Ryan McNeil

CSC84040

Assignment 2

Part A

1. Files in ./src.
   1. config.py
      1. Sets configuration settings. LLM & embedding model options, defaults for settings like temperature & max\_tokens, defines the app\_name, paths, and contains a list of stopwords to ignore in retriever processing (mostly conjunctions and articles which are irrelevant to or hinder understanding document context).
   2. app.py
      1. As in typical software design, app.py is responsible for actually running the app, using the functions and scripts from the other files. It sets up libraries including streamlit for the frontend. Imports config (settings like temperature & embedding models). Sets up a session state (maintains state across user uses). Defines the widgets: *Sidebar* for selecting LLM, embedding model, and parameters, these are saved to the session\_state; *Mainpage*, which displays the app’s title. Finally, app.py calls run\_upload\_and\_settings() for uploading PDF & settings, then run\_chatbot() to actually interact with the LLM.
   3. app\_sections.py
      1. app\_sections.py defines the functions for uploading & processing PDFs and building the chatbot interface. Contains the code for handling uploads & LLM interaction. It defines run\_upload\_and\_settings(), which runs the upload and settings container, taking in the uploaded PDF and saves it to a the “raw” folder. It defines run\_chatbot(), which provides the main functionality for interacting with the chatbot; initializes a session, uses UnstructuredPDFLoader to read in the PDF, creates the knowledge base using the selected embedding model using create\_knowledge\_base(), shows the chat messages (saves them in session\_state), retrieves the generated context using get\_retriever() and generates responses based on user input and generate\_kn\_response().
   4. app\_utils.py
      1. Defines functions for document processing, embedding, and retrieval, including the BM25 algorithm. Initialize\_session\_state() sets up defaults for session states (things like chat history & parameters). Embedding\_function uses hugging face embeddings to encode documents (you cache it for efficient retrieval). The BM25 algo is defined, which retrieves documents based on TF-IDF scoring. Get\_retriever() gets the user-selected embedding model (BM25 or FAISS). Create\_konwledge\_base() processes and saves docs as embeddings using the selected model. Num\_tokens\_from\_string() calculates the number of tokens in doc chunk (for models where you have a limited number of tokens). Generate\_response() connects to the model API to generate a response from a prompt (with context). Generate\_kb\_response() retrieves most relevant docs, puts them together and actually generates the response by calling generate\_response().
2. Solved using RecursiveCharacterTextSplitter. Solution in code.
3. *Description of what I’m doing also in comments in code.* For question 2 I use RecursiveCharactertextSplitter to grab chunks of text, set to split chunks to a length of 1000 charcters, with an overlap of 200 characters (20%). I then initialize a new docs list object (because your later code implies that the original ‘docs’ is to have been overwritten with the chunked data, my code reads from docs\_orig, and writes to docs). I loop to split each doc into chunks and append the collection of chunks from each document to the overall docs object along with its metadata.
4. I provide some framework for all three, but decided to use Google, for which I have provided the appropriate code. Solution in code.
5. It is problematic that the code currently assumes the answer is contained in exactly 1 chunk and the vector search will grab the right chunk because:
   1. Some questions may rely on information spread across multiple chunks from different sections of the text. E.g. if a doc defines something in one section, then references it in another. E.g. if we are comparing or contrasting things that come up in different parts of the text & thus different chunks.
   2. Necessary context may be spread across a section of the text that is longer than our chunk size.
   3. Multiple chunks may be necessary for the LLM to do something like summarize the text.
6. Here, I simply increase the value of k, the number of documents being used for context. Alternatively, you could update the code in generate\_kb\_response on line 211 to something like:
   1. n\_chunks = 3   
      relevant\_docs = retriever.get\_relevant\_documents(prompt)[:n\_chunks]
   2. I provide both solutions in the code.
7. For Q6, the first solution simply increases the k number of documents that BM25 will return to an arbitrary value >1. The second actually sets the number of chunks that will be returned in a prompt as generated by generate\_kb\_response.

Part B

1. [Solution in code with comments] I grab keywords, score each doc by keyword relevance using BM25, and grab the top k relevant docs.

Part C

1. requirements.txt and Dockerfile provided in submission.
2. Solution in video.

Part D

1. After trying several files, I think:
   1. BM25 excels in matching keywords, finding references to a specific thing defined in the text (likely due to TF-IDF scoring).
   2. FAISS excels in semantic search. Its embedding design means it can learn things like synonyms and overall meaning, rather than specific term matches.
   3. I would use BM25 for asking for specific facts & definitions from the text; tasks suitable for direct text retrieval. I would use FAISS for asking more semantic questions, or for things like summaries.
2. Discuss:
   1. Overall project: Successfully builds a RAG using LLM(s) and either BM25 or an embedding model (FAISS) for retrieving context. It seems to have some performance problems (seems to only occur in docker, not when running through streamlit in terminal), and finicky API calls (I encounter failed API calls intermittently, even though my code does work most of the time. I’m guessing this is happening on the LLM side, not to do with the Lucy code itself). Given an understanding of the advantages/disadvantages of each of our retriever models and of the various LLMs available, a user could quickly use this to scour medium sized texts. I believe FAISS combined with one of the more recent and more powerful models, like ChatGPT could be particularly powerful for asking complex, nuanced questions of challenging texts.
   2. What is a decoder transformer model?: A decoder model is used for generative tasks. It takes a sequence of words and predicts the next iteratively to create a response. Decoders use self-attention and causal masking to prevent themselves from “seeing” future words (or ‘tokens’). An encoder, on the other hand, focuses on encoding (read: ‘producing an understanding of’) inputs. Encoders (like BERT) attend to a whole text sequence at once, instead of only tokens to the left of the one being predicted. They learn with self-attention across a whole sequence, without masking. Encoders are better for comprehension tasks like classification or sentiment analysis, while decoders are better for things like text generation. RAG typically uses encoder models to retrieve relevant docs & a decoder to generate a response.
   3. What types of questions the chatbot can answer: With RAG, the chatbot could handle fact-based queries (“what is the definition of x”), semantic queries (“what does the author think about x”), comparisons (“how does x differ from y”), and summary questions (“what does this paper find?”).
   4. How might one improve RG to handle a broader range of questions to my pdf?:
      1. You could configure the retriever to retrieve multiple relevant chunks and somehow process them before they’re presented to the decoder. Better for synthesizing spread-out information.
      2. Add a mechanism to adjust the retrieval method based on the user’s question. A fact-based query could have the retriever grab 1 or 2 chunks and BM25, and a summary query could have the retriever grab many chunks and use FAISS.
      3. Use an encoder model like BERT for retrieval. Potentially more powerful.
      4. Adjust the code to open a chat session instead of sending prompts and receiving isolated responses. This would allow users to ask follow-up questions.
3. Summary & discussion of all work + insights:
   1. I’ve added code to: split a read-in file into chunks for an embedding model. Added code to query Google’s LLM model (an older version with free tokens). Amended the retriever code to consider more documents and amended the code that produces a prompt so that it includes more relevant context in the prompt to the decoder. Implemented the BM25 algorithm such that it performs TF-IDF scoring and grabs the most relevant chunks from the uploaded document by score for use in the raw prompt to be sent to the LLM. Amended the requirements.txt so that the app correctly runs with all dependencies. Produced a Dockerfile that can be used to build and spin up the app in a docker container for hosting. Built and spun up said container and tested it. Tested extensively to experiment with each retriever model’s strengths and weaknesses. The final product successfully features a chatbot that allows users to upload a PDF and ask questions of it through an LLM. I’ve also reviewed the difference between and strengths & weaknesses of encoder and decoder models, and between the BM25 algorithm and embedding models like FAISS. I’ve also identified some limitations of and proposed some possible improvements to the existing app.

Extra Credit

1. How might one update ./src/app\_utils.py to also incorporate the chat history. E.g. if a user asks a follow-up question that is only understood given the previous LLM response. What complications could arise and how could you handle them?
   1. You could modify generate\_kb\_response() to keep track of the chat history and always include it in prompts, or to retrieve it from st.session\_state[‘messages’] instantiated globally. You could shorten this history by using the retriever to also only grab the most relevant previous chat messages. Alternatively, you could have a rolling history window that only looks back so far. Or, you could implement some method of history summarization using the decoder.
   2. Complications could include:
      1. Quickly exceeding the **token limit** as prompts including chat history grow
         1. Could be mitigated by the abovementioned methods for shortening the history.
      2. Even if you had unlimited tokens, an **excessively large chat history** could contribute lots of irrelevant information to each prompt, drowning out the users most recent actual question.
         1. Also could be mitigated by the abovementioned methods for shortening the history.
         2. *or* by introducing relevance filtering via a retriever to pass only parts of the history to the overall prompt that are relevant to the user’s question.
      3. **Context drift** where the model may lose focus or relevant context as the chat history grows and tangents or misunderstandings occur
         1. Could be mitigated by introducing a topic-tracking mechanism, reducing the chat history, or somehow modifying the prompt to privilege or weight the current user question and document summary *over* the chat history.