Intelligent chunking methods for code documentation RAG

Task 1 overview

The implementation of the RAG pipeline is organized into several main steps, described in detail below:

1. **Prepare a Dataset** – For this project, the chosen dataset is **Wikitext**, primarily because it contains the largest number of user queries. The dataset consists of a **corpus** (a collection of documents), a set of **queries** relevant to those documents, and **labeled excerpts** from the corpus that correspond to each query.
2. **Implement the Chunking Algorithm** – We use the **FixedTokenChunker** algorithm. As the name suggests, this algorithm splits the corpus into fixed-size chunks of tokens, with optional overlapping between neighboring chunks.
3. **Define Retrieval Quality Metrics** – To evaluate the performance of the retrieval system, we implement **Precision** and **Recall** metrics. These are computed based on the overlap of tokens between retrieved chunks and ground truth (golden) excerpts.
4. **Prepare an Embedding Function** – We import a pre-trained embedding model and write a function to convert both the user queries and the chunked corpus text into vector representations (embeddings).
5. **Develop the Retrieval Evaluation Pipeline** – A main pipeline function is implemented, receiving necessary parameters and connecting all components into a complete **RAG pipeline**.
6. **Experiment and Analyze Results** – We conduct experiments by varying chunking hyperparameters and analyzing how they affect performance.

Precision and Recall

From experiments is clear that there is a very important trade-off between precision and recall metrics. In our pipeline, **Precision** and **Recall** are defined as:

$$\boldsymbol{Precision(C)} = \frac{|\boldsymbol{t_e} \cap \boldsymbol{t_r}|}{|\boldsymbol{t_r}|}$$

$$\boldsymbol{Recall(C)} = \frac{|\boldsymbol{t_e} \cap \boldsymbol{t_r}|}{|\boldsymbol{t_e}|}$$

where $\boldsymbol{t_e}$ and $\boldsymbol{t_r}$ denote set of tokens among relevant all relevant excerpts (labels) and set of tokens of all retrieved chunks, respectively.

Insights from Experiments

From our experiments, we observe a clear trade-off between **precision** and **recall**:

- For a **fixed chunk overlap**, increasing the **chunk size** improves recall (because there's a higher chance of including relevant tokens). However, this often leads to lower precision, since more irrelevant tokens may also be included.
- Increasing **chunk overlap** generally improves both precision and recall because it increases the redundancy between neighboring chunks, making it more likely to capture relevant tokens. However, large overlaps come at a cost: increased storage and computation, and higher redundancy.
- The **number of retrieved chunks (top-k)** is another important hyperparameter. Retrieving more chunks generally improves recall (more chances to include relevant parts), but may reduce precision (more irrelevant chunks retrieved).

Hyperparameter Tuning

Given the sensitivity of precision and recall to chunking parameters, it's important to **perform grid search** across:

- Chunk size
- Overlap
- Number of retrieved chunks (top-k)

However, grid search becomes expensive quickly, especially when searching over multiple hyperparameters. In our project, we evaluate performance using **12 different triplets** of chunk size, overlap, and number of retrieved chunks.

Chunk size	Chunk overlap	Top k	Avg precision	Avg recall	Num chunks
200	50	1	0.236	0.669	178
200	50	5	0.101	0.888	178
200	50	10	0.066	0.934	178
200	100	1	0.237	0.677	266
200	100	5	0.113	0.878	266
200	100	10	0.075	0.922	266
400	50	1	0.140	0.679	76
400	50	5	0.060	0.909	76

400	50	10	0.039	0.969	76
400	100	1	0.150	0.722	89
400	100	5	0.063	0.930	89
400	100	10	0.041	0.970	89

Limitations and Future Directions

The method used in this project is one of the **simplest** available. The major limitation lies in the fact that we split text **purely by token count**, which can cut:

- In the middle of a **sentence**, or
- Between sentences in a **paragraph**, breaking semantic meaning

As a result, retrieval quality suffers. More intelligent chunking methods are needed, such as:

- **Sentence-aware Chunking with Overlap**
- **Recursive Text Splitting**
- **Semantic Chunking**
- **Adaptive Chunking with Attention Budget**
- **Discourse-aware or Graph-based Chunking**

These methods aim to preserve semantic boundaries, improve embedding quality, and enhance retrieval performance.

Conclusion

The project demonstrates a full retrieval pipeline and highlights the impact of chunking hyperparameters on precision and recall. Although FixedTokenChunker is simple, it has clear limitations. Future improvements will require the use of more sophisticated chunking techniques that respect semantic structure and document coherence.