CISC 6210: Natural Language Processing

Final Project

Student Name: Rajeshwar Vempaty **Date:** May 8,2024

Project Title: Interactive PDF Knowledge Extraction System

Project Overview:

The objective of this project is to develop a question-answering (QA) system that utilizes a corpus of PDF documents (primarily research papers). By leveraging the capabilities of LangChain and the ChatGPT API, the system will be able to interpret and answer queries related to the content of these documents. The design focuses on creating a user-friendly interface that allows for seamless interaction with the PDF content, making the extracted information accessible and useful for various applications.

Objectives:

- **Text Extraction and Preprocessing:** Efficiently convert PDF documents into clean, searchable text.
- **Text Chunking for Efficient Processing:** Segment the text into manageable parts to optimize retrieval and processing.
- **Semantic Embedding of Text:** Transform text chunks into vector embeddings for semantic search capabilities.
- **Conversational Interface Development:** Employ a conversational AI to interact with users and understand their queries.
- **LangChain Integration for Enhanced Functionality:** Utilize LangChain for handling complex data interactions necessary for retrieving and answering questions from the PDF corpus.

Technologies and Libraries:

- **Streamlit:** User interface creation.
- **PyPDF2 and pdfplumber:** PDF text extraction.
- **FAISS:** High-performance similarity search and clustering of dense vectors.
- **LangChain:** Integration of AI models with data sources for interactive applications.
- **Hugging Face Transformers:** Text embedding using state-of-the-art language models.

System Architecture:

1. <u>PDF Text Extraction and Preprocessing:</u>
    - Tools Used: PyPDF2, pdfplumber
    - Functionality: The system extracts text using pdfplumber, which handles various PDF layouts and formats. Post-extraction, the text is processed to remove any irrelevant elements such as headers, footers, and formatting, ensuring that the text is clean and ready for NLP operations.

PDF Text Extraction and Preprocessing Code Snippet:

```
# Text Extraction
def get_pdf_text(pdf_docs):
    text = ""
    for pdf in pdf_docs:
        try:
            with pdfplumber.open(pdf) as pdf_reader:
                for page in pdf_reader.pages:
                    text += page.extract_text() or ''
        except Exception as e:
            logging.error(f"Failed to process PDF {pdf.name}: {str(e)}")
            st.error(f"Error processing {pdf.name}. Make sure it's not corrupted
and is in a supported format.")
    return text

# Cleaning Text
def clean_text(text):
    text = text.lower()
    patterns_to_remove = [
        r'\b[\w.-]+?@\w+?\.\w+?\b',  # emails
        r'\[[^\]]*\]',  # text in square brackets
        r'Figure \d+: [^\n]+',  # figure captions
        r'Table \d+: [^\n]+',  # table captions
        r'^Source:.*$',  # source lines
        r'[^\x00-\x7F]+',  # non-ASCII characters
        r'\bSee Figure \d+\b',  # references to figures
        r'\bEq\.\s*\d+\b',  # equation references
        r'\b(Table|Fig)\.\s*\d+\b',  # other ref styles
        r'<[^>]+>'  # HTML tags
    ]
    for pattern in patterns_to_remove:
        text = re.sub(pattern, '', text, flags=re.MULTILINE)
    text = re.sub(r'\s+', ' ', text).strip()  # Normalize whitespace
    return text
```

2. Text Chunking:
    - Tool Used: Regular Expressions
    - Functionality: This process involves using regular expressions to identify logical sections within the text (such as chapters or headings). These sections are then chunked into smaller parts, each suitable for processing and embedding.

Text Chunking Code Snippet:

```
def get_text_chunks(text):
    header_pattern =
re.compile(r'\n\s*(Abstract|Introduction|Methods|Methodology|Results|Discussion|C
onclusion)\s*\n', flags=re.IGNORECASE)
    section_pattern =
re.compile(r'^(Abstract|Introduction|Methods|Methodology|Results|Discussion|Concl
usion)$', re.IGNORECASE)

    sections = header_pattern.split(text)
    chunks = []
    current_chunk = []
    current_length = 0
    current_offset = 0

    for section in sections:
        if section_pattern.match(section):
            if current_chunk:
                chunks.append((section, current_offset, current_offset +
current_length))
                current_chunk = []
                current_length = 0
            current_chunk.append(section)
            current_offset += len(section) + 1
        else:
            words = section.split()
            for word in words:
                if current_length + len(word) + 1 > CHUNK_SIZE:
                    if current_chunk:
                        chunks.append((' '.join(current_chunk).strip(),
current_offset, current_offset + current_length))
                    current_chunk = [word]
                    current_length = len(word) + 1
                else:
                    current_chunk.append(word)
                    current_length += len(word) + 1

    if current_chunk:
        chunks.append((' '.join(current_chunk).strip(), current_offset,
current_offset + current_length))

    return [chunk[0] for chunk in chunks]
```

3. Text Embedding and Vector Store Creation:
    - Tools Used: Hugging face Transformers, FAISS
    - Functionality: Text chunks are transformed into dense vector embeddings using models from Hugging face. These embeddings are stored in a FAISS index for efficient similarity searches.

Text Embedding and Vector Store Creation Code Snippet:

```
def get_vectorstore(text_chunks, model_type='huggingface', model_name=None):

    try:
        if model_type == 'huggingface' and model_name:
            embeddings = HuggingFaceInstructEmbeddings(model_name=model_name)
            print(f"Using Hugging Face model: {model_name}")
        else:
            embeddings = OpenAIEmbeddings()
            print("Using default OpenAI embeddings.")

        vectorstore = FAISS.from_texts(texts=text_chunks, embedding=embeddings)
        print("Vector store created successfully.")
        return vectorstore
    except Exception as e:
        logging.error("Failed to create vector store: %s", str(e))
        raise ValueError("Error in creating vector store. Please check the
embedding model details.")
```

4. Semantic Search and Retrieval:
    - Tools Used: FAISS
    - Functionality: When a user query is received, the system computes the semantic similarity between the query's embedding and the stored text embeddings. The most relevant chunks are then retrieved based on their similarity scores.

5. LangChain Integration and Conversational AI:
    - Tools Used: LangChain, ChatGPT API
    - Functionality: LangChain is utilized to manage the data interaction layer, ensuring that the AI can access the vector store and utilize the retrieved information to generate contextually appropriate responses.

LangChain Integration and Conversational AI Code Snippet:

```
def get_conversation_chain(vectorstore):
    llm = ChatOpenAI()
    memory = ConversationBufferMemory(memory_key='chat_history',
return_messages=True)
    return ConversationalRetrievalChain.from_llm(llm=llm,
retriever=vectorstore.as_retriever(), memory=memory)
```

Framework:

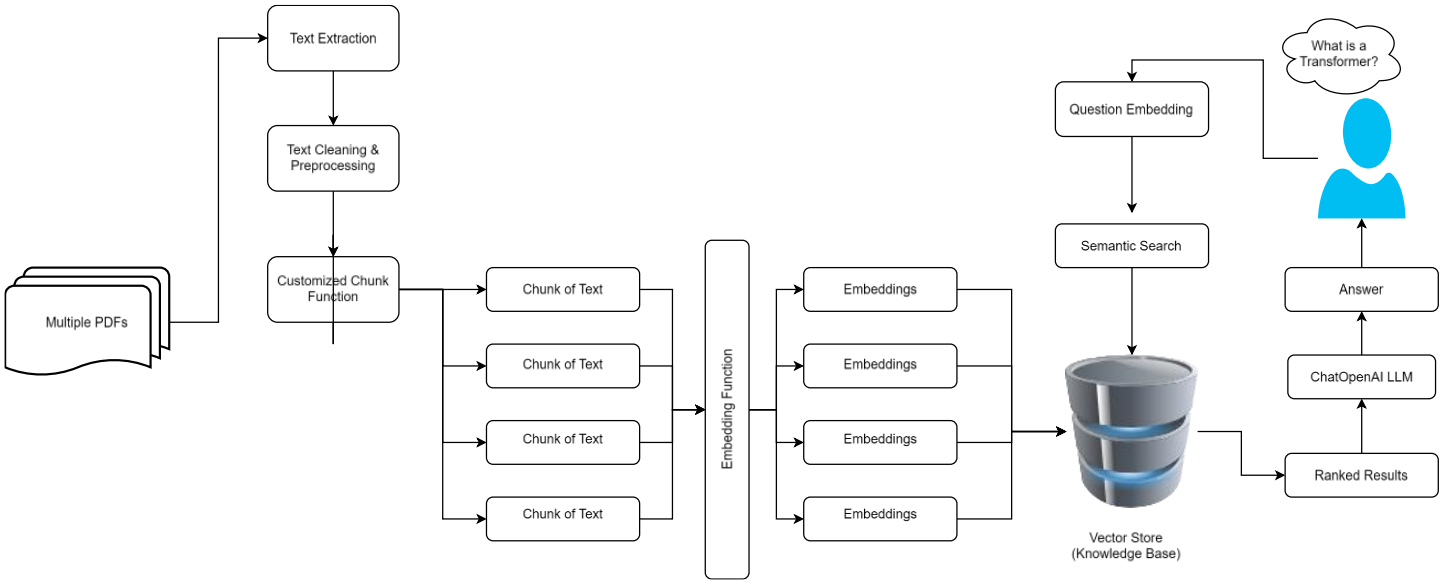


Figure 1: Framework.

User Interaction:

The system's interface, built with Streamlit, is designed for ease of use. Users can upload PDFs, input questions, and view answers through a clean and intuitive layout. This interaction is facilitated by backend Python scripts that manage the session state and user inputs.

Example User Flow:

1. PDF Upload: User uploads one or more PDFs using the file upload widget.
2. Question Input: User types a question into the text box provided.
3. Processing and Response: The system processes the question, retrieves relevant information, and displays the answer.

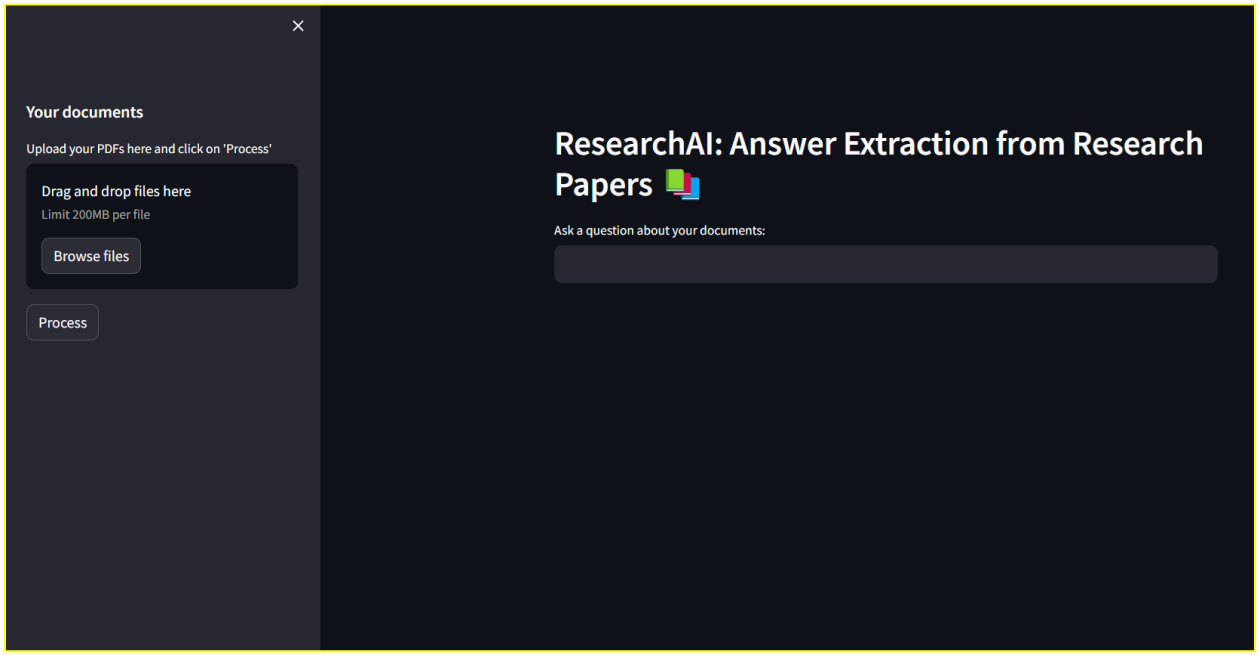
Challenges and Solutions:

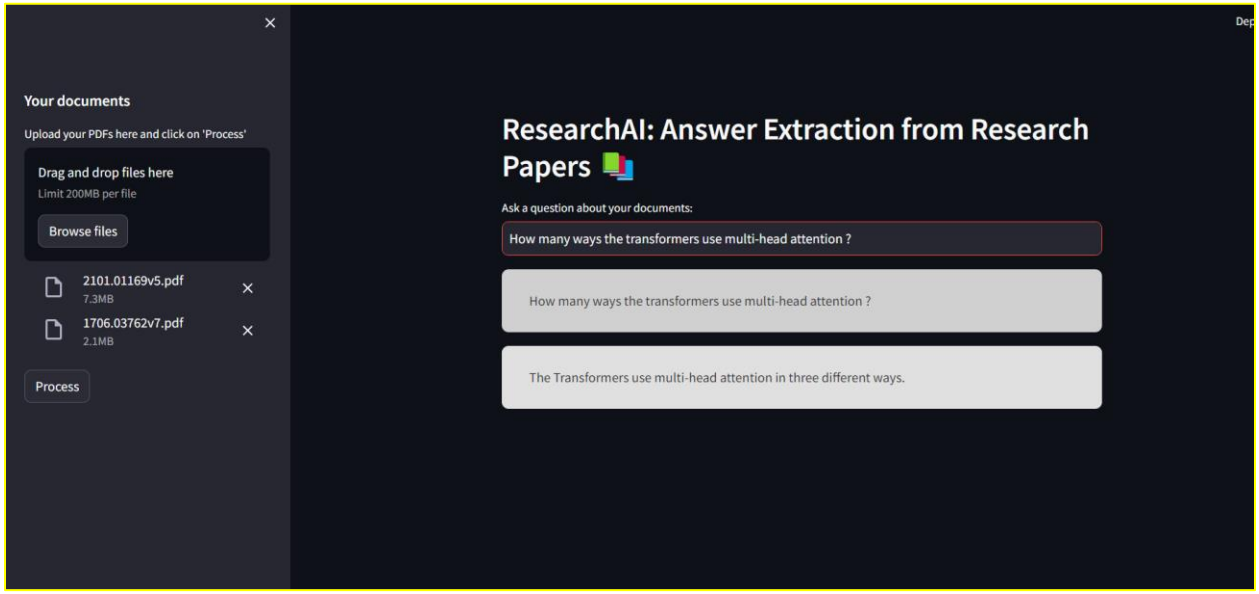
In developing our question-answering system using a PDF corpus, we faced challenges in text extraction, processing, and retrieval. We overcame difficulties in extracting clean text from varied PDF layouts using PyPDF2 and pdfplumber and addressed large text volumes and computational demands during embedding by segmenting text into manageable chunks and employing FAISS for efficient vector storage. Additionally, integrating these processes into a seamless conversational model using LangChain and the ChatGPT API was challenging. We enhanced context retention and response relevance by implementing conversational retrieval chains and utilizing conversation buffer memory, thereby creating a robust system capable of efficiently managing and querying large volumes of text data.

Conclusion:

This design document outlines the development of a QA system that integrates sophisticated AI and NLP technologies to provide a powerful tool for extracting and interacting with information contained within PDF documents. By combining Lang Chain's advanced data handling capabilities with the intuitive conversational abilities of the ChatGPT API, this system promises to be a valuable resource for users needing to query large volumes of text efficiently and effectively.

Screenshots





Installation and Execution Guide:

- **Prerequisites**
  Python: The system is developed in Python, so ensure Python 3.8 or newer is installed on your system. Visual Studio Code: Recommended as the development environment to leverage features like IntelliSense, code navigation, and integrated terminal for a seamless experience.
- **Installation Steps**
  Clone the Repository:
  First, clone the repository containing the project code to your local machine using Git. If you don't have Git installed, download, and install it from git-scm.com.

  git clone https://your-repository-url.git
  cd your-project-folder

  Set Up a Virtual Environment:
  It is best practice to use a virtual environment for Python projects to manage dependencies efficiently. You can create one using:

  python -m venv venv

  Activate the virtual environment:
  Windows: *.\venv\Scripts\activate*
  MacOS/Linux: *source venv/bin/activate*

  Install Dependencies:
  Install all required packages using pip. These packages include Streamlit, PyPDF2, pdfplumber, langchain, faiss, dotenv, and sentence_transformers among others.

  pip install streamlit PyPDF2 pdfplumber python-dotenv faiss-cpu sentence-transformers langchain

  Note: If you are using a GPU and want to leverage it, install faiss-gpu instead of faiss-cpu for better performance.

  Environment Variables:
  If the project uses environment variables (e.g., API keys), ensure that the .env file is set up correctly in the project's root directory with the required variables.

- **Running the Application**
  Once installation is complete, you can run the application through Visual Studio Code's integrated terminal:

  Start the Streamlit App:
  Make sure you are still in the project's root directory and the virtual environment is activated. Run the application using Streamlit:

  streamlit run app.py

  Access the Web Interface:
  After executing the command, Streamlit will start the server and provide a local URL (usually http://localhost:8501) which you can open in a web browser to interact with the application.

Using the Application:
Upload PDF files using the provided interface, enter your queries related to the PDF content, and receive answers dynamically generated by the system.

- **Troubleshooting**
  If you encounter any package dependency issues, try updating pip and retrying the installation:

  pip install --upgrade pip
  pip install -r requirements.txt

Source Code: https://github.com/rajeshwar-vempaty/NLP_Project