Text Splitters in LangChain for Generative AI Applications

1. What is Text Splitting and Why is it Important?

Text splitting is the process of breaking down large textual data (like articles, PDFs, HTML pages, or books) into smaller, more manageable pieces, often called "chunks," that an LLM can process effectively.

Key Definition: "Text splitting is the process of breaking large chunks of text like articles, PDFs, HTML pages or books into smaller manageable pieces that an LLM can handle effectively."

Why Text Splitting is Essential for LLM-Powered Applications:

- **Overcoming LLM Context Length Limitations:** LLMs have a finite "context length" – a maximum amount of text they can process at once. Large documents often exceed these limits. Text splitting allows "us to process documents that would otherwise exceed these limits."
- *Example:* If an LLM has a 50,000-token limit, a 100,000-word PDF cannot be sent as a single input. Splitting allows the LLM to process it in parts.
- **Improving Quality of Downstream Tasks:** Text splitting significantly enhances the performance of various LLM tasks:
- **Embedding:** Splitting large texts into smaller chunks before generating embeddings (converting text to numerical vectors) results in "better capture of semantic meaning." Large texts lead to less accurate embeddings because capturing the full semantic meaning in a single vector is difficult.
- **Semantic Search:** When performing semantic search (finding documents similar in meaning to a query), using chunks leads to "more precise" and "improved" search quality compared to searching on large, unbroken texts.
- **Summarization:** LLMs can "drift" or "hallucinate" (generate incorrect information) when summarizing very large documents. Text splitting "has been empirically proven" to yield "better results" for summarization.
- **Optimizing Computational Resources:** Working with smaller text chunks is "more memory efficient and allows for better parallelization of processing tasks," reducing computational requirements and storage needs.

2. Types of Text Splitting Techniques

The are mostly four main types of text splitting techniques, with a focus on their implementation in LangChain.

2.1. Length-Based Text Splitting (Character Text Splitter)

This is the simplest and fastest method. It involves pre-defining a chunk size (e.g., in characters or tokens) and then splitting the text at fixed intervals.

- **Mechanism:** The text is traversed, and chunks are created every time the predefined character/token limit is reached.
- **Example:** If the chunk size is 100 characters, the first 100 characters form chunk 1, the next 100 form chunk 2, and so on.
- **Advantages:** "Very simple conceptually."
- Easy to implement.
- Fast execution.

- **Disadvantages:Ignores Linguistic Structure:** Does not consider grammar, linguistic structure, or semantic meaning.
- **Abrupt Cuts:** Often cuts "text in the middle of a word, paragraph, or sentence." This can lead to a loss of context and degrade the quality of embeddings or LLM outputs, as "half the information is in one chunk and half in another."
- **chunk_overlap Parameter:** This parameter defines how many characters/tokens overlap between consecutive chunks.
- **Purpose:** To "retain context" that might otherwise be lost due to abrupt cuts. By starting the next chunk slightly before the previous one ended, "information that was cut can be saved."
- **Benefit:** Ensures "information is similar" between chunks, helping to pass on context.
- **Trade-off:** A larger overlap means more chunks and increased computational load.
- **Recommendation:** For RAG-based applications, "10 to 20% is a good number" for chunk overlap relative to chunk size.
- **LangChain Implementation:** Uses the CharacterTextSplitter class, specifying chunk_size, chunk_overlap, and separator. It can be used directly on raw text or integrated with DocumentLoaders using split_documents.

2.2. Text Structure-Based Text Splitting (Recursive Character Text Splitter)

This technique leverages the inherent hierarchical structure of text (paragraphs, sentences, words, characters) to make more intelligent splitting decisions.

- **Mechanism:** Uses a predefined list of separators (e.g., \n\n for paragraphs, \n for lines/sentences, for words, "" for characters). It attempts to split using the higher-level separators first. If a chunk is still too large, it progressively moves to lower-level separators until the chunk_size is met.
- "It first tries to create chunks based on paragraphs, but if it cannot create chunks based on paragraphs, then it tries to create chunks based on sentences, if it doesn't work on sentences, then it tries to create them based on words, and if it doesn't work on words, then finally it chunkees at the character level."
- **Goal:** To avoid abrupt cuts within meaningful units (words, sentences, paragraphs) as much as possible.
- **Advantages:**"One of the most used text splitting techniques."
- **Contextual Awareness:** More intelligent splitting than length-based methods, preserving context by breaking at natural boundaries.
- Adaptable to chunk_size: Larger chunk_size allows for paragraph-based splits; smaller sizes lead to sentence or word-based splits.
- **Disadvantages:** Still relies on pre-defined rules, not semantic understanding.
- **LangChain Implementation:** Uses the RecursiveCharacterTextSplitter class, specifying chunk_size and chunk_overlap. This is generally recommended as the best general-purpose splitter.

2.3. Document Structure-Based Text Splitting (Extension of Recursive Character Text Splitter)

This is an extension of the Recursive Character Text Splitter designed for specific document types that don't follow typical plain text structures, such as code or Markdown.

- **Mechanism:** Similar to the recursive splitter, but uses a *different set of separators* tailored to the specific document type's constructs (e.g., class, def for Python code; headings, lists for Markdown; HTML tags for HTML). After applying these specific separators, it can fall back to general paragraph, sentence, and word separators if chunks are still too large.
- **Purpose:** To correctly split structured content based on its inherent logical divisions.

- **Examples:Python Code:** Splits based on class definitions, def (function) definitions, etc.
- **Markdown:** Splits based on headings (#, ##), lists (-), etc.
- **HTML:** Splits based on HTML tags.
- **LangChain Implementation:** Uses RecursiveCharacterTextSplitter.from_language() by specifying the Language enum (e.g., Language.PYTHON, Language.MARKDOWN, Language.JAVASCRIPT, Language.HTML).

2.4. Semantic Meaning-Based Text Splitting (Experimental)

This advanced technique attempts to split text based on changes in semantic meaning or topic rather than just length or structural rules.

1. **Mechanism:** Splits text into small units, typically sentences.
2. Generates embedding vectors for each sentence using an embedding model (e.g., OpenAI's embedding model).
3. Calculates the "similarity" (e.g., cosine similarity) between consecutive sentence embeddings.
4. Identifies "break points" where the similarity between adjacent sentences drops significantly, indicating a topic change.

- "Wherever they feel that the similarity has suddenly become very low, that point indicates that the topic has changed here."
- **Threshold Type:** Various criteria can be used to determine a "significant drop" in similarity, such as "standard deviation," "percentile," "interquartiles," or "gradient."
- **Advantages:Contextually Superior:** Aims to produce chunks that are highly coherent semantically, even if they span structural boundaries like paragraphs. This would be ideal for cases where a single paragraph discusses multiple distinct topics (e.g., agriculture and IPL in one paragraph).
- **Disadvantages:Experimental Stage:** Currently "experimental" in LangChain and "not used much."
- **Accuracy Issues:** The speaker notes that in their experience, its "performance was not very accurate."
- **Computational Cost:** Requires embedding generation and similarity calculations, which are more computationally intensive.
- **Future Outlook:** This concept is "very promising" and could become a "main text splitter" as embedding models improve.
- **LangChain Implementation:** Uses SemanticChunker from langchain_experimental, requiring an embedding model and a threshold_type (e.g., standard_deviation) to be specified.