Intro RAG Bake-Off for Technical Q&A

Ángel Valencia Padilla

Table of contents

Abstract

This project explores different Retrieval-Augmented Generation (RAG) pipelines for a multiple-choice Q&A, based on a Meta's research paper introducing *REFRAG*. Using a curated dataset, we compared Okapi BM25, Dense and Hybrid retrieval methods over both paragraph based and recursively built chunks, and we assessed their improvements over the baseline LLM.

The report adopts a hybrid approach by presenting the experimental methodology and results for the Q&A task combined with a theoretical introduction to RAG and the studied methods.

1. Introduction

We will begin with a brief introduction to Retrieval-Augmented Generation (RAG), explaining some of the motivation behind this method, as well as the main components of a simple pipeline. Readers familiar with RAG systems can skip section 1.1.

1.1. RAG Background

Large language models (LLMs) have revolutionized many industries including software development, scientific research, education, customer services, etc. However their great power has a great limitation: the expense of their training. Training a fully functional LLM that can understand language and perform simple tasks is extremely expensive in regards to the computational cost and training time they require. For example, OpenAI's ChatGPT 4 training cost and duration are estimated to be above $100 Million and 6 months respectively [1].

The inability to train LLMs frequently often results in outdated models, that will not perform adequately on recent data they haven´t been trained on. On the other hand, the computational expenses of their training makes it so people can only rely on general models, trained on general task and public available data, rarely trained on private specific data. For example, many business would benefit from leveraging a LLM's knowledge and automation on their private data, or academics would benefit from a LLM assistant updated on recent discoveries, but general LLMs are not enough for these tasks.

Retrieval-Augmented Generation (RAG) is a method that incorporates information retrieval into LLMs generation. The purpose of RAG is to enable the LLM to reason over some selected source of external information which does not need to belong to the LLM's training data. Therefore with RAG we can leverage LLMs on private or updated data, introducing great opportunities for using them in complex task like those we mentioned before.

Given an information source where we would like to leverage LLMs on, we can build a RAG system which will work as follows. Once we are given a query, an information retriever searches the information source in order to find pieces of relevant information for it, in order to construct

a prompt where retrieved information is placed as context and the initial query is said to be answered. This way, the LLM can use external information to reason, as if it had been part of its training.

A basic RAG pipeline can be considered a composition of 3 stages:

- **Indexing**: The process in which the information source is processed into manageable and easy to retrieve pieces. This includes splitting text into small *chunks*, embedding them (in case of *dense retrieval*) and storing them for later use.
- **Retrieval**: Once a query is given, retrieving the adequate chunks relevant to the query.
- **Answer Generation**: Building the final prompt containing the retrieved context and the query, and parsing the final LLM's answer.

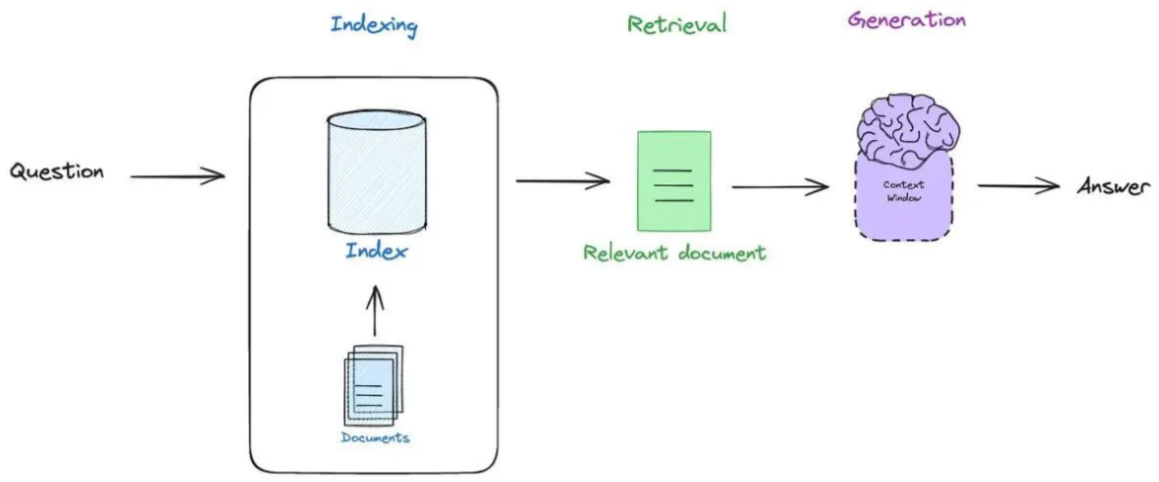

Figure 1: A Basic RAG Pipeline. Source: [2]

1.2. Project Description

We built a Q&A system over the research paper from Meta *REFRAG*, to compare how different retrieval setups help a LLM to answer multiple-choice questions. In particular, we used Google's *Gemini Flash 2.5 Lite* as LLM and we tested it over a 70 multiple-choice questions dataset, where each example contained one correct answer and three distractors as shown in the following schema:

```yaml
question: What is the primary mechanism through which the REFRAG framework achieves a reduction in computational complexity for attention?
answers:
  A: "It eliminates the need for a key-value cache, thus reducing memory requirements."
  B: It modifies the core LLM architecture to use a linear attention mechanism instead of a quadratic one.
  C: It changes the scaling factor of attention computation from the number of tokens to the number of chunks.
  D: It offloads all attention computations to a specialized lightweight encoder model.
correct_answer: C
paper_reference: "REFRAG makes several novel modifications to the decoding process: Instead of using tokens from retrieved passages as input, REFRAG leverages pre-computed, compressed chunk embeddings as approximate representations, feeding these embeddings directly into the decoder. This approach offers three main advantages: 1) It shortens the decoder's input length, improving token allocation efficiency; 2) It enables reuse of pre-computed chunk embeddings from retrieval, eliminating redundant computation; and 3) It reduces attention computation complexity, which now scales quadratically with the number of chunks rather than the number of tokens in the context."
```

Figure 2: Questions Dataset Schema

We built a modular, and reproducible experimental setup, mostly with python, where we implemented Okapi BM25, dense and hybrid retrieval methods. We implemented two different chunking methods: paragraph based and recursive chunking, and we tested the different retrieval and chunking combinations over the Q&A task. In sections 2-4, we will go into detail on how we implemented every step of the project, while giving the theoretical foundations behind the methods and design choices. In section 5 we will deep down on the evaluation process we followed, covering the metrics we used to measure performance, and the results we obtained.

2 Indexing

2.1. RAG Indexing Background

On a RAG pipeline, retrieving the full information source for every query is inefficient. LLMs context window is limited, and they may lose precision when large redundant context is given. It is also really expensive in the terms of tokens per prompt, something unwanted if we are dealing with commercial models. Therefore, intelligently splitting our source of information into pieces (chunks) is a must. The chunking method is important, since different strategies may produce better chunks in terms of retrieval ease, quality of information, and understanding for the LLM.

Afterwards, chunks are stored in an accessible database for future retrieval. In particular, when dealing with modern retrieval methods, is common to use a trained embedding model to encode

the semantic meaning of chunks into numerical vectors that get stored in a vector database, so natural language processing techniques can be applied in the retrieval step afterwards.

2.2 Project Implementation

For splitting the *REFRAG* paper we chose to implement and evaluate two of the most classic chunking methods.

Recursive Chunking: Splitting the document into a flexible chunk length with a recursive algorithm that iterates through text boundaries in descending order of importance (eg. [“\n\n”, “\n”, ” “,” “]), splitting until chunks satisfy the adequate length.We used *Langchain*’s implemented version `RecursiveCharacterTextSplitter` with a maximum chunk length of 1000 characters. We also added a 200 characters overlap between consecutive chunks, to prevent information loss caused by the split.

```
method: recursive
num_chunks: 131
avg_length: 952.0610687022901
min_length: 368
max_length: 999
lengths_examples[10]: 997,964,971,939,907,900,922,899,946,933
```

Figure 3: Recursive Chunks Summary

Paragraph Base Chunking: Splitting the document by its paragraphs perfectly aligns with the structured nature of technical papers, where similar topics are grouped together and commonly in the same paragraph. We loaded the paper as an html document to identify paragraphs, and set a minimum and maximum paragraph lengths of 100 and 2000 characters respectively. We merged consecutive smaller paragraphs, with the intention of representing tables or sparse data, and recursively splitted larger ones with 2000 characters chunk length and 200 characters overlap.

```
method: paragraph
num_chunks: 143
avg_length: 788.4615384615385
min_length: 110
max_length: 1999
lengths_examples[10]: 256,1249,587,381,1992,1996,1721,1769,336,919
```

Figure 4: Paragraph Chunks Summary

Finally, after we made sure to include some metadata in our chunks (id, chunk length, chunking method, ...), we stored them both as a `.jsonl` files, for rapid access and visualization, and in *Chroma* vector databases, for future dense retrieval. For embedding the chunks we used a *Sentence-Transformers* model hosted on *Hugging Face*: *all-MiniLM-L6-v2*.


```
metadata:
  id: paragraph_chunk_025
  paragraph_index: 18
  split_index: 0
  orig_tag: p
  length: 578
  method: paragraph
page_content: "REFRAG trained under different compression rates. Figure 10 illustrates the training
trajectory of REFRAG under different compression rates in the continual pre-training task. We observe a
performance regression as the compression rate increases; however, even at a compression rate of 32, our
model remains competitive (as shown in table 1 ). In contrast, a compression rate of 64 appears to be
overly aggressive, resulting in diminished performance. These findings suggest a practical limit to the
compression rate beyond which the model's capability is significantly reduced."
```


Figure 5: Chunk Example

3. Retrieval

3.1. Retrieval Methods Background

Retrieval may be the key step of a RAG pipeline, since the LLM´s answer to a query is strictly dependent on the given retrieved context. The key idea is, for a given query, establishing an order of relevance in the available 'documents' (datasets, chunks or any pieces of information we are dealing with) by assigning to each one a relevance score. Then the top k documents according to this score are retrieved, where the k parameter is fixed according to the overall task needs. Some of the most common retrieval methods are the following.

3.1.1. Sparse Retrieval

This type of methods are based on keyword matching. The score they assign to each document is mainly based on the exact match of words between the document and the query, even though they may take into account other features to maximize accuracy. For example, the **TF-IDF** method scores a term (this generalizes to a full query) taking into account not only term frequency (TF), but also how rare a term is across all documents (IDF).

$$\text{TF-IDF}(t, d, D) = \text{TF}(t, d) \times \text{IDF}(t, D)$$

$$\mathrm{TF}(t,d) = \frac{f(t,d)}{\sum_{t' \in d} f(t',d)}$$

$$\mathrm{IDF}(t,D) = \log \frac{N}{1+n_t}$$

Where: t = term, d = document, D = set of documents, $f(t,d)$ = frequency of term t in document d, N = total number of documents in the source, n_t = number of documents containing term t.

The **BM25 Score** is a modern version of TF-IDF. Some of its improvements include document length normalization, a much more smoothed IDF and more saturated TF, with which it has clearly shown an overall more accurate and robust relevance scoring.

$$\mathrm{BM25}(q,d) = \sum_{t \in q} \mathrm{IDF}^*(t) \cdot \frac{f(t,d) \cdot (k_1+1)}{f(t,d) + k_1 \cdot \left(1 - b + b \cdot \frac{|d|}{\mathrm{avgdl}}\right)}$$

$$\mathrm{IDF}^*(t) = \log \frac{N - n_t + 0.5}{n_t + 0.5}$$

Where: q = query, $|d|$ = number of words in document d, avgdl = average document length in the source, k_1 = TF saturation parameter (typical 1.2–2.0), b = document length normalization parameter (typical 0.75).

Sparse methods are exceptionally good when specific terms matter. For example, error codes, technical terms, IDs, names, etc. And they are a fast and simple solution for simple RAG tasks.

3.1.2. Dense Retrieval

This type of methods are based on semantic meaning, rather than exact word matching. They leverage natural language processing to compute a similarity score that purely depends on the query and document meaning. For this retrieval method, a proper indexing step should have used a trained embedding model to embed the chunks as vectors, which should be stored in an accessible vector database.

Embedding models are trained to represent semantically closer texts in positionally closer vectors in a high dimensional vector space. This way, once a query is given, embedding it enables to score document relevancy by measuring their proximity to the query inside of the vector space they belong. Popular ways to compute the query and document similarity are:

Cosine Similarity:

$$CosSim(q, d) = \frac{q \cdot d}{||q|| \cdot ||d||}$$

Dot Product Similarity:

$$DotSim(q, d) = q \cdot d$$

L2 Similarity:

$$L2Sim(q, d) = -||q - d||_2$$

Where $q, d \in \mathbb{R}^n$ are the vector representations of the documents to compare (query and document in the RAG case).

A more advanced technique that is worth to mention is Maximal Margin Relevance (MMR). MMR is an iterative method which selects documents in a way that rewards similarity to the query while penalizing similarity to already selected documents. This method ensures diversity in the retrieved documents so more distinct but relevant information is obtained.

MMR Algorithm:

Input: - Query vector $q \in \mathbb{R}^n$ - Candidate documents ($D = \{d_1, d_2, ..., d_n\}$ - Number of documents to select k - Trade-off parameter between relevance and diversity $\lambda \in [0, 1]$

Output: - Selected documents S

1. Initialize $S \leftarrow \emptyset$
2. Repeat until $|S| = k$:
    - For each $d \in D \setminus S$, compute the MMR score:

$$\text{MMR}(d) = \lambda \, \text{Sim}(d, q) - (1 - \lambda) \max_{d_j \in S} \text{Sim}(d, d_j)$$

    - Select $d^* = \arg\max_{d \in D \setminus S} \text{MMR}(d)$
    - Add d^* to S
3. Return S

Dense methods outperform sparse ones when meaning is more important that matching the exact words. This is the case for most common situations, where the query is in natural language and not technical keywords, and synonyms and paraphrasing are the norm.

3.1.3. Hybrid Retrieval

This method takes the best of both worlds, by combining a sparse and a dense retriever into a single one. This way, it benefits from the ability to understand semantic meaning in natural language, while being precise when it comes to keywords that may lack meaning by themselves.

When given a query, it gets processed by both the sparse and the dense retrievers. Then ensembling methods are applied to assign final scores that take into account both the sparse and dense rankings. Common methods include:

Weighted Sum:

$$score(d) = \alpha \cdot score_{sparse}(d) + \beta \cdot score_{dense}(d)$$

Where $\alpha, \beta \in \mathbb{R}$ are the optimized weights for summing the normalized sparse and dense scores respectively, and d is the document to score.

Reciprocal Rank Fusion (RRF):

$$RRFScore(d) = \sum_{r \in R} \frac{1}{k_{rrf} + rank_r(d)}$$

Where R is the set of retrievers (one sparse and one dense), $rank_r(d)$ is their respective ranking for document d and k_{rrf} is a normalization constant (typical 60).

3.2. Project Implementation

For our technical Q&A task, we implemented the various retrieval methods in order to compare their performance. Therefore, for every retriever we picked a default and fixed $k = 3$ number of top retrieved chunks. We compared the following methods with the specified settings:

- **BM25**: Used *Langchain*'s built in default `BM25Retriever`.
- **Dense**: Used the *Chroma´*s vector database (where we stored our chunks) build in retriever, with Cosine Similarity as the default similarity metric. We included MMR search with $\lambda = 0.8$.
- **Hybrid**: We implemented a hybrid retriever that merged the top k chunks from the BM25 retriever, with the top k from the Dense one. Then, RRF was used to assign new scores and select the top k out of the $2k$ scored documents.

4. Answer Generation

4.1. Answer Generation Background

The final step of a RAG pipeline is obtaining and parsing the LLM's answer to the query. For this task, the default approach would be formatting a prompt where the retrieved chunks are given as context and the query is said to be answered. However there are several techniques that can improve this process.

For example, prompt engineering techniques are commonly used to improve answer quality, like including direct instructions or a role description for the LLM in the prompt. Other advanced techniques may summarize long context for better context memory, break down everything into smaller prompts and merge the answers, or use answers to previous queries as new context.

4.2. Project Implementation

In our prompt for the experiments, retrieved chunks were given as context and the multi-choice question (query) with the four possible answers where specified. A technical expert role was given and direct instructions asked the LLM to follow a specific answer format, so that we could easily parse the output and extract the given answer and a reference to the chunk it obtained it from. For the latter task, context chunks were displayed next to their ID's. The baseline LLM received this same prompt, however the context section was empty.

```
prompt_template =  """
You are a technical expert. Answer this multiple choice question using the provided chunks as context.

Context (Chunks IDs in brackets):
{context}

Question:
{question}

Options:
{options}

Please provide:
- Your answer (one letter only, no explanation, no punctuation, just an uppercase letter).
- The IDs of the retrieved chunks you used to answer (comma-separated, e.g. chunk_000, chunk_123).

You must use this format:

Answer: [single letter]
Sources: [comma-separated ids]

If the answer is not present in the provided context, respond EXACTLY:

Answer: X
Sources: NONE
"""
```

Figure 6: Implemented Prompt Template

After prompting the LLM, using regular expressions we identified the option chosen and referenced chunks easily, thanks to the formatting instructions we gave the LLM.

5. Evaluation and Results

5.1. Computing Metrics

Experiments consisted in iterating over the combinations of chunking and retrieval methods. Respectively: paragraph and recursive chunking, combined with BM25, dense and hybrid retrieval, also including the baseline model (no retrieval). Each combination of chunking and retrieval answered the 70 questions. We made sure to include multiple runs (five) to ensure the experiments had statistical relevance. In the process, significant results were obtained at each question and run iteration, like:

- Answer Correctness: Did the LLM answer the question correctly?
- Source Correctness: Did the LLM cited a chunk that actually contained the answer?
- Is X: Did the LLM choose not to answer because of missing context?

- Latency: How much seconds did this iteration cost?

```
retrieval_method: dense
chunking_method: recursive
run: 5
question_id: 35
question: How is curriculum learning applied to the reconstruction task to address optimization difficulty as k and the number of chunks increase?
answers:
  A: By starting with all chunks reconstructed at once and progressively pruning tokens using scheduled sampling.
  B: By freezing the encoder and gradually unfreezing layers of the decoder based on validation loss.
  C: By beginning with reconstructing a single chunk and gradually increasing to multiple chunks while adjusting the data mixture toward harder examples over time.
  D: By replacing ground-truth tokens with model predictions earlier in training to induce robustness to errors.
correct_answer: C
paper_reference: "Curriculum learning incrementally increases task difficulty, enabling the model to gradually and effectively acquire complex skills. For the reconstruction task, training begins with reconstructing a single chunk: the encoder receives one chunk embedding c1 for x1:k and the decoder reconstructs the k tokens using the projected chunk embedding. Subsequently, the model reconstructs x1:2k from two chunk embeddings, and so forth. To continuously adjust task difficulty, we vary the data mixture over time, starting with easier tasks and gradually shifting toward more difficult tasks."
retrieved_chunks[3]: recursive_chunk_018,recursive_chunk_018,recursive_chunk_029
llm_answer: C
llm_source[1]: recursive_chunk_018
is_x: false
answer_correct: true
source_correct: true
latency: 0.516
```

Figure 7: Results Example

Excluding source correctness, the remaining metrics were trivial to compute after we parsed the LLM's response and timed the experiment execution. For evaluating source correctness, every question in our dataset contained a reference from the paper where the answer was found, but looking for an exact match of the chunk with the reference was not accurate, since the construction of chunks commonly splitted the reference but maintained the crucial information, or sometimes text formatting disturbed the exact character matching, which resulted in abundant false negatives.

To implement an accurate source evaluation that did not directly depend on exact text match, we formatted a specific source evaluation prompt, to leverage the natural language understanding of our LLM by instructing him to classify a chunk as relevant for a question if and only if it contained the crucial part of the reference needed for answering successfully. This resulted in more accurate and robust measurements.

```
You are a scientific assistant. Your task is to determine wheter a collection of text chunks from
a scientific paper is a valid source of information to answer a question.

You will consider the following:
- Question: the question to consider.
- Reference: a excerpt from the paper containing the information needed to answer the question.
- Chunks: the collection of chunks to consider. It could be a single chunk.

You must consider the collection as valid if and only if verifies both conditions:
- It contains some part of the reference (accept some variation in text formatting).
- The part of the reference the chunk contains has the main information needed to answer the question.

Your anser must consist of just one letter (no explanation or punctuation, just an uppercase letter):
- 'Y' (if the chunk is valid)
- 'N' (if the chunk is not valid)

Question:
{question}

Reference:
{paper_reference}

Chunks:
{chunks_text}
```

Figure 8: Source Evaluation Prompt

After each experiment, we computed general metrics from the iteration results we collected. In particular, we computed:

- Answer accuracy and standard deviation
- Source attribution accuracy and standard deviation
- Coverage: Proportion of answered questions
- Average latency.

Retrieval Method	answer_accuracy	source_accuracy	coverage	answer_std	source_std	avg_latency
baseline	0.66	0.0	0.84	0.0	0.0	0.5
bm25	0.89	0.88	0.96	0.0	0.0	0.5
dense	0.87	0.72	0.93	0.0	0.0	0.6
hybrid	0.91	0.88	0.96	0.0	0.0	0.62

Figure 9: Recursive Chunking Metrics

Retrieval Method	answer_accuracy	source_accuracy	coverage	answer_std	source_std	avg_latency
baseline	0.66	0.0	0.84	0.0	0.0	0.46
bm25	0.94	0.76	0.97	0.0	0.0	0.59
dense	0.81	0.58	0.89	0.0	0.0	0.58
hybrid	0.91	0.73	0.96	0.0	0.0	0.61

Figure 10: Paragraph Chunking Metrics

5.2. Chunking Methods Analysis

Averaging across retrieval methods we found the same answer accuracy for both chunking methods, while source attribution was favored recursive chunking.

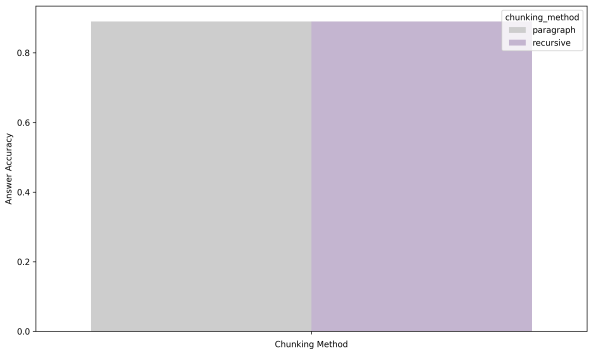

Figure 11: Answer Accuracy

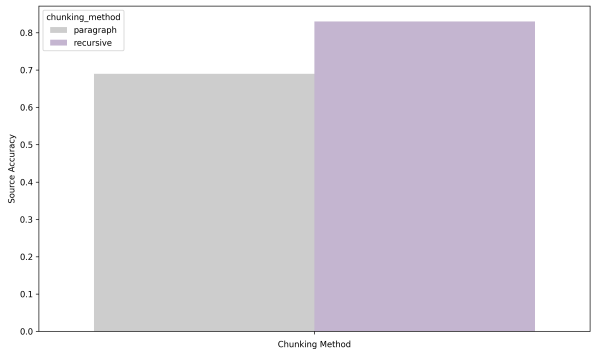

Figure 12: Source Accuracy

Source attribution is a stricter metric than answer accuracy, since relevant insights for answering a question can be obtained from non relevant chunks that form a sufficient relevant context when presented together. And also the LLM's internal knowledge could be sufficient to deduce simple questions without proper context, as we can see on the baseline model accuracy score. While in contrast, if source attribution is correct, a relevant chunk is presented and it should suffice to answer the question successfully.

Therefore, the source attribution difference between the methods may have been caused by references divided by paragraphs or contained in tables, captured by recursive chunking but missed by paragraph chunking, even though still being a relevant context when presented with other nearby paragraph chunks, in a way that answer accuracy remains overall equal between the two chunking methods, despite the difference in source attribution.

While the overall answer accuracy was the same for both methods, we found it behaved somewhat different on the specific retrieval methods, where paragraph chunks aligned better with the BM25 method, and recursive ones benefited dense retrieval. This may also be an expected outcome since BM25's sensitivity to keywords makes smaller and clean chunks like paragraphs better for retrieval than larger recursive chunks, where overlap makes keywords to spread across various chunks, and a larger size is noisier in TF-IDF terms. In contrast, dense retrieval does benefit for the richer semantic context given by the overlap and the larger size.

A consequence of the complementary increases in BM25 and dense retrieval scores across the chunking methods is how hybrid retrieval, which combines both, maintained the same answer accuracy, despite the change in source attribution, demonstrating its robustness as retrieval method.

5.3. Retrieval Methods Analysis

For analyzing the retrieval methods we averaged their metrics across the chunking methods. And obtained the following ones.

Retrieval Method	answer_accuracy	source_accuracy	coverage	answer_std	source_std	avg_latency
baseline	0.66	0.0	0.84	0.0	0.0	0.48
bm25	0.92	0.82	0.96	0.0	0.0	0.54
dense	0.84	0.65	0.91	0.0	0.0	0.59
hybrid	0.91	0.8	0.96	0.0	0.0	0.62

Figure 13: Retrieval Methods Average Metrics

As expected, the source attribution accuracy ranking maintained in terms of answer accuracy as well. BM25 obtained the best results with an average 92% answer accuracy, slightly over hybrid (91%) and clearly outperforming dense retrieval (84%). And of course all three retrieval-based methods outperformed the baseline LLM (66%) by a very significant difference.

As we saw earlier, BM25 benefited from paragraph chunks the most. Therefore, the BM25 and paragraph chunks combination emerged as the optimal choice, with 94% accuracy. Probably, the technical aspect of the paper was responsible for the BM25 dominance. Numerous questions included technical terms that reappeared in the paper reference, like *REFRAG*, *CEPE*, *TTFT*, etc. Therefore BM25, which responds excellently to specific technical keywords, may have had an edge for finding relevant chunks, and it outperformed dense retrieval so much, that taking into account dense rankings on the hybrid retrieval RRF score may have made hybrid slightly more biased, therefore achieving slightly worse results than pure BM25. This is a clear

indicator of how retrieval method selection is crucial and must be task specific for optimizing results.

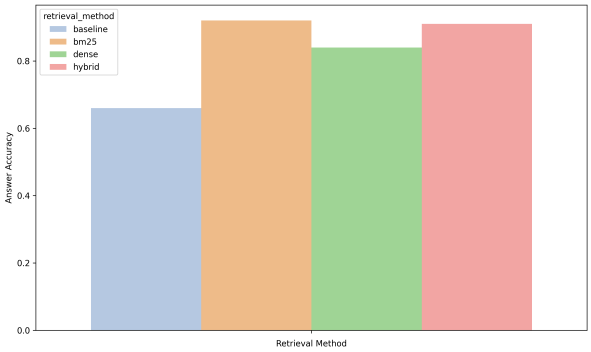

Figure 14: Answer Accuracy

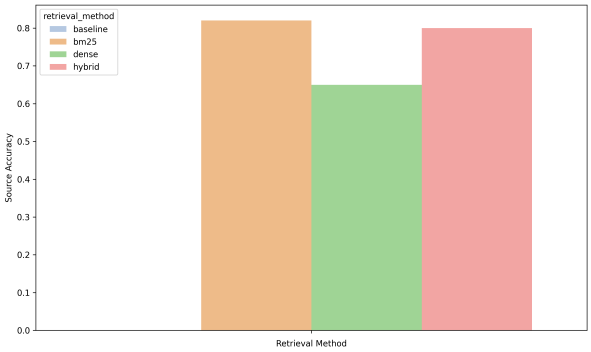

Figure 15: Source Accuracy

In terms of variance, all methods showed no variance over the answer and source correctness along the multiple runs, proving the stability of RAG as a system. We can also see how most successful methods also had higher coverage scores, suggesting the LLM was on a great part responsible enough to determine questions that were unanswerable with the given chunks. In terms of latency once again we obtained expected results, with the baseline model being the fastest method in average, since no retrieval is involved, followed by BM25 and dense retrieval afterwards, since apart from the keyword and similarity search, the embedding computation for dense retrieval means a feed forward pass through the embedding neural network, adding latency. And clearly the most time expensive method is hybrid retrieval, since both computations of BM25 and dense are required, plus the RRF algorithm.

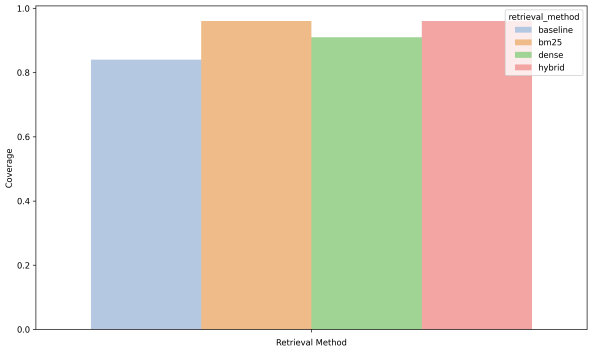

Figure 16: Coverage

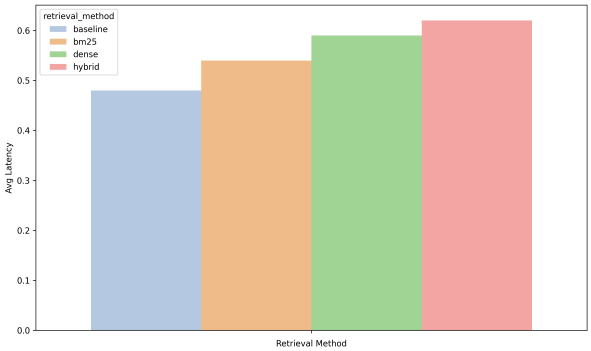

Figure 17: Latency

6. Conclusion

The core objective was to understand the impact of different chunking and retrieval methods in terms of answer accuracy and overall reliability, and thanks to our experiments we reached several key findings.

First, all RAG-based approaches (BM25, dense, and hybrid retrieval) clearly outperformed the baseline LLM, confirming the utility of augmenting LLMs with external, relevant context for precision tasks. For this specific technical Q&A task, the sparse retrieval method BM25 combined with paragraph based chunking emerged as the optimal choice, delivering the highest answer accuracy (94%) with a significantly low latency. Its performance could be attributed to its sensitivity to the precise technical keywords and acronyms prevalent in the research paper, combined with the smaller and better structured chunks without overlap obtained from paragraph chunking, that better align with BM25's keyword nature.

Furthermore, while the choice between paragraph-based and recursive chunking did not affect overall answer accuracy, recursive chunking proved superior for source attribution, suggesting it better preserves integral pieces of information, probably because of the overlap factor.

These results are inevitably bounded to this project: a single technical paper and a specific LLM. Future work could validate these findings across diverse document types and larger sources of information, or explore more advanced retrieval and chunking optimization. Ultimately, this study proves a key aspect of retrieval: method selection must be task-aware. When operating in a known, keyword-rich domain, we proved that sparse methods like BM25 are highly effective. However for general-purpose or uncertain applications, a hybrid retrieval strategy may provide a more robust and reliable RAG system.

Acknowledgements

This project was part of the *Concurso de Modelización de Problemas de Empresa*, an event organised in the *Facultad de Ciencias Matemáticas, UCM*. The problem proposal was given by the enterprise *Management Solutions*.

Author: *Ángel Valencia Padilla*

References

[1]: Epoch AI Analysts. (2024). *Estimated training cost and duration of large scale AI models such as GPT-4*. Hakia.

[2]: Image from *Navigating the basics of RAG indexing – Part 1 for beginners*, Medium, 2024.

Lin, X., Ghosh, A., Low, B. K. H., Shrivastava, A., & Mohan, V. (2025). *REFRAG: Rethinking RAG Based Decoding.* arXiv preprint arXiv:2509.01092.

Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.-t., Rocktäschel, T., Riedel, S., & Kiela, D. (2020).
Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. NeurIPS 2020.

Robertson, S., Walker, S., Jones, S., Hancock-Beaulieu, M., & Gatford, M. (1995). *Okapi at TREC-3.* Proceedings of the Third Text Retrieval Conference (TREC).

Salton, G., & Buckley, C. (1988).
Term-weighting approaches in automatic text retrieval. Information Processing & Management, 24(5), 513–523.

Karpukhin, V., Oguz, B., Min, S., Lewis, P., Wu, L., Edunov, S., Chen, D., & Yih, W. (2020). *Dense Passage Retrieval for Open-Domain Question Answering.* EMNLP 2020.

Carbonell, J., & Goldstein, J. (1998). *The Use of MMR, Diversity-Based Reranking for Reordering Documents and Producing Summaries.* SIGIR 1998.

Reimers, N., & Gurevych, I. (2019).
Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. EMNLP-IJCNLP 2019.

Retrieval-Augmented Generation for AI-Generated Content: A Survey. (2025). Data Science and Engineering (Springer).