AI Teaching Assistant Chatbot using RAGs

Anonymous CVPR submission

Paper ID *****

Abstract

Retrieval-Augmented Generation (RAG) has emerged as a pivotal technique to enhance Large Language Models (LLMs) by integrating external knowledge sources during the generation process. This study delves into the implementation and optimization of various RAG methodologies, including Naive RAG, Semantic Chunking RAG, Re-ranker RAG, Hybrid RAG, Self RAG, and Graph RAG. Leveraging tools such as FAISS vector databases, Google Generative AI embedding models, and Whisper transcription models, we develop a comprehensive RAG system. Our findings indicate that each RAG variant offers distinct advantages in terms of retrieval accuracy, response generation quality, and computational efficiency. This research provides a comparative analysis of these RAG implementations, offering insights into their applicability across diverse natural language processing tasks.

1. Introduction

Large Language Models (LLMs) have demonstrated remarkable capabilities in various natural language processing tasks, including text generation, question answering, and summarization [1, 4, 7]. However, these models often suffer from hallucinations and static knowledge limitations, especially when dealing with knowledge-intensive queries [3, 5]. One promising approach to address these challenges is Retrieval-Augmented Generation (RAG), which enhances LLMs by integrating external knowledge sources during the text generation process [3, 7].

RAG systems combine two key components: a retriever that fetches relevant documents from a large corpus and a generator that uses the retrieved documents to produce responses [4, 6]. The retriever can be based on sparse methods, such as BM25, or dense methods, such as FAISS vector databases, which encode documents into high-dimensional embeddings for efficient similarity search [2, 5]. Recent advancements in dense retrieval models, such as Dense Passage Retrieval (DPR) [5] and ColBERT [6], have significantly improved the retrieval accuracy and relevance in RAG systems.

Different variations of RAG have been proposed to address specific challenges in information retrieval and generation. Naive RAG divides documents into fixed-size chunks and retrieves the most relevant chunks for each query [7]. Semantic Chunking RAG improves upon this by splitting documents based on semantic coherence rather than fixed character lengths, making retrieval more accurate and contextually relevant [4]. Re-ranker RAG incorporates a cross-encoder to re-rank the retrieved documents, improving retrieval precision [6].

Hybrid RAG systems combine multiple retrieval strategies, such as BM25 and dense retrieval, to balance precision and recall [2, 3]. Self RAG introduces reflective mechanisms to iteratively refine the retrieval process based on the generated responses, reducing hallucinations and improving the quality of generated content [9]. Graph RAG, on the other hand, models relationships between document chunks as a graph structure, enabling more advanced reasoning and multi-hop retrieval [8].

The main contributions of this work are as follows:

- We implement and evaluate six different RAG methodologies, including Naive RAG, Semantic Chunking RAG, Re-ranker RAG, Hybrid RAG, Self RAG, and Graph RAG.
- We leverage state-of-the-art tools such as FAISS, BM25, and Google Generative AI embeddings to build a comprehensive RAG system.
- We provide a comparative analysis of these RAG implementations, highlighting their strengths and limitations across various knowledge-intensive tasks.

By systematically comparing these RAG methods, we aim to provide actionable insights for deploying RAG systems in real-world applications, including question answering, knowledge-grounded dialogue, and summarization tasks [1, 3, 7].

2. Related work

Large language models (LLMs), which have shown excellent ability in information retrieval and summarization, have been adopted as irreplaceable tools for scientific research and other domains. However, such powerful tools do have their limitations when faced with Knowledge-Intensive Tasks, as their knowledge base would have difficulties in real-time updating, and the parameter scale would have memory limitations. Despite all these difficulties, the Retrieval Augmented Generation(RAG) provides us with a promising future.

2.1. Retrieval-Augmented Generation

Retrieval-Augmented Generation(RAG) is a natural language processing framework which acquires the ability to acquire dynamic knowledge and respond in real-time. RAG proposed a new framework, which combines a retriever and a generator, introducing an external knowledge base to deal with the limitation of the generation model in Knowledge-Intensive Tasks, thus bringing us a significant performance improvement. The two key components of RAG are the retriever and the generator, a retriever can match relevant documents from its inner and external knowledge base, and a generator can use these documents to form a high-quality output.

2.2. Retriever

A retriever is what a dynamic knowledge basement is all about, and its ability to retrieve relevant documents in an external knowledge basement directly determines the ability of this model to perform the Knowledge-Intensive Tasks. A retriever's work are as follows:

- Information Retrieval: retrieve relevant documents.
- Dimension Reduction: reduce the variables to improve computational efficiency.
- Filtering: filter the relevant documents.
- Context Augmentation: add information based on existing contents to provide a complete contents for generator.

Retrievers are typically divided into Sparse Retrieval and Dense Retrieval.

2.3. Generator

A generator is another key component in the Retrieval-Augmented Generation framework, which is used to produce answers based on the relevant documents being retrieved.

The main task of a generator is to:

- Generate answers based on retrieved context.
- Enhance the quality of the response.
- Handle complex language tasks.

Generators in RAG are mostly pre-trained language models, such as Transformer-based models and the Sequence-to-Sequence(Seq2Seq) Model.

2.4. Sparse Retrieval

Sparse retrieval represents documents as sparse vectors and uses keywords to find relevant documents. When dealing with large-scale text collections and does not require complex model training beforehand, this method is great efficiency. BM25 is a probabilistic model used in sparse retrieval. It applies weighted adjustments to term frequencies and normalizes document lengths, which is very suitable for sparse matching. While such model significantly improves phrase matching, its ability to understand semantics is limited.

2.5. Dense Retrieval

Dense retrieval is a new method proposed in recent years. It uses deep learning models to generate embeddings, which can be used in encoding documents and queries into low-dimensional dense vectors. These vectors perform well in capturing semantic information, enabling more intelligent text matching. Facebook AI Similarity Search(FAISS) is an open-source library, which is designed for fast similarity search of dense vectors. It offers various search algorithms and is capable of handling billions of vectors efficiently.

3. Approach

This session is about the datasets processing and the algorithm. We took our course materials as our datasets, and we built different functions to construct all kinds of RAGs to test their performance.

3.1. Datasets

The datasets we used come from the lectures materials, including lecture videos, pre/post class readings, lectures' powerpoint documents and the labs. When facing the 200Mb limitation of each single document in calling API, we divided the videos larger than 200Mb into several parts, each smaller than the limitation.

3.2. Naive RAG

Naive RAG is the prototype of all RAG models and is the most simple one. Naive RAG searches for relevant documents using a simple retrieval strategy with fixed text fragmentation length and cosine similarity for embedding-based retrieval.

Naive RAG splits text into fixed-length chunks:

```
paragraphs = [text[i:i + chunk_size]
for i in range(0, len(text), chunk_size)]
```

Then do the embeddings:

```
similarities =
cosine_similarity([query_embedding],
paragraph_embeddings)

most_similar_indices =
np.argsort(similarities[0])[::-1][:n_results]
```

3.3. Semantic Chunking RAG

Semantic Chunking RAG enables the model to retrieve relevant documents based on semantic meanings, making it possible to match phrases that differs from the query.

To construct semantic chunking, we use the codes as follows:

```
semantic_text_splitter =
SemanticChunker
(GoogleGenerativeAIEmbeddings
(model="models/embedding-001"))

semantic_chunks =
semantic_text_splitter.
create_documents([text])
```

And the semantic retriever is built as follows:

```
semantic_vector_store =
FAISS.from_documents
(semantic_chunks, embeddings)

semantic_retriever =
semantic_vector_store.as_retriever()
```

In that way we can build a semantic chunking RAG.

3.4. Re-ranker RAG

The key point about Re-rank RAG is that after the retriever finishes its task retrieving all the relevant documents, Re-rank RAG would have cross-encoder to re-rank the relevance of the retrieved documents. By doing so, this would help improve the precision and the relevance.

The codes are as follows

```
model = HuggingFaceCrossEncoder
(model_name="BAAI/bge-reranker-base")

compressor = CrossEncoderReranker
(model=model, top_n=3)

compression_retriever =
ContextualCompressionRetriever
(base_compressor=compressor,
base_retriever=semantic_retriever)
```

Only the most relevant documents would be accepted, thus significantly improves the performance of the RAG model.

3.5. Hybrid RAG

Hybrid RAG is the combination of traditional RAG(naive RAG) with RAG models with semantic understandings, which would provide with better performances.

The codes are as follows:

```
bm25_retriever =
BM25Retriever.from_documents
([Document(page_content=text)])

ensemble_retriever =
EnsembleRetriever
(retrievers=[bm25_retriever,
compression_retriever], weights=[0.5, 0.5])
```

BM25 and semantic retrievers are combined together to reconstruct the weight.

3.6. Self RAG

Self RAG uses an inner generator to generate content, which was added into its knowledge base to be used for training. This would leads to quick self-iteration and requires less training data.

For our project, we use semantic text splitter to get text chunks and use FIASS to construct the vector store and retrieve.The initial answer is generated based on the query and the most similar chunk. Then, within the set number of iteration rounds, relevant documents are retrieved based on the answer. The new context and the previous answer are combined and input into the LLM to obtain a new answer again. The RAG rules are strictly followed, and answers are generated based solely on the given context, until no more documents can be retrieved or the iteration limit is reached.

```
retrieved_docs = semantic_retriever.
get_relevant_documents(refined_answer)

if not retrieved_docs:
    break
more_context = "\n\n".join
([doc.page_content for doc in retrieved_docs])

refine_prompt = [
    {
        "role": "system",
        "content": (
            f"Below is the initial answer
            to the user's query: ..."
        ),
```

```
    },
    {"role": "user", "content": query},
]
new_answer =
llm.invoke(refine_prompt).content
refined_answer = new_answer
```

This enables the initial context to be used as a query to retrieve additional relevant documents.

3.7. Graph RAG

Graph RAG is a new framework which enables the RAG model to deal with graph-based knowledge representations. The key point of Graph RAG is about generating the Knowledge Graphs(KG), which would be used to match the query, making graph content available for RAG models to retrieve.

For traditional rags, there are still limitations. LLMs have limited context windows, preventing global understanding by processing large volume of text. Besides, traditional RAG is not able to capture the holistic relationships of the chunks, which can be fixed by the Knowledge Graphs.

Here is the brief summary of the steps of graph rag. First, we use llm prompts to extract nodes and edges form the splitted text chunks, where the nodes refers to the entities, the edges represent the relationships between entities, the attributes describe these relationships. All these three elements are stored as triples (subject-relation-object). Then we use community detection algorithms such as Leiden to partition the graph into communuties of closely related entities.For the last step, we generate summaries for these communities, and then leverage these structures when perform RAG-based tasks.

For our project we wrote a simplified rag method which has the similar idea of graph rag. We built a graph with only subject and objects, which means it's a undirected graph, and there's no relation description between two nodes, the nodes is connected according to the cos-similarity with the key called neighbors. Then we append the neighbors text as 'relevant context' for final input.

The codes are as follows

```
 graph[i] = {
        'content': chunk.page_content,
        'embeddings': embeddings.embed
        _query(chunk.page_content),
        'neighbors': []
    }
for i in range(len(semantic_chunks)):
    paths = 0
    for j in range(i + 1, len(semantic
    _chunks)):
        similarity = cosine_similarity(
        [graph[i]['embeddings']],
        [graph[j]['embeddings']])[0][0]
        if similarity > 0.5 and paths <
        max_paths_per_chunk:
            graph[i]['neighbors'].append(j)
            graph[j]['neighbors'].append(i)
            paths += 1
```

4. Experimental Results

We use Streamlit to carry the project code, which allows us to build a simple UI for texting queries and generating answers. First, we load the Google Gemini API Key to call the API and import Whisper for video-to-text processing. This project supports multiple types of files, including text, PDF, MP3, and MP4, which enables us to load as many lecture materials as possible. Additionally, the function rag-implementation is used to map and manage various RAG models, allowing us to conveniently switch to different RAG models, making it easy to compare the features and capabilities of different RAG implementations.

To evaluate the performance of different RAG models, we designed dozens of diverse questions and uploaded files with answers inside. The uploaded files can be of various formats.For each question and each answer, We would give it a score of how well the responses generated by different RAG models are by comparing the standard answers and the generated answers. This approach allows us to get quantifiable scores for each RAG model.

Here are some examples of the query and the responses:

RAG model: Self RAG

Query: What is the use of rag?

Response:Retrieval-Augmented-Generation (RAG) improves the performance of Large Language Models (LLMs).especially for tasks requiring accurate and detailed information. lt addresses LLM limitations such asinfrequent training data updates and lack of domain knowledge in specialized areas. RAG retrievesrelevant information from a knowledge base to augment the LLM's response, overcoming issues likehallucinations and inability to cite sources.

Score: 80, this text can tell what RAG is, yet it didn't take all the key points we consider relevant.

RAG: Semantic Chunking RAG

Query: What can PEFT do?

Response: Parameter-Efficient Fine-tuning (PEFT) enables fine-tuning large language models iLLMs) withsignificantly fewer trainable parameters, reducing computational and storage costs while achievingcomparable performance to full fine-tuning, This allows tuning large LLMs on consumer hardware andavoids issues like catastrophic forgetting. PEFT methods train a small set of extra parameters added to thepretrained LLM, allowing a single LLM to be used for multiple tasks by simply switching these small sets ofweights, The r PEFT library provides these techniques integrated with s Transformers and rAccelerate. Currently

supported PEFT methods include LoRA, Prefix Tuning, Prompt Tuning, and P-TuningIt can be applied to various modalities, including image classification and stable difusion dreambooth.

Score: 60, though the text can take most of the key points we set, it had some meaningless emoji in its text, which is considered as a significant drawback.

RAG Model: Self RAG

Query: What is the advantage of graph rag?

Response: Graph RAG uses knowledge graphs to provide substantial improvements in question-and-answerperformance when reasoning about complex information,especially where Baseline RAG struggles.Specifically, it addresses Baseline RAG's weaknesses in connecting disparate information andunderstanding summarized semantic concepts over large data collections or individual large documents.

Score: 90, it is not very long, but contains most of the key points.

RAG Model: Graph RAG

Query: How does Retrieval-Augmented Generation work?

Response: RAGs operate with a few main steps to enhance generative Al outputs: Retrieval and pre-processing: RAGs leverage powerful search algorithms to query external data, such asweb pages, knowledge bases, and databases. (from documentID'3432f181-f38d-4367-bc85-dofad520f244') Grounded generation: The pre-processed retrieved information is then seamlessly incorporated into thepre-trained LLM. This integration enhances the LLM's context, providing it with a more comprehensiveunderstanding of the topic. This augmented context enables the LLM to generate more precise,informative, and engaging responses. RAG operates by first retrieving relevant information from adatabase using a query generated by the LLM. This retrieved information is then integrated into the LLM'squery input, enabling it to generate more accurate and contextually relevant text. Retrieval is usuallyhandled by a semantic search engine that uses embeddings stored in vector databases and sophisticatedranking and query rewriting features, ensuring that the results are relevant to the query and will answerthe user's question,(from document ID 'a33d4c4c-f247-4ec3-b226-06c67623d2b4')

Score: 85, it gets all of the key points, yet it is too long, and its phrase is too similiar to the original file, which might be demonstrating that the model as some weakness in summarizing.

It can be seen that in this project, different types of RAG models can successfully perform in text retrieving and provide responding. Among them, the naive RAG, as the most basic RAG model, performs the worst. The other models have improved their performance through various methods.

5. Conclusion

This article is mainly about different models of RAG, we talked about the principles of Retrieve Augmented Generator, mostly about the retriever and the generator and the sparse retriever and the dense retriever. We also construct six types of RAG models to make a teaching assistant chatbot. We tested the performance of different type of RAG models, and these all contributed to the improvement of the chatbot, especially the graph rag enabled our model to deal with multimodal tasks.

References

[1] Sebastian Borgeaud, Arthur Mensch, Jordan Hoffmann, Trevor Cai, Eliza Rutherford, Katie Millican, and George Irving. Improving language models by retrieving from trillions of tokens. In *ICLR*, 2022. 1

[2] Luyu Gao and Jamie Callan. Condenser: A pre-training architecture for dense retrieval. 2021. 1

[3] Kelvin Guu, Kenton Lee, Zora Tung, Panupong Pasupat, and Ming-Wei Chang. Retrieval augmented language model pre-training. In *ICLR*, 2020. 1

[4] Gautier Izacard and Edouard Grave. Leveraging passage retrieval with generative models for open-domain question answering. In *NeurIPS*, 2020. 1

[5] Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen tau Yih. Dense passage retrieval for open-domain question answering. 2020. 1

[6] Omar Khattab and Matei Zaharia. Colbert: Efficient and effective passage search via contextualized late interaction over bert. 2020. 1

[7] Patrick Lewis, Ethan Perez, Aleksandara Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Kuttler, Mike Lewis, Luke Zettlemoyer, and Sebastian Riedel. Retrieval-augmented generation for knowledge-intensive nlp tasks. In *NeurIPS*, 2020. 1

[8] Gaspard Mialon, Riccardo Dessì, Marco Lomeli, Jean-Baptiste Alayrac, Antoine Joulin, Edouard Grave, and Bea Krause. Augmented language models: A survey. *arXiv preprint arXiv:2101.00296*, 2021. 1

[9] Ramesh Thoppilan, Daniel De Freitas, Jamie Hall, Noam Shazeer, Apoorv Kulshreshtha, Heng-Tze Cheng, Alicia Jin, Patrick Nguyen, Zhifeng Chen, and Alexis Lee. Lamda: Language models for dialog applications. In *NeurIPS*, 2022. 1