Dan Bigman 02/06/2025

Dylan Rodriguez Flores

IronHack - Puerto Rico

**Retrieval Augmented Generation (RAG) Challenge Report**

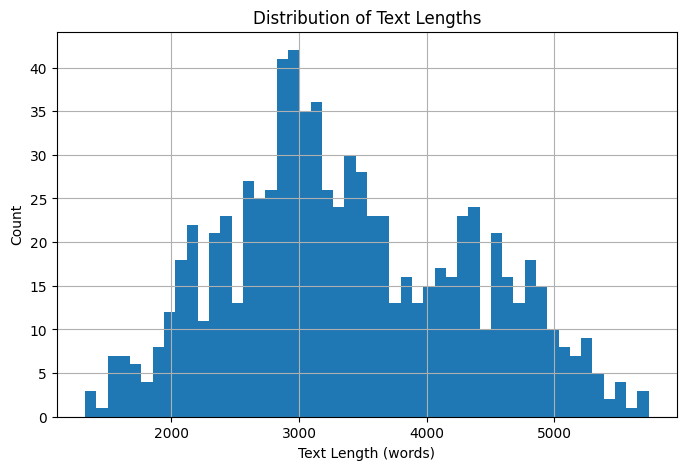
1. **Introduction**

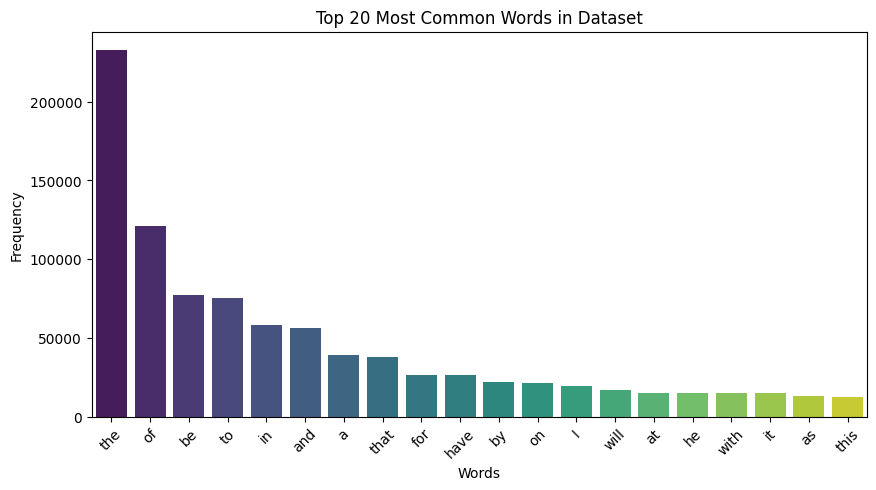
Retrieval Augmented Generation (RAG) is a novel approach that combines the strengths of retrieval-based and generation-based models to provide accurate and contextually relevant responses. By leveraging a vector database to retrieve relevant documents and a large language model (LLM) to generate responses, RAG can significantly enhance the capabilities of applications in various domains such as customer support, knowledge management, and content creation.

1. **Exploratory Data Analysis**

For the initial inspection we examined data structure, formats, and metadata of the 821 text files. After this, we identify any missing values, duplicates, or inconsistencies in the documents to normalize the set as much as possible. Once that was done, we proceeded with the data cleaning & preprocessing. To achieve this, we removed noise and normaliz the text in the files. If required, the text documents were split into smaller, coherent chunks. To generate visualizations for the report, we utilized libraries (e.g., Matplotlib, Seaborn) to generate charts showing data distributions, word frequencies, etc.

This would then reveal document insights and challenges that might affect the retrieval and generation stages.





1. **Connection Setups**
2. Embedding and Storing Chunks
   1. Generate Document Embeddings

The selected tools for the embeddings were:

* all-MiniLM-L6-v2 for English text.

Processing Steps:

* Tokenize and chunk your documents.
* Generate embeddings for each chunk using Sentence Transformers.
* Validate that the embeddings capture semantic meaning by checking a few sample outputs.
  1. Connect to a Vector Database

The chosen library was ChromaDB for storing and retrieving embeddings and due to being the default tool used in lessons, made it easier to adapt to the project.

Integration Steps:

* Pre-process: Ensure each document chunk has an associated embedding.
* Store: Upload the embeddings along with metadata (e.g., document IDs, original text) to ChromaDB.
* Retrieve: Develop logic to perform similarity searches (e.g., cosine similarity) on stored embeddings given a query.
  1. Integration Frameworks

Optional Libraries - Leverage frameworks like LangChain or LlamaIndex for easier integration.

1. Connecting to a Large Language Model (LLM)
   1. LLM Selection & Setup

Use the OpenAI API for generating responses. The generated API key allows us to connect with the LLM and send queries and prompts that the model can then answer based on the information in the documents.

* 1. Integration Workflow
     1. Query Processing: When a user submits a query, first use the vector DB to retrieve the most relevant document chunks.
     2. Prompt Construction: Combine the retrieved documents with the query to create a context-rich prompt.
     3. Response Generation: Pass the prompt to the LLM and retrieve the generated answer.
     4. Testing: Iterate on the prompt structure and retrieval parameters to fine-tune the quality of responses.

1. **Evaluation Metrics**
2. Performance Testing
   1. Manual Evaluation: Tested the system with multiple queries to understand its strengths and weaknesses.
   2. Automated Evaluation: Created a test set with queries and expected responses.
3. Metrics & Documentation
   1. Define success metrics (e.g., relevance, accuracy, response time).
   2. Document your evaluation process and results in the project report.
   3. Identify areas for improvement and possible refinements to the pipeline.
4. **Conclusion**

Developing this Retrieval-Augmented Generation (RAG) system was a complex yet rewarding challenge that required integrating multiple technologies, including document chunking, embedding generation, vector storage, and Large Language Model (LLM) inference. Throughout the process, several technical hurdles arose, from managing large-scale data efficiently in Google Colab to ensuring persistent storage for embeddings and ChromaDB collections.

One of the primary challenges was handling system limitations, especially regarding memory constraints and execution time. Moving computations to Google Colab helped, but ensuring the pipeline remained efficient required optimizing batch processing, embedding storage, and retrieval mechanisms. Another major challenge was adapting to changes in OpenAI’s API, particularly ensuring compatibility with the latest version, which necessitated revising the method of calling the LLM for response generation.