LAB 2

The 5 Levels of Text Splitting

One of the most effective strategies for improving the performance of language model applications- such as RAG (Retrieval-Augmented Generation)—is splitting large datasets into smaller, manageable "chunks". This process ensures that models receive precise context without exceeding token limits.

Level 1: Character Splitting

Character splitting is the most basic form of text segmentation. It involves dividing text into fixed-size chunks of $N$ characters, regardless of the content's structure or meaning.

- **Key Concepts**:
  - **Chunk Size**: The fixed number of characters per segment (e.g., 50, 100, or 100,000).
  - **Chunk Overlap**: A specified amount of shared text between sequential chunks to prevent the loss of context at split points.
- **Pros**: Highly simple and easy to implement.
- **Cons**: Rigid; it often cuts through words or sentences, ignoring the document's natural structure.

Implementation Example (Manual):

Python
```
text = "This is the text I would like to chunk up. It is the example text for this exercise"
chunks = []
chunk_size = 35

for i in range(0, len(text), chunk_size):
    chunk = text[i:i + chunk_size]
    chunks.append(chunk)
```

Level 2: Recursive Character Text Splitting

This method improves upon Level 1 by using a hierarchy of separators to keep related text (like paragraphs or sentences) together as much as possible.

- **Mechanism**: The splitter attempts to split by the first separator in a list. If the resulting chunk is still too large, it moves to the next separator.
- **Standard Separators (LangChain)**:
  1. "\n\n" (Paragraphs)
  2. "\n" (Lines)
  3. " " (Words)
  4. "" (Individual characters)

Implementation Example:

Python
```
from langchain_text_splitters import RecursiveCharacterTextSplitter

text_splitter = RecursiveCharacterTextSplitter(chunk_size=450, chunk_overlap=0)
docs = text_splitter.create_documents([text])
```

Level 3: Document Specific Splitting

This level involves tailoring the chunking strategy to specific file formats by utilizing their inherent structural markers (e.g., Markdown headers or Python class definitions).

Markdown Splitting

Focuses on structural elements like headers and code blocks.

- **Primary Separators**: \n#{1,6} (Headers), ``` (Code blocks), and --- (Horizontal lines).

Code Splitting (Python/JS)

Splits code based on logical boundaries to maintain functional context.

- **Python Separators**: \nclass, \ndef, \n\tdef (Indented functions).
- **JS Separators**: \nfunction, \nconst, \nlet, \nif, \nfor.

Level 4: Semantic Splitting

Semantic splitting moves away from static counts and focuses on the **meaning** of the text.

- **Strategy**: It uses an "embedding walk" to analyze the semantic similarity between sentences. Chunks are formed where there is a significant shift in meaning, ensuring that each segment contains a coherent idea.

Level 5: Agentic Splitting

The most advanced and experimental level, Agentic Splitting, utilizes an agent-like system to determine how to split text.

- **Strategy**: It employs a language model to dynamically assess the text and decide on the most effective split points based on context, intent, and complex reasoning, rather than predefined rules.

Evaluation Frameworks

To ensure your chunking strategy is effective, you must test it using retrieval evaluations. Key frameworks include:

- **LangChain Evals**: Tools for guided evaluation of language model outputs.
- **Llama Index Evals**: Comprehensive modules for evaluating RAG pipelines.
- **RAGAS Evals**: Specifically designed metrics for automated evaluation of Retrieval-Augmented Generation.