Directions to Navigate code:

NOTE: All the files should be run from the root directory/folder

**Config.py**

All the model configurations are present in **config.py** in the root directory. In case someone wants to test different models, MODEL\_NAME corresponding to that particular model can be uncommented.

The same can be done to change transformer embedding models.

Config.py serves as a one-stop destination to carry out the entire process with different models seamlessly.

**Scripts folder**

The scripts folder consists of all the code written to support and test the models. It also consists of the code to perform data preprocessing and indexing.

1. For data preprocessing, the **scripts/preprocess\_pmc.py** and **scripts/preprocess\_pubmed.py** files should be run individually.
2. For FAISS indexing **scripts/index\_doxuments1.py** should be run.
3. To test the code on a single customized query, **scripts/test\_query.py** should be run
4. To test model performance on the MedQA dataset, **scripts/test\_scores.py should be run.**
5. To test the model outside the RAG framework, open **scripts/test\_scores.py** or **scripts/test\_query.py** depending on the use case and comment the following line of code:

from models.generator\_LC import generate\_answer

and uncomment the following line of code:

#from models.generateNR import generate\_answer

**Models folder**

The models folder contains all the code required to set up the retriever model, generator model and to load the models inside these.

1. To make any changes/inspect how the models are being loaded open the **models/load\_model.py** file.
2. Note that the model is changed from config.py and not from the above file.
3. To inspect the retriever architecture, open the **models/retriever\_MedCPT.py** file.
4. The generator with the RAG framework is present in the **models/generator\_LC.py** file.
5. The generator without the RAG framework is present in the **models/generateNR.py** file.

The following dependencies might need to be installed to run the code.

transformers

torch

faiss-cpu

sentence-transformers

langchain

numpy

sklearn

pandas  
tqdm

matplotlib

re

Huggingface\_hub

NOTE: You will need to request access to the hugging face models used in this project to run the code. The models used are open-source, so the access is provided almost instantly. This can be followed by the **huggingface-cli login** command on the terminal and pasting your access token.

# 

# **2/21/25**

# **RAG Model Implementation Progress**

## **Overview**

We are implementing a Retrieval-Augmented Generation (RAG) model using Meditron-7B for medical diagnosis. This involves integrating a dense retriever (FAISS) with the Meditron model to generate evidence-based medical responses by retrieving relevant context from a knowledge base.

## **Objectives**

1. Build an efficient RAG pipeline for medical diagnosis.
2. Use FAISS for document retrieval and Meditron-7B for response generation.
3. Ensure consistent and accurate retrieval by maintaining a synchronized knowledge base.

## **Challenges Faced**

1. **Python Import Errors**: Confusion between module name and file name (retrieve.py) leading to circular imports and ModuleNotFoundError.
2. **IndexError: list index out of range**: Inconsistent document ordering during indexing and retrieval.
3. **FAISS Indexing Issues**: Invalid indices caused by mismatch between indexed and loaded documents.

## **Solutions Implemented**

### **1. Resolved Import Errors**

* Added \_\_init\_\_.py in the Scripts directory to mark it as a package.
* Used absolute imports and modified sys.path for consistent module resolution.

### **2. Consistent Document Handling**

* Implemented consistent document loading and saving strategy.
* Documents are saved as index/documents.json during indexing.
* The same file is loaded during retrieval to maintain index consistency.

### **3. Improved Index Validation and Error Handling**

* Checks if indices are within range before accessing the documents list.
* Logs invalid indices and provides detailed error messages.
* Captures and logs JSON parsing errors for better debugging.

## **Current Status**

* The RAG pipeline successfully retrieves relevant medical documents and generates evidence-based responses.
* FAISS index is created and saved consistently, preventing IndexError.
* generate.py is integrated with retrieve.py using absolute imports, resolving previous module import errors.

## **Next Steps**

1. Expand the Knowledge Base:
   * Add more JSON files to data/knowledge\_base to improve retrieval accuracy.
2. Fine-Tuning and Optimization:
   * Optimize hyperparameters for better generation quality.
   * Experiment with different retrieval strategies (e.g., BM25).
3. Deployment and Testing:
   * Deploy the model as a REST API for real-time medical query processing.
   * Conduct rigorous testing to validate the accuracy of differential diagnoses.

## **Conclusion**

Significant progress has been made in building a robust RAG pipeline. The implemented solutions ensure consistent indexing and retrieval, providing accurate and contextually relevant medical responses. The next phase will focus on scaling the knowledge base and optimizing the model for deployment.

## **3/8/25**

I developed a Retrieval-Augmented Generation (RAG) system designed to address medical queries by combining information from two Kaggle datasets. The system retrieves relevant documents from a PMC Patients dataset and a PubMed Articles dataset, then uses a large language model to generate a concise, accurate answer. Initially, I experimented with Meditron-7B and earlier versions of DeepSeek, but I eventually transitioned to using deepseek-ai/DeepSeek-R1-Distill-Qwen-7B. This model has proven to be much better at following prompt instructions and generating the desired concise answer.

## **My Objectives and Approach**

1. Data Acquisition and Preprocessing:  
   * I downloaded the PMC Patients dataset and the PubMed Articles dataset from Kaggle.
   * In the PMC dataset, I encountered inconsistent column names; the text content was labeled “patient.” I resolved this by renaming it to “abstract” for consistency with the rest of the pipeline.
   * In the PubMed dataset, the main content was provided in a column named “article.” I renamed this to “title” so that both datasets would conform to a standardized format.
   * I cleaned both datasets by removing rows missing critical fields (such as title and abstract) and ensuring data types (e.g., numeric values in “age” or “year”) were consistent. The cleaned data was then saved as JSON files for later indexing.
2. Document Indexing and Retrieval:  
   * I utilized a pre-trained, general-purpose sentence transformer to convert the text from the “abstract” field into vector embeddings.
   * I built separate FAISS indices for the PMC Patients and PubMed datasets, enabling fast retrieval of the top three relevant documents from each dataset based on a user query.
   * For retrieval, I computed an embedding for the user query and then conducted similarity searches on both FAISS indices to fetch the most similar documents.
3. Prompt Engineering and Answer Generation:  
   * I designed a prompt that includes the retrieved document context along with the user’s query. The prompt instructs the LLM to generate a concise, accurate answer without repeating the context.
   * Initially, I used Meditron-7B and earlier DeepSeek versions, but I observed that these models sometimes output extra context or did not strictly follow the prompt instructions.
   * I then switched to using deepseek-ai/DeepSeek-R1-Distill-Qwen-7B. This model has shown improved adherence to prompt instructions, generating more concise and targeted answers.
   * I refined the prompt and implemented post-processing techniques to extract only the answer portion from the model’s output.
4. System Integration and Testing:  
   * I developed an end-to-end test script that ties all components together. This script accepts a medical query, retrieves relevant context from both datasets via the FAISS indices, constructs a detailed prompt, and feeds it to the LLM to generate an answer.
   * During testing, I encountered issues such as ensuring the attention mask was properly passed during tokenization and managing unwanted prompt repetition. I addressed these issues through additional prompt engineering and post-processing strategies.

## **Challenges I Encountered**

* Inconsistent Dataset Structures: The two Kaggle datasets were formatted differently and had different naming conventions. I resolved this by renaming columns (for example, “patient” to “abstract” and “article” to “title”) and cleaning the data to remove incomplete entries.
* Indexing and Retrieval: After standardizing the data, I successfully built separate FAISS indices for each dataset using sentence embeddings. This allowed efficient similarity searches based on user queries.
* Prompt and Generation Issues: Although my prompt was designed to instruct the model to output only the answer, earlier models sometimes included extra context. I refined the prompt and applied post-processing to extract only the answer segment.
* Model and Hardware Considerations: My earlier experiments with DeepSeek highlighted that certain versions relied on FP8 quantization, which required an NVIDIA GPU. By switching to deepseek-ai/DeepSeek-R1-Distill-Qwen-7B, I obtained a model variant that not only meets my hardware capabilities but also follows prompt instructions more reliably.

## **Current Status and Next Steps**

* Data Preprocessing: I have successfully preprocessed and cleaned both the PMC Patients and PubMed Articles datasets. The cleaned JSON files are ready for indexing.
* Document Indexing and Retrieval: I built efficient FAISS indices for both datasets and integrated a retrieval mechanism that fetches the top relevant documents based on a user query.
* Answer Generation: I integrated a prompt-engineered generation module using deepseek-ai/DeepSeek-R1-Distill-Qwen-7B. This model has demonstrated improved adherence to prompt instructions and generates concise, accurate answers.
* Future Enhancements: I plan to further refine prompt engineering and explore additional fine-tuning strategies to enhance the model’s performance. Additionally, I will continue monitoring new model releases and hardware improvements to further optimize the system’s capabilities.

## **Conclusion**

I have built a modular, end-to-end Retrieval-Augmented Generation system for medical queries by:

* Preprocessing and standardizing two distinct Kaggle datasets.
* Creating FAISS indices for fast retrieval using sentence embeddings.
* Designing prompts that effectively combine retrieved context with user queries.
* Generating concise answers using deepseek-ai/DeepSeek-R1-Distill-Qwen-7B, which has proven to be better at following prompt instructions.

This project has given me valuable insights into data cleaning, efficient document retrieval, prompt engineering, and the challenges of deploying advanced language models. I am now well-positioned to further refine the system and explore additional enhancements to improve its accuracy and reliability.

## **3/25/25**

## **Overview of Work Completed**

The primary goal of this project was to build a robust RAG system capable of answering user queries using real biomedical data. This involved a multi-stage pipeline consisting of data collection, preprocessing, vector-based indexing, retrieval implementation, and the integration of a generative model to produce final answers. The work proceeded in the following stages:

### **Data Acquisition and Preparation**

Two primary datasets were used: one from PMC Patients and another from PubMed articles, both sourced from Kaggle. These datasets represent clinical and research aspects of biomedical literature respectively. The raw CSV files were inspected, cleaned, and converted into structured JSON format. Invalid entries, missing abstracts, and inconsistent field types were removed or standardized to ensure downstream compatibility.

### **Preprocessing and Cleaning**

Separate preprocessing scripts were developed for both datasets. These scripts loaded the data, removed incomplete or malformed records, standardized column names, and ensured that fields like age and gender followed consistent formats. The cleaned data was saved in JSON format, making it suitable for chunking and embedding.

### **Chunking and Embedding**

To address the context length limitations of transformer models, documents were chunked using a sliding window approach. Each document was split into overlapping chunks of 100 to 500 tokens. This helped preserve semantic continuity across the chunks. These text segments were then converted into dense vectors using the PubMedBERT embedding model (NeuML/pubmedbert-base-embeddings). Embeddings were processed in batches and saved efficiently.

This was later updated to MedCPT embeddings.

### **Indexing with FAISS**

A key component of the system was the use of FAISS for indexing the embeddings. Two separate indexes were created: one for PMC documents and one for PubMed documents. Each embedding vector was stored in an IndexFlatL2 FAISS index. Corresponding metadata (such as document text and chunk information) was stored in separate JSON files. This allowed fast nearest neighbor search and retrieval of relevant document chunks during inference.

### **Retriever Implementation**

The retriever component embedded incoming user queries using the same embedding model and used FAISS to find the most relevant chunks from each dataset. This dual-retrieval strategy ensured that both clinical and research-oriented content were considered when answering a query. The top results were returned as context for the generation module.

### **Generator and Transition to DeepSeek**

Initially, the system used Meditron-7B for generation but it struggled to follow instruction prompts. The project transitioned to the deepseek-ai/DeepSeek-R1-Distill-Qwen-7B model via API. This significantly improved coherence and response accuracy. Prompts were structured to include system role definitions, retrieved context, and the user’s question. Responses were generated through the DeepSeek API and returned to the user.

### **Full Pipeline Integration**

A centralized script (test\_query.py) tied all components together. It collected a user query, retrieved relevant chunks from both indexes, constructed the prompt, called the generation API, and displayed the final answer. The system was modular, allowing updates to individual components like embedding models or LLMs without affecting the rest of the pipeline.

## **Summary of Contributions**

* Cleaned and standardized two biomedical datasets.
* Implemented a chunking strategy for long documents.
* Generated dense embeddings using a biomedical transformer model.
* Created and persisted FAISS indexes for fast retrieval.
* Built a retriever that queries both datasets in parallel.
* Integrated a DeepSeek API-based language model for generation.
* Orchestrated the full system in a modular and testable fashion.

### **4/5/25**

### **Testing Code: test\_scores.py**

The test\_scores.py script was developed to evaluate the performance of our RAG system by calculating various evaluation metrics. It is designed to take a batch of medical queries, pass them through the entire retrieval-augmented generation pipeline, and compare the model's predictions against ground truth labels. The process involved several steps:

1. **Setting Up the Evaluation Script**
   * Created a script named test\_scores.py to automate the evaluation process for multiple queries.
   * Defined a function evaluate\_medqa\_hf() to handle the full testing pipeline.
2. **Integration with Existing Pipeline**
   * Imported the retriever and generator functions to form a cohesive pipeline.
   * Passed each query through the retriever to obtain relevant contexts and then through the generator to produce the final answer.
3. **Metrics Calculation**
   * Implemented accuracy calculation by comparing predicted answers against ground truth labels.
   * Incorporated additional metrics such as BLEU score, F1-Score (Macro), and DDF-1 Score for more comprehensive evaluation.
   * USMLE Pass Rate was added to assess performance against standardized medical evaluation criteria.
4. **Output Parsing and Extraction**
   * Developed a function extract\_final\_answer() to properly parse the model’s output.
   * Ensured compatibility with our structured prompt format: "Final Answer: [Option Letter] - [Explanation]".
   * Enhanced parsing robustness to handle cases where only the option letter is returned without explanation.
5. **Logging and Result Display**
   * Implemented logging for raw outputs for further inspection.
   * Displayed evaluation metrics at the end of the run for immediate feedback.
6. **Refinement and Debugging**
   * Iteratively refined the script to handle edge cases, particularly when the model’s output was incomplete or incorrectly formatted.
   * Improved robustness by adding checks for missing or malformed answers.
7. **Final Implementation**
   * Incorporated the script into the project pipeline, enabling systematic testing of various models and retrieval strategies.
   * The script now serves as the primary evaluation tool for assessing model performance across multiple queries.