

Lessoncode1(PDF Summary chat)

This project is a **multi-functional AI assistant** that allows users to:

1. **Upload a PDF** → The AI extracts text (including OCR for images).
2. **Chat with the extracted content** → Users can ask questions based on the document.
3. **Render AI-generated HTML or Markdown** → If the AI response contains structured content (like an HTML page or Markdown-formatted text), it can be previewed in a browser.

The project consists of **three Python files**, each responsible for a specific task:

File	Purpose
pdf_chatbot.py	Extracts text from PDFs, applies OCR to images, and stores extracted text.
html_renderer.py	Detects whether AI-generated content is HTML or Markdown and renders it in a browser.
pdf_chat_interface.py	Provides a web-based chatbot interface where users can upload PDFs, ask questions, and preview structured responses.

This project enhances **AI-driven document interaction**, making it easier to extract knowledge from PDFs while **properly displaying formatted AI responses**.

Now, let's go into **detailed explanations of each file**

Extracting Text from PDFs (pdf_chatbot.py)

This file is responsible for:

- **Extracting plain text** from PDFs.
- **Applying OCR to images** inside the PDF to capture embedded text.
- **Saving and reloading extracted text** to avoid redundant processing.

How It Works

```
import fitz # PyMuPDF
import pytesseract
from PIL import Image
```

```
import tempfile
import os
```

- **PyMuPDF** (**fitz**) extracts text from PDFs.
- **Pytesseract** performs OCR on images inside the PDF.
- **PIL** processes extracted images before OCR.

Extracting Text (Including OCR)

```
def extract_text_from_pdf(pdf_path):
    """Extracts text from a PDF file using OCR if necessary."""
    doc = fitz.open(pdf_path)
    text = ""

    for page in doc:
        text += page.get_text("text") + "\n"

        img_list = page.get_images(full=True)
        for img_index, img in enumerate(img_list):
            xref = img[0]
            base_image = doc.extract_image(xref)
            image_bytes = base_image["image"]

            with tempfile.NamedTemporaryFile(delete=False, suffix=".png") as temp_img:
                temp_img.write(image_bytes)
                temp_img_path = temp_img.name

            img = Image.open(temp_img_path)
            text += pytesseract.image_to_string(img) + "\n"

    return text.strip()
```

- Extracts text **directly from PDF pages**.
- Extracts **images from the PDF** and applies OCR.
- **Combines both sources** into a final text output.

Saving & Loading Extracted Text

```
def save_pdf_text(text, output_file="pdf_text.txt"):
    with open(output_file, "w", encoding="utf-8") as f:
        f.write(text)

def load_pdf_text(file_path="pdf_text.txt"):
    if os.path.exists(file_path):
        with open(file_path, "r", encoding="utf-8") as f:
            return f.read()
    return ""
```

- **Saves extracted text** for future queries.
- **Loads text** to allow ongoing chatbot conversations.

2 Detecting & Rendering HTML/Markdown (html_renderer.py)

This file ensures that **AI-generated responses** are properly formatted and previewed.

How It Works

```
import re
import tempfile
import webbrowser
import markdown
```

- **Detects if the AI response is HTML or Markdown.**
- **Renders HTML or converts Markdown to HTML** for proper display.

Detecting Response Type

```
def detect_and_render_html_or_md(response):
    """Detects HTML or Markdown in the response and renders appropriately."""
    if "<!doctype html>" in response.lower() or "<html" in response.lower():
        render_html(response)
    return "HTML preview opened in a new tab."
```

```

elif "***" in response or "# " in response: # Detects Markdown-like syntax
    render_markdown(response)
    return "Markdown preview opened in a new tab."
return "No HTML or Markdown detected."

```

- **Checks for HTML structure** (`<!DOCTYPE html>` or `<html>`).
- **Checks for Markdown formatting** (`*bold*` , `# Heading` , etc.).
- **Calls the appropriate rendering function.**

Rendering HTML

```

def render_html(html_content):
    """Saves and opens HTML in a new tab."""
    with tempfile.NamedTemporaryFile(delete=False, suffix=".html", mode="w", encoding="utf-8") as temp_file:
        temp_file.write(html_content)
        temp_file_path = temp_file.name
    webbrowser.open(f"file://{temp_file_path}")

```

- **Saves HTML in a temporary file.**
- **Opens it in a browser** for preview.

Rendering Markdown

```

def render_markdown(md_content):
    """Converts Markdown to HTML and opens it in a new tab."""
    html_content = markdown.markdown(md_content)
    render_html(html_content)

```

- **Converts Markdown to HTML** before rendering.
- **Uses the `markdown` library** to preserve formatting.

3 PDF Chatbot Web Interface (pdf_chat_interface.py)

This file creates a **Gradio-based chatbot** that allows:

1. **Uploading PDFs.**
2. **Asking questions about the extracted content.**
3. **Previewing AI-generated HTML or Markdown responses.**

Running AI Chat with PDF Context

```
def run_ollama_prompt(prompt, context, model="deepseek-r1:1.5b"):
    """Runs a chat prompt with PDF context using Ollama CLI."""
    full_prompt = f"Context:\n{context}\n\nUser Question:\n{prompt}"
    command = ["ollama", "run", model, full_prompt]

    try:
        result = subprocess.run(command, capture_output=True, text=True, c
heck=True, encoding="utf-8")
        response = result.stdout.strip()
        response = re.sub(r'<think>.*?</think>', '', response, flags=re.DOTAL
L).strip()
        return response
    except subprocess.CalledProcessError as e:
        return f"Error running Ollama: {e}"
```

- **Sends extracted PDF text as context** to the AI model.
- **Filters out** `<think>` tags to keep responses clean.

Handling HTML/Markdown Previews

```
def handle_html_preview(response):
    """Handles HTML/Markdown preview without replacing the response."""
    preview_message = detect_and_render_html_or_md(response)
    return response, preview_message
```

- **Passes AI responses to** `detect_and_render_html_or_md()` .
- **Returns a status message** indicating whether HTML/Markdown was detected.

Building the Gradio UI

```

with gr.Blocks() as app:
    gr.Markdown("# Chat with Your PDF")

    with gr.Row():
        pdf_upload = gr.File(label="Upload PDF")
        model_selector = gr.Dropdown(models, label="Select Model", value="deepseek-r1:1.5b")

    user_query = gr.Textbox(label="Ask a question about the PDF")
    submit_button = gr.Button("Get Answer")
    response_output = gr.Textbox(label="Response", interactive=False)
    preview_message_output = gr.Textbox(label="Preview Status", interactive=False)

    with gr.Row():
        html_preview_button = gr.Button("Render HTML/Markdown Preview")

    submit_button.click(fn=chat_with_pdf, inputs=[user_query, model_selector], outputs=response_output)
    pdf_upload.change(fn=load_pdf_and_chat, inputs=[pdf_upload, user_query, model_selector], outputs=response_output)
    html_preview_button.click(fn=handle_html_preview, inputs=response_output, outputs=[response_output, preview_message_output])

app.launch()

```

- **Provides a chatbot interface** that integrates AI, PDF processing, and HTML rendering.