Case Study: Building a Mini RAG System

1. Introduction

    In this case study, the main goal was to design a document search and retrieval system for a company's internal knowledge base. The primary goal of this system is to combine external knowledge (from documents) with the generative capabilities of a language model to provide relevant answers to user queries. The system uses FAISS to retrieve similar document chunks and uses GPT-4o-mini to generate human-like responses based on the retrieved context.

2. Implementation

- Task 1: Data cleaning and chunking text. ('clean.py')

  This task processes .txt documents and splits them into chunks. Each chunk is under 300 tokens, with an overlap of 50 tokens between adjacent chunks. The preprocessing steps include:
  - Removing stop words, punctuation, and special characters from the text.
  - Tokenizing the text into sentences.
  - Splitting the sentences into chunks to ensure context is maintained.

  The final output from this task is a JSON file containing document titles and their chunks.

- Task 2: Embedding creation and vector storage and retrieval. ('embedding.py' and 'vector_database.py')

  Used pre-trained embedding models, distilbert-base-nli-stsb-mean-tokens and all-MiniLM-L6-v2 to create vector embeddings for each chunk. The embeddings are then saved in a JSON file for each model's embeddings, which includes the embeddings and the documents metadata. (Moving forward, I used the MiniLM-L6-v2).

  The embeddings are normalized. Used FAISS to store and search the vectors, which allows us to retrieve similar chunks. Used the IndexFlatIP (cosine similarity) index to store the embeddings, which allows fast nearest-neighbor searches based on vector similarity. This one returns the top-3 chunks and the metadata based on the query's embedding.

- Task 3: Contextual query handling and RAG pipeline simulation. ('query_handling.py' and 'pipeline.py')

  Handled multi-turn question (since this isn't a real time model, I implemented query history for the context). The query is pre-process first with the same embedding model. For this one I implemented weighted query, to make the current query has more say to the context. Then it's passed to FAISS to retrieve the top-3 similar chunks.

  Once we retrieve the top-3 relevant document chunks using FAISS, I combined these chunks into a single context that will be fed into the OpenAI GPT-4o-mini model combined with the query and generate response.

- Task 4: Scaling and performance tuning. (‘pipeline_optimize.py’)

  FAISS partitioning was implemented by switching to IndexIVFFlat with nlist = 15 to improve retrieval performance for larger datasets. This approach requires sufficient training data. Embedding normalization was applied to ensure the consistency and accuracy of similarity calculations. To address token limits, context token length truncation was implemented. Then, caching was removed due to the lack of repeated queries, though caching could be reintroduced if repeated queries become a part of the system in the future.

3. Challenges Faced

- Problem with comparing models ("all-MiniLM-L6-v2" vs. "distilbert-base-nli-stsb-mean-tokens"): The models output embeddings with different dimensions, which made it hard to compare them using similarity functions. Tried PCA, but it didn’t work.
- FAISS partitioning didn’t work too well. The IndexIVFFlat approach required more chunks to train. Using nlist = 15 led to only 1 chunk being retrieved, as the partitioning didn't work well with the dataset size.
- Caching didn’t work due to no repeated queries. Since the system isn’t real-time input-based, caching was removed. It could work if the user inputs the query themselves.