**Project Report**

**An AI-Powered Document Assistant with RAG**

Date: July 13, 2025

Author: Kavya Bhardwaj

**CHAPTER 1**

**Concept and Architecture**

**1.1 Project Overview**

This report explores the creation and inner workings of our AI-Powered Document Assistant. At its heart, this project is a Retrieval-Augmented Generation (RAG) pipeline designed to intelligently answer questions about a specific document—in this case, the AI Training Document.pdf. Our goal was to build more than just a search tool; we wanted to create an interactive chatbot that could provide accurate, context-aware answers and transparently show how it arrived at them, complete with confidence scores and source attribution.

To bring this to life, we developed a user-friendly web interface with Streamlit. This frontend allows for real-time conversations and lets users tweak the pipeline's parameters, making it a powerful and accessible tool for anyone looking to quickly draw insights from dense documents.

**1.2 The RAG Model: A Two-Brain Approach**

Retrieval-Augmented Generation (RAG) is a clever AI architecture that combines the best of two worlds in natural language processing (NLP):

1. **The Librarian (Retrieval):** This part of the system is a master at finding and pulling the most relevant snippets of information from a large text collection. We use a sentence-transformer model and a FAISS vector index to handle this.
2. **The Storyteller (Generation):** This is a creative Large Language Model (LLM)—in our case, microsoft/DialoGPT-medium—that excels at crafting fluent, human-like responses.

Instead of just letting the Storyteller answer from its general knowledge, the RAG model first has the Librarian find relevant passages from our document. These passages are then handed to the Storyteller, giving it a solid, factual foundation to build its answer on. This simple but powerful technique helps us avoid common LLM pitfalls like "hallucination" (making things up) by grounding every response in the facts provided by the source document.

**1.3 System Architecture**

The project is built as a clean, modular pipeline. Each component has a specific job, and the data flows logically from one stage to the next.

1. **Ingestion & Preprocessing (The Prep Stage):**
   * **Input:** The source PDF document.
   * **Process:** We start by extracting all the text from the PDF. Then, we break this text down into smaller, overlapping "chunks." This is a crucial step that ensures we don't lose the meaning of sentences that cross our chunk boundaries.
   * **Output:** A neat collection of text chunks, ready for the next stage.
2. **Vectorization & Indexing (The Library):**
   * **Input:** The text chunks from the prep stage.
   * **Process:** Each chunk is fed into an embedding model (all-MiniLM-L6-v2), which converts it into a numerical vector. Think of this as giving each chunk a unique coordinate in a "meaning space." These vectors are then organized in a FAISS vector database.
   * **Output:** A highly efficient, searchable vector index (index.faiss) and its corresponding map back to the original text chunks (chunks.pkl).
3. **Runtime Query Processing (The RAG Loop):**
   * **Input:** A user's question, typed into the Streamlit app.
   * **Process:**
     + The question is also turned into a vector using the same embedding model.
     + We use this query vector to search our FAISS index for the text chunks with the closest "meaning coordinates."
     + A prompt is built on the fly, packaging the user's question with the most relevant chunks we found.
     + This complete package is handed to our generative LLM, DialoGPT-medium.
   * **Output:** A generated answer, a confidence score, and the source chunks used to create it.
4. **User Interface (The Conversation):**
   * **Input:** The final answer and all its metadata.
   * **Process:** The Streamlit app presents the information in a clean, conversational format. Users can not only read the answer but also peek behind the curtain to see which parts of the document the AI used.
   * **Output:** An engaging and transparent chat experience.

**CHAPTER 2**

**Implementation and Methods**

**2.1 Getting the Data Ready (**Preprocess.ipynb**)**

A RAG system is only as good as its knowledge base. We took great care in our preprocessing pipeline to turn the raw PDF content into a high-quality, structured dataset.

* **PDF Text Extraction:** We chose the PyMuPDF library (fitz) to pull text from data/AI Training Document.pdf. It's fast, reliable, and handles different PDF layouts gracefully.
* **Smart Text Chunking:** We used a RecursiveCharacterTextSplitter from the langchain library to break the text into smaller pieces. We settled on a chunk\_size of 300 characters with a 50-character chunk\_overlap. This configuration means each chunk is small enough for precise retrieval but large enough to hold a complete thought, and the overlap ensures we don't awkwardly split sentences.

**2.2 Creating and Storing Embeddings (**embed.ipynb**)**

This is where we translate our text into a language machines can understand and organize it for rapid search.

* **The Embedding Model:** We're using sentence-transformers/all-MiniLM-L6-v2. It's a fantastic model that's both lightweight and highly effective at capturing the semantic meaning of text, making it perfect for our use case.
* **The Vector Store:** For our database, we went with FAISS (Facebook AI Similarity Search). It's an incredibly fast library for finding the "nearest neighbors" in a vast collection of vectors. We use the IndexFlatL2 index, which calculates the simple Euclidean distance between vectors to find the best matches.

**2.3 The Core Pipeline (**rag\_pipeline.py**)**

The ImprovedRAGPipeline class is the engine of our chatbot, handling the real-time logic.

* **Retrieval:** When a question comes in, it's encoded into a vector. FAISS then searches the index for the k most similar text vectors. We calculate a similarity score and filter out any chunks that don't meet our similarity\_threshold, ensuring only the most relevant information proceeds.
* **Generation:** We carefully construct a prompt for the LLM. This isn't just a question; it's a command that tells the model to answer *only* using the context we've provided. This "grounding" technique is our main strategy for preventing the model from going off-topic or inventing information. DialoGPT-medium then takes this prompt and generates the final answer.
* **Helpful Features:**
  + **Confidence Scoring:** We've included a simple confidence score based on the length of the answer. It's a basic but useful heuristic to gauge how sure the model is.
  + **Hallucination Detection:** We also have a basic check for hallucinations. It works by seeing how many words in the answer also appear in the source chunks. A low overlap might indicate the model is pulling in outside information, and we can flag this for the user.

**CHAPTER 3**

**The User Experience and What's Next**

**3.1 The Streamlit Interface (**app.py**)**

The front door to our project is a web app built with Streamlit. We chose it because it lets us build beautiful, interactive UIs quickly, using only Python.

* **Layout and Interaction:** The app has a simple two-column design. The main chat window is where the conversation happens. The sidebar is the control panel, offering both customization and transparency.
* **Interactive Controls:** We've put the user in the driver's seat. Sliders in the sidebar let you adjust:
  + **Number of chunks (**k**):** How much context should the AI consider?
  + **Similarity threshold:** How relevant must a chunk be to get included?
  + **Max tokens:** How long should the AI's answer be?
* **Transparency Features:**
  + **Performance Stats:** The sidebar shows key stats like which model is running and whether it's using a CPU or GPU.
  + **Source Context Expander:** Every answer from the AI comes with a dropdown. Clicking it reveals the exact text chunks the model used to form its answer, complete with similarity scores. This "show your work" feature is key to building user trust.

**3.2 Performance and Evaluation**

The get\_performance\_stats method gives us a quick look at the system's setup. When a GPU is available, we use half() precision, which makes the LLM significantly faster and more responsive. Our embedding cache also prevents re-calculating vectors for questions that have already been asked. Ultimately, the system's success is judged by the quality of its answers, but features like hallucination detection and confidence scores give us useful metrics to track that quality.

**3.3 Conclusion**

This project successfully delivers a complete, end-to-end RAG pipeline that transforms a static document into a dynamic, conversational knowledge base. It showcases how blending retrieval and generation can lead to AI tools that are not only smart but also reliable and transparent. The Streamlit UI is the perfect finishing touch, making the powerful backend accessible and easy for anyone to use.

**3.4 What's Next?**

The current system is a fantastic foundation, but there are many exciting directions we could take it in the future:

* **Smarter Models:** We could upgrade to newer, more powerful embedding and language models (like those from the Llama or Mistral families) to boost both comprehension and the quality of the generated text.
* **Better Hallucination Checks:** We can implement more sophisticated fact-checking mechanisms, perhaps using a separate AI model to verify that the answer logically follows from the source text.
* **Multi-Document Support:** The next big step is to expand the system to handle a whole library of documents, allowing users to ask questions across a much larger set of information.
* **Improved Re-ranking:** We could add a second-pass "re-ranking" step. After getting an initial set of results, a more powerful (but slower) model could re-order them to find the absolute best context for the LLM.
* **Conversational Memory:** We could give the chatbot a memory, enabling it to understand follow-up questions and maintain a coherent conversation over multiple turns.