CSCE 685 DIRECTED STUDIES
Final Report

Q&A chatbot for TAMU CS Department based on Retrieval Augmented Generation pipeline using unstructured data.

BY

CHINMAY AJIT SAWKAR

Under the guidance of DR. Duncan M. (Hank) Walker

INDEX

Objective

With the advances in Large Language models in recent years the application of these models in language understanding and performing complex question answering have also greatly increased. Although these large language models are trained on billions of documents, and knowledge bases in terms of next token prediction these systems are still not capable to answer questions related to institutional data or propriety data. Uploading propriety data to large language models which use the data for continuous training also raises issues for data privacy and data leaks. It is also infeasible and inefficient to train these large language models in the whole data as training requires high computational costs and it would still be a problem as we would have to train the model on every new data. Thus, to tackle this problem Retrieval Augmented Generation commonly known as RAG is used which uses efficient retrieval techniques to fetch the data relevant to the user query from a large corpus of documents and rephrase the query to large language model for question answering.

The goal of this directed study project is to explore different techniques for Retrieval Augmented Generation and build a question answering chat not that can answer question related to any user specific data. For testing the implementation of the RAG model, we use the Texas A&M Computer Science courses syllabus data for answering question related to the syllabus for the courses offered. The essential step in RAG pipeline if how we capture the data into chunks which can be indexed for efficient retrieval. Breaking unstructured data into semantically logical chucks which will also have information about the global topic and document was a major challenge that was tackled in this project. The text chucks were formed with document title and details like position in metadata which was then transformed into embedding vectors using Google Gemini model. These embeddings were stored in vector database for quick retrieval. Finally, when there is a user question the text chunks with high semantic similarity were fetched, along with chat history and compressed into a prompt with relevant data as context for the model to answer. The large language model can now answer the question from the context provided in the prompt. A application was made using Streamlit fin python for displaying the answers in a chat format for the user.

Literature Study

Large Language Models and their limitations

Large Language Models or LLMs have been majorly improved the Natural Language Processing capabilities in the last few years. LLMs are deep learning models that are trained on vast amount of data and are specifically characterized by their large number of parameters which help them generate text based on their trained data like humans. Their applications lie in question answering, text translation, summarization, sentence completion. However, in terms of question answering there are few shortfalls of LLM models. Firstly, as they have been trained on static data, they so they can only answer any question based on the data. If there is new data the model needs to be retained on the new data which is time consuming if there are many scenarios where data is constantly increasing. It's also costly for organizations to train such large models with billions of parameters frequently. Another limitation with LLM is that they are trained on large corpus of data sources which may be similar in nature thus the have a false confidence and answer incorrectly based on a similar source of data and thus not reliable for consumer applications. There are many cases in which LLM's provide false information or irrelevant answers with complete confidence, also known as hallucination in NLP terms. There is also a limitation of data security which looms as organization would have to continuously train their own LLM for securing the data sharing.

RAG – Introduction and Benefits

Retrieval Augmented Generation is a technique to help LLMs for generating better outputs by providing the LLM the relevant knowledge base outside the data on which the LLM is trained required for generating the right output. Thus, it enhances the capabilities and correctness of the LLM's in answering questions related to an organizational data or related to a particular domain without the requirement to retrain on the data. The benefits of RAG over LLM are that they are cost effective as re training is not required. It's easy to update the data sources with latest information just by adding the new information in the retrieval system. RAG also improves the trust in the answers of LLM as it can also provide the source of information that it generates and thus it can be validated by the user directly if required.

RAG Architecture

The RAG architecture primarily consists of two main components retrieving relevant information to the user query, generating accurate response based on the user query and the relevant documents retrieved. However, there are more finer steps which are equally important in RAG pipeline accuracy and efficiency. These are the main steps in RAG flow:

1. **Data processing and Text Chunking**: The first step in RAG pipeline is getting the data into text from different data sources like HTML pages, PDFs, CSV, tables and making optimal chunks for the data to be stored as embeddings. Embeddings are numerical

representations of text data that represent the semantic meaning behind the chunk of text data.

2. **Query Processing**: The user query needs to be pre-processed to filter the stop words, interpret misspelled words. Also follow up questions need to be rephrased using the conversation history to include the previous context.
3. **Retrieval System:** Once we have vector representations of text data, we can build a efficient retrieval system that can efficiently search for the chunks of data relevant to the user query. The retrieval system should be fast and be able to accurately rank the chunks according to their similarity with the user query.
4. **LLM Inference on augmented context:** The user query along with the relevant documents chunks fetched by the retrieval system are passed as context to the LLM for answering the question. The response is the presented to the user and updated in conversation history.

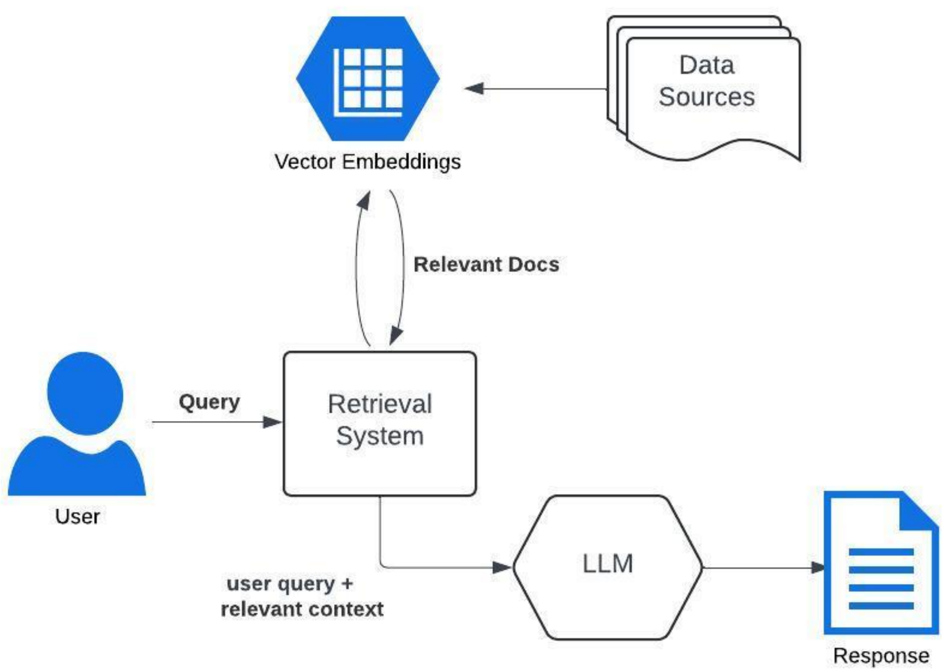


Fig. A general flow for RAG which comprises of two main parts Retrieval System and response generation using LLM.

Methodology

This section dives deep into the implementation details and challenges faced while building the RAG system in this project. The RAG system is the backend for the chat bot that can be used for answering questions based on any domain or organization data which normal NLP techniques or using LLM models would fail to answer accurately and efficiently. For the context of this project, I have considered the syllabus for different courses at the Texas A&M CS Department. The openly available syllabus documents which in PDF format and has a equivalent complexity as a real life use case as each document has different structure, similar type of content and which adds enough complexity to test the RAG system. The RAG system must accurately fetch the data corresponding to the user query and the chucks need to be done optimally so that all the relevant data is fetched.

Data Preprocessing and Text Chunking Strategies

The data preprocessing stage is crucial to the accuracy of the RAG system as the text chunking strategies determine the retrieval of relevant information. Although there are multiple strategies for storing the data, choosing the correct method of making chunks of data depends on the type of data. With different sources like HTML pages, PDF documents, code files, csv files the data first needs to be loaded into text format. The next step is to divide this data into small chunks or parts which would be easily to retrieve, have a logical semantic meaning and be small enough to fit in the context window of large language model.

There are different chucking strategies that I explored to be used based on type of data.

- **Character splitting:** This strategy is done by splitting based on a particular character. This is quite simple strategy, but it does not consider the document structure and can cut abruptly in between paragraphs, sentences.
- **Token Splitting:** In this strategy the text is split based on characters, but the size of the chunk is based on number of tokens. Thus, we can ensure that each chunk fits the number of tokens allowed in LLM context window.
- **Recursive Character Splitting**: This strategy splits text based on a set of characters that are taken as a parameter and split recursively based on them. That is, it splits based on the first character, then splits the resulting chunks which are longer than the specified chunk size (also taken as a parameter) based on the next character in the set provided. The advantage of this strategy is that it makes the chunks such that semantically similar data is chucked together as it first separates paragraphs into chunks, then splits long paragraphs and finally sentences.

Even after using these strategies there is a challenge for indexing long PDFs into chunks as each document might have similar sections for e.g., in this case all syllabus documents for different courses have a Textbooks section. Thus, for a model to accurately retrieve correct textbooks for a particular subject we must smartly add that information to each chunk. This is done by using the metadata for the chunk which contains the document title, file name, page number of the chunk etc. This helps associate each chunk to appropriate document. In this project I used the Recursive Character Splitting with the metadata included in each chunk. The parameters that were to be optimized were the chunk size and the chunk overlap. The chunk size must be small enough to fit in context window of our model but large enough to accommodate largest of paragraphs and text with semantic meaning. The chunk overlap is used for specifying the overlap of characters in chunks so that the context of the text is preserved and chuck are not cut weirdly. The chunk size was set to 1500 characters and the chunk overlap was set to 200 characters.

Retrieval System

The next step after dividing the text into chunks is converting the chunks into a numerical representation such that they can be stored compared and retrieved easily. These numerical vector representations are called embeddings and are generated using a deep learning model and stored in a vector store database.

Text embeddings

These are the numerical representations of text in a higher dimensional vectors and are used to capture the semantic meaning of the text and relationship between words. This is done using transformer-based models that are trained on large corpus of text using tasks like masked language modelling and next token prediction. Using these trained models, we can get mapping of text to a high dimensional numerical embedding space that preserves the contextual meaning. For example, in embedding space the embedding for a word 'queen' and 'king' would be having more similarity than a text embeddings of words 'queen' and 'earth'. The similarity is measured using cosine similarity which measures the mathematical similarity between vectors. Its value ranges from -1 to 1 where 1 signifies exactly similar, -1 signifies that the vectors are opposite while 0 is when there is no similarity between vectors. In this project we use the Gemini Embeddings model to get the embedding from the text chunks that we had created.

Vectorstores

For efficient storing of the vector embeddings, we use a special databases called vectorstores which are capable of efficient storing and retrieval of high dimensional vectors. These databases use special metric-indexing data structures like k-d trees, ball trees and techniques like locality sensitive hashing (LSH) for storing these high dimensional vectors such that they can be compared and retrieved efficiently. They are also memory intensive as they employ vector quantization and compression for efficient storage. In this project I use the Chroma database for efficient storage and retrieval of vector embeddings. Chroma database is also scalable as it uses distributed also provide distributed indexing, partitioning and load balancing options which can be useful for large scale deployments.

Retriever

The retriever is the main component of the retrieval system which returns the relevant documents or text chunks which are most similar to the user query. But first we must rephrase the user query using the LLM API to accommodate the chat history and the previous context into the follow up question. For example, if the first question is ‘Which is the course at Texas A&M related to Operating Systems?’, to which our LLM answers correctly with the correct course name and detail. If the follow up question to this is ‘What is the grading policy in this course?’ then it would be impossible to determine the whole context of the question if we use the follow up question as is. The retrieval system would not be able to fetch the correct Grading policy without the correct context of the question as for which subject the grading policy needs to be fetched. This is perfectly solved by rephrasing the query with the conversation history.

The rephrased query is then preprocessed to filter out the filler words, stop words and then converted to embedding space so that it can be compared to the text embedding stored in the Vectorstores. The retrieval strategy used starts with using simple Vector Store base retriever which fetches the top chunks with maximum similarity to the query embedding. There are three ways a base vectorstore based retriever can fetch the top similar chunck:

1. **Similarity Based Search**: This is the simplest and default method and takes input a parameter k and fetches top k similar chucks based on vector cosine similarity.
2. **Maximum Marginal Relevance Based Search:** In this method the searches returned are also diverse in nature that is the retriever does not overload the search results with similar kind of chunks thus helps in cases where data is duplicated.
3. **Similarity Score Threshold Search**: In this method instead of specifying number of results we specify the threshold for similarity and the retriever fetches all chunks with similarity greater than the specified threshold.

In this project, I use the basic similarity-based search as do not have similar documents and it is difficult to determine the optimal threshold for similarity. Also, I had added contextual compression on top of base retrieval which will filter out the redundant searches.

Contextual Compression

The document chunks retrieved from the base retriever can be directly sent to the LLM as context for response generation but there are two fallbacks of this method. Firstly, the actual answer may only be a small part of a retrieved document while other text being irrelevant, and this can lead to incorrect or vague responses. Secondly, it would be inefficient in terms of cost and token limit to send these large chunks to LLM as context as it is common issue with large language models to be lost in the long context passed as highlighted by [2] Lui F. et al. (2023). To solve this problem I use contextual compression, which compresses the retrieved documents into a context that this relevant to the query. This is done by filtering the irrelevant content and removing the redundant content in multiple retrieved chunks. The steps followed are as follows:

1. Use Recursive Character Splitter to reduce the retrieved documents to even smaller chunks.

2. Redundant elements are filtered using the *EmbeddingRedundantFilter* and the *EmbeddingsFilter* is used to filter only the relevant elements based on the similarity to the query.
3. The *LongContextReodrer* is used to re-order the relevant parts of the context to the start and end of the context as LLM face the issue of losing information from long contexts which can be solved having some of the relevant information at the end as well.

Below is the code snippet used for creating the base retriever and the contextual compression retriever.

```
def create_retriever(
    vector_store,
    embeddings,
    base_retriever_search_type="similarity",  # Specify the default search type for the base retriever
    base_retriever_k=16,  # Specify the default value for 'k' (number of retrievals) for the base retriever
    compression_retriever_k=20,  # Specify the default value for 'k' for the compression retriever
):
    # Create the base retriever using the provided vector store and settings
    base_retriever = Vectorstore_backed_retriever(
        vectorstore=vector_store,
        search_type=base_retriever_search_type,
        k=base_retriever_k,
        score_threshold=None,
    )

    # Create the compression retriever using embeddings and the base retriever
    compression_retriever = create_compression_retriever(
        embeddings=embeddings,
        base_retriever=base_retriever,
        k=compression_retriever_k,
    )
```

Response Generation

Once the context is extracted from the documents using the retriever and the contextual compression. The query and the compressed context is sent to the LLM API for response generation. This is done using a customized RAG prompt template which instructs the LLM to answer the question based only on the context provided. This is required as LLMs can sometimes return irrelevant responses even if it does not know the data or the context to answer the question this is known as hallucination. The model I use in this project is Google Gemini Pro with a token limit of 32670 and 1.6 trillion parameters using a Google AI Studio API from with 6 requests per minute limit.

```python
# response generation
response_generation_llm = ChatGoogleGenerativeAI(
    google_api_key=st.session_state.google_api_key,
    model="gemini-pro",
    temperature=st.session_state.temperature,
    top_p=st.session_state.top_p,
    convert_system_message_to_human=True,
)
```

The main parameters that need to be optimized are the *temperature* and the *top-p* parameters.

- **Temperature**: The temperature determines the amount of randomness or unpredictability that is added to the responses. During the response generation the LLM assigns probability to different tokens based on the previous context. Thus, having a high temperature value increases randomness in these probabilities and thus make the response more creative. While if the temperature is kept low then the LLM response is more predictable and deterministic. The best value for temperature after trails was set to be 0.4.
- **Top-p**: This is a parameter that govern the cutoff point for cumulative probability distribution for token selection when the LLM is generating text. A higher value of top-p makes the model generate a diverse and varied responses, while a lower value of top-p makes the responses more coherent as a smaller number of tokens are selected. For this project I use a top-p value of 0.96

The response generation is handled by using the *ConversationalRetrievalChain* from *LangChain* which handles the integration with *Gemini* API and memory management. To maintain a history of the chat and update it with the current response from LLM is an important task. This is done by maintaining a *ConversationBufferMemory* which keeps a track of last k questions and responses in the conversation and is then passed in the next follow up question for context. In this project we can also have the option to view the source of data which led to the response from LLM. This is an important benefit of RAG over fine tuning LLMs that users can get the actual data from which the responses which increases the trust in the responses as they can be verified manually. Below is the code snippet used for creating a *ConversationalRetrievalChain.*

```python
# Create the ConversationalRetrievalChain
chain = ConversationalRetrievalChain.from_llm(
    condense_question_prompt=condense_question_prompt,
    combine_docs_chain_kwargs={"prompt": answer_prompt},
    condense_question_llm=standalone_query_generation_llm,
    llm=response_generation_llm,
    memory=memory,
    retriever=retriever,
    chain_type=chain_type,
    verbose=False,
    return_source_documents=True,
)
```

Implementation and Chat-bot design

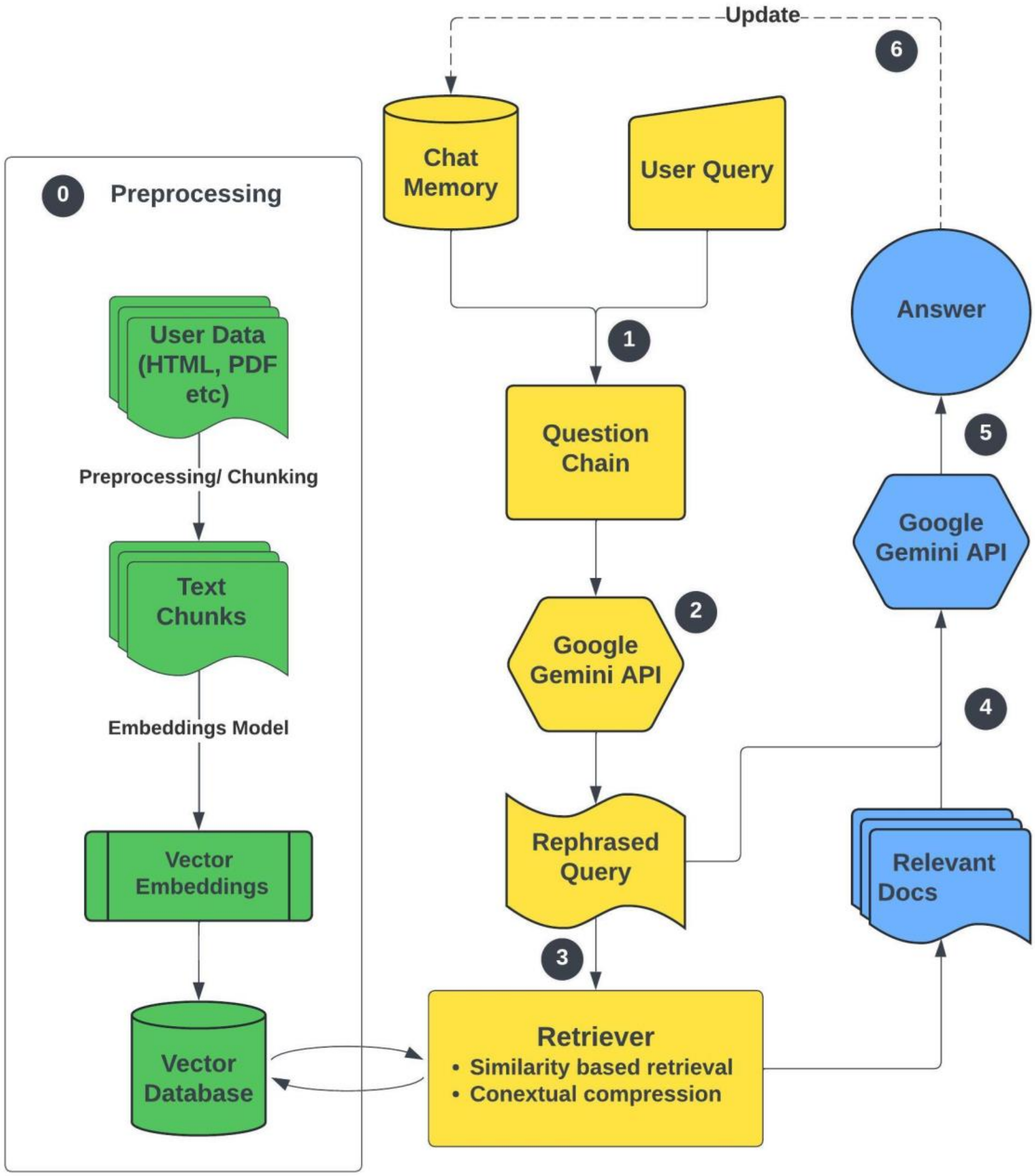


Figure 1. RAG Implementation Flow

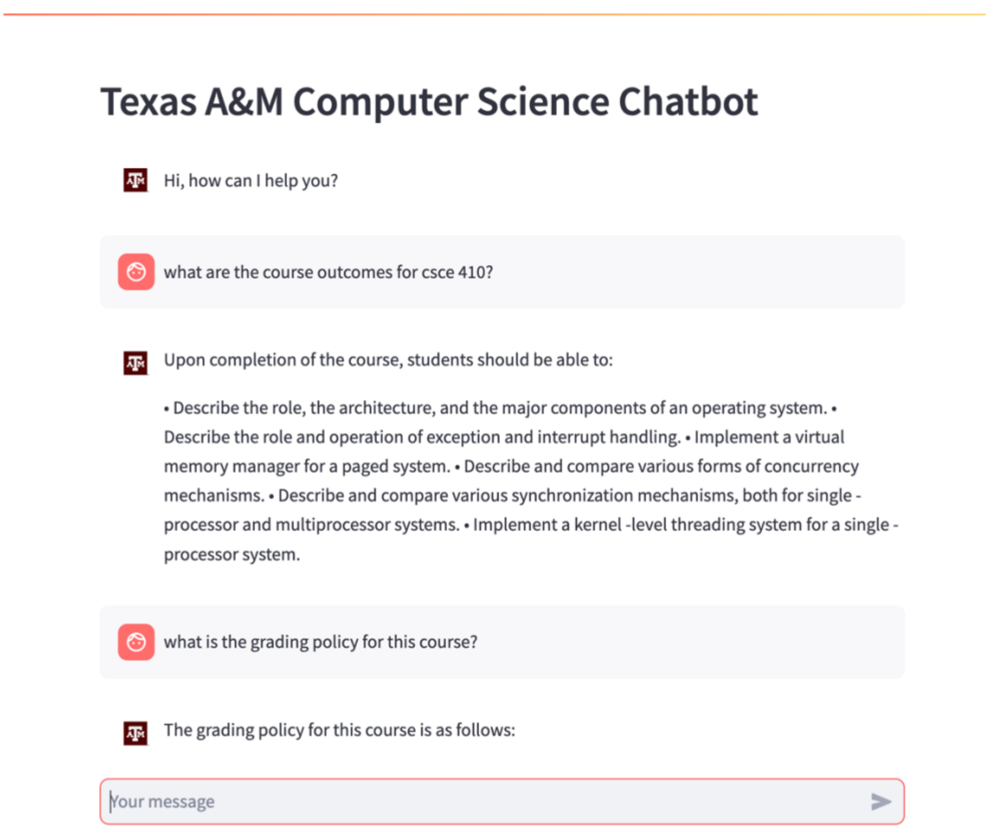
0. **Data Preprocessing and Indexing**
1. **Getting user query and conversation history**
2. **Rephrasing query based on conversation context.**
3. **Retrieval and Contextual Compression**
4. **Rephrased query and Context from relevant docs sent to LLM.**
5. **Response Generation**
6. **Updating Chat History**

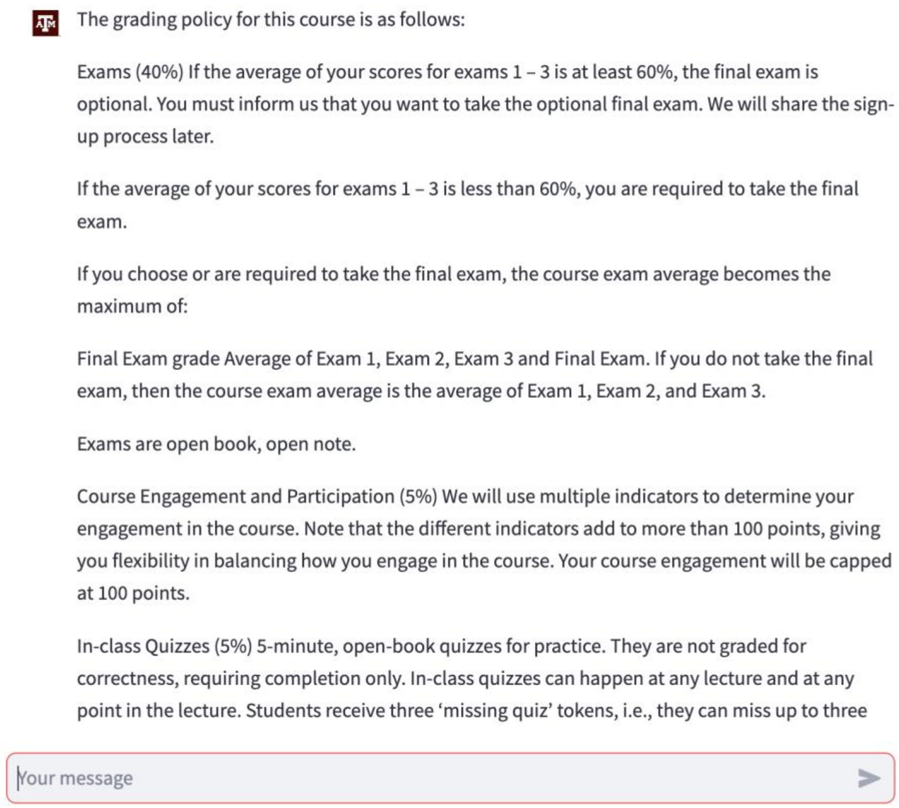
The above flow chart explains the RAG implementation flow for the project. The software part for the project was built using *Streamlit.* It is a python framework used for building interactive web applications. Streamlit is a great choice for building interface for Machine Learning tools, rapid development and integration with various machine learning and data science API. The benefits of using Streamlit are that we can develop both backend development and frontend interface in Python, updates in real time and is easy for sharing and deployment.

Results

The implantation of the RAG system is tested by building a chat bot is designed for Texas A&M Computer Science Department. The syllabus for 25 courses which were openly available online were chosen as dataset. Due to the limit on resources for this project I have currently deployed the app locally on MacBook Pro M2 processor, with Google Gemini Pro API for LLM inference. As the main purpose for the project was to build a RAG system the Chat bot is designed such that users can also upload our own documents and then use the chat bot to query specific questions based on the chat. The hyperparameters like top-p, temperature, chunk size, overlap were experimentally decided based on the responses. The chat bot was successfully able to answer queries from the user as shown below.

Example 1.

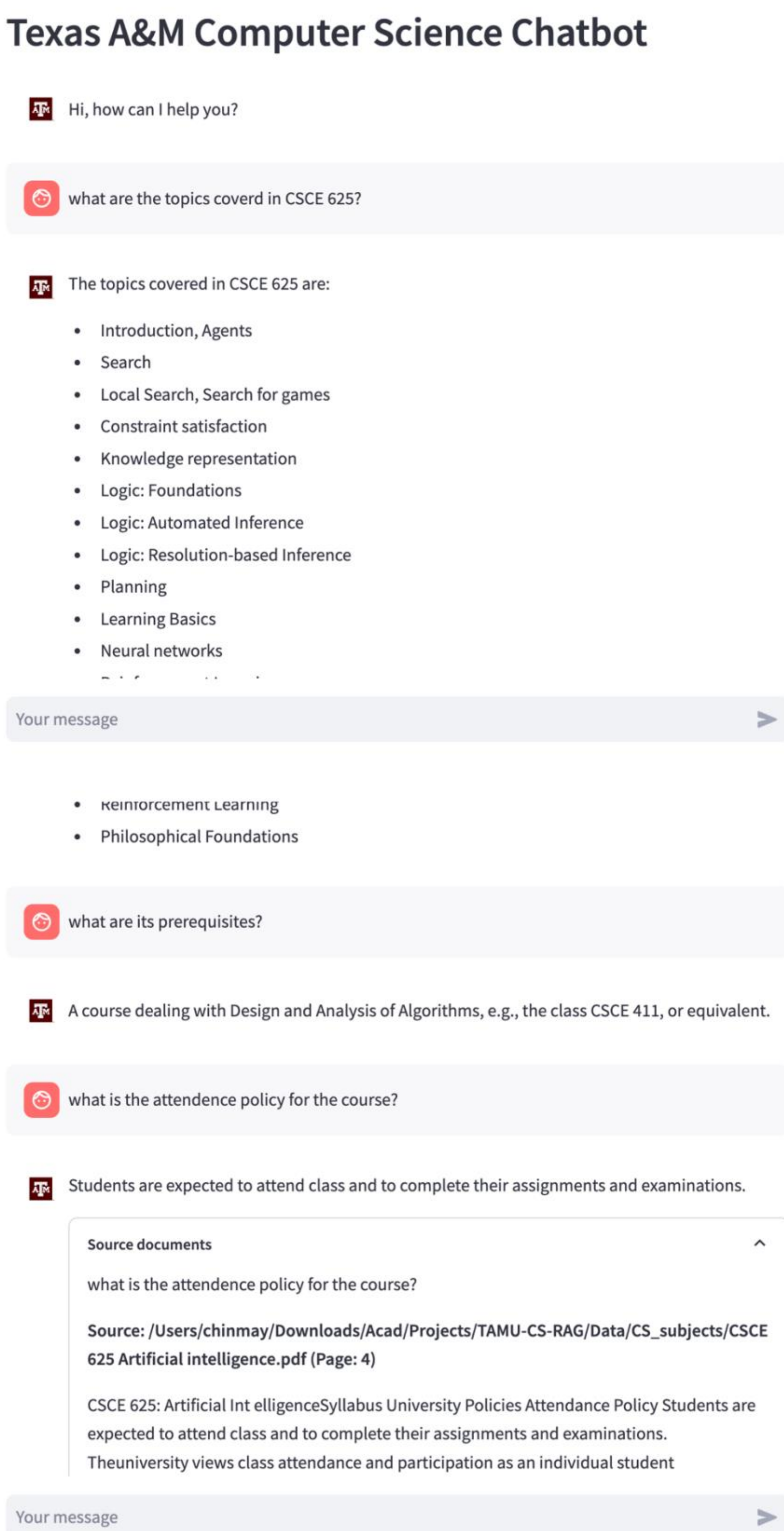




We can see that the retriever system works correctly fetching the course outcomes as per the first query. The next query is however a follow up question the initial query was about CSCE 410 and the next query was about the grading policy for that course the RAG system takes into account the chat history and updates the query for answering correctly.

In another example (Example 2) as shown below the model answers correctly about the topics of the course, then again, we ask if there are any prerequisites for the course upon which the model answers correctly. Finally, when we ask about the attendance policy of the course, we get a generic answer which is mentioned in the syllabus. This can be verified as the RAG chatbot also provides the text chunk and the file and page number from which the model found the relevant answer. This is quite helpful prospect of RAG that we can verify the source of the information.

Example 2:



Texas A&M Computer Science Chatbot
Hi, how can I help you?
what are the topics coverd in CSCE 625?
The topics covered in CSCE 625 are:
Introduction, Agents
Search
Local Search, Search for games
Constraint satisfaction
Knowledge representation
Logic: Foundations
Logic: Automated Inference
Logic: Resolution-based Inference
Planning
Learning Basics
Neural networks
Your message
Reinforcement Learning
Philosophical Foundations
what are its prerequisites?
A course dealing with Design and Analysis of Algorithms, e.g., the class CSCE 411, or equivalent.
what is the attendence policy for the course?
Students are expected to attend class and to complete their assignments and examinations.
Source documents
what is the attendence policy for the course?
Source: /Users/chinmay/Downloads/Acad/Projects/TAMU-CS-RAG/Data/CS_subjects/CSCE 625 Artificial intelligence.pdf (Page: 4)
CSCE 625: Artificial Int elligenceSyllabus University Policies Attendance Policy Students are expected to attend class and to complete their assignments and examinations. Theuniversity views class attendance and participation as an individual student
Your message

Conclusion

The Retrieval Augmented Generation is an important application for efficient and accurate question answering on organizational data. The advances in this field are crucial for automating question answering, summarization tasks in various domains. For example, software engineers could easily explain the code documentation to other software engineers or get accurate answers to their questions without having to go through length documentation. The major challenges faced in this project were building the chucking strategies that would accurately get the relevant content retrieved from the documents. Some other chunking strategies like Parent Document Chunking can also be very useful to improve the performance as the whole document is retrieved in this strategy thus the chances of getting irrelevant chunks is reduced. The performance of the Retrieval Augmented Generation pipeline can be improved by adding a re-ranker which is used to reorder the relevant information after retrieval.

References

1. Gao, Y., Xiong, Y., Gao, X., Jia, K., Pan, J., Bi, Y., Dai, Y., Sun, J. and Wang, H., 2023. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997*.

2. Liu, N.F., Lin, K., Hewitt, J., Paranjape, A., Bevilacqua, M., Petroni, F. and Liang, P., 2024. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics*, *12*, pp.157-173.

3. Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.T., Rocktäschel, T. and Riedel, S., 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. Advances in Neural Information Processing Systems, 33, pp.9459-9474.

4. Siriwardhana, S., Weerasekera, R., Wen, E., Kaluarachchi, T., Rana, R. and Nanayakkara, S., 2023. Improving the domain adaptation of retrieval augmented generation (RAG) models for open domain question answering. *Transactions of the Association for Computational Linguistics*, *11*, pp.1-17.

5. https://docs.streamlit.io/ Streamlit Documentation

6. https://github.com/langchain-ai - LangChain is a framework for developing applications powered by large language models (LLMs).