Ryan McNeil

Assignment 3

1. Read through the code files under “./src” and describe what the following files are doing.
   1. Config.py : defining some variables & settings for our program. AI decoder models; embedding model (just the one); a temperature (something like “randomness”); Max Tokens for our decoder model; an app name (more of an intro statement). And directories for documents.
   2. app.py : creates the Streamlit web app. Loads the .env; imports the config; initializes the session state; sets the name; creates a sidebar in the app for selecting a model from the “Models” list, set temperature, & store those selections in the session state; runs upload & settings to handle file uploads; runs the chatbot to interact with the chatbot interface.
   3. app\_sections.py : defines functions for the app. run\_upload\_and\_settings() runs the upload and settings container. allows user to choose a file, copies it to the ‘raw’ folder & saves it to the session state. run\_chatbot() sees when the start button is clicked, builds a vector database from the uploaded PDF file, and allows for chatting; generates responses for user prompts.
   4. app\_utils.py : contains utility/helper functions for our app. Initializes API, defines our embedding function, defines utility functions. get\_embedding(): generates embedding vector from sentence. get\_retriever(): loads vector retriever from FAISS database. Initialize\_session\_state() initializes session state variables. Num\_tokens\_from\_string() calculates the number of tokens in a string. Generate\_response() generates responses to user prompts using the selected model & API. Create\_knowledge\_base() processes documents to compute embedding vectors & builds the FAISS database. Generate\_kb\_response() generates responses from the knowledge base based on prompts.
2. *Code in app\_utils.py lines 131-148*
3. Explain your choices form question 2. Describe what happens when you use a transformer based encoder model and how this relates/informs to embedding a document for storage in a vector database.
   1. *Code is commented in app\_utils.py*.
   2. I chose to use RecursiveCharacterTextSpliter because it is the standard text splitter in langchain that splits text and allows for overlap. I chose to overwrite ‘docs’ with the chunked data, because it appears that that is what later code is expecting *and* we already have the ‘orig\_docs’ saved. I chose chunk\_size = 1000 because it simply seemed like a reasonable value for this task. I chose overlap = 200 because this is 20% of 1000 (seems to be the standard overlap proportion for similar functions). Some applications seem so use chunk sizes of 5000+, but I’m guessing we would use shorter documents for testing purposes here.
   3. Using a transformer-based encoder, the input document(s) is broken into smaller chunks (or tokens) and encoded by the model in a way that preserves context for each token. Smaller chunks mean we’re more likely to be under the token max of our embedding model. Transformer-based embeddings are typically vectors of numeric values (which can represent words and meanings in high-dimensional space), which can be stored in vector databases (unlike the original unembedded data). Vector databases (like FAISS) allow efficient storage & retrieval of these embeddings.
4. *Code in app\_utils.py lines 91-134*
5. The code currently assumes the answer is contained in exactly 1 chunk and the vector search (embedding model similarity) will grab the right chunk. Discuss why this is problematic and what type of questions might be difficult to answer.
   1. The most obvious problem is if the best answer to a prompt/question (or the necessary context for that answer) is spread across multiple chunks. Relying on only one chunk can lead to inaccuracies in this case.
   2. Vector similarity may not sufficiently identify the best answer/chunk. Maybe a question or answer is ambiguous and requires a very complex understanding of the text beyond one chunk. Maybe the dialect the text was written in is different from the dialect of the question, resulting in decreased vector similarity that is unrelated to sentence meaning.
6. *Code in app\_utils.py in generate\_kb\_response function; lines 183 – 223*
7. Discuss your modification in 6 and the limitations, pros, and cons when determining how much context to bring in.
   1. I modified the code to grab our relevant chunks **plus** the chunks *around* and *between* those chunks. The buffer I arbitrarily set to 3 (as in, grab everything from 3 chunks before the first relevant chunk to 3 chunks after the last relevant chunk.).
   2. The Pros are increased context, better text generation, more accurate answers and a more robust model understanding of the text.
   3. The cons are the increased computational cost; additional short-term memory usage to store the additional text chunks (though, I’m assuming we’re working in contexts where this is negligible); potentially exceeding the maximum prompt length; possibly diluting the power of the most relevant chunks by surrounding them with a collection of other less relevant chunks.
8. Discuss the overall project, what a decoder transformer model is (and how different from encoder), what types of questions the chatbot could answer, and ways one might improve RAG to handle a broader range of questions on your pdf.
   1. The overall project takes documents for context and processes them to create an embedding representation of them, and uses them to generate contextualized text responses using any one of multiple text generation APIs. It can potentially circumvent the input limits of using APIs like ChatGPT directly by storing and processing documents locally and selectively providing a few chunks from a large body of work that are best for generating relevant text responses.
   2. A decoder transformer model “decodes” embeddings to generate text (or other output) in response to some prompt. Good for things like AI chatbots or machine translation. In contrast, an encoder processes input (in this context, text) to encode it into a vector representation. Encoder-decoder models (unlike decoder-only models) can potentially provide contextual representations of text that can be fine-tuneed on decoder tasks.
   3. This app allows you to use AI to explore text documents quickly and easily. This chatbot could answer questions about the content of the text(s), answer questions based on the content of the text(s), interpret the text(s) for clarification or explaining things to user(s). The chatbot could even give advice for improving the documents, or generating new documents based on the content of the documents uploaded.
   4. RAG (Retrieval-Augmented Generation) can be improved by improving retrieval mechanisms, e.g. using multiple retrieval models and combining them somehow (as with MM-RAG); by fine-tuning your model on a particular domain or dataset related to the kinds of prompts you want to ask. RAG could also be improved by implementing a step in the model where the user prompts are rewritten/paraphrased by an interpreter (prior to text generation) to better match the language in the uploaded documents. It can also be improved by implementing a feedback loop where the model learns from user interactions.
9. EXTRA CREDIT: Discuss how one might update “./src/app\_utils.py” (but don’t update) to also incorporate the chat history. For example, 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. Update the structure to store the chat history locally and input it in each prompt as additional context.
   2. OR Update the structure to store the chat history in a history component of the database, or a queue of previous interactions.
   3. One complication would be ambiguity. The model might not understand how to query the chat history for relevant context. If we’re including chat history as part of the “relevant context” portion of the code, this may also contribute to excessively long prompts, dilution of our relevant docs. You may want to dynamically adjust the context padding we implemented in question 6 to increase how many chunks you’re grabbing for context, or eventually you may wind up with a situation where most of the “most relevant docs” are from the chat history and *not* from the uploaded documents.
   4. Another complication would be memory & performance. How would the history be stored? Where? For how long? Storing/processing a large chat history could impact RAM & model performance. It could also be a data privacy/security concern.