AGENTIC AI CSCR3214

Assignment-1 Complete Working Of Code and Documentation

B.TECH 3rd YEAR

SEMESTER: 6th

SESSION: 2025-2026

Submitted By:

Jafar Alsaleh [2023855441]

Submitted To:

Dr. Ayush Kumar Singh

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

SHARDA SCHOOL OF ENGINEERING & TECHNOLOGY

SHARDA UNIVERSITY, GREATER NOIDA

1. Overview

This project implements a Retrieval-Augmented Generation (RAG) system that answers user queries using information extracted from PDF documents. The pipeline consists of:

1. Text extraction from PDFs
2. Text chunking
3. Embedding generation
4. Vector indexing using FAISS
5. Retrieval of relevant chunks
6. Context-based answer generation

The system ensures that answers are grounded in the uploaded documents.

2. Step-by-Step Code Execution Flow

Step 1: Installing Dependencies

The notebook begins by installing required libraries:

- `pdfplumber` – Extracts text from PDF files.
- `sentence-transformers` – Generates dense vector embeddings.
- `faiss-cpu` – Stores and searches embeddings efficiently.
- `transformers` – Used for answer generation (FLAN-T5).
- `torch`, `numpy`, `tqdm` – Supporting libraries.

These libraries enable document processing, semantic search, and text generation.

Step 2: Importing Required Modules

All necessary modules are imported, including:

- File handling utilities
- Text cleaning functions
- Embedding and retrieval tools
- Generation model interface

Helper functions are defined for:

- Text cleaning
- Saving and loading FAISS index
- Normalizing embedding vectors

Step 3: Problem Definition Section

A markdown section clearly defines the objective:

Build a Retrieval-Augmented Generation system that answers user queries using only information present in uploaded PDF documents.

This satisfies the assignment's requirement for a clearly defined purpose.

Step 4: Dataset Handling

Input Requirements:

- File Type: PDF only
- Content Type: Text only
- Source: Public or self-created PDFs

The code loads three PDFs from the working directory.

Text Extraction Process

Function used:

`extract_text_from_pdf()`

Process:

1. Opens PDF using `pdfplumber`
2. Iterates over all pages
3. Extracts text
4. Cleans formatting
5. Removes Project Gutenberg boilerplate (header/footer)
6. Returns cleaned full text

This ensures that irrelevant metadata does not affect retrieval.

Step 5: Text Chunking Strategy

Function used:

`chunk_text()`

Parameters:

- Chunk Size: 2000 characters
- Chunk Overlap: 300 characters

Why Chunking Is Required

Large documents cannot be embedded as a whole. Therefore:

- The text is split into manageable chunks
- Overlap preserves contextual continuity
- Each chunk becomes a retrievable unit

The output of this stage:
A list of structured text chunks with metadata including:

- Source file name
- Chunk ID
- Preview text

Step 6: Embedding Generation

Model used:

`all-MiniLM-L6-v2`

Process:

1. Load embedding model
2. Convert each text chunk into a 384-dimensional vector
3. Store embeddings as NumPy arrays

Output shape example:

`(1596, 384)`

This means:

- 1596 chunks
- Each represented by a 384-dimension embedding

These embeddings represent semantic meaning of text chunks.

Step 7: FAISS Vector Database Creation

FAISS (Facebook AI Similarity Search) is used for fast similarity search.

Steps:

1. Normalize embedding vectors
2. Initialize FAISS IndexFlatIP
3. Add normalized embeddings to index
4. Store metadata separately

Why normalization?
Because cosine similarity is computed using inner product on normalized vectors.

The FAISS index now enables fast nearest-neighbor search.

Step 8: Query Embedding & Retrieval

When a user enters a query:

1. The query is embedded using the same embedding model.
2. The query vector is normalized.
3. FAISS searches for top-k most similar chunks.
4. The system retrieves:
    a. Chunk text
    b. Similarity score
    c. Source metadata

This ensures semantic search rather than keyword search.

Step 9: Context Assembly

Function:

`assemble_context()`

Process:

- Combines top-k retrieved chunks
- Limits total character size
- Prevents token overflow

This assembled context is passed to the generator.

Step 10: Prompt Construction

Function:

`make_prompt()`

The system constructs a structured prompt:

- Provides retrieved context
- Includes user question
- Instructs model to summarize in own words
- Restricts answer to context only

This reduces hallucination.

Step 11: Answer Generation

Two generation options exist:

Option A: OpenAI API (if key provided)

High-quality generation.

Option B: Local FLAN-T5 model

Offline generation using:

`google/flan-t5-small`

The generator:

1. Receives structured prompt
2. Processes context
3. Produces grounded answer

Step 12: End-to-End Query Function

Function:

`answer_query()`

Pipeline:

1. Retrieve relevant chunks
2. Assemble context
3. Generate answer
4. Return:
    a. Query

b. Generated answer
c. Source files
d. Retrieved chunk previews

This function completes the RAG cycle.

Step 13: Testing

Minimum 3 queries are executed.

For each query:

- The query is printed
- Generated answer is displayed
- Source documents are shown
- Retrieved chunks are previewed

This satisfies assignment testing requirements.

Step 14: Index Saving (Optional)

The FAISS index and metadata are saved locally.

Files created:

- `faiss_index.bin`
- `index_meta.pkl`

This allows reuse without recomputing embeddings.

How the Complete RAG Pipeline Works Conceptually

1. Documents are converted into vector space.
2. Query is converted into vector space.
3. FAISS finds semantically closest document chunks.
4. Retrieved chunks are injected into prompt.
5. Generator produces contextualized answer.
6. Output remains grounded in source documents.

Why This Implementation Is Valid RAG

- Uses dense vector embeddings
- Uses FAISS vector store
- Performs semantic retrieval
- Injects retrieved context into generation
- Produces grounded answers
- Avoids direct fine-tuning
- Fully modular architecture

This matches standard RAG architecture used in industry.

Final Summary

This project successfully demonstrates:

- End-to-end RAG pipeline implementation
- PDF-based knowledge retrieval

- Embedding-based semantic search
- FAISS vector indexing
- Context-grounded text generation
- Query testing with documented outputs

The system fulfills all assignment requirements including architecture explanation, dataset specification, embedding selection, chunking strategy, and testing with outputs.