Summary of my research on Advanced RAG Techniques

RAG(Retrieval Augmented Generation): is the process of optimizing the output of an LLM, so it references an additional knowledge base outside of its training data sources. RAG extends the already powerful capabilities of LLMs to specific domains or an organization's internal knowledge base, all without the need to retrain the model.

The standard RAG technique uses the same text chunk for embedding and synthesis. The issue with this approach is that the embedding-based retrieval works well with smaller chunks whereas LLM needs more context and bigger chunks to synthesize a good or relevant answer.

There are two techniques used to fix this issue in the standard RAG techniques:

Sentence Window Retrieval

In sentence window retrieval, we perform retrieval of pieces of document then return a number of sentences surrounding the relevant sentence we retrieved, synthesis of the LLM is then generated from this relevant sentence and the window of sentences above and below it.

Let's think of a large text data, and let's say the relevant sentence is found somewhere in the middle. The window of the sentences which are sentence windows above and below the relevant sentence are passed along with the relevant sentence to the LLM to perform it's response. We can control the window size, which is the sentences that will be passed along with the relevant sentence.

Steps to use sentence window retrieval:

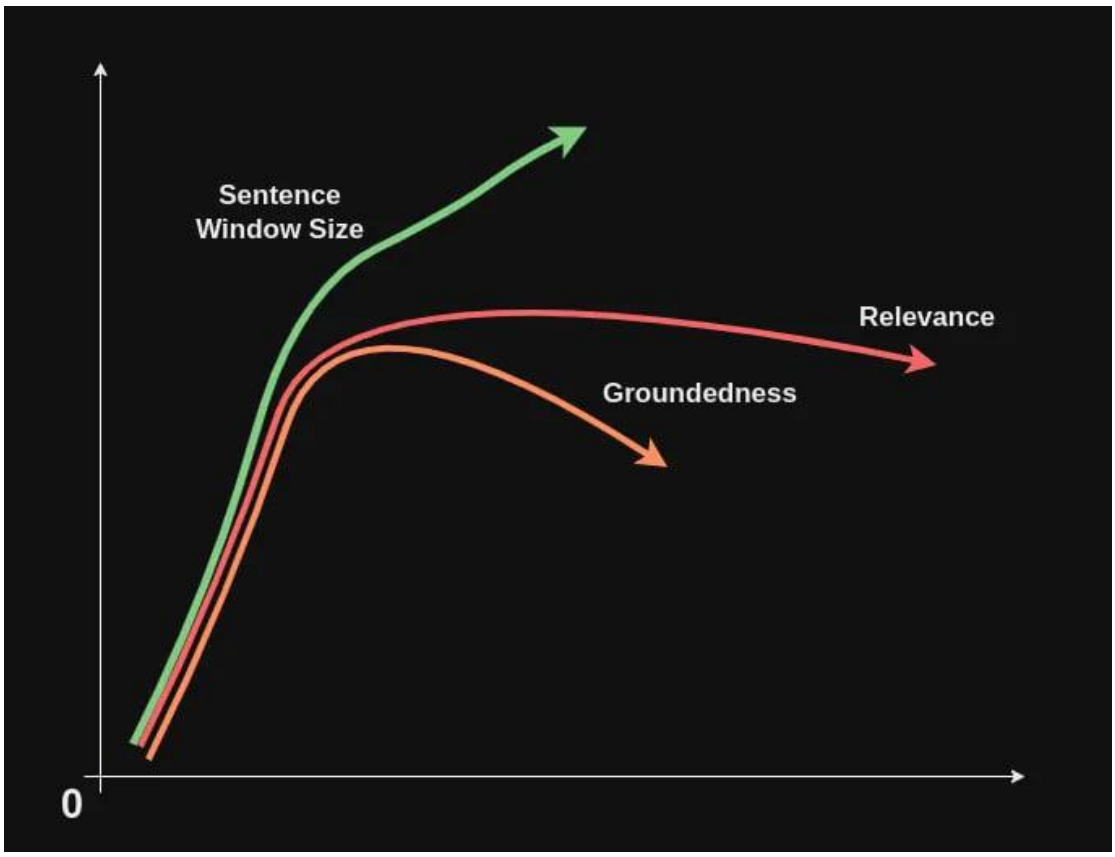
1. Loading Document
2. Sentence Window Retriever Setup
    - ✓ We will use SentenceWindowNodeParser from llama index to break down the document into individual sentences, then add the surrounding sentences within the allowed window to each individual sentence to create a large context.
3. Building Indexes
    - ✓ After the sentences with their surrounding contexts are created, we'll then create embedding to store it in the vector database.
4. Creating Meta Data Replacement Post Processor
    - ✓ This processor comes into play after we performed retrieval of the relevant chunk. It replaces the metadata around the retrieved node with the actual surrounding text that lies within the sentence window.
5. Adding a Re-Ranker
    - ✓ This re-ranks the retrieved sentences based on their relevance. Basically we use reranking to match the query against the existing nodes to find the most relevant node.
6. Adding Query Engine

Evaluation

Groundedness: As sentence window increases, the groundedness will increase up to a certain point. This is because the LLM has more context to base it's response other than hallucinating or generating a wrong response.

When the sentence window is small, the response the LLM will generate will have lower groundedness as the context does not provide enough information to the LLM to base it's response. On the other hand, if the window size is too large, the groundedness will decrease as the LLM is provided with large context to base it's response, it ends up deviating from the provided information since it's provided with a knowledge thet's too large to compose a response with all these information within it.

Relevance: As sentence window increases, the relevance of the response will increase up to a certain point. This is because with more context the LLM may or may not get distracted and falls back to it's training data for generating the response. With too small context, the LLM begins to hallucinate and the relevance drops.



To get started into evaluating:

1. Create a file containing questions
2. Read the questions and use for loop to pass each questions to ***TruLens*** to get an evaluation.
3. Use sentence window size 3 and 6 to evaluate
4. Compare the result

Evaluation result usually looks like this:

From the image we can see that relevance and groundedness measures are displayed.

Auto-Merging Retrieval

Auto-Merging is a retrieval technique that leverages a hierarchical document structure. When a document is too long, it is split into smaller documents or chunks, where we can think of the smaller documents as the children of the original document and the original document as the parent. This results in a hierarchical tree structure where each smaller document is a child of a previous larger document. The leaves of the tree are the documents which don't have any children, and the root is the original document.

Steps to use auto-merging retrieval:

1. Loading Document
2. Auto-Merging Setup
   - ✓ We will use **HierarchicalDocumentSplitter** to split int into smaller documents. We'll need to specify the block sizes. For example, let's say we have one big text, and the block size is {512, 256, 128}. The HierarchicalDocumentSplitter will create parent nodes from the big text using 512 chunk size, and from that it will continue creating the child nodes with the size 256 and so. The last node is called leaf_node, in this case it'll have a chunk size of 128.
3. Load into storage
   - ✓ We define a docstore to load all nodes into.
   - ✓ We define VectorStoreIndex, it will contain all the leaf_nodes.
4. Define a Retriever
   - ✓ The retriever will be imported from **AutoMergingRetriever** class from llama_index and it will take a user query.

- ✓ When the set of smaller chunks linking to a parent chunk exceeds some threshold then it'll merge these smaller chunks linking to a parent chunk, so it'll return the parent chunk instead of individual small chunks.

5. Plug it into Query Engine

Evaluation

We run evaluations on each of the retrievers: correctness, semantic similarity, relevance, and faithfulness. We can import all of these classes from *llma_index.evaluations.*

We can also use TruLens to evaluate the response.

More about technical implementation will be discussed in the notebook.

References

1. https://ai.gopubby.com/advance-retrieval-techniques-in-rag-part-03-sentence-window-retrieval-9f246cffa07b
2. https://haystack.deepset.ai/blog/improve-retrieval-with-auto-merging#:~:text=Auto%2DMerging%20is%20a%20retrieval,original%20document%20as%20the%20parent.
3. https://docs.llamaindex.ai/en/stable/examples/retrievers/auto_merging_retriever/