identify the text from scanned documents, there are still documents that contain full pages in an image format. You've probably seen that when you read a great article and try to select a sentence, but instead you select the whole page. This can be a result of a limitation in the specific OCR machine or its complete absence. That way, in order not to leave this information undetected in this article, I tried to create a process that also considers these cases and takes the most out of our precious and information-rich PDFs.

The Theoretical Approach

With all these different types of PDF files in mind and the various items that compose them, it's important to perform an initial analysis of the layout of the PDF to identify the proper tool needed for each component. More specifically, based on the findings of this analysis, we will apply the appropriate method for extracting text from the PDF, whether it's text rendered in a corpus block with its metadata, text within images, or structured text within tables. In the scanned document without OCR, the approach that identifies and extracts text from images will perform all the heavy lifting. The output of this process will be a Python dictionary containing information extracted for each page of the PDF file. Each key in this dictionary will present the page number of the document, and its corresponding value will be a list with the following 5 nested lists containing:

1. The text extracted per text block of the corpus
2. The format of the text in each text block in terms of font family and size
3. The text extracted from the images on the page
4. The text extracted from tables in a structured format
5. The complete text content of the page



Sample
Logo Text

This is an example title.

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Title 1 with a longer title than usual	Title 2	Title 3
Value 11	Value 12	Value 13
Value 21	Value 22	Value 23

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

```
Sample

Logo Text


This is an example title.

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
incididunt  ut  labore  et  dolore  magna  aliqua.  Ut  enim  ad  minim  veniam,  quis
nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.
Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu
fugiat  nulla  pariatur.  Excepteur  sint  occaecat  cupidatat  non  proident,  sunt  in
culpa qui officia deserunt mollit anim id est laborum.


|Title 1 with a longer title than usual|Title 2|Title 3|
|Value 11|Value 12|Value 13|
|Value 21|Value 22|Value 23|

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
incididunt  ut  labore  et  dolore  magna  aliqua.  Ut  enim  ad  minim  veniam,  quis
nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.
Duis  aute  irure  dolor  in  reprehenderit  in  voluptate  velit  esse  cillum  dolore  eu
fugiat  nulla  pariatur.  Excepteur  sint  occaecat  cupidatat  non  proident,  sunt  in
culpa qui officia deserunt mollit anim id est laborum.
```

Image by the author

That way, we can achieve a more logical separation of the extracted text per source component, and it can sometimes help us to more easily retrieve information that usually appears in the specific component (e.g., the company name in a logo image). In addition, the metadata extracted from the text, like the font family and size, can be used to easily identify text headers or highlighted text of greater importance that will help us further separate or post-process the text in multiple different chunks. Lastly, retaining the structured table information in a way that an LLM can understand will enhance significantly the quality of inferences made about relationships within the extracted data. Then these results can be composed as an output the all the textual information that appeared on each page.

You can see a flowchart of this approach in the images below.

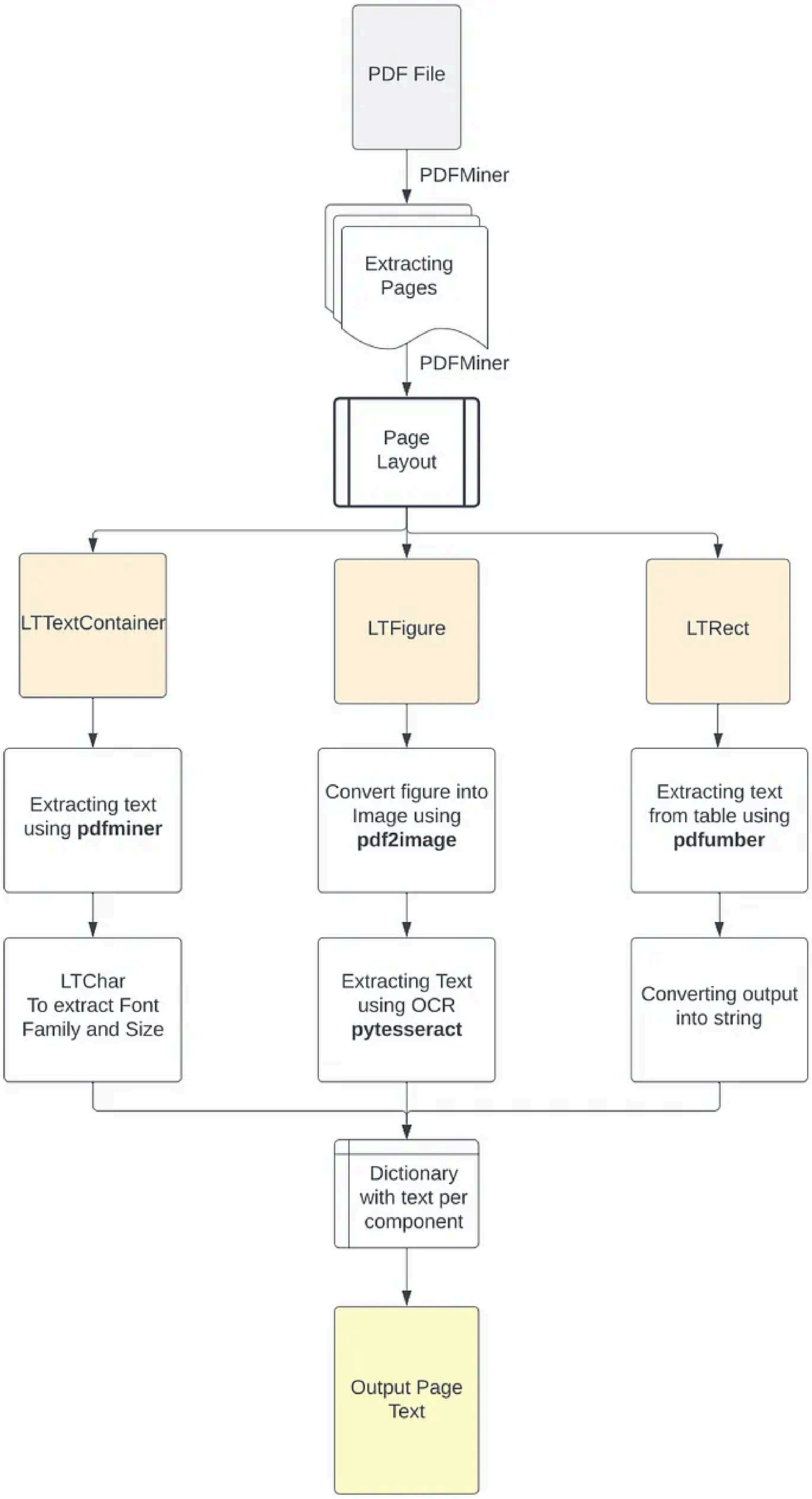

PDF File
PDFMiner
Extracting Pages
PDFMiner
Page Layout
LTTextContainer
LTFigure
LTRect
Extracting text using **pdfminer**
Convert figure into Image using **pdf2image**
Extracting text from table using **pdfumber**
LTChar To extract Font Family and Size
Extracting Text using OCR **pytesseract**
Converting output into string
Dictionary with text per component
Output Page Text

Image by the author

Installation of all the necessary libraries

Before we start this project, though, we should install the necessary libraries. We assume that you have Python 3.10 or above installed on your machine. Otherwise, you can install it from here. Then let's install the following libraries:

PyPDF2: To read the PDF file from the repository path.

```
pip install PyPDF2
```

Pdfminer: To perform the layout analysis and extract text and format from the PDF. (the .six version of the library is the one that supports Python 3)

```
pip install pdfminer.six
```

Pdfplumber: To identify tables in a PDF page and extract the information from them.

```
pip install pdfplumber
```

Pdf2image: To convert the cropped PDF image to a PNG image.

```
pip install pdf2image
```

PIL: To read the PNG image.

```
pip install Pillow
```

Pytesseract: To extract the text from the images using OCR technology

This is a little trickier to install because first, you need to install Google Tesseract OCR, which is an OCR machine based on an LSTM model to identify line recognition and character patterns.

You can install this on your machine if you are a Mac user through **Brew** from your terminal, and you are good to go.

```
brew install tesseract
```

For Windows users, you can follow these steps to install the link. Then, when you download and install the software, you need to add their executable paths to Environment Variables on your computer. Alternatively, you can run the following commands to directly include their paths in the Python script using the following code:

```
pytesseract.pytesseract.tesseract_cmd = r'C:\Program Files\Tesseract-OCR\tesser
```

Then you can install the Python library

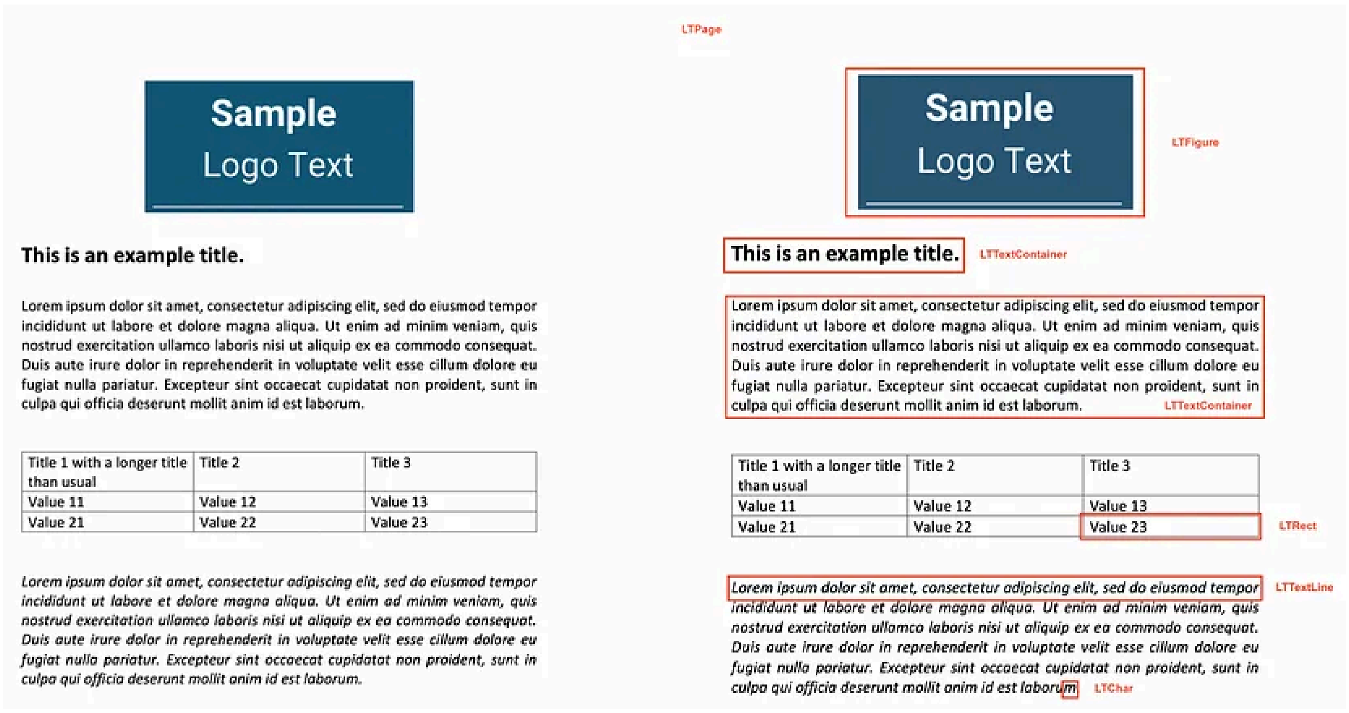
```
pip install pytesseract
```

Lastly, we will import all the libraries at the beginning of our script.

```python
# To read the PDF
import PyPDF2
# To analyze the PDF layout and extract text
from pdfminer.high_level import extract_pages, extract_text
from pdfminer.layout import LTTextContainer, LTChar, LTRect, LTFigure
# To extract text from tables in PDF
import pdfplumber
# To extract the images from the PDFs
from PIL import Image
from pdf2image import convert_from_path
# To perform OCR to extract text from images
import pytesseract
# To remove the additional created files
import os
```

So now we are all set. Let's move to the fun part.

Document's Layout Analysis with Python



This is an example title.

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Title 1 with a longer title than usual	Title 2	Title 3
Value 11	Value 12	Value 13
Value 21	Value 22	Value 23

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Image by the author

For the preliminary analysis, we used the PDFMiner Python library to separate the text from a document object into multiple page objects and then break down and examine the layout of each page. PDF files inherently lack structured information, such as paragraphs, sentences, or words as seen by the human eye. Instead, they understand only the individual characters of the text along with their position on the page. That way, the PDFMiner tries to reconstruct the content of the page into its individual characters along with their position in the file. Then, by comparing the distances of those characters from others it composes the appropriate words, sentences, lines, and paragraphs of text. (4) To achieve that, the library:

Separates the individual pages from the PDF file using the high-level function extract_pages() and converts them into **LTPage** objects.

Then for each LTPage object, it iterates from each element from top to bottom and tries to identify the appropriate component as either:

- **LTFigure** which represents the area of the PDF that can present figures or images that have been embedded as another PDF document in the page.
- **LTTextContainer** which represents a group of text lines in a rectangular area is then analysed further into a list of **LTTextLine** objects. Each one of them represents a list of **LTChar** objects, which store the single characters of text along with their metadata. (5)
- **LTRect** represents a 2-dimensional rectangle that can be used to frame images, and figures or create tables in an LTPage object.

Therefore, based on this reconstruction of the page and the classification of its elements either into **LTFigure**, which contains the images or figures of the page, **LTTextContainer**, which represents the textual information of the page, or **LTRect**, which will be a strong indication of the presence of a table, we can apply the appropriate function to better extract the information.

```
for pagenum, page in enumerate(extract_pages(pdf_path)):

    # Iterate the elements that composed a page
    for element in page:

        # Check if the element is a text element
        if isinstance(element, LTTextContainer):
            # Function to extract text from the text block
            pass
            # Function to extract text format
            pass
        # Check the elements for images
        if isinstance(element, LTFigure):
            # Function to convert PDF to Image
            pass
            # Function to extract text with OCR
            pass

        # Check the elements for tables
        if isinstance(element, LTRect):
            # Function to extract table
            pass
            # Function to convert table content into a string
            pass
```

So now that we understand the analysis part of the process, let's create the functions needed to extract the text from each component.

Define the function to extract text from PDF

From here on, extracting text from a text container is really straightforward.

```
# Create a function to extract text

def text_extraction(element):
    # Extracting the text from the in-line text element
    line_text = element.get_text()

    # Find the formats of the text
    # Initialize the list with all the formats that appeared in the line of tex
    line_formats = []
    for text_line in element:
        if isinstance(text_line, LTTextContainer):
            # Iterating through each character in the line of text
            for character in text_line:
                if isinstance(character, LTChar):
                    # Append the font name of the character
                    line_formats.append(character.fontname)
                    # Append the font size of the character
                    line_formats.append(character.size)
    # Find the unique font sizes and names in the line
    format_per_line = list(set(line_formats))
```

```
        # Return a tuple with the text in each line along with its format
        return (line_text, format_per_line)
```

So to extract text from a text container, we simply use the **get_text**() method of the LTTextContainer element. This method retrieves all the characters that make up the words within the specific corpus box, storing the output in a list of text data. Each element in this list represents the raw textual information contained in the container.

Now, to identify this text's format, we iterate through the LTTextContainer object to access each text line of this corpus individually. In each iteration, a new **LTTextLine** object is created, representing a line of text in this chunk of corpus. We then examine whether the nested line element contains text. If it does, we access each individual character element as LTChar, which contains all the metadata for that character. From this metadata, we extract two types of formats and store them in a separate list, positioned correspondingly to the examined text:

- The font family of the characters, including whether the character is in bold or italic format
- The font size for the character

Generally, characters within a specific chunk of text tend to have consistent formatting unless some are highlighted in bold. To facilitate further analysis, we capture the unique values of text formatting for all characters within the text and store them in the appropriate list.

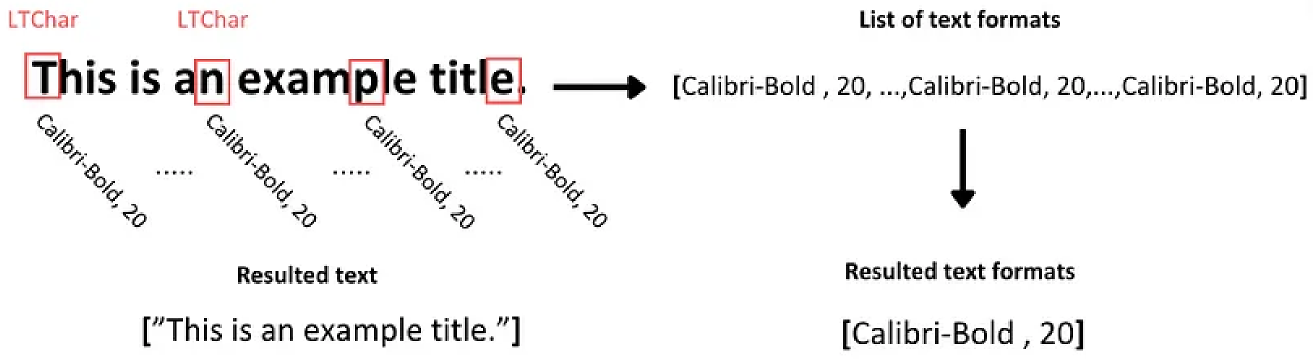


Image by the author

Define the function to extract text from Images

Here I believe it is a more tricky part.

How to handle text in images found in PDF?

Firstly, we need to establish here that image elements stored in PDFs are not in a different format from the file, such as JPEG or PNG. That way in order to apply OCR software on them we need first to separate them from the file and then convert them into an image format.

```
# Create a function to crop the image elements from PDFs
def crop_image(element, pageObj):
    # Get the coordinates to crop the image from the PDF
    [image_left, image_top, image_right, image_bottom] = [element.x0,element.y0
    # Crop the page using coordinates (left, bottom, right, top)
    pageObj.mediabox.lower_left = (image_left, image_bottom)
    pageObj.mediabox.upper_right = (image_right, image_top)
    # Save the cropped page to a new PDF
    cropped_pdf_writer = PyPDF2.PdfWriter()
    cropped_pdf_writer.add_page(pageObj)
    # Save the cropped PDF to a new file
    with open('cropped_image.pdf', 'wb') as cropped_pdf_file:
        cropped_pdf_writer.write(cropped_pdf_file)

# Create a function to convert the PDF to images
def convert_to_images(input_file,):
    images = convert_from_path(input_file)
    image = images[0]
    output_file = "PDF_image.png"
    image.save(output_file, "PNG")

# Create a function to read text from images
def image_to_text(image_path):
    # Read the image
    img = Image.open(image_path)
    # Extract the text from the image
    text = pytesseract.image_to_string(img)
    return text
```

To achieve this, we follow the following process:

1. We use the metadata from the LTFigure object detected from PDFMiner to crop the image box, utilising its coordinates in the page layout. We then save it as a new PDF in our directory using the **PyPDF2** library.

2. Then we employ the **convert_from_file**() function from the **pdf2image** library to convert all PDF files in the directory into a list of images, saving them in PNG format.

3. Finally, now that we have our image files we read them in our script using the **Image** package of the **PIL** module and implement the **image_to_string**() function of pytesseract to extract text from the images using the tesseract OCR engine.

As a result, this process returns the text from the images, which we then save in a third list within the output dictionary. This list contains the textual information extracted from the images on the examined page.

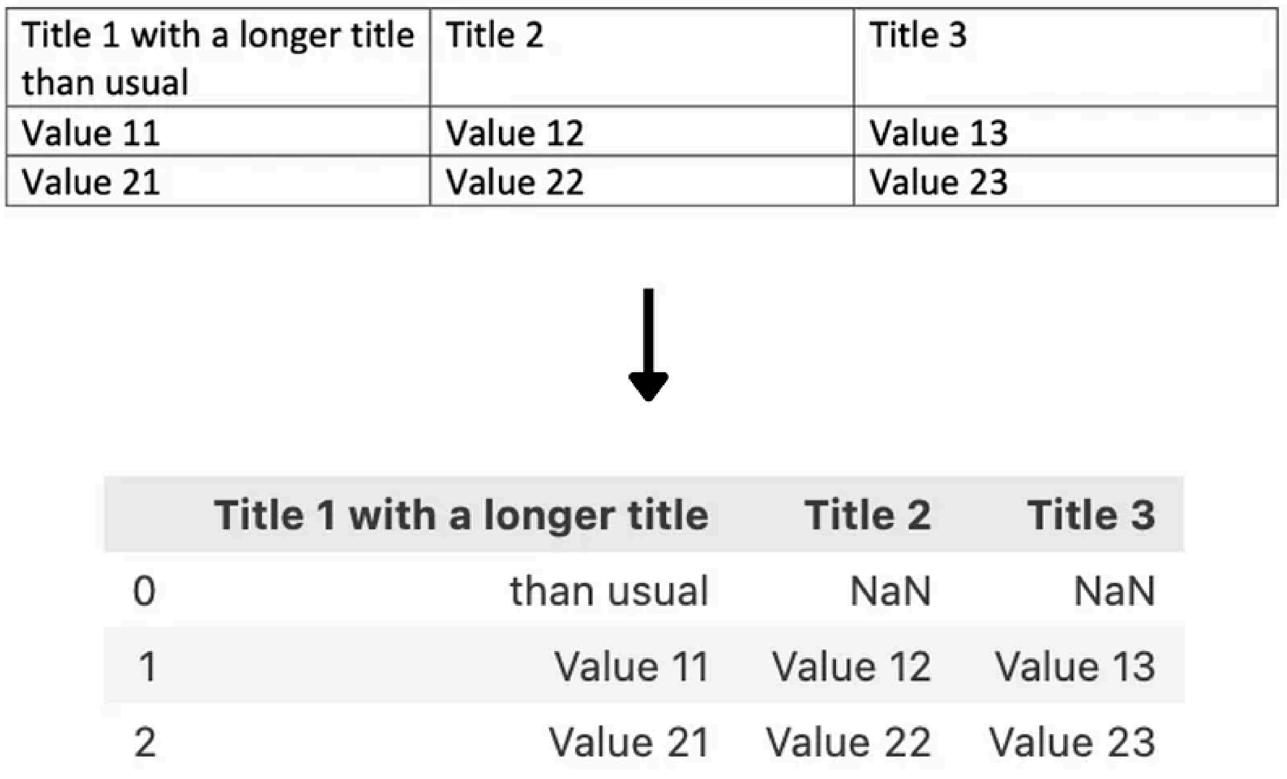
Define the function to extract text from Tables

In this section, we will extract a more logically structured text from tables on a PDF page. This is a slightly more complex task than extracting text from a corpus because we need to take into account the granularity of the information and the relationships formed between data points presented in a table.

Although there are several libraries used to extract table data from PDFs, with **Tabula-py** being one of the most well-known, we have identified certain limitations in their functionality.

The most glaring one in our opinion comes from the way that the library identifies the different rows of the table using the line-break special character \n in the table's text. This works pretty well in most of the cases but it fails to capture correctly when the text in a cell is wrapped into 2 or more rows, leading to the addition of unnecessary empty rows and losing the context of the extracted cell.

You can see the example below when we tried to extract the data from a table using tabula-py:



Title 1 with a longer title than usual	Title 2	Title 3
Value 11	Value 12	Value 13
Value 21	Value 22	Value 23

	Title 1 with a longer title	Title 2	Title 3
0	than usual	NaN	NaN
1	Value 11	Value 12	Value 13
2	Value 21	Value 22	Value 23

Image by the author

Then, the extracted information is outputted in a Pandas DataFrame instead of a string. In most cases, this can be a desirable format but in the case of transformers that take into account text, these results need to be transformed before feeding into a model.

For this reason, to tackle this task we used the **pdfplumber** library for various reasons. Firstly, it is built on pdfminer.six which we used for our preliminary analysis, meaning that it contains similar objects. In addition, its approach to table detection is based on line elements along with their intersections that construct the cell that contains the text and then the table itself. That way after we identify a cell of a table, we can extract just the content inside the cell without carrying how many rows needed to be rendered. Then when we have the contents of a table, we will format it in a table-like string and store it in the appropriate list.

```python
# Extracting tables from the page

def extract_table(pdf_path, page_num, table_num):
    # Open the pdf file
    pdf = pdfplumber.open(pdf_path)
    # Find the examined page
    table_page = pdf.pages[page_num]
```

```
        # Extract the appropriate table
        table = table_page.extract_tables()[table_num]
        return table

    # Convert table into the appropriate format
    def table_converter(table):
        table_string = ''
        # Iterate through each row of the table
        for row_num in range(len(table)):
            row = table[row_num]
            # Remove the line breaker from the wrapped texts
            cleaned_row = [item.replace('\n', ' ') if item is not None and '\n' in
            # Convert the table into a string
            table_string+=('|'+'|'.join(cleaned_row)+'|'+'\n')
        # Removing the last line break
        table_string = table_string[:-1]
        return table_string
```

To achieve that, we created two functions, **extract_table()** to extract the contents of the table into a list of lists, and **table_converter()** to join the contents of those lists in a table-like string.

In the **extract_table()** function:

1. We open the PDF file.
2. We navigate to the examined page of the PDF file.
3. From the list of tables found on the page by pdfplumber, we select the desired one.
4. We extract the content of the table and output it in a list of nested lists representing each row of the table.

In the **table_converter()** function:

1. We iterate in each nested list and clean its context from any unwanted line breaks coming from any wrapped text.
2. We join each element of the row by separating them using the | symbol to create the structure of a table's cell.
3. Finally, we add a line break at the end to move to the next row.

This will result in a string of text that will present the content of the table without losing the granularity of the data presented in it.

Adding all together

Now that we have all the components of the code ready let's add them all up to a fully functional code. You can copy the code from here or you can find it along with the example PDF in my Github repo here.

```python
# Find the PDF path
pdf_path = 'OFFER 3.pdf'

# create a PDF file object
pdfFileObj = open(pdf_path, 'rb')
# create a PDF reader object
pdfReaded = PyPDF2.PdfReader(pdfFileObj)

# Create the dictionary to extract text from each image
text_per_page = {}
# We extract the pages from the PDF
for pagenum, page in enumerate(extract_pages(pdf_path)):

    # Initialize the variables needed for the text extraction from the page
    pageObj = pdfReaded.pages[pagenum]
    page_text = []
    line_format = []
    text_from_images = []
    text_from_tables = []
    page_content = []
    # Initialize the number of the examined tables
    table_num = 0
    first_element= True
    table_extraction_flag= False
    # Open the pdf file
    pdf = pdfplumber.open(pdf_path)
    # Find the examined page
    page_tables = pdf.pages[pagenum]
    # Find the number of tables on the page
    tables = page_tables.find_tables()


    # Find all the elements
    page_elements = [(element.y1, element) for element in page._objs]
    # Sort all the elements as they appear in the page
    page_elements.sort(key=lambda a: a[0], reverse=True)
```

```
        # Find the elements that composed a page
        for i,component in enumerate(page_elements):
            # Extract the position of the top side of the element in the PDF
            pos= component[0]
            # Extract the element of the page layout
            element = component[1]

            # Check if the element is a text element
            if isinstance(element, LTTextContainer):
                # Check if the text appeared in a table
                if table_extraction_flag == False:
                    # Use the function to extract the text and format for each text
                    (line_text, format_per_line) = text_extraction(element)
                    # Append the text of each line to the page text
                    page_text.append(line_text)
                    # Append the format for each line containing text
                    line_format.append(format_per_line)
                    page_content.append(line_text)
                else:
                    # Omit the text that appeared in a table
                    pass

            # Check the elements for images
            if isinstance(element, LTFigure):
                # Crop the image from the PDF
                crop_image(element, pageObj)
                # Convert the cropped pdf to an image
                convert_to_images('cropped_image.pdf')
                # Extract the text from the image
                image_text = image_to_text('PDF_image.png')
                text_from_images.append(image_text)
                page_content.append(image_text)
                # Add a placeholder in the text and format lists
                page_text.append('image')
                line_format.append('image')

            # Check the elements for tables
            if isinstance(element, LTRect):
                # If the first rectangular element
                if first_element == True and (table_num+1) <= len(tables):
                    # Find the bounding box of the table
                    lower_side = page.bbox[3] - tables[table_num].bbox[3]
                    upper_side = element.y1
                    # Extract the information from the table
                    table = extract_table(pdf_path, pagenum, table_num)
                    # Convert the table information in structured string format
                    table_string = table_converter(table)
                    # Append the table string into a list
                    text_from_tables.append(table_string)
                    page_content.append(table_string)
                    # Set the flag as True to avoid the content again
                    table_extraction_flag = True
                    # Make it another element
                    first_element = False
                    # Add a placeholder in the text and format lists
                    page_text.append('table')
                    line_format.append('table')

                # Check if we already extracted the tables from the page
                if element.y0 >= lower_side and element.y1 <= upper_side:
                    pass
                elif not isinstance(page_elements[i+1][1], LTRect):
                    table_extraction_flag = False
                    first_element = True
                    table_num+=1


        # Create the key of the dictionary
        dctkey = 'Page_'+str(pagenum)
        # Add the list of list as the value of the page key
        text_per_page[dctkey]= [page_text, line_format, text_from_images,text_from_

    # Closing the pdf file object
    pdfFileObj.close()

    # Deleting the additional files created
    os.remove('cropped_image.pdf')
    os.remove('PDF_image.png')

    # Display the content of the page
    result = ''.join(text_per_page['Page_0'][4])
    print(result)
```

The script above will:

Import the necessary libraries.

Open the PDF file using the **pyPDF2** library.

Extract each page of the PDF and iterate the following steps.

Examine if there are any tables on the page and create a list of them using **pdfplumner.**

Find all the elements nested in the page and sort them as they appeared in its layout.

Then for each element:

Examine if it is a text container, and does not appear in a table element. Then use the **text_extraction**() function to extract the text along with its format, else pass this text.

Examine if it is an image, and use the **crop_image**() function to crop the image component from the PDF, convert it into an image file using the **convert_to_images**(), and extract text from it using OCR with the **image_to_text**() function.

Examine if it is a rectangular element. In this case, we examine if the first rect is part of a page's table and if yes, we move to the following steps:

1. Find the bounding box of the table in order not to extract its text again with the text_extraction() function.
2. Extract the content of the table and convert it into a string.
3. Then add a boolean parameter to clarify that we extract text from Table.
4. This process will finish after the last LTRect that falls into the bounding box of the table and the next element in the layout is not a rectangular object. (All the other objects that compose the table will be passed)

The outputs of the process will be stored in 5 lists per iteration, named:

1. page_text: contains the text coming from text containers in the PDF (placeholder will be placed when the text was extracted from another element)
2. line_format: contains the formats of the texts extracted above (placeholder will be placed when the text was extracted from another element)
3. text_from_images: contains the texts extracted from images on the page
4. text_from_tables: contains the table-like string with the contents of tables
5. page_content: contains all the text rendered on the page in a list of elements

All the lists will be stored under the key in a dictionary that will represent the number of the page examined each time.

Afterwards, we will close the PDF file.

Then we will delete all the additional files created during the process.

Lastly, we can display the content of the page by joining the elements of the page_content list.

Conclusion

This was one approach that I believe uses the best characteristics of many libraries and makes the process resilient to various types of PDFs and elements that we can encounter, with PDFMiner however do the most of the heavy lifting. Also, the information regarding the format of the text can help us with the identification of potential titles that can separate the text into distinct logical sections rather than just content per page and can help us to identify the text of greater importance.

However, there will always be more efficient ways to do this task and even though I believe that this approach is more inclusive, I am really looking forward to discussing with you new and better ways of tackling this problem.

📖 References:

1. https://www.techopedia.com/12-practical-large-language-model-llm-applications

2. https://www.pdfa.org/wp-content/uploads/2018/06/1330_Johnson.pdf

3. https://pdfpro.com/blog/guides/pdf-ocr-guide/#:~:text=OCR technology reads text from, a searchable and editable PDF.

4. https://pdfminersix.readthedocs.io/en/latest/topic/converting_pdf_to_text.html#id1

5. https://github.com/pdfminer/pdfminer.six

Data Science Programming Data NLP