# RAG Search AI Using FAISS - Detailed Documentation

## 1. Introduction

This document provides a detailed explanation of the Retrieval-Augmented Generation (RAG) AI search system implemented using FAISS. It covers the Embedding Layer, Search Layer, and Generation Layer, along with the chosen models, dataset columns used for embeddings, vector database approaches, and potential improvements for fine-tuned search queries.

## 2. System Architecture

The AI-driven search system is structured as follows:

1. Embedding Layer - Converts text data into vector representations.

2. Search Layer - Utilizes FAISS for efficient similarity search.

3. Generation Layer - Uses OpenAI's LLM to generate responses from retrieved results.

## RAG Architecture 🏗️

Retrieval-Augmented Generation (RAG) is an advanced AI approach that combines information retrieval and generative AI models. It enhances responses by retrieving relevant documents from a knowledge base before generating a response. This approach ensures accurate, up-to-date, and contextually rich answers.

### 📌 RAG Architecture Diagram (Steps Flow):

User Query ---> Embedding Layer (Convert Query to Vector)  
 ↓  
 FAISS Index (Search for Similar Matches)  
 ↓  
 Metadata Filtering (Apply Conditions)  
 ↓  
 Generation Layer (Format Response)  
 ↓  
 User Receives Final Output

## 3. Embedding Layer

The all-MiniLM-L6-v2 model from the SentenceTransformers library is used to generate dense vector embeddings for the given datasets. The embedding layer processes the following columns:

### Email Search AI:

- summary (Short description of email threads)

- body (Complete email content)

### Fashion Search AI:

- p\_attributes (Product attributes including material, color, and style)

- img (Image embeddings are not currently implemented but can be added)

Each text-based entry is encoded into a numerical vector using all-MiniLM-L6-v2.

### Python Code for Embedding Layer

from sentence\_transformers import SentenceTransformer  
  
# Load embedding model  
embedding\_model = SentenceTransformer("all-MiniLM-L6-v2")  
  
def generate\_embeddings(texts):  
 return embedding\_model.encode(texts, convert\_to\_numpy=True)

## 4. Search Layer

The FAISS (Facebook AI Similarity Search) library is used for fast vector search. The system builds a FAISS index as follows:

1. Extracts the relevant text column from the dataset.

2. Converts it into vector embeddings.

3. Stores embeddings in a FAISS IndexFlatL2 structure.

4. Performs similarity search using L2 distance (Euclidean distance).

### Python Code for FAISS Indexing and Search

import faiss  
import numpy as np  
  
# Function to create FAISS index  
def create\_faiss\_index(data\_vectors):  
 dimension = data\_vectors.shape[1]  
 index = faiss.IndexFlatL2(dimension)  
 index.add(data\_vectors)  
 return index  
  
# Function to search in FAISS index  
def search\_faiss(index, query\_vector, k=5):  
 distances, indices = index.search(query\_vector, k)  
 return indices

### What is FAISS?

FAISS (Facebook AI Similarity Search) is a library designed for efficient similarity search and clustering of dense vectors. It enables:

- Fast nearest neighbor search in large datasets.

- Memory-efficient indexing to store high-dimensional embeddings.

- Support for various indexing strategies like Flat, IVFFlat, and HNSW.

## 5. Generation Layer

Once relevant results are retrieved from FAISS, OpenAI’s text generation model (GPT-4/GPT-3.5-turbo) is used to synthesize a meaningful response. The model takes retrieved text as context and generates a natural language response tailored to the user query.

### Python Code for Generating Responses

import openai  
  
def generate\_response(context):  
 response = openai.ChatCompletion.create(  
 model="gpt-4",  
 messages=[{"role": "system", "content": "You are a helpful assistant."},  
 {"role": "user", "content": context}]  
 )  
 return response["choices"][0]["message"]["content"]

## 6. Alternative Vector Database Approaches

Apart from FAISS, other vector databases can be used:

- ChromaDB - Good for real-time search with filtering capabilities.

- Weaviate - Allows hybrid search combining text and metadata.

- Pinecone - Scalable and cloud-based solution for production-level search.

Each approach has different trade-offs in terms of accuracy, speed, and scalability.

## 7. Potential Enhancements for Fine-Tuned Search Queries

To improve search accuracy, the following enhancements can be made:

1. Multi-Modal Search - Include image embeddings using CLIP to search via text and images.

2. Hybrid Search - Combine FAISS vector search with keyword-based filtering for better results.

3. Re-ranking Strategies - Use a transformer-based re-ranking model to prioritize high-quality results.

4. Context Awareness - Implement query expansion techniques to understand user intent better.

5. Metadata-Based Filtering - Apply filters like product category, email sender, or date for refined searches.

### Search Layer Python Code on p\_attribute or sender:

def search\_fashion(query, k=5, min\_price=None, max\_price=None, fabric=None):  
 query\_vector = embedding\_model.encode([query])  
   
 # Perform FAISS search  
 distances, indices = faiss\_index.search(query\_vector, k)  
   
 # Retrieve matching results  
 results = [fashion\_data[i] for i in indices[0]]  
   
 # Apply metadata filtering  
 filtered\_results = [  
 item for item in results   
 if (min\_price is None or item["price"] >= min\_price) and  
 (max\_price is None or item["price"] <= max\_price) and  
 (fabric is None or fabric.lower() in item["fabric"].lower())  
 ]  
   
 return filtered\_results

## 8. Conclusion

The current implementation efficiently retrieves relevant emails and fashion products using FAISS. Further optimizations like multi-modal search, re-ranking, and hybrid search techniques can improve precision and relevance. Depending on the use case, alternative vector databases like ChromaDB or Weaviate can also be explored.