Technical Documentation: Algerian University Fields RAG Chatbot

Executive Summary

This document provides comprehensive technical documentation for an AI-powered chatbot system that leverages Retrieval-Augmented Generation (RAG) to answer questions about university fields in Algeria. The system employs a two-stage response generation process with vector-based document retrieval using FAISS indexing.

System Architecture

High-Level Architecture

The chatbot follows a modular architecture with clear separation of concerns:

```
┌─────────────────┐    ┌─────────────────┐    ┌─────────────────┐
│   Flask API     │    │  RAG Pipeline   │    │  Vector Store   │
│  (routes.py)    │◄──►│  (chatbot.py)   │◄──►│  (FAISS Index)  │
└─────────────────┘    └─────────────────┘    └─────────────────┘
         ▲                      ▲                      ▲
         │                      │                      │
         ▼                      ▼                      ▼
┌─────────────────┐    ┌─────────────────┐    ┌─────────────────┐
│   Web Client    │    │   Gemini API    │    │  PDF Documents  │
│  (Frontend)     │    │  (LLM Service)  │    │ (Knowledge Base)│
└─────────────────┘    └─────────────────┘    └─────────────────┘
```

Component Breakdown

1. Web Application Layer

- **Framework**: Flask with CORS support
- **Entry Point**: `main.py`
- **Application Factory**: `__init__.py`
- **API Routes**: `routes.py`

2. RAG Processing Pipeline

- **Core Logic**: `chatbot.py` - performs the entire RAG workflow
- **Document Retrieval**: `retriever.py` - Vector similarity search implementation
- **External API Integration**: `utils.py` - Gemini API communication

3. Data Processing Layer

- **Document Loader**: `pdf_loader.py` - PDF text extraction and preprocessing
- **Vector Store**: FAISS in-memory index for similarity search

Technical Implementation Details

1. Document Processing Pipeline

PDF Text Extraction

```
# Location: pdf_loader.py
Process: PDF → Raw Text → Document Collection
```

Implementation Details:

- Uses `PyPDF2` library for PDF parsing
- Processes all PDF files from `data/fields/` directory
- Extracts text from each page and concatenates
- Maintains filename mapping for source attribution

Technical Specifications:

- Input: PDF files (*.pdf format)
- Output: List of document strings and corresponding filenames
- Error Handling: Empty pages handled gracefully with fallback to empty string

Text Chunking Strategy

```
# Location: retriever.py
Chunk Size: 500 characters
Overlap: None (sequential chunking)
```

Chunking Logic:

- Fixed-size chunking with 500-character windows
- Sequential processing without overlap
- Filters empty chunks to optimize index size
- Maintains document provenance through `doc_map` array

2. Vector Embedding and Indexing

Embedding Model

- **Model**: `all-MiniLM-L6-v2` from SentenceTransformers
- **Dimensions**: 384-dimensional dense vectors
- **Language**: Multilingual support (English, French, Arabic)

FAISS Index Configuration

```
# Vector Store Specifications
Index Type: IndexFlatL2 (L2 distance)
Search Algorithm: Exhaustive search
Memory Usage: In-memory storage
Persistence: Runtime only (rebuilt on startup)
```

Index Construction Process:

1. Load and chunk all PDF documents
2. Generate embeddings for each chunk using SentenceTransformer
3. Build FAISS index with L2 distance metric
4. Store chunk-to-document mapping for source attribution

3. Retrieval Mechanism

Similarity Search Algorithm

```
# Location: retriever.py → retrieve_context()
Parameters:
- top_k: 5 (default)
- threshold: 0.5 (similarity threshold)
```

Retrieval Process:

1. **Query Encoding**: Convert user question to 384-dim vector
2. **Similarity Search**: Find top-k most similar chunks using FAISS
3. **Distance Conversion**: Transform L2 distance to similarity score
4. **Threshold Filtering**: Only include chunks above similarity threshold
5. **Context Aggregation**: Concatenate relevant chunks into context string

Similarity Calculation:

```
similarity = 1 - (l2_distance / 4)  # Normalized similarity score
```

4. Two-Stage Response Generation

Stage 1: Initial Response Generation

Location: `chatbot.py → generate_final_response()`

Process Flow:

```
Query → Retrieve Context → Check Relevance → Generate Initial Response
```

Conditional Logic:

- **With Context**: Uses retrieved chunks as context for domain-specific answers
- **Without Context**: Falls back to general knowledge responses

Prompt Structure (With Context):

```
You are a helpful academic assistant.
Context: {retrieved_chunks}
Question: {user_question}
Answer:
```

Stage 2: Response Improvement

Location: `utils.py → improve_response()`

Enhancement Process:

- **Input**: Raw response from Stage 1
- **Goal**: Improve clarity, structure, and professionalism
- **Output**: Polished, well-formatted final response

Improvement Prompt:

```
Please rewrite and improve the answer below. Make it concise, to the point.
Retain all essential details and key concepts, organize it clearly in
paragraphs and bullet points when needed, and use professional yet
accessible language.
```

5. LLM Integration

Gemini API Configuration

- **Model**: Gemini Pro (Google Generative AI)
- **API Endpoint**: Configurable via `GEMINI_URL`

- **Authentication**: API key-based authentication

Request Structure

```
{
  "contents": [{
    "parts": [{"text": "prompt_content"}]
  }]
}
```

Error Handling

- Status code validation
- Detailed error logging for debugging
- Graceful fallback with error messages

API Specification

Endpoint Documentation

POST /chat

Purpose: Process user questions and return AI-generated responses

Request Format:

```
{
  "question": "string"
}
```

Response Format:

```
{
  "response": "string"
}
```

Error Responses:

```
{
  "error": "No question provided"
}
```

Configuration Management

Environment Variables

```
# config.py
GEMINI_API_KEY: str    # Google Gemini API authentication
GEMINI_URL: str        # Gemini API endpoint URL
```

Directory Structure

```
ai_chatbot/
├── app/                  # Application modules
│   ├── __init__.py       # Flask app factory
│   ├── routes.py         # API endpoints
│   ├── chatbot.py        # RAG worflow
│   ├── retriever.py      # Vector search logic
│   ├── pdf_loader.py     # Document processing
│   └── utils.py          # External API calls
├── data/
│   └── fields/           # PDF knowledge base
├── main.py               # Application entry point
├── config.py             # Configuration settings
├── requirements.txt      # Python dependencies
└── Procfile              # Deployment configuration
```

Deployment Specifications

Dependencies

```
Flask==2.3.3
flask-cors==4.0.0
sentence-transformers==2.2.2
scikit-learn==1.3.0
faiss-cpu==1.7.4
PyPDF2==3.0.1
requests==2.31.0
```

Resource Requirements

- **RAM**: Minimum 512MB (recommended 1GB for larger document sets)
- **CPU**: Single core sufficient for moderate load
- **Storage**: Minimal (documents loaded at runtime)

Limitations and Considerations

Current Limitations

1. **Index Persistence**: Vector index is rebuilt on every startup
2. **Memory Constraints**: All embeddings stored in RAM
3. **Document Updates**: Requires application restart for new PDFs
4. **Scalability**: Single-threaded processing for embedding generation
5. **Context Window**: Limited to top-5 chunks regardless of content length

Troubleshooting Guide

Common Issues

1. **Empty Responses**: Check GEMINI_API_KEY configuration
2. **No Relevant Context**: Verify PDF files in data/fields/ directory
3. **Import Errors**: Ensure all dependencies installed via requirements.txt

4. **Memory Issues**: Monitor RAM usage during embedding generation

Debug Information

- API response status codes logged to console
- Failed API calls include detailed error information
- Vector search results include similarity scores for analysis

Future Enhancement Opportunities

Technical Improvements

1. **Advanced Chunking**: Semantic chunking based on sentence boundaries
2. **Hybrid Search**: Combining dense and sparse retrieval methods
3. **Model Fine-tuning**: Domain-specific embedding model training
4. **Multi-modal Support**: Integration of images and tables from PDFs

Functional Enhancements

1. **Source Attribution**: Returning specific document sources with responses
2. **Conversation History**: Multi-turn conversation support
3. **Query Expansion**: Automatic query enhancement for better retrieval
4. **Feedback Loop**: User rating system for response quality improvement