A SELF-REFLECTIVE RETRIEVAL AUGMENTED GENERATION SYSTEM TO ELIMINATE HALLUCINATION IN LLM GENERATION

Haotong Hu[1], Garret Washburn[2]

[1]Basis Bilingual School, Shenzhen, Shenzhen, China
[2]Computer Science Department, California State Polytechnic University, Pomona, CA 91768

ABSTRACT

In a Retrieval Augmented Generation system, the possibility of hallucination is ever prevalent, as it has been increasingly common for an AI model to produce inaccurate generations given an input, which can become increasingly more devastating for a RAG system due to its node-like generation. To solve this problem, we introduced a series of self-reflective nodes, creating a self-RAG model. To create our program, we established a series of nodes that utilized ChatGPT for generation as well as the Langchain python library and its workflow functions to piece together the nodes. A major challenge faced during the planning and development stages was the creation of the self-reflective nodes within the workflow, as these proved to be difficult to implement. To test the accuracy and capacity of this proposed Self-RAG model, multiple experiments were carried out and the result ensures a quality generation. This model should be implemented because it avoids misleading and inaccurate answers.

KEYWORDS

Rag, LLM, Self-RAG, Self-Reflective

1. INTRODUCTION

Modern day AI, while most times useful, often provides inaccurate answers to questions that require information that the AI model simply does not have access too. Additionally, in the event that the AI model does have the information necessary to answer a question, it can still produce an answer that is very unclear or not conducive to proper learning or communication. These inaccuracies in the AI responses can have great impact depending on the application, considering practical AI is becoming more and more prominent in many industries such as software development and even medical fields. “In a 2018 Deloitte survey of 1,100 US managers whose organizations were already pursuing AI, 63% of companies surveyed were employing machine learning in their businesses.” Such a problem, though seemingly small, would lead to detrimental effects in the long run. Software developed based on inaccurate generative AI would have problems in its algorithms that are hard to locate in the long run. If these problems are not addressed immediately, there would be more and more problems leading to the collapse of the system or the production of completely inaccurate output. In the medical field, the patient's life is dependent on such an erroneous system.

David C. Wyld et al. (Eds): DMSE, CSEIT, NeTCoM, SPM, CIoT, NCS, NLPD, ArIT, CMLA – 2024
pp. 19-26, 2024. CS & IT - CSCP 2024 DOI: 10.5121/csit.2024.141402

Included in this paper, are three related methodologies that attempt to solve the same problem that the methodology proposed in this essay does. These three methodologies are RQ-Rag, DR-RAG, and VistaRAG.

RQ-Rag is a RAG model that attempts to prevent hallucination by refining the query, making sure the model doesn't go off track. Although this model refines the query, it is still not self - reflective upon the answers generated which may result in an inaccurate or unrelated answer from an ineffective question.

DR-RAG is another RAG model designed to prevent hallucination by dynamically retrieving relevant information from the documents prior to answering the query. The refined documents are then used to answer the query, however, this poses the problem of potentially grabbing irrelevant or discarding relevant information to answer the query accurately.

VistaRAG is a RAG system that aims to reduce inaccurate and misleading results by checking the generation against real life data. Such an implementation is typically used for automotive cars. The limitations of this system is that while there is comparison with real life data, there is no assurance that the real life data is correct. Furthermore, the real life data doesn't necessarily reflect the particular situation one is in and thus may lead to ineffective decisions.

By implementing self-RAG into the question-answering AI model, hallucination and information without accuracy could be erased or at the very least, decreased by a large margin. The self-RAG system solves this problem by checking the relevant documents and generating an answer only through the relevant information, as the nature of the AI model processing system is self-reflective and able to determine if it has grabbed useless information. This eliminates the chance that unrelated ideas and facts are put into the final generated answer. Not only would this system check the relevant documents, it would also self assess the generated answers it produces. By double checking - or checking however many times until the model deems it's answer appropriate - the AI model could successfully locate the relevant information and generate an accurate answer through the information given. self-RAG serves as a better model than other models, like hybrid search, because even though hybrid search could better locate the relevant document, it does not re-asses itself to make sure the information it extracted is correct. For instance, if the hybrid search model extracts deceitful information in the beginning, no matter how profound the generation is, the information is still useless or misleading from the start as it is unable to go back and grab more information that is actually useful, making the model move further but in the wrong direction. Thus, to ensure that the correct information is given from the start and processed later in the process,a self-RAG system is the most effective solution.

Within this paper, to verify the accuracy of the proposed Self-RAG model methodology, experiments were performed to test its capability and accuracy. Within the first experiment displayed, the focal point of the experiment was to examine the accuracy of the model compared to how many different questions were asked. What we found from this experiment was that the model, at around over 3 different questions, started to become inaccurate or completely unrelated to any question. In the second experiment, the intention was to find the relationship between the intricacy of one question containing various sub-questions and the accuracy of the generation. What we found was that the model stayed consistently accurate, despite the complexity of the original question. This result piqued our interest, as in the former experiment we found the model to break at around 4 questions, which contradicts how the model performed in the second experiment. We believe the second experiment's result occurred because the Self-RAG model was able to rephrase the question to be easier to answer, as opposed to the former experiment which included a few differing questions.

2. CHALLENGES

In order to build the project, a few challenges have been identified as follows.

2.1. The create_workflow Function

One major component of the self-reflective-RAG algorithm that is presented in this paper is the create_workflow function. This function acts as place to 'connect the dots,' or nodes in this case, that make up the entire self-reflective-RAG model. One challenge that was faced during the development process was the managing of the 'state' parameter. Within the workflow, defined by the langchain library, the state is a json string that is passed around to all of the workflow functions, and is constantly updated to reflect the part of the process that the model is currently in. Managing this 'state' parameter was particularly difficult because all of the necessary information needed for the self-reflective-RAG model to function is stored in the state and is updated in every workflow node. Keeping the data updated was quite a task.

2.2. The create_docs() Function

Another significant aspect of the self-reflective-RAG algorithm is the create_docs() function which loads all the given webpages as resources that could be parsed into documents understandable by the AI model. The challenging aspect we encountered during the development process was creating the retriever for the documents and having it be accessible for the workflow functions. Originally, the creation of the retriever was it's own node in the workflow, however, this would not work, as the retriever must be created with all of the documents beforehand, and cannot be accessed from the workflow. Eventually, the solution that we came to was to pass the retriever as an item in the input 'state,' so that it may be accessed by all necessary workflow nodes.

2.3. The Overall Grading Process

Probably the most important aspect of this project we had to consider was how the overall grading process of the self-reflective-RAG model worked. Originally, the relevance of each document was checked and documents that were graded as irrelevant were deleted. The rest of the relevant documents went through the final grading and generating. However, when the final generated answer appeared, it is evident that only the most relevant generation was used, not all the relevant generations that came out of the algorithm. Upon reviewing this strategy for self-reflecting, we determined that grading the generations was a waste of time, as the algorithm would only grab the most relevant generation anyway. To fix this problem, we decided to include all generations that passed the grading to be combined into one more comprehensive generation.

3. SOLUTION

Taking a look at the overall structure of the self-reflective-RAG model, the three major components that are linked together are the workflow, the retriever for the documents, and the actual workflow functions themselves. The first step in the processing of the self-reflective-RAG model is the creation of the retriever, which involves loading all of the url's given to the model as webpages and converting them into documents which can be processed but the AI model. After this, the documents are parsed and checked to see if they are relevant to the question. Upon creating the usable documents, the workflow is created, which involves setting up the flow for all of workflow nodes so that the data may be properly processed. After creating the workflow, the input state is created, which only includes the question and documents, and fed into the workflow.

Upon being fed into the workflow, the workflow continues to make changes with the state, and upon visiting every node in the state, the returned state now contains the final generation which can be accessed and displayed. To implement this program, we used the langchain framework. During our generation, the rag prompt is called from the langchain framework hub that langchain has made available so that we did not have to create our own prompt. The most used modules from the langchain library that are implemented into this project are from the langchain_openai library, as the ChatOpenAI module allows the program to make queries to ChatGPT within the workflow.

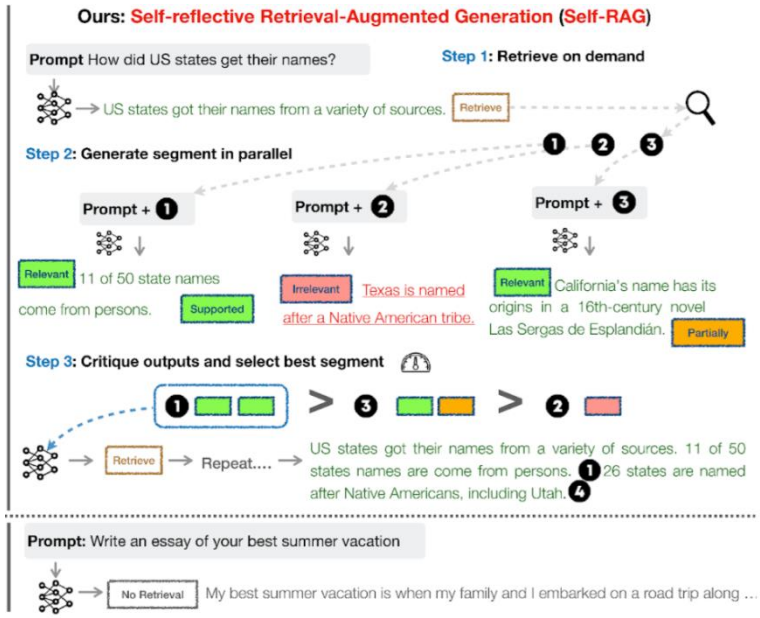

Figure 1. Overview of the solution

The purpose of the create_workflow() function is to link all of the workflow nodes together, in order to create the whole node by node workflow. Inside of this function, all of the nodes are defined and also fit properly into what will become the ‘workflow’ of the self-reflective-RAG processing system.

```python
def create_workflow():
  workflow = StateGraph(GraphState)

  # Define the nodes
  workflow.add_node("grade_documents", grade_documents)  # grade documents
  workflow.add_node("generate", generate)  # generatae
  workflow.add_node("transform_query", transform_query)  # transform_query
  workflow.add_node("prepare_for_final_grade",
                    prepare_for_final_grade)  # passthrough
  workflow.add_node("grade_generation_v_documents",
                    grade_generation_v_documents)
  workflow.add_node("grade_generation_v_question", grade_generation_v_question)

  # Build graph
  workflow.set_entry_point("grade_documents")
  workflow.add_edge("grade_documents", "generate")
  workflow.add_conditional_edges(
      "grade_documents",
      decide_to_generate,
      {
          "transform_query": "transform_query",
          "generate": "generate",
      },
  )
  workflow.add_edge("transform_query", "grade_documents")

  workflow.add_edge(
      "generate", "grade_generation_v_documents"
  )  # generate -> grade_generation_v_documents -> grade_generation_v_question
  workflow.add_edge("grade_generation_v_documents",
                    "grade_generation_v_question")
  workflow.add_edge("grade_generation_v_question", "prepare_for_final_grade")
  workflow.add_edge("prepare_for_final_grade", END)

  return workflow.compile()
```

Figure 2. Screenshot of code 1

The image above showcases the function to create the workflow. Within the first half of the function, all of the nodes are defined utilizing the ‘add_node’ function that is a part of the workflow object defined at the very top of the function. To actually define the path of the workflow, various functions are employed. To start, the ‘set_entry_point’ function is used to define the entry point of the workflow, which is set to the grade_documents node. At the grade_documents node, all of the documents inside fo the retriever are graded and if deemed useless will be dropped and not used in the algorithm. After setting the entry point, all of the edges are defined, starting with the ‘grade_documents’ to the ‘generate’ edge. Upon hitting the generate edge, generations that seek to answer the question are created using each document. After creating each generation, we define all the necessary nodes for generating the final answer including nodes such as ‘transform_query,’ ‘grade_generation_v_documents,’ and ‘grade_generation_v_question.’

The intended use of the create_docs() function is to create the retriever, which loads urls that are given by the user into webpages and converts them to documents that will be passed into the workflow and used by the nodes. The documents would then be split into different chunks and each chunk would be contained in the retriever.

```python
def create_docs(documents):
  # urls = [
  #     "https://en.wikipedia.org/wiki/International%20Federation%20of%20
  #     "https://www.pewresearch.org/short-reads/2023/02/09/key-facts-as-
  #     "https://en.wikipedia.org/wiki/List%20of%20FIFA%20World%20Cup%20r
  # ]
  docs = [WebBaseLoader(url).load() for url in documents]
  docs_list = [item for sublist in docs for item in sublist]

  text_splitter = RecursiveCharacterTextSplitter.from_tiktoken_encoder(
      chunk_size=250, chunk_overlap=0)
  doc_splits = text_splitter.split_documents(docs_list)

  # Add to vectorDB
  vectorstore = Chroma.from_documents(
      documents=doc_splits,
      collection_name="rag-chroma",
      embedding=OpenAIEmbeddings(),
  )
  #overwrite retriever
  return vectorstore.as_retriever()
```

Figure 3. Screenshot of the calendar

Within in the create_docs function, the first thing that is done is the urls that are passed as documents to this function are loaded by their url. After each url is loaded, the loaded docs are then separated into sublists, so that they are in smaller chunks and easier to understood by a llm. After this step, the text_splitter variable is initialized, and in it is stored the actual text_splitter object specified with how big each chunk should be. The doc_splits variable stores what is returned with the text_splitter object is called to split the docs_list. Finally, the vectorstore object is initialized, containing the doc_splits as the documents, the “rag-chroma” model as the collection_name, and the OpenAI Embeddings as the embedding. The vectorstore is then returned as a retriever object to be used in the workflow. This retriever will go on to be used solely to get relevant documents pertaining to the question that is asked to the rag system.

The unique background functions that are linked together in the create_workflow() function to create the workflow path are what are called the workflow functions or the ‘nodes.’ These nodes are all unique in that they each serve a different purpose in the workflow, however, each is only allow the one parameter ‘state,’ as this standardizes the parameters and ensures all will be able to be linked together linearly.

```
def prepare_for_final_grade(state):
    """
    Passthrough state for final grade.

    Args:
        state (dict): The current graph state

    Returns:
        state (dict): The current graph state
    """

    print("---FINAL GRADE---")
    state_dict = state["keys"]
    question = state_dict["question"]
    documents = state_dict["documents"]
    generated_prompts = state_dict["generation"]

    prompt = PromptTemplate(
        template=
        """You are a English writer that summarizes multiple statements into one statement. \n
        Here is a collection of statements: \n\n {generation} \n\n
        summarize these statements into one statement.""",
        input_variables=["generation", "question"],
    )

    # Grader
    model = ChatOpenAI(temperature=0, model="gpt-4-0125-preview", streaming=True)

    print("BELOW IS THE GENERATED PROMPTS")
    print(generated_prompts)

    # Prompt
    chain = prompt | model | StrOutputParser()
    final_answer = chain.invoke({
        "generation": generated_prompts,
        "question": question
    })

    return {
        "keys": {
            "documents": documents,
            "question": question,
            "generation": generated_prompts,
            "final_answer": final_answer
        }
    }
```

Figure 4. Screenshot of code 3

The pepare_for_final_grade node is one of the several nodes used. It also appears to be the final node in the workflow. The very first that is always done in all of the workflow nodes, is the retrieval of the items within the state parameter that is passed between the nodes. This retrieval is necessary as it permits the use of the generate items from the previous nodes within the current node, wherever that may be. After retrieving these items from the state, the prompt template is defined, which is essentially the instruction that is given to chatgpt, including some defined input variables and where to fit them into the prompt. After, the ChatOpenAI model and the chain are created. Finally then, the chain is invoked containing the actual input variables to be used, and then the new state is created and returned, having been updated with the new item to be stored.

4. EXPERIMENT

4.1. Experiment 1

One potential blind spot that we have identified in the application is how long and how many questions can be posed to our rag system. It is important that this complexity be handled effectively, as it can generate potentially incorrect answers or waste resources.

For this experiment, the goal is to test how the amount of questions posed to the rag system with the appropriate documents will affect the generation that is returned. To test this, we intend to continue to ask differing questions, adding a question and a documen as we move along, until the generation we receive implies an incorrect or incomplete answer. By conducting the experiment this way, we intend to discover how the amount of questions we psoe will affect the resource management by measuring the time and how accurate the rag system is.

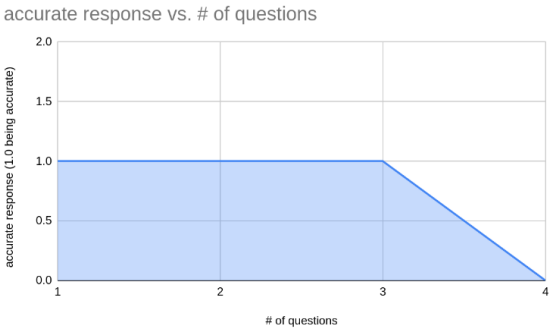

Figure 5. Figure of experiment 1

After conducting this experiment, the most surprising outcome we encountered was that the rag system was incredibly accurate even going up to three questions! In the generation that was provided by the rag system, all three questions were answered accurately and backed up with factual information from the documents. We also noticed that this was done in a timely manner, as a matter of fact, what took the longest amount of time was the actual collection of the documents from the web. After around 3 questions is when the rag system started to drift off target, and would fall victim to finding relevelant documetns to be irrelevant or just deemed generations to not be relevant to the question. We predict the reason this being that the generations that are graded are not only graded on what question that pertains to it, but also the other questions posed, which leads to the document being graded irrelevant. Generally, there is just too much information being passed around to be accurate.

4.2. Experiment 2

One potential blindspot in the self-rag system, is the potential that in the process of creating the generation, important information relevant to the question is excluded. It is important that this does not occur, as the self-rag model should answer the prompt exactly.

In this experiment, to test and see how detailed the AI model can remain when being asked intricate questions, we will ask questions of various increasing intricacies and examine how accurate and whether the self-rag model touches on all points relevant to the question. To conduct this experiment, we will start with a baseline question, and continue to increase it's intricacy by asking for more relevant information one at a time. We will continue to test this, until either the self-rag model becomes inaccurate or excludes information relevant to the answer. The importance of addressing this issue is critical, as ensuring the inclusion of all data relevant to the question is the desired outcome.

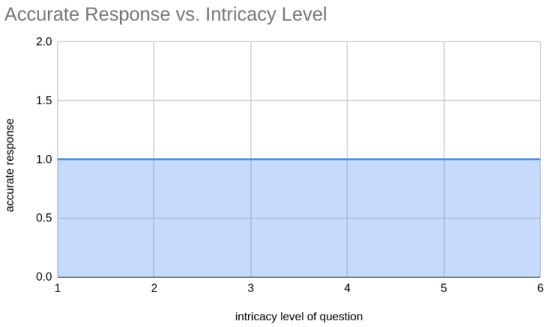

Figure 6. Figure of experiment 2

During and after experimentation, the first thing we noticed was how consistently accurate and encompassing the self-rag system remained despite the intricacy of the question. As a matter of fact, we were unable to get the AI model to become inaccurate, despite the intricacy of our questions. We believe these results were obtained and remained consistent due the components of each question being related to the same central topic. This enables the self-rag model to not have to switch contexts, as it previously had to do in the former experiment. We believe these results were obtained primarily due to the nature of the self-rag system, specifically, it's ability to reflect on itself and reshape the question to be tailored to receive a particular response. Additionally, a key component of the self-rag model is the ability to compare the generation with the original question and determine if it answers the prompt accurately, which ensures the output is accurate and all encompassing.

5. RELATED WORK

Another RAG system that aims to tackle hallucination of generative AI is the RQ-RAG system. RQ-RAG, or Refine Query for Retrieval Augmented Generation, is a RAG system that attempts to solve the problem of generation hallucination by spending it's time focusing on the refining of the question to invoke a more particular and informative answer. This solution is effective, in that test results demonstrate accurate results, however, there is an inherent deficiency apparent in the potential for the RQ-RAG model could refine the question in the wrong direction, producing an output that is no longer relevant to the question asked. Moving further with this issue, there is not any apparent way for the RQ-RAG model to look back and determine if it's refined query is still relevant to the original query. Compared to the RQ-RAG model, the Self-RAG model proposed in this paper improves upon where the RQ-RAG lacks by including self-reflection in the RAG process.

Another attempt at improving a basic RAG system is DR-RAG, a RAG model designed specifically to dynamically retrieve all information relevant to the query. DR-RAG involves a two-step process, of first retrieving all necessary information for answering a query from the documents and utilizing those as the new documents for the LLM to answer the query with. One potential problem with how the DR-RAG model discards information in an attempt to prevent hallucination, is the possibility of discarding relevant information or even retrieving useless information. Additionally, the DR-RAG model does not reflect on itself, and ensure there is no hallucination or useless responses or feedback.

VistaRag is a RAG system that works specifically for automotive driving. Different from the Self-RAG system that checks its generated answers by looking back at itself, the VistaRag system compares the generated answers to real world data, typically stored in a database. The issue that arises when the VistaRag model is employed in use is the possibility of the RAG model still hallucinating. The comparison to the real world data, while still effective, is not enough to ensure the accuracy of the prediction. However, a self reflective check, while still having the possibility to hallucinate, will add an extra layer of caution for the model.

6. CONCLUSIONS

During and after the development of the Self-RAG methodology seen in this paper, there are some apparent concerns that have to do with the safety and potential accuracy of the node system. After reflecting on how the documents are essentially web pages that can be loaded into the model, there then becomes the possibility of using the Self-RAG system for dangerous things, as well as prompt injection vulnerabilities. If given more time, we intend to remedy by establishing

the node that verifies the documents as well as the question before proceeding with the generation. Since in the current Self-RAG methodology, there is only validation that the answer generated answers the question, not that the generated answer aligns with the documents uploaded, this could lead to misleading answers or inappropriate contents.

This concludes the deep-dive on the presented Self-RAG methodology. Throughout all of the effort that went into research and development, I learned a lot about the Self-RAG system as well as systemic AI. Thank you for reading.

REFERENCES

[1] Asai, Akari, et al. "Self-rag: Learning to retrieve, generate, and critique through self-reflection." arXiv preprint arXiv:2310.11511 (2023).
[2] Asai, Akari, et al. "Self-RAG: Self-reflective Retrieval Augmented Generation." NeurIPS 2023 Workshop on Instruction Tuning and Instruction Following. 2023.
[3] Pandya, Keivalya, and Mehfuza Holia. "Automating Customer Service using LangChain: Building custom open-source GPT Chatbot for organizations." arXiv preprint arXiv:2310.05421 (2023).
[4] Davenport, Thomas, and Ravi Kalakota. "The potential for artificial intelligence in healthcare." Future healthcare journal 6.2 (2019): 94.
[5] Jeong, Minbyul, et al. "Improving Medical Reasoning through Retrieval and Self-Reflection with Retrieval-Augmented Large Language Models." arXiv preprint arXiv:2401.15269 (2024).
[6] Jeong, Minbyul, et al. "Improving Medical Reasoning through Retrieval and Self-Reflection with Retrieval-Augmented Large Language Models." arXiv preprint arXiv:2401.15269 (2024).
[7] Chan, Chi-Min, et al. "RQ-RAG: Learning to Refine Queries for Retrieval Augmented Generation." arXiv preprint arXiv:2404.00610 (2024).
[8] Jiang, Zhengbao, et al. "Active retrieval augmented generation." arXiv preprint arXiv:2305.06983 (2023).
[9] Hei, Zijian, et al. "DR-RAG: Applying Dynamic Document Relevance to Retrieval-Augmented Generation for Question-Answering." arXiv preprint arXiv:2406.07348 (2024).
[10] Salemi, Alireza, and Hamed Zamani. "Evaluating Retrieval Quality in Retrieval-Augmented Generation." arXiv preprint arXiv:2404.13781 (2024).
[11] Yu, Hao, et al. "Evaluation of Retrieval-Augmented Generation: A Survey." arXiv preprint arXiv:2405.07437 (2024).
[12] Yan, Shi-Qi, et al. "Corrective Retrieval Augmented Generation." arXiv preprint arXiv:2401.15884 (2024).
[13] Zhao, Penghao, et al. "Retrieval-Augmented Generation for AI-Generated Content: A Survey." arXiv preprint arXiv:2402.19473 (2024).
[14] Alkaissi, Hussam, and Samy I. McFarlane. "Artificial hallucinations in ChatGPT: implications in scientific writing." Cureus 15.2 (2023).
[15] Jeong, Cheonsu. "Generative AI service implementation using LLM application architecture: based on RAG model and LangChain framework." Journal of Intelligence and Information Systems 29.4 (2023): 129-164.