5 Levels of Text Splitting

1. Introduction

Modern large language models (LLMs) face practical limitations related to context length and memory management. When processing long documents, naive input strategies often result in truncated information or degraded performance. Text splitting (also referred to as text chunking) is a foundational preprocessing technique that addresses these limitations by dividing large texts into manageable segments while attempting to preserve semantic coherence.

This project systematically demonstrates **five progressive levels of text splitting**, ranging from simple character-based methods to more semantically informed approaches. The objective is to provide a conceptual and practical understanding of how different splitting strategies affect downstream LLM performance, such as retrieval-augmented generation (RAG), summarization, and question answering.

2. Objectives

The primary objectives of this project are:

- To explain the necessity of text splitting in LLM-based systems.
- To demonstrate multiple text splitting strategies with increasing sophistication.
- To analyze the advantages and limitations of each splitting level.
- To provide reproducible Python implementations for educational and practical use.

3. System Requirements

3.1 Software Requirements

- Python 3.8 or above
- Jupyter Notebook
- Standard Python libraries (no heavy dependencies required for basic demonstrations)

3.2 Hardware Requirements

- Any standard system capable of running Python and Jupyter Notebook
- Minimum 4 GB RAM recommended

4. Methodology: Levels of Text Splitting

4.1 Level 1: Character-Based Splitting

Description:
Character-based splitting divides text into fixed-size chunks based purely on character count, without considering linguistic structure.

Implementation Overview:

- A predefined chunk_size is selected.
- The text is iterated character-wise and sliced accordingly.

Advantages:

- Extremely simple to implement
- No dependency on linguistic rules

Limitations:

- May split words or sentences arbitrarily
- Poor semantic preservation

Use Case:

- Rarely recommended, suitable only for low-stakes or experimental preprocessing

4.2 Level 2: Recursive Character Splitting

Description:
This method improves upon simple character splitting by attempting to split text using a hierarchy of separators (e.g., paragraphs, sentences, words) before falling back to character limits.

Advantages:

- Better structural awareness than pure character splitting
- Reduces abrupt semantic breaks

Limitations:

- Still not fully semantically aware
- Depends heavily on well-defined separators

4.3 Level 3: Document or Sentence-Based Splitting

Description:
At this level, splitting is performed along natural linguistic boundaries such as sentences or paragraphs.

Advantages:

- Preserves syntactic integrity
- More readable chunks

Limitations:

- Chunk sizes may vary significantly
- Long sentences or paragraphs may still exceed model limits

4.4 Level 4: Token-Based Splitting

Description:
Token-based splitting uses tokenizer logic similar to that of LLMs to ensure each chunk fits within a defined token limit.

Advantages:

- Aligns closely with LLM processing constraints
- Prevents token overflow errors

Limitations:

- Requires access to or implementation of tokenizers
- Slightly higher computational overhead

4.5 Level 5: Semantic-Based Splitting

Description:
Semantic splitting divides text based on meaning rather than length alone, often using embeddings or similarity measures to group related sentences.

Advantages:

- Highest semantic coherence
- Ideal for RAG and knowledge retrieval systems

Limitations:

- Computationally expensive
- Requires embedding models and similarity calculations

5. Implementation Details

Each level is implemented incrementally within the Jupyter Notebook using Python. Code cells demonstrate:

- Text preprocessing
- Chunk generation logic
- Output inspection for validation

6. Results and Observations

- Simpler splitting techniques are easy to implement but result in poor contextual integrity.
- Advanced splitting strategies significantly improve semantic continuity and downstream task performance.
- There is a trade-off between computational complexity and chunk quality.

7. Applications

The techniques demonstrated in this project are applicable to:

- Retrieval-Augmented Generation (RAG) systems
- Chatbots with long-term memory
- Document summarization
- Question-answering over large corpora
- Search and indexing pipelines

8. Conclusion

This project highlights that effective text splitting is a critical preprocessing step for building reliable and scalable LLM-based applications. While basic methods may suffice for small inputs, advanced semantic-aware splitting is essential for production-grade systems handling large and complex documents.

Future work may involve benchmarking these splitting strategies quantitatively using retrieval accuracy or response quality metrics.