Project Log

Problem Description:

To achieve the desired objective in an effective manner, we need to create a framework for converting unstructured PDF documents into a structured format. Such a structured format can then be leveraged to provide the required accessibility-related
functionalities.

Key Objectives:

Phase 1: This phase will focus on handling ‘text’ in PDFs. The system is expected to take any PDF as input and execute the
following steps:

1. Convert PDF into an editable format (e.g., .txt, HTML, MS-Word)
2. Leverage NLP to identify and tag the following components within the document:
3. Document details: Title, publication date, authors
4. Content: Chapters (title and number) and paragraphs; Title of tables, charts, images, and other graphics
5. References: Footnotes, sources, copyright messages, and other references

Flow Description (Meeting -I)

The flow of the entire problem statement can be divided into two parts.

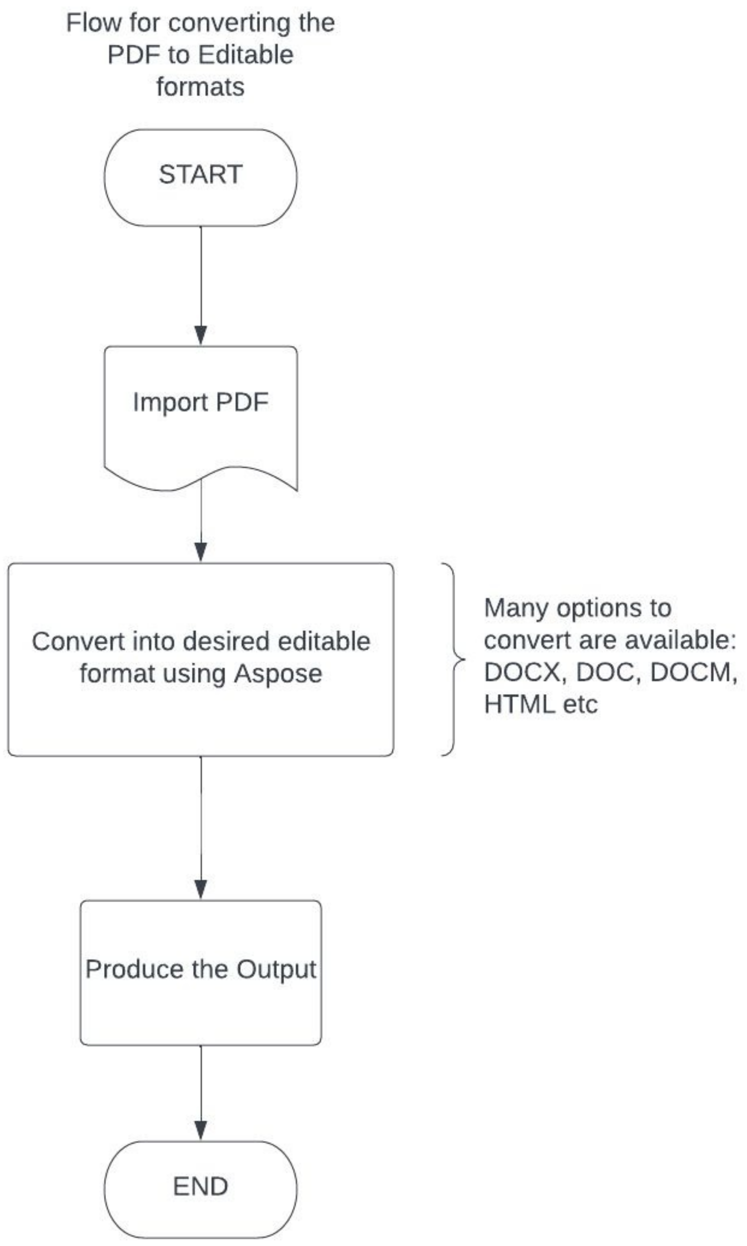
1. Taking the Input PDF and converting it into a desired editable format
2. Extracting document details using NLP

Part 1:

ASPOSE.PDF is one of the most useful APIs which can be used for converting a PDF document into other formats. Aspose adds a PDF manipulation component to our existing .NET projects and would be the go-to-api in this problem statement. However Aspose.PDF works on a subscription basis and there are free alternatives like

1. ZetPDF
2. PDFium
3. Api2PDF

These free alternatives however are not as effective as Aspose.


Flow for converting the PDF to Editable formats
START
Import PDF
Convert into desired editable format using Aspose
Many options to convert are available: DOCX, DOC, DOCM, HTML etc
Produce the Output
END

Part 2:

There are several methods available to extract data from a pdf file and provide structure to an unstructured document. Let us list them all out and explain them in brief.

1. Using the Google Docs Api

   The google docs api is an excellent resource for performing many operations, one of them being providing and defining a proper structure to the document.The documentation is sufficient for understanding and implementing the entire API. The pdf file which was taken as input needs to be converted to a Google DOC file.

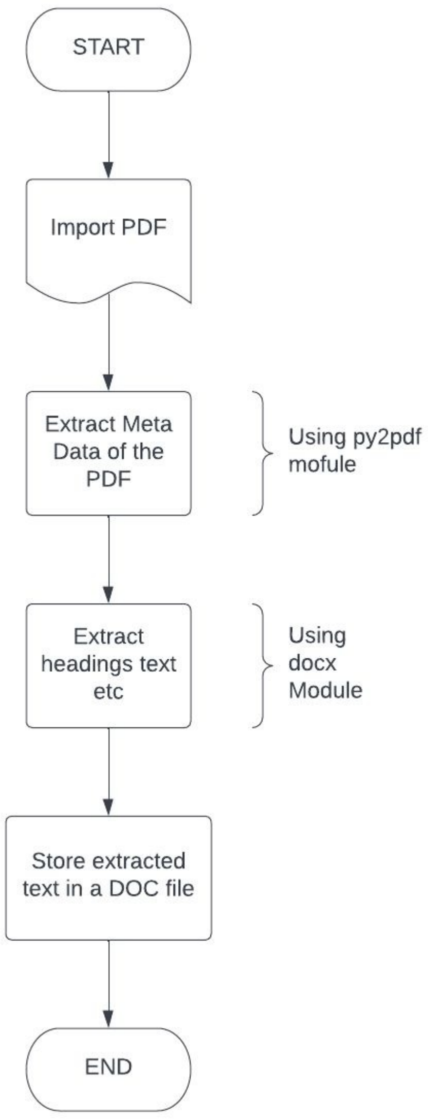
2. Directly extracting data from the PDF file

   Python modules such as Py2PDF are excellent resources for extracting data from pdf files. Modules like docx are able to extract text with ease from any pdf file. However structuring of the extracted data may be a problem due to the inconsistent interaction between the python libraries and pdf files like reading indents as entire paragraphs, irregular spacing, etc.

3. Converting the PDF file into an HTML file and extracting the structure

   Extracting structure from an HTML file is comparatively easier and HTML tags help in classifying contents with minimum effort. Python modules like docx can be easily utilized to extract text and structure from an HTML file. There may be cases of inconsistent interaction between the converted HTML file and the python libraries, However this can be overlooked and improved upon as no data loss occurs.

Flow for extracting data


START
Import PDF
Extract Meta Data of the PDF
Using py2pdf mofule
Extract headings text etc
Using docx Module
Store extracted text in a DOC file
END

Points of Discussion (Meeting -II):

1. **Workflow Structure**

1. Create weekly tasks and aim at finishing the task in the decided time depending upon the task
2. Weekly meetings every wednesday
    - 2.1 To be discussed in this meeting-
        - a. updates on work completed in the previous week
        - b. next task and the time required to finish it
        - c. issues faced and possible solutions

*if a new task requires more/less time meeting can be shifted accordingly
*if a sudden meeting is to be called a notice at least 1 day prior is to be given so that necessary changes in the schedule can be made (applicable for both parties)

2. **Things Needed**

1. Detailed information about the Tech Stack (includes all the concerned technologies that have been used up till this point)
2. At Least 1-2 hours of time commitment whenever a meeting is called.
3. All the methodologies and approaches which have been tried and tested uptill now.
4. Testing of certain operations which can be effectively tested by the company.
5. Specific sample data required at different stages of development

3. **Approach methods**

<u>Plan -A: Using Web Scraping APIs:</u>

Scrapers are basically used to extract data from specified sources. Web scraping tools work with specific sources such as applications, operating systems, or browsers.

Scraper API will be used to extract data from HTML. To combine multiple sets of existing data into a single source, avoiding costly duplication of work.

Some of the APIs which can be used are:

1.WebScrapingAPI:

HTML formatted responses
latest anti-bot detection tools
handles proxies, browsers, and CAPTCHAs
integration with any development language
Javascript rendering

2. ScraperAPI

extracted data in HTML, JPEG, or plain text formats
speed and reliability
automatically retries failed requests
full customization

3. ScrapingAnt

Output preprocessing — analyze and work with direct text output without dealing with HTML
Chrome page rendering
low latency rotating proxies
Javascript rendering
high-end AWS solutions
high speed and availability

4. ScrapingBot

scrapes and extracts valuable data from any webpage without getting blocked
extracts and parse the data in structured JSON
fast and reliable
easy integration
Javascript rendering
handles proxies and browsers

Plan -B: Develop code using RegEx and Beautifulsoup or similar:

Beautiful Soup is a Python library that extracts data from HTML and XML files. It collaborates with the parser to provide idiomatic ways to navigate, search, and modify the parse tree.
We can use Beautiful Soup to efficiently extract titles and paragraphs from any HTML document given that we have a proper HTML format

```
data = ''
for data in soup.find_all("p"):
   print(data.get_text())
```

Beautiful Soup is a powerful web scraping library in python and since we are using HTML as our base format it can be used.

We can also use regex to extract headings and paragraphs from the given HTML document by matching and removing all tags

The prerequisite of this method is that the converted html document needs to be precise.

This plan can be divided into 2 sub phases.

1. Identifying and testing out the different packages/libraries that can be utilized for scraping
2. Building a model using HEPS or similar modified approaches

What is HEPS?

HEPS stands for Heading based Page Segmentation which simulates the behavior of human readers. The HTML document is transformed into a DOM (Document Object Model) Tree which facilitates the page segmentation and data extraction becomes possible.

Final Methodology Used and Notable Outcomes

Requirements

The following Python libraries are required to run this project:
pandas
os
zipfile
bs4 (BeautifulSoup)
csv
re

Installation

To install the required libraries, run the following command:

```
pip install pandas os zipfile beautifulsoup4 csv regex
```

Usage

Here's a brief overview of how to use your project:
Clone the repository to your local machine.
Install the required libraries (see above).
Run the Python script(s) in the repository.

Contributing

If you would like to contribute to this project, please follow these steps:
Fork the repository.
Create a new branch for your feature or bug fix.
Make your changes and commit them with a descriptive message.
Push your changes to your fork.
Create a pull request.

Methodology

The basic steps that we have included in developing our framework that produces a final tagged html document are as follows

1. Accept a pdf file as input
2. Convert the pdf file into
    a. Html file using cloud convert api
    b. Doc file using ilovepdf api (powered arpyse.net)
3. Using NLP extract the following features from the html file and store them in a csv file
    a. Section headings
    b. Figure headings
    c. Table headings
4. Using NLP extract the following the features from the doc file and store them in a separate csv file
    a. Header
    b. Footer
5. Using the csv file containing the extracted features from the html file tag the final html document and produce the output.

Approach for extracting header and footer

1)Using BeautifulSoup and zipfile

This approach processes a collection of Microsoft Word DOCX files in a specified input directory and extracts the header and footer from each file. It then writes the extracted header and footer to a corresponding CSV file in an output directory. The code checks if the header and footer XML files exist in the zip file, and if so, uses BeautifulSoup to parse the XML files and extract the text content from them. The code also defines functions to check if a string is empty or contains only whitespace characters. It then loops through all files in the input directory and extracts the header and footer text content from each DOCX file. The extracted text content is then written to a CSV file with an appropriate header row, and with the "is_header" and "is_footer" flags set accordingly. If the header and footer text are empty, an empty row is written to the CSV file with both flags set to false. The output CSV files are created in a specified output directory.
Reference file: header_footer_extraction.py

2)Using OCR

The first step of this approach involves using a PDF library such as PyPDF2 or pdftotext to load the PDF file into the code. Once the PDF file is loaded, the pages of the PDF are converted into temporary images using a PDF library such as PyMuPDF.

Next, a copy of each page is created using the Python Imaging Library (PIL) or OpenCV. Note that every page has 2 corresponding images - one in color and one in grayscale.
The next step involves using a thresholding algorithm such as Otsu's method or adaptive thresholding to convert the first copy of the image into black and white pixels. This step essentially creates a binary image where black pixels represent text and white pixels represent the background.
Once the binary image is created, we traverse the pixels from bottom up, looking for the first set of black pixels. When we encounter the first set of black pixels, we check if the pixels following the black set of pixels are completely white for multiple lines. If they are, then we create a frame that includes the black pixels. We can use a simple algorithm to find the bounding box of the black pixels.
In the second image, we focus on this frame only and use Optical Character Recognition (OCR) tools such as Tesseract or Google Cloud Vision to extract the text from this frame. The extracted text can then be stored in a CSV file.
Overall, this approach involves converting the PDF pages into temporary images, converting these images into binary images using thresholding, finding the frame that includes the text using a simple algorithm, and finally extracting the text using OCR tools. This approach can be useful when dealing with PDFs that contain images, tables, or other complex elements, as it allows us to focus only on the text and extract it in a structured format.
Reference file: header_footer_ocr.py

The code processes a collection of Microsoft Word DOCX files located in an input directory and extracts the header and footer from each file. It then writes the extracted header and footer to a corresponding CSV file in an output directory.

The code uses the following libraries:

os: to work with file paths and directories.
zipfile: to open the DOCX file as a zip file and access its contents.
BeautifulSoup: to parse the XML files inside the DOCX file and extract text content from them.
csv: to write the extracted header and footer to a CSV file.
Here are the main steps of the code:

Define a function extract_header_footer that takes a path to a DOCX file as input and returns the header and footer text content from the file as a tuple. The function first opens the DOCX file as a zip file and checks if the header and footer XML files exist in the zip file. If they exist, the function uses BeautifulSoup to parse the XML files and extract the text content from them. The function then returns the header and footer text content as a tuple.

Define a function is_empty that takes a string as input and returns True if the string is empty or contains only whitespace characters, and False otherwise.

Define input and output directories as input_dir and output_dir, respectively. If the output directory does not exist, create it using os.makedirs(output_dir).

Loop through all files in the input directory using os.listdir(input_dir). For each file, check if it has a .docx extension using file_name.endswith('.docx').

If the file is a DOCX file, extract the header and footer text content from the file using the extract_header_footer function. Construct the output file path by replacing the extension with .csv.

Open the output file using open(output_file, 'w', newline=''). Write the header row to the file using csv.writer.writerow(['Text', 'is_header', 'is_footer']).

If the header text is not empty, write it to the CSV file with the is_header flag set to True. If the footer text is not empty, write it to the CSV file with the is_footer flag set to True. If both header and footer text are empty, write an empty row to the CSV file with both is_header and is_footer flags set to False. The row is written using csv.writer.writerow().

The loop continues with the next file in the input directory, until all files have been processed.

Approach for extracting section headings.

The extracted section headings will be tagged as **"s_heading"** in the final html document.
The approach that we have used for extracting the headings is as follows

This Python code reads HTML files from a directory called "html_db", extracts headings from each file, and saves the heading data into a corresponding CSV file. The CSV files are stored in a directory called "results/csv_db", which is created if it doesn't already exist.

The code uses the following libraries:

re for regular expressions
BeautifulSoup for parsing HTML
pandas for working with data in tabular format

os for file handling
The code defines a list called "pool" that contains some common words used in headings such as "Introduction", "Abstract", "Conclusion", etc. It also defines a function called "is_heading" that takes in a text string and checks if it contains any of the words in the "pool" list.

The code then loops through all the HTML files in the "html_db" directory, loads each file using the open() function, and parses the HTML using BeautifulSoup. It then searches for all div elements with specific classes and extracts the text of those elements, checks if each extracted text string is a heading by calling the "is_heading" function, and saves the data into a list called "data".

The code then creates a Pandas DataFrame using the "data" list and assigns column names based on the information extracted from the headings. It then saves the DataFrame into a CSV file with the same name as the corresponding HTML file in the "results/csv_db" directory.

Finally, the code prints out a message for each HTML file indicating that it has been converted to a CSV file.

Approach for extracting table headings.

This Python code reads CSV files containing headings data from a directory called "results/csv_db", identifies table headings from the headings data, and marks those rows in the CSV files.

The code uses the following libraries:

pandas for working with data in tabular format
os for file handling
The code defines a list called "pool" that contains some common words used in table headings such as "Table", "Tab", etc.

The code then loops through all the CSV files in the "results/csv_db" directory, loads each file using the pd.read_csv() function, and adds a new column called "is_table_heading" with the default value of "False" to each DataFrame.

It then loops through each row in the DataFrame and splits the text into words. If any of the first three words contain a word from the "pool" list, the code marks the corresponding row as a table heading by setting the value of the "is_table_heading" column to "True".

If the text is already marked as a heading or a figure heading, the code removes the corresponding marks by setting the value of the "is_heading" and "is_figure_heading" columns to "False".

Finally, the updated DataFrame is saved into the same CSV file, and the code prints out a message for each CSV file indicating that it has been updated with table headings.

Approach for extracting figure headings

This code updates CSV files in a directory containing HTML parsing results by identifying whether each text in the CSV file is a figure heading or not.

The code loads a list of CSV files from a directory named "results/csv_db", and then iterates through each file. For each CSV file, the code reads it into a pandas DataFrame using pd.read_csv(). Then, the code adds a new column to the DataFrame named "is_figure_heading" and initializes all the rows with False.

Next, the code loops through each row in the DataFrame and checks whether any of the first three words in the "Text" column contain a word from the "pool" list, which contains variations of the word "figure". If a row contains a figure heading, the code sets the corresponding "is_figure_heading" column to True.

Additionally, if the text is already marked as a "heading" in the "is_heading" column, the code marks the row as not being a heading anymore by setting the corresponding "is_heading" column to False.

Finally, the updated DataFrame is saved back to the original CSV file using df.to_csv(), which overwrites the original file.

The code then prints a message indicating that the CSV file has been updated with figure headings.

Notable Outcomes

- The HTML to PDF conversion requires more than one api as the html document generated from the cloud convert api
- The project at the current state does not require a model and can be completed by using a rule based approach

Team Members

Ankur Kulkarni	kulankur5@gmail.com
Ishaan Mane	20ishaanmane@gmail.com
Pragya Shukla	pragyas0211@gmail.com