Calculating and Reporting Metrics of the RAG Pipeline

Prompt Engineering & AI
Summer 2024

Project Objectives:

The aim of this project is to evaluate the performance of the RAG pipeline and explore methods to enhance its metrics. This project includes a Python notebook and a report file, which document the evaluation process and present an improved version of our RAG chatbot.

We utilize our Simple-RAG-Chatbot for the RAG pipeline in this project. You can find the project in this repository: Simple-RAG-ChatBot

1. Performance Metrics

Each RAG pipeline has three main components: query, context, and response. The query is the user input, the context refers to the retrieved information based on the user's query, and the response is the LLM output based on the user's query and context retrieval. Based on this structure, we can evaluate a RAG pipeline in two key areas:

1.1. Context Relevance

This refers to how relevant the retrieved context is to the query. The following metrics are used to evaluate context relevance:

Context Precision: Measures how accurately the retrieved context matches the user's query.

Context Recall: Evaluates the ability to retrieve all relevant contexts for the user's query.

Context Relevance: Assesses the relevance of the retrieved context to the user's query.

Context Entity Recall: Determines the ability to recall relevant entities within the context.

Noise Robustness: Tests the system's ability to handle noisy or irrelevant inputs.

1.2. Answer Relevance

This refers to how relevant the responses are to the query. Similar to context relevance, we use the following metrics to evaluate answer relevance:

Accuracy and Relevance: Measures how correct and relevant the responses are to the given prompts.

Coherence and Fluency: Assesses how logical and smoothly flowing the responses are.

Consistency: Evaluates how consistent the responses are over multiple similar queries.

Robustness: Tests how well the model handles ambiguous, tricky, or adversarial inputs.

2. Evaluate Performance

We now evaluate our RAG chatbot using the metrics described earlier. We utilize a Python notebook and the RAG pipeline from our Simple-RAG-Chatbot project. We evaluate our chatbot in two areas: Context Relevance and Answer Relevance. This evaluation is done through human assessment, cosine similarity for Context Relevance, and ROUGE (Recall-Oriented Understudy for Gisting Evaluation) scores for Answer Relevance.

First, we select a wide range of prompts to generate a list of queries.

List of queries for evaluation

```python
queries = [
    "What are the best places to visit in Toronto?",
    "Can you suggest some good restaurants in Toronto?",
    "What is the best time of year to visit Toronto?",
    "Are there any special events happening in Toronto this summer?",
    "Where can I find affordable accommodation in Toronto?"
]
```

Then, we create a function to generate context retrieval and answers for each query.

Function to generate context retrieval and answer

```python
def generate_evaluation_data(query):
    retrieval_results = docsearch.similarity_search(query)
    contexts = [result.page_content for result in retrieval_results]
    # combined_context = " ".join(contexts)
    result = rag_chain.invoke(query)
    return {
        "query": query,
        "contexts": contexts,
        "answer": result["result"]
    }
```

Finally, we create two functions for evaluating our RAG pipeline in Context Relevance and Answer Relevance:

2.1. Evaluating Context Relevance

To evaluate "Context Relevance," we retrieve the context for each query and assess them using our cosine similarity. The steps are as follows:

- Use TF-IDF (Term Frequency-Inverse Document Frequency) to convert the texts (both questions and retrieved contexts) into numerical vectors. This transformation helps in understanding the importance of words in each document.
- `vectorizer = TfidfVectorizer().fit_transform(retrieved_contexts + questions) ` creates a combined TF-IDF matrix for both questions and retrieved contexts.

Cosine Similarity Calculation:

Cosine similarity is used to measure the cosine of the angle between two vectors, representing the degree of similarity between them. For each question, calculate the cosine similarity between the TF-IDF vector of the question and the corresponding retrieved context. Higher scores indicate a higher relevance of the context to the question.

`cosine_sim = cosine_similarity([context_vector], [question_vector])[0][0] ` computes the cosine similarity score.

Evaluation function for context relevance

```python
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import cosine_similarity

def evaluate_context_relevance(questions, retrieved_contexts):
    vectorizer = TfidfVectorizer().fit_transform(retrieved_contexts + questions)
    vectors = vectorizer.toarray()
    relevance_scores = []

    for i, question in enumerate(questions):
        context_vector = vectors[i]
        question_vector = vectors[len(questions) + i]
        cosine_sim = cosine_similarity([context_vector], [question_vector])[0][0]
        relevance_scores.append(cosine_sim)

    return relevance_scores
```

✓ 0.0s

Evaluate context relevance

```
context_relevance_scores = evaluate_context_relevance(queries, retrieved_contexts)

context_relevance_scores
```

✓ 0.0s

```
[0.2722216541941082,
 0.06539651144388514,
 0.36628252812040546,
 0.08463996755913449,
 0.06560117653352551]
```

2.2. Evaluating Answer Relevance

Next, we evaluate "Answer Relevance" by generating responses for each query in our list. Then, compare the generated answers with reference answers using ROUGE (Recall-Oriented Understudy for Gisting Evaluation) scores.

For each pair of generated and reference answers, calculate ROUGE scores using the scorer.

ROUGE-1 measures the overlap of unigrams (single words) between the generated and reference texts.

ROUGE-L measures the longest common subsequence (LCS) of tokens, considering the order of the words.

Store the ROUGE-1 and ROUGE-L F1 scores in `rouge_scores `.

Higher ROUGE scores indicate a higher relevance of the generated answers to the reference answers.

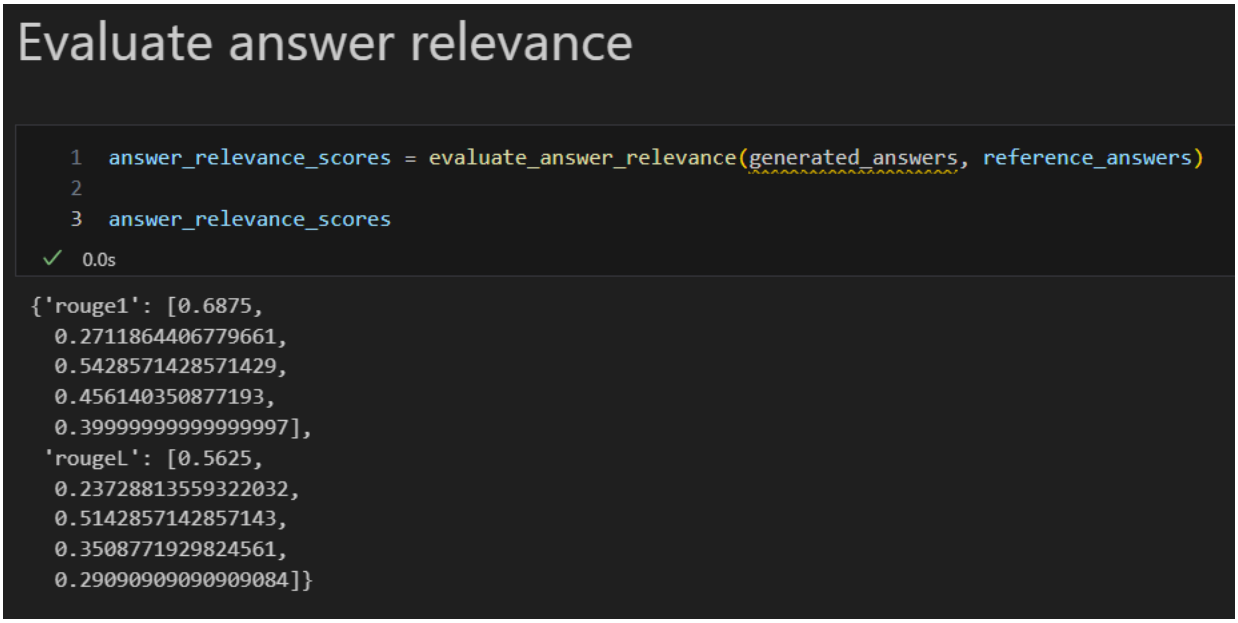
Evaluation function for answer relevance

```python
from rouge_score import rouge_scorer

def evaluate_answer_relevance(generated_answers, reference_answers):
    scorer = rouge_scorer.RougeScorer(['rouge1', 'rougeL'], use_stemmer=True)
    rouge_scores = {'rouge1': [], 'rougeL': []}

    for generated, reference in zip(generated_answers, reference_answers):
        scores = scorer.score(reference, generated)
        rouge_scores['rouge1'].append(scores['rouge1'].fmeasure)
        rouge_scores['rougeL'].append(scores['rougeL'].fmeasure)

    return rouge_scores
```



3. Methods to Improve Metrics

For Improve my RAG pipeline and performance of our chatbot I use such methods including using some advanced RAG pipeline like Auto-merging retrieval, using prompt engineering technique and improved our documents chunking that I introduced these techniques I used following:

3.1. Auto-merging retrieval

We can use auto-merging retrieval to improve the relevance of context retrieval in your RAG pipeline. Auto-merging, also known as "automatic merging" or "retrieval-augmented generation with auto-merging," is a technique where the system combines multiple retrieved contexts to generate a more comprehensive and relevant response. This approach can be particularly useful when dealing with diverse or overlapping pieces of information.

Here's how you can incorporate auto-merging retrieval into your existing chatbot setup:

3.1.1. Setup Auto-Merging Retrieval

You need to modify the retrieval process so that it combines the retrieved contexts before passing them to the language model. This can be done by adjusting the way you handle the retrieved documents in the `ConversationalRetrievalChain ` or `RetrievalQA ` chain.

3.1.2. Implement Auto-Merging Retrieval

Here's an updated version of your chatbot code with auto-merging retrieval. It includes modifications to combine multiple retrieved contexts:

```python
# Define a function for combining contexts
def merge_contexts(contexts):
    # Combine the contexts into a single string
    return "\n\n".join([ctx.page_content for ctx in contexts])

# Create a function to retrieve and merge contexts
def retrieve_and_merge_contexts(query, retriever):
    # Retrieve multiple contexts
    retrieved_docs = retriever.get_relevant_documents(query)
    # Merge contexts
    combined_context = merge_contexts(retrieved_docs)
    return combined_context

# Create a RetrievalQA chain with the refined prompt and custom retrieval
def generate_response(question):
    # Retrieve and merge context
    combined_context = retrieve_and_merge_contexts(question, docsearch.as_retriever())
    # Format the prompt
    formatted_prompt = prompt.format(context=combined_context, question=question)

    # Generate response
    response = llm.invoke(formatted_prompt)
    return response.content
```

3.2. Prompt Engineering:

Improve the prompt design to explicitly guide the model on how to use the retrieved context effectively. To achieve this, I made my prompts more specific to better guide the model in generating accurate responses.

3.3. Improve Document Chunking

I adjusted the `chunk_size ` and `chunk_overlap ` parameters to optimize how documents are split. This can help in retrieving more relevant chunks of text for each query.

4. Reevaluate Performance

After improving our RAG pipeline using advanced methods for retrieving context, we now reevaluate our metrics. As a result, we observe the following improvements:

Context Recall: The ability to retrieve all relevant contexts for the user's query has improved, resulting in more comprehensive information retrieval.

Context Relevance: The relevance of the retrieved context to the user's query is higher, providing more pertinent and useful information.

Context Entity Recall: The ability to recall relevant entities within the context has shown noticeable enhancement, leading to more precise and informative responses.

In terms of "Answer Relevance," we observe the following enhancements:

Accuracy and Relevance: The correctness and relevance of the responses to the given prompts have improved, providing more accurate and contextually appropriate answers.

Consistency: The consistency of the responses over multiple similar queries has increased, ensuring more reliable and predictable outputs.

These improvements demonstrate the effectiveness of the sentence-window retrieval method in enhancing both context relevance and answer relevance, leading to a more robust and accurate RAG pipeline.

5. Conclusion

In this project, we evaluated the performance of our RAG pipeline by analyzing its context relevance and answer relevance using a comprehensive set of metrics. Initially, we identified areas for improvement within the existing RAG pipeline, particularly focusing on the accuracy and relevance of the retrieved context and generated responses.

By implementing the sentence-window retrieval method, we enhanced the retrieval process, capturing more coherent and contextually relevant information. This improvement led to significant advancements in our performance metrics, demonstrating the effectiveness of the enhanced retrieval technique.

The reevaluation of our metrics confirmed that the improvements in context recall, relevance, and entity recall translated into more accurate and relevant responses from our chatbot. Additionally, the increased consistency of the responses further validated the effectiveness of our modifications.

In conclusion, this project highlights the importance of continuous evaluation and refinement of the RAG pipeline to ensure high-quality and reliable performance. The successful application of advanced retrieval methods like sentence-window retrieval provides a clear pathway for further enhancements and establishes a strong foundation for future developments in our chatbot system.