PolicyNav Documentation

Overview

PolicyNav is a Streamlit-based web application designed to help users upload, extract, and interact with policy documents such as PDFs and images. The app uses OCR and PDF parsing to extract text, saves the extracted text to XML, and provides a chat interface for user interaction. It also supports working with sample datasets, such as those stored in the datasets folder.

Features

Upload PDF or image files (PNG, JPG, JPEG)
Extract text from scanned or digital PDFs and images
Save extracted text as XML files
View PDF metadata
Chat interface for interacting with extracted text
Use sample datasets for testing and demonstration

Pre-trained Datasets

Sample datasets are stored in the backend/datasets folder. For example:

finance_policy_sample.csv contains sample finance policy data, including policy names, ministries, years, and descriptions.

You can add your own public policy datasets (CSV, XML, PDF, images) to this folder for analysis and experimentation.

Start the Streamlit app:
streamlit run app.py

Upload a PDF or image file.
View extracted text and metadata.
Extracted text is automatically saved as an XML file in the temp_files directory.
Interact with the file using the chat interface.
Use sample datasets from the datasets folder for testing.

Use of Datasets

The datasets folder contains sample policy datasets such as finance_policy_sample.csv.
You can upload these datasets or your own documents through the app for extraction and analysis.
The app is flexible and works with any policy-related document or dataset relevant to your research.

Libraries and Tools Used

streamlit for building the web interface
os for file and directory operations
xml.etree.ElementTree for saving extracted text as XML
dotenv for loading environment variables
time for initialization delay
PyMuPDF for PDF text and metadata extraction
pytesseract for OCR text extraction from images and scanned PDFs

pdf2image for converting PDF pages to images
Pillow for image processing
opencv-python-headless for advanced image preprocessing
fastapi, watchdog, uvicorn, spacy included for possible API and NLP extensions

External Tool

Tesseract OCR required for OCR functionality

Contact

For questions or contributions, please open an issue or pull request on GitHub.

Project Main Code (app.py)

```
import streamlit as st
import os
import xml.etree.ElementTree as ET

from dotenv import load_dotenv
load_dotenv()

import time
time.sleep(1)  # ek second ka gap

from backend.ocr import extract_text_from_scanned_pdf
from backend.pdf_loader import extract_pdf_text, get_pdf_metadata, is_scanned_pdf

st.set_page_config(
    page_title="Upload policy documents",
    page_icon="📄",
    layout="wide",
    initial_sidebar_state="expanded"
)

def main():
    st.markdown(
        """
        <div style="background: linear-gradient(90deg, #667eea 0%, #764ba2 100%);
                    padding: 2rem;
                    border-radius: 10px;
                    margin-bottom: 2rem;
                    text-align: center;
                    color: white;
                    box-shadow: 0 4px 6px rgba(0, 0, 0, 0.1);">
            <h1 style="margin: 0; font-size: 3rem; font-weight: bold;">
                File Upload Center
            </h1>
            <p style="margin: 0.5rem 0 0 0; font-size: 1.2rem; opacity: 0.9;">
                Upload a PDF or image and chat with its content
            </p>
        </div>
        """,
        unsafe_allow_html=True
```

```
    )

    uploaded_file = st.file_uploader(
        "Choose a file to upload",
        accept_multiple_files=False,
        help="Upload a PDF or image file to extract text and interact with it",
        type=["pdf", "png", "jpg", "jpeg"]
    )

    if uploaded_file:
        file_path = save_uploaded_file(uploaded_file)
        if file_path:
            extracted_text = process_uploaded_file(file_path)
            if extracted_text:
                chat_with_file_llm(extracted_text)

def save_uploaded_file(uploaded_file):
    try:
        temp_dir = "temp_files"
        os.makedirs(temp_dir, exist_ok=True)
        file_path = os.path.join(temp_dir, uploaded_file.name)
        with open(file_path, "wb") as f:
            f.write(uploaded_file.getbuffer())
        return file_path
    except Exception as e:
        st.error(f"Failed to save the uploaded file: {e}")
        return None

def save_text_to_xml(extracted_text, file_path):
    root = ET.Element("Document")
    text_elem = ET.SubElement(root, "ExtractedText")
    text_elem.text = extracted_text

    xml_file_path = file_path + ".xml"
    tree = ET.ElementTree(root)
    tree.write(xml_file_path, encoding="utf-8", xml_declaration=True)
    return xml_file_path

def process_uploaded_file(file_path):
    st.markdown("### File Details")
    if file_path.endswith(".pdf"):
        metadata = get_pdf_metadata(file_path)
        st.write("**PDF Metadata:**")
        st.json(metadata)

        if is_scanned_pdf(file_path):
            st.info("The PDF appears to be scanned. Extracting text using OCR...")
            success, ocr_text, _ = extract_text_from_scanned_pdf(file_path)
            if success:
                st.text_area("Extracted Text (OCR)", ocr_text, height=300)
                xml_path = save_text_to_xml(ocr_text, file_path)
                st.success(f"Extracted text saved to XML: {xml_path}")
                return ocr_text
            else:
                st.error("Failed to extract text using OCR.")
                return None
        else:
```

```
            st.info("The PDF is a digital document. Extracting text...")
            text = extract_pdf_text(file_path)
            st.text_area("Extracted Text", text, height=300)
            xml_path = save_text_to_xml(text, file_path)
            st.success(f"Extracted text saved to XML: {xml_path}")
            return text
    elif file_path.lower().endswith((".png", ".jpg", ".jpeg")):
        st.info("Processing image file...")
        success, ocr_text, _ = extract_text_from_scanned_pdf(file_path)
        if success:
            st.text_area("Extracted Text (OCR)", ocr_text, height=300)
            xml_path = save_text_to_xml(ocr_text, file_path)
            st.success(f"Extracted text saved to XML: {xml_path}")
            return ocr_text
        else:
            st.error("Failed to extract text from the image.")
            return None
    else:
        st.error("Unsupported file type. Please upload a PDF or image file.")
        return None

def chat_with_file_llm(extracted_text):
    st.markdown("### Chat with the File")
    if "messages" not in st.session_state:
        st.session_state.messages = []

    for msg in st.session_state.messages:
        with st.chat_message(msg["role"]):
            st.markdown(msg["content"])

    if user_query := st.chat_input("Ask me something about the file..."):
        st.session_state.messages.append({"role": "user", "content": user_query})

        response = "LLM functionality has been removed. No automated answer available."
        st.session_state.messages.append({"role": "assistant", "content": response})

        with st.chat_message("assistant"):
            st.markdown(response)

if __name__ == "__main__":
    main()
```

Backend

ocr.py

```
import pytesseract
from pdf2image import convert_from_path
from PIL import Image
import cv2
import numpy as np
import os
import logging

logging.basicConfig(level=logging.INFO)
```

```
logger = logging.getLogger(__name__)

TESSERACT_PATH = os.environ.get("TESSERACT_PATH", r"C:\Program
Files\Tesseract-OCR\tesseract.exe")
if os.path.exists(TESSERACT_PATH):
   pytesseract.pytesseract.tesseract_cmd = TESSERACT_PATH
else:
   logger.warning(f"Tesseract not found at {TESSERACT_PATH}. OCR may not work
properly.")

def preprocess_image(image):
   gray = cv2.cvtColor(np.array(image), cv2.COLOR_RGB2GRAY)
   denoised = cv2.fastNlMeansDenoising(gray, h=10)
   _, thresh = cv2.threshold(denoised, 150, 255, cv2.THRESH_BINARY +
cv2.THRESH_OTSU)
   return Image.fromarray(thresh)

def extract_text_from_scanned_pdf(pdf_path, dpi=300):
   try:
       images = convert_from_path(pdf_path, dpi=dpi)
       ocr_text = ""
       ocr_data_pages = []
       for img in images:
           processed = preprocess_image(img)
           text = pytesseract.image_to_string(processed, lang="eng")
           ocr_data = pytesseract.image_to_data(processed, lang="eng",
output_type=pytesseract.Output.DICT)
           ocr_text += text + "\n"
           ocr_data_pages.append((img, ocr_data))
       return True, ocr_text, ocr_data_pages
   except Exception as e:
       logger.error(f"OCR extraction failed: {e}")
       return False, "", []



print(cv2.__version__)
```

pdfloade.py

```
import fitz
import logging
from backend.ocr import extract_text_from_scanned_pdf

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

def extract_pdf_text(pdf_path: str) -> str:
   """
   Extract text from a PDF file, using OCR if the PDF is scanned.
   """
   try:
```

```
        doc = fitz.open(pdf_path)
        text = ""
        for page in doc:
            text += page.get_text()
        doc.close()
        if len(text.strip()) < 100:
            logger.info("PDF appears to be scanned. Using OCR...")
            success, ocr_text, _ = extract_text_from_scanned_pdf(pdf_path)
            if success:
                return ocr_text
        return text
    except Exception as e:
        logger.error(f"Text extraction failed: {e}")
        return ""

def get_pdf_metadata(pdf_path: str) -> dict:
    """
    Extract metadata from the PDF.
    """
    try:
        doc = fitz.open(pdf_path)
        metadata = {
            "author": doc.metadata.get("author", ""),
            "title": doc.metadata.get("title", ""),
            "page_count": len(doc),
            "creation_date": doc.metadata.get("creationDate", ""),
            "modification_date": doc.metadata.get("modDate", "")
        }
        doc.close()
        return metadata
    except Exception as e:
        logger.error(f"Metadata extraction failed: {e}")
        return {}

def is_scanned_pdf(pdf_path: str) -> bool:
    """
    Returns True if the PDF appears to be scanned (little or no extractable
text).
    """
    try:
        doc = fitz.open(pdf_path)
        text = ""
        for page in doc:
            text += page.get_text()
        doc.close()
        return len(text.strip()) < 100
    except Exception as e:
        logger.error(f"Error checking if PDF is scanned: {e}")
        return False
```

Pdfwriter.py

```
import fitz
import logging
```

```
from typing import List, Tuple, Optio
from backend.ocr import extract_text_from_scanned_pdf
from PIL import ImageDraw

logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

def draw_redaction(page, rect, redaction_type, text=None, font_size=11,
custom_mask_text=None):
    try:
        if redaction_type == "black_bar":
            page.draw_rect(rect, color=(0, 0, 0), fill=(0, 0, 0), overlay=True)
        elif redaction_type == "white_bar":
            page.draw_rect(rect, color=(1, 1, 1), fill=(1, 1, 1), overlay=True)
        elif redaction_type == "custom" and custom_mask_text:
            page.draw_rect(rect, color=(1, 1, 1), fill=(1, 1, 1), overlay=True)
            page.insert_text(rect.tl, text=custom_mask_text,
fontsize=font_size, color=(0, 0, 0))
        elif redaction_type == "masked":
            page.draw_rect(rect, color=(1, 1, 1), fill=(1, 1, 1), overlay=True)
            page.insert_text(rect.tl, text="*" * len(text), fontsize=font_size,
color=(0, 0, 0))
        elif redaction_type == "numbered" and text:
            page.draw_rect(rect, color=(1, 1, 1), fill=(1, 1, 1), overlay=True)
            page.insert_text(rect.tl, text=text, fontsize=font_size, color=(0,
0, 0))
        elif redaction_type == "random":
            page.draw_rect(rect, color=(1, 1, 1), fill=(1, 1, 1), overlay=True)
            page.insert_text(rect.tl, text=text, fontsize=font_size, color=(0,
0, 0))
        elif redaction_type == "partial":
            # For digital PDFs, partial redaction is handled in redactor.py and
not needed here.
            pass
        logger.info(f"Redacted {rect} with {redaction_type}")
    except Exception as e:
        logger.error(f"Drawing redaction failed: {e}")

def find_text_instances(page, text_to_find):
    try:
        text_instances = page.search_for(text_to_find)
        return text_instances
    except Exception as e:
        logger.error(f"Finding text instances failed: {e}")
        return []

def redact_pdf(
    input_pdf,
    output_pdf,
    redaction_type,
    entities_to_redact,
    custom_mask_text=None,
    threshold=0.2,
    scanned=False
):
    """
    Redact PII in a PDF based on detected entities.
    For scanned PDFs, use OCR bounding boxes and draw on images.
    For digital PDFs, redact text layer as before.
    """
    if not scanned:
        # Digital PDF: redact as before
        try:
            doc = fitz.open(input_pdf)
            redactions_applied = False
            entity_counters = {}
            if redaction_type == "numbered":
                for _, entity_type, _, _ in entities_to_redact:
                    if entity_type not in entity_counters:
                        entity_counters[entity_type] = 0
```

```
                    entity_counters[entity_type] += 1
            for page_num in range(len(doc)):
                page = doc.load_page(page_num)
                for entity_text, entity_type, start, end in entities_to_redact:
                    text_instances = find_text_instances(page, entity_text)
                    replacement_text = None
                    if redaction_type == "random":
                        replacement_text = generate_fake_data(entity_type,
entity_text)
                        if replacement_text and len(replacement_text) >
len(entity_text):
                            replacement_text =
replacement_text[:len(entity_text)]
                    elif redaction_type == "masked":
                        replacement_text = "*" * len(entity_text)
                    elif redaction_type == "numbered":
                        entity_counters[entity_type] -= 1
                        count_number = entity_counters[entity_type] + 1
                        replacement_text = f"{entity_type} {count_number}"
                    elif redaction_type == "custom":
                        replacement_text = custom_mask_text
                    elif redaction_type == "partial" and entity_type in ["PAN",
"AADHAAR", "CREDIT_CARD"]:
                        replacement_text = partial_redact(entity_text,
entity_type)
                    for rect in text_instances:
                        draw_redaction(
                            page=page,
                            rect=rect,
                            redaction_type=redaction_type,
                            text=replacement_text,
                            custom_mask_text=custom_mask_text
                        )
                        redactions_applied = True
            doc.save(output_pdf, garbage=4, deflate=True)
            doc.close()
            return redactions_applied
        except Exception as e:
            logger.error(f"PDF redaction failed: {e}")
            return False
    else:
        # Scanned PDF: redact on images using OCR bounding boxes
        try:
            success, _, ocr_data_pages =
extract_text_from_scanned_pdf(input_pdf)
            if not success:
                return False
            from PIL import Image
            import re
            redacted_images = []
            for img, ocr_data in ocr_data_pages:
                draw = ImageDraw.Draw(img)
                for i, word in enumerate(ocr_data["text"]):
                    orig = word.strip()
                    if not orig:
                        continue
                    for entity_text, entity_type, _, _ in entities_to_redact:
                        # For partial, match last 4 digits/letters
                        if (
                            redaction_type == "partial"
                            and entity_type in ["PAN", "AADHAAR",
"CREDIT_CARD"]
                            and orig[-4:] == entity_text.strip()[-4:]
                        ) or (orig == entity_text.strip()):
                            x, y, w, h = ocr_data['left'][i],
ocr_data['top'][i], ocr_data['width'][i], ocr_data['height'][i]
                            if redaction_type == "black_bar":
                                draw.rectangle([x, y, x + w, y + h],
fill="black")
                            elif redaction_type == "white_bar":
```

```
                                draw.rectangle([x, y, x + w, y + h],
fill="white")
                            elif redaction_type == "masked":
                                draw.rectangle([x, y, x + w, y + h],
fill="white")
                                draw.text((x, y), "*" * len(orig),
fill="black")
                            elif redaction_type == "random":
                                draw.rectangle([x, y, x + w, y + h],
fill="white")
                                draw.text((x, y),
generate_fake_data(entity_type), fill="black")
                            elif redaction_type == "custom" and
custom_mask_text:
                                draw.rectangle([x, y, x + w, y + h],
fill="white")
                                draw.text((x, y), custom_mask_text,
fill="black")
                            elif redaction_type == "numbered":
                                draw.rectangle([x, y, x + w, y + h],
fill="white")
                                draw.text((x, y), f"{entity_type}",
fill="black")
                            elif redaction_type == "partial" and entity_type in
["PAN", "AADHAAR", "CREDIT_CARD"]:
                                masked = partial_redact(orig, entity_type)
                                draw.rectangle([x, y, x + w, y + h],
fill="white")
                                draw.text((x, y), masked, fill="black")
                    redacted_images.append(img.convert("RGB"))
                # Save all images as PDF
                if redacted_images:
                    redacted_images[0].save(output_pdf, save_all=True,
append_images=redacted_images[1:])
                    return True
                return False
            except Exception as e:
                logger.error(f"Image-based PDF redaction failed: {e}")
                return False
```

Datasets

Indian education policy