Working of Code: 5 Levels of Text Splitting

Introduction

This notebook demonstrates five different levels of text splitting (chunking) strategies used in natural language processing and large language model applications. Text splitting is essential when working with large documents that exceed token limits or when building retrieval-augmented generation (RAG) systems with better context management.

The notebook uses LangChain's text splitting utilities to demonstrate practical implementations of each strategy, progressing from simple character-based splitting to sophisticated agentic approaches.

Overview of the Five Levels

- **Level 1: Character Splitting:** Simple static character chunks of data
- **Level 2: Recursive Character Text Splitting:** Recursive chunking based on a list of separators
- **Level 3: Document Specific Splitting:** Various chunking methods for different document types (PDF, Python, Markdown)
- **Level 4: Semantic Splitting:** Embedding walk based chunking
- **Level 5: Agentic Splitting:** Experimental method of splitting text with an agent-like system

Level 1: Character Splitting

Concept

Character splitting is the most basic form of text chunking. It divides text into fixed-size chunks based purely on character count, regardless of content structure or meaning.

Key Parameters

- **Chunk Size:** The number of characters per chunk (e.g., 35, 100, 1000)
- **Chunk Overlap:** The number of characters that overlap between consecutive chunks to avoid splitting context

How It Works

Step 1: Define the sample text

```
text = "This is the text I would like to chunk up. It is the example text for this exercise"
```

Step 2: Manual implementation using Python

The code iterates through the text with a fixed step size (chunk_size) and extracts chunks:

```
chunks = []
chunk_size = 35  # Characters

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

This produces three chunks:

```
['This is the text I would like to ch',
 'unk up. It is the example text for ',
 'this exercise']
```

Step 3: Using LangChain's CharacterTextSplitter

LangChain provides a more sophisticated implementation that works with Document objects:

```
from langchain_text_splitters import CharacterTextSplitter

text_splitter = CharacterTextSplitter(
    chunk_size=35,
    chunk_overlap=0,
    separator='',
    strip_whitespace=False
)

text_splitter.create_documents([text])
```

Pros and Cons

- **Pros:** Simple to implement and understand

- **Cons:** Very rigid; doesn't respect sentence boundaries, paragraphs, or document structure

Level 2: Recursive Character Text Splitting

Concept

Recursive character splitting is a more intelligent approach that attempts to split text at natural boundaries. It uses a hierarchy of separators (like double newlines, single newlines, spaces) and recursively applies them until chunks are small enough.

How It Works

Step 1: Define separator hierarchy

The splitter tries separators in order: paragraphs (\n\n), lines (\n), sentences (.), then spaces.

```
from langchain_text_splitters import RecursiveCharacterTextSplitter

text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=65,
    chunk_overlap=0
)
```

Step 2: Process sample text

```
some_text = """When writing documents, writers will use document structure to group
content. This can convey to the reader which ideas are related. For example, closely
related ideas are in sentences. Similar ideas are in paragraphs. Paragraphs form a
document. \n\nParagraphs are often delimited with a carriage return or two carriage
returns. Carriage returns are the "backslash n" you see embedded in this string.
Sentences have a period at the end, but also spaces and words too"""
```

Step 3: Split and analyze chunks

The algorithm tries to split at paragraph boundaries first, then sentences, then words:

```
chunks = text_splitter.create_documents([some_text])

# Results in multiple chunks that respect natural boundaries:
# Chunk 1: "When writing documents, writers will use document structure to"
# Chunk 2: "group content. This can convey to the reader which ideas are"
# etc.
```

Customization

You can customize the separator list for specific needs:

```
custom_text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=450,
    chunk_overlap=0,
    separators=[
        "\n\n",  # Paragraph breaks (highest priority)
        "\n",     # Line breaks
        " ",      # Spaces
        ""        # Character-by-character (last resort)
    ]
)
```

Advantages

- Respects natural document structure
- Better context preservation than simple character splitting
- Flexible and customizable separator hierarchy

Level 3: Document-Specific Splitting

Concept

Different document types have different structures. Document-specific splitters are optimized for particular formats like Markdown, Python code, HTML, or LaTeX.

Markdown Splitter

Splits Markdown documents while respecting headers, lists, and code blocks:

```
from langchain_text_splitters import MarkdownHeaderTextSplitter

markdown_document = """# Title\n\n## Section 1\n\nThis is content...\n\n## Section
2\n\nMore content..."""

headers_to_split_on = [
    ("#", "Header 1"),
    ("##", "Header 2"),
    ("###", "Header 3"),
]

markdown_splitter =
MarkdownHeaderTextSplitter(headers_to_split_on=headers_to_split_on)
md_header_splits = markdown_splitter.split_text(markdown_document)
```

Each resulting chunk includes metadata about which headers it belongs to, preserving document hierarchy.

Python Code Splitter

Specialized for Python code, respecting function and class boundaries:

```
from langchain_text_splitters import RecursiveCharacterTextSplitter, Language

python_splitter = RecursiveCharacterTextSplitter.from_language(
    language=Language.PYTHON,
    chunk_size=50,
    chunk_overlap=0
)

python_code = """
def hello_world():
    print("Hello, World!")

class MyClass:
    def __init__(self):
        self.value = 42
"""

python_docs = python_splitter.create_documents([python_code])
```

This splitter understands Python syntax and tries to keep complete functions and classes together.

Supported Languages

LangChain supports specialized splitting for many programming languages:

- Python
- JavaScript
- TypeScript
- Java
- C++
- Go
- Ruby
- Rust
- PHP
- HTML
- LaTeX
- Markdown

Level 4: Semantic Splitting

Concept

Semantic splitting uses embeddings to understand the meaning of text and creates chunks based on semantic similarity rather than just structure. This ensures that semantically related content stays together.

How It Works

Step 1: Install required packages

```
pip install langchain-experimental sentence-transformers
```

Step 2: Set up the semantic splitter

```
from langchain_experimental.text_splitter import SemanticChunker
from langchain_community.embeddings import HuggingFaceEmbeddings

# Initialize embeddings model
embeddings = HuggingFaceEmbeddings(model_name="all-MiniLM-L6-v2")

# Create semantic splitter
text_splitter = SemanticChunker(embeddings)
```

Step 3: Split text semantically

```
text = """Your long document text here..."""

docs = text_splitter.create_documents([text])
```

Algorithm Details

The semantic chunker works by:

1. Breaking text into sentences
2. Creating embeddings for each sentence
3. Computing similarity between consecutive sentences
4. Grouping sentences together when similarity is high
5. Creating chunk boundaries when similarity drops

Breakpoint Types

The SemanticChunker supports different breakpoint strategies:

- **Percentile:** Splits at the top percentile of similarity differences
- **Standard Deviation:** Splits when similarity drops by more than one standard deviation
- **Interquartile:** Uses interquartile range to determine breakpoints

Advantages

- Preserves semantic coherence within chunks
- More intelligent than structure-based splitting

- Better for retrieval-augmented generation (RAG) systems

Disadvantages

- Computationally more expensive (requires embeddings)
- Requires an embeddings model
- Slower processing time

Level 5: Agentic Splitting

Concept

Agentic splitting is an experimental approach that uses a language model as an "agent" to make intelligent decisions about how to split text. The agent analyzes the content and determines optimal chunk boundaries based on context, meaning, and task requirements.

How It Works

Step 1: Install dependencies

```
pip install langchain langchain-experimental openai
```

Step 2: Set up the agentic splitter

```
from langchain_experimental.text_splitter import AgenticChunker
from langchain_openai import ChatOpenAI

# Initialize the language model
llm = ChatOpenAI(temperature=0, model="gpt-4")

# Create the agentic splitter
agentic_splitter = AgenticChunker(llm=llm)
```

Step 3: Define chunking strategy

You can provide instructions to guide the agent's chunking decisions:

```
text = """Your document text..."""

chunks = agentic_splitter.create_documents(
    [text],
    instructions="Split this document into chunks that each focus on a single main
idea or concept."
)
```

Agent Decision Process

The agentic splitter works through these steps:

6. Analyzes the document structure and content
7. Identifies key topics, themes, and transitions
8. Considers the user's instructions about chunking strategy
9. Makes intelligent decisions about where to place boundaries
10. Can reason about context that should stay together
11. Adapts to different document types and content

Example Use Cases

- **Research papers:** Split by methodology, results, and conclusions
- **Legal documents:** Split by clauses while keeping related provisions together

- **Technical documentation:** Split by features or components with intelligent context preservation

Advantages

- Most intelligent and context-aware splitting method
- Can adapt to different document types and requirements
- Accepts natural language instructions
- Can reason about complex document structures

Disadvantages

- Most computationally expensive (requires LLM API calls)
- Slowest processing time
- Results may vary based on the LLM used
- Still experimental and may not be production-ready

Comparison and Selection Guide

When to Use Each Level

- **Character Splitting (Level 1):** Quick prototyping, simple applications, when document structure doesn't matter
- **Recursive Character Splitting (Level 2):** General-purpose text splitting, when you want to respect natural boundaries, most common use case
- **Document-Specific Splitting (Level 3):** Working with code, Markdown, or structured documents, when format-specific handling is important
- **Semantic Splitting (Level 4):** RAG systems, when semantic coherence is critical, when you have computational resources for embeddings
- **Agentic Splitting (Level 5):** Complex documents requiring intelligent analysis, when you need adaptive splitting strategies, experimental projects

Key Considerations

- **Chunk Size:** Consider your model's context window and the optimal size for retrieval
- **Chunk Overlap:** Prevents context loss at boundaries but creates data duplication
- **Evaluation:** Always test your chunking strategy with retrieval evaluations
- **Performance vs. Quality:** Balance computational cost with chunking quality for your use case
- **Document Type:** Different document types may benefit from different strategies

Evaluation Frameworks

It's crucial to evaluate your chunking strategy's performance in real retrieval scenarios. The notebook recommends these frameworks:

- **LangChain Evals:** *https://python.langchain.com/docs/guides/evaluation/*
- **Llama Index Evals:** *https://docs.llamaindex.ai/en/stable/module_guides/evaluating/root.html*
- **RAGAS Evals:** *https://github.com/explodinggradients/ragas*

Summary

The notebook provides a comprehensive progression from simple to sophisticated text splitting strategies. Starting with basic character splitting and advancing through recursive methods, document-specific approaches, semantic understanding, and finally agent-based decision making, each level offers different trade-offs between simplicity, performance, and quality.

For most applications, Level 2 (Recursive Character Splitting) provides a good balance of simplicity and effectiveness. Level 3 should be used when working with structured documents. Levels 4 and 5 are best suited for advanced applications where semantic coherence is critical and computational resources are available.

The key to success is evaluating your chosen strategy against real-world performance metrics using the recommended evaluation frameworks. Remember that the best chunking strategy depends on your specific use case, document types, and application requirements.