AI/ML Intern - RAG-based Document Chatbot (Technical Task)

By AVINASH M [Email: avinashmanivannan03@gmail.com]

Summary:

This assignment presents the development and implementation of a Retrieval-Augmented Generation (RAG) chatbot for PDF documents. The system enables users to upload PDF files and query their contents through natural language questions. By combining document retrieval techniques with large language model capabilities, the chatbot provides contextually relevant answers based on the document's content.

Project Overview:

Purpose:

The RAG PDF Chatbot was developed to address the challenge of efficiently extracting information from PDF documents. Rather than manually reading through lengthy documents, users can simply ask questions and receive targeted answers derived from the document's content.

Implementation Approach:

The system follows the RAG architecture, which enhances traditional language models by incorporating a retrieval component. This approach grounds the model's responses in the specific content of the uploaded document, improving accuracy and relevance while reducing hallucinations.

Technical Implementation:

Architecture:

The solution implements a multi-stage pipeline:

1. Document Processing: PDF text extraction, chunking, and vectorization.
2. Vector Storage: Creation of a searchable index of document chunks.
3. Question-Answering: Query processing, context retrieval, and response generation.

Technology Stack:

Frontend:

- **Streamlit**: Provides an interactive web interface for document upload and question submission

Backend:

- **Document Processing**:
  - PyPDFLoader (primary extractor)
  - pytesseract + pdf2image (OCR fallback)
  - PyMuPDF (secondary fallback)
- **NLP & Vector Operations**:
  - LangChain: Framework for text processing and RAG workflow
  - Sentence Transformers: Document and query embedding
  - FAISS: Vector similarity search
- **Large Language Model**:
  - Meta-LLaMA-3-8B-Instruct-Turbo (via Together API)

Critical Components:

PDF Text Extraction:

A robust, multi-method approach was implemented to handle various PDF types:

```
def extract_text_from_pdf(pdf_path):
    try:
        from langchain.document_loaders import PyPDFLoader
        loader = PyPDFLoader(pdf_path)
        pages = loader.load()
        if all(len(p.page_content.strip()) == 0 for p in pages):
            raise ValueError("Empty pages from PyPDFLoader")
        print("Extracted using PyPDFLoader")
        return [Document(page_content=p.page_content) for p in pages]
    except Exception as e:
        print(f"PyPDFLoader failed: {e}")
```

```
try:
    print("Trying OCR (pdf2image + pytesseract)...")
    images = convert_from_path(pdf_path)
    docs = [Document(page_content=pytesseract.image_to_string(img)) for img in images]
    if all(len(doc.page_content.strip()) == 0 for doc in docs):
        raise ValueError("OCR returned empty text.")
    print("Extracted using OCR")
    return docs
except Exception as e:
    print(f"OCR failed: {e}")
```

Document Chunking and Embedding:

Text is divided into manageable chunks and converted to vector embeddings:

```
text_splitter = RecursiveCharacterTextSplitter(chunk_size=500, chunk_overlap=50)
chunks = text_splitter.split_documents(docs)
embedding_model = HuggingFaceEmbeddings(model_name="all-MiniLM-L6-v2")
db = FAISS.from_documents(chunks, embedding_model)
```

Context Retrieval:

For each question, relevant document chunks are retrieved:

```
def get_top_k_chunks(query, db, k=3):
    results = db.similarity_search(query, k=k)
    return "\n\n".join([doc.page_content for doc in results])
```

Response Generation:

The LLM generates answers based on the retrieved context:

```
def generate_llama_response(query, context):
    messages = [
        {"role": "system", "content": "You are a helpful assistant. Use the given context to answer."},
        {"role": "user", "content": f"Context:\n{context}\n\nQuestion:\n{query}"}
    ]
```

```
client = together.Client(api_key=os.environ.get("TOGETHER_API_KEY"))
response = client.chat.completions.create(
    model="meta-llama/Meta-Llama-3-8B-Instruct-Turbo",
    messages=messages,
    max_tokens=512,
    temperature=0.5,
    top_p=0.9
)
return response.choices[0].message.content.strip()
```

Implementation Challenges and Solutions:

Challenge 1: Handling Different PDF Types:

Problem: PDFs can exist in various formats—text-based, scanned images, or mixed—presenting extraction challenges.

Solution: Implemented a cascading extraction pipeline with multiple fallback methods:

1. First attempt: PyPDFLoader for native text extraction
2. Second attempt: OCR with pytesseract for image-based PDFs
3. Final fallback: PyMuPDF for alternative extraction

This approach significantly improved text extraction reliability across different PDF types.

Challenge 2: Optimal Context Retrieval:

Problem: Determining how many chunks to retrieve as context for the LLM required balancing completeness with relevance.

Solution: Experimented with different retrieval parameters:

- Adjusted chunk sizes and overlap values
- Tested various k values for top-k retrieval
- Implemented a system prompt that constrains the model to only use provided context

The final implementation uses k=5 chunks with 500-character chunks and 50-character overlap, providing sufficient context while maintaining response accuracy.

Challenge 3: Response Quality Control:

Problem: Even with retrieved context, LLMs can sometimes generate responses that aren't grounded in the document.

Solution: Enhanced the system prompt to explicitly:

- Instruct the model to only answer based on provided context
- Direct the model to acknowledge when information isn't found
- Set temperature=0.5 for balanced creativity and accuracy

Example system prompt:

"You are a helpful assistant. ONLY answer based on the given context. If the answer is not in the context, respond with: 'No relevant information found in the document.'"

User Interface:

The application provides two interfaces:

Streamlit Web Interface:

The web interface offers a user-friendly experience:

- File upload component for PDF selection
- Text input for natural language questions
- Formatted response display
- Processing status indicators

```
uploaded_file = st.file_uploader("Upload a PDF file", type=["pdf"])


if uploaded_file:
   with open("temp.pdf", "wb") as f:
      f.write(uploaded_file.read())
   st.success("File uploaded successfully!")


   with st.spinner("Extracting and indexing text..."):
      docs = extract_text_from_pdf("temp.pdf")
      text_splitter = RecursiveCharacterTextSplitter(chunk_size=500, chunk_overlap=50)
      docs = text_splitter.split_documents(docs)
      embedding_model = TogetherEmbeddings(
         model="togethercomputer/m2-bert-80M-32k-retrieval",
```

```
    together_api_key=os.environ["TOGETHER_API_KEY"]
)

db = FAISS.from_documents(docs, embedding_model)
```

Command-Line Interface:

A command-line version is also available for programmatic access or headless environments.

Evaluation:

Strengths:

- **Robust Text Extraction**: Multiple fallback methods increase success rate across PDF types
- **Contextual Responses**: Answers are grounded in document content
- **User-Friendly Interface**: Simple upload and query process
- **Flexibility**: Both web and command-line interfaces available

Limitations:

- **Processing Speed**: Initial document processing can be slow for large PDFs
- **Context Window**: Limited by chunk retrieval approach
- **Single Document Focus**: Currently operates on one document at a time

Future Improvements:

Several enhancements could be implemented in future iterations:

1. **Multi-Document Support**: Enable queries across multiple uploaded documents
2. **Conversation Memory**: Implement chat history for multi-turn interactions
3. **Metadata Filtering**: Add capability to filter by document sections or metadata
4. **Performance Optimization**: Improve processing speed for large documents
5. **Enhanced Evaluation**: Implement automatic evaluation metrics
6. **Additional Document Formats**: Extend support beyond PDFs

Running Instructions:

Prerequisites:

- Python 3.8+
- Tesseract OCR engine
- Poppler (for pdf2image)
- Together API key

Installation:

pip install -r requirements.txt

Configuration:

Create a .env file with:

TOGETHER_API_KEY=your_api_key_here

Running the Web Application:

streamlit run app.py

Running the Command-Line Version:

python rag_chatbot_without_ui.py

Conclusion:

The RAG PDF Chatbot demonstrates an effective application of retrieval-augmented generation for document question-answering. By combining advanced text processing, vector similarity search, and large language models, the system enables users to interact naturally with PDF documents and extract relevant information efficiently.

The implementation successfully addresses key challenges in document processing and response generation, providing a foundation for further development and enhancement of document intelligence capabilities.

Document Prepared By:

Avinash M
[Phone: +91-9884039666]
[Email: avinashmanivannan03@gmail.com]
[LinkedIn: linkedin.com/in/m-avinash]
[GitHub: github.com/avinashmanivannan03]