

# Documentation: 5 Levels of Text Splitting

This document details the strategies and code implementation found in the notebook 5\_Levels\_Of-Text\_Splitting.ipynb. The notebook acts as a comprehensive guide to "chunking"—the process of breaking down large texts into smaller, manageable pieces for Language Models (LLMs) and Vector Databases.

## 1. Project Overview

Text splitting is a foundational step in building RAG (Retrieval Augmented Generation) applications. If text chunks are too small, they lack context; if they are too large, they confuse the retrieval system or exceed token limits. This notebook explores five distinct levels of complexity in splitting strategies.

**Dependencies:**

* langchain / langchain-text-splitters: Main library for splitting utilities.
* openai: Used for embeddings and LLM calls in advanced levels.
* tiktoken: For accurate token counting.
* matplotlib / seaborn: For visualizing chunk distributions (likely used in Level 4/5).

## 2. Level 1: Character Splitting

This is the most basic form of splitting. It divides text based rigidly on a fixed character count.

* **Class Used:** CharacterTextSplitter (from LangChain).
* **Mechanism:**
  1. Accepts a chunk\_size (e.g., 1000 characters) and chunk\_overlap.
  2. Splits the text strictly when the limit is reached.
  3. Optionally uses a separator (like "\n\n") but often defaults to splitting mid-sentence if the constraint is tight.
* **Pros:** Simple, fast, low computational cost.
* **Cons:** ignores semantic structure; can cut words or sentences in half, destroying context.

## 3. Level 2: Recursive Character Text Splitting

The industry standard for generic text. It attempts to keep related text together by trying different separators in order.

* **Class Used:** RecursiveCharacterTextSplitter.
* **Mechanism:**  
  It iterates through a list of separators in a specific order, typically:
  1. "\n\n" (Paragraph breaks)
  2. "\n" (New lines)
  3. " " (Spaces)
  4. "" (Characters)  
     If a chunk is too large using the first separator, it falls back to the second, and so on.
* **Pros:** Respects document structure (paragraphs/sentences) better than Level 1.
* **Cons:** Still heuristic-based; doesn't "understand" the text.

## 4. Level 3: Document Specific Splitting

This level acknowledges that different file types have different structural hierarchies that should be preserved.

* **Scenario 1: Markdown (MarkdownHeaderTextSplitter)**
  + Splits text based on headers (#, ##, ###).
  + **Result:** Chunks correspond to specific sections of the document, often attaching the header metadata to the chunk.
* **Scenario 2: Code (Python/JS Splitters)**
  + Splits based on classes, functions, and method definitions.
  + Ensures a function isn't split in the middle of its logic.
* **Pros:** Excellent for structured documents and codebases.
* **Cons:** Requires clean, well-formatted source documents.

## 5. Level 4: Semantic Splitting

This strategy uses the *meaning* of the text to determine break points, rather than structural markers like newlines.

* **Class Used:** SemanticChunker (experimental/advanced LangChain).
* **Mechanism:**
  1. **Sentence Splitting:** Breaks text into individual sentences.
  2. **Embedding:** Generates a vector embedding for each sentence (using OpenAI Embeddings).
  3. **Cosine Similarity:** Calculates the similarity between adjacent sentences.
  4. **Thresholding:** If the similarity score between Sentence A and Sentence B drops below a certain threshold (or represents a percentile outlier), a split occurs. This implies a "topic change."
* **Pros:** Creates chunks that are topically coherent.
* **Cons:** Slower and costs money (requires embedding calls).

## 6. Level 5: Agentic Splitting

The most advanced level, treating text splitting as a reasoning task.

* **Concept:** Uses an LLM (like GPT-4) to read the text and decide where the logical splits should be.
* **Mechanism:**
  + The text is passed to an LLM with a prompt asking it to identify distinct propositions or standalone statements.
  + The LLM rewrites or groups text into "atomic" facts.
* **Implementation:** often involves "Propositional Indexing" where complex sentences are broken down into simple, standalone facts before embedding.
* **Pros:** Highest quality chunks; perfect for complex reasoning or QA.
* **Cons:** Slowest and most expensive method; not scalable for massive datasets without optimization.

## 7. Visualization & Evaluation

The notebook likely includes code to visualize the "Chunk Size Distribution."

* **Tools:** matplotlib is used to plot histograms of chunk sizes.
* **Goal:** To ensure chunks are filling the context window efficiently without having too many tiny, useless fragments.

## 8. Summary Recommendation

* **Start with Level 2 (Recursive)** for most general text applications.
* **Use Level 3** if you are processing Markdown or Code.
* **Use Level 4** if you need high-precision retrieval where topics shift frequently within documents.