Practical Lab File for Agentic AI

Sharda School of Engineering and Technology

Name – Mohd Mujeeb
Sys ID- 2023248197
Semester/Year- 6th sem, 3rd Year

Faculty-In-Charge/Submitted To
Mr. Ayush Singh

5 Levels of Text Splitting

A Comprehensive Guide to Text Chunking Strategies

Introduction

Ever tried to put a long piece of text into ChatGPT only to be told it's too long? Or struggled to give your application better long-term memory? **One of the most effective strategies to improve the performance of your language model applications is to split your large data into smaller pieces.**

This is called **splitting** or **chunking** (we'll use these terms interchangeably). In the world of multi-modal AI, splitting also applies to images and other data types.

This notebook demonstrates 5 different levels of text splitting, from basic to advanced, showing how each method improves upon the previous one.

Table of Contents

Level 1: Character Splitting

Character splitting is the most basic form of splitting up your text. It is the process of simply dividing your text into N-character sized chunks regardless of their content or form.

Overview

This method isn't recommended for production applications, but it's a great starting point for understanding the basics of text chunking.

Pros and Cons

Pros:

- Easy and simple to implement

Cons:

- Very rigid and doesn't take into account the structure of your text

Key Concepts

- **Chunk Size:** The number of characters in each chunk
- **Chunk Overlap:** How many characters from the end of one chunk appear at the beginning of the next
- **Separator:** The character or string used to split the text

Manual Implementation

Here's how to implement character splitting manually in Python:

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

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

Using LangChain CharacterTextSplitter

LangChain provides a more sophisticated approach with the CharacterTextSplitter class. This handles the chunking process and wraps results in Document objects with metadata.

```
from langchain_text_splitters import CharacterTextSplitter

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

text_splitter.create_documents([text])
```

Understanding Chunk Overlap

Chunk overlap blends chunks together so that the tail of Chunk #1 will be the same as the head of Chunk #2, and so on. This helps maintain context across chunk boundaries.

Example with overlap of 4 characters:

```
text_splitter = CharacterTextSplitter(
    chunk_size=35,
    chunk_overlap=4,
    separator=''
)
```

Result: The text "o ch" at the tail of Chunk #1 matches the "o ch" at the head of Chunk #2, creating continuity between chunks.

Limitations

- Splits text at arbitrary positions without considering meaning or structure
- May split words in the middle
- Doesn't respect sentence or paragraph boundaries
- Not suitable for production applications

Level 2: Recursive Character Text Splitting

The problem with Level 1 is that we don't take into account the structure of our document at all. We simply split by a fixed number of characters.

The Recursive Character Text Splitter improves on this by specifying a series of separators which will be used to split documents in a hierarchical manner.

How It Works

The splitter tries to split text using a list of separators in order:

- \n\n: Paragraph breaks (highest priority)
- \n: Line breaks
- : Spaces
- : Individual characters (lowest priority)

The splitter attempts to keep related content together by prioritizing natural break points in the text.

Implementation

```
from langchain_text_splitters import RecursiveCharacterTextSplitter

text = """
One of the most important things I didn't understand about the world
when I was a child is the degree to which the returns for performance
are superlinear.

Teachers and coaches implicitly told us the returns were linear.
"You get out," I heard a thousand times, "what you put in."
They meant well, but this is rarely true.
"""

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

text_splitter.create_documents([text])
```

Key Features

- More chunks end with periods (.), indicating natural sentence boundaries
- Respects paragraph breaks first before splitting further
- If a chunk is too large after paragraph splitting, it moves to the next separator (line breaks, then spaces)
- Provides "snap to separator" functionality with built-in cushion for better chunk boundaries

Adding Metadata

You can enhance chunks with metadata for better tracking and filtering:

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

for i, doc in enumerate(docs):
    doc.metadata = {
        "source_file": "file.txt",
        "chunk_no": i
    }
```

Advantages Over Level 1

- Respects document structure (paragraphs, sentences)
- Creates more meaningful chunks that preserve context
- Better for semantic coherence
- Suitable for most general-purpose text splitting needs

Level 3: Document Specific Splitting

Level 3 introduces document-type awareness. Instead of treating all text the same, we use specialized splitters optimized for different formats and programming languages.

Markdown Splitting

The Markdown splitter uses Markdown-specific separators to respect document structure:

```
from langchain_text_splitters import MarkdownTextSplitter

markdown_text = """
# Fun in California

## Driving

Try driving on the 1 down to San Diego

### Food

Make sure to eat a burrito while you're there

## Hiking

Go to Yosemite
"""

splitter = MarkdownTextSplitter(chunk_size=40, chunk_overlap=0)
splitter.create_documents([markdown_text])
```

Key Feature: Chunks gravitate towards Markdown sections (headers), preserving the logical structure of the document.

Python Code Splitting

Python-specific splitter with separators optimized for Python syntax:

Separator hierarchy:

- \nclass
- \ndef
- \n\tdef
- \n\n
- \n
-
-

```
from langchain_text_splitters import PythonCodeTextSplitter

python_text = """
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

p1 = Person("John", 36)

for i in range(10):
    print(i)
"""

python_splitter = PythonCodeTextSplitter(
    chunk_size=100,
    chunk_overlap=0
)

python_splitter.create_documents([python_text])
```

JavaScript Code Splitting

JavaScript-specific splitter with JS syntax awareness:

Separator hierarchy:

- \nfunction
- \nconst
- \nlet
- \nvar
- \nclass
- \nif
- \n\n
- \n
-
-

```
from langchain_text_splitters import RecursiveCharacterTextSplitter, Language

javascript_text = """
// Function is called, the return value will end up in x
let x = myFunction(4, 3);

function myFunction(a, b) {
    // Function returns the product of a and b
    return a * b;
}
"""

js_splitter = RecursiveCharacterTextSplitter.from_language(
    language=Language.JS,
    chunk_size=65,
    chunk_overlap=0
)

js_splitter.create_documents([javascript_text])
```

Benefits of Document-Specific Splitting

- Preserves syntactic structure of code
- Keeps related code blocks together (classes, functions)
- Respects language-specific conventions
- Better context preservation for code analysis and generation
- Maintains document hierarchy (Markdown headers)

Level 4: Semantic Chunking

Semantic chunking uses embeddings and similarity measures to create chunks based on *meaning* rather than just structure. This is a significant leap forward in chunking sophistication.

How It Works

The process involves:

1. Break text into sentences
2. Generate embeddings for each sentence using a model (e.g., OpenAI embeddings)
3. Calculate cosine similarity between consecutive sentences
4. Group sentences together when similarity is high
5. Create chunk boundaries when similarity drops below a threshold

Implementation

```
from langchain_experimental.text_splitter import SemanticChunker
from langchain_openai import OpenAIEmbeddings

tesla_text = """
Tesla's Q3 Results
Tesla reported record revenue of $25.2B in Q3 2024.
The company exceeded analyst expectations by 15%.
Revenue growth was driven by strong vehicle deliveries.

Model Y Performance
The Model Y became the best-selling vehicle globally.
Customer satisfaction ratings reached an all-time high.
"""

text_splitter = SemanticChunker(
    OpenAIEmbeddings()
)

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

Understanding Embeddings and Similarity

Here's how to manually calculate semantic similarity:

```
from openai import OpenAI
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity

client = OpenAI()

# Generate embeddings
sentences = [
    "Tesla reported record revenue.",
    "The company exceeded expectations.",
    "Revenue growth was strong."
]

embeddings = []
for s in sentences:
    response = client.embeddings.create(
        model="text-embedding-3-small",
        input=s
    )
    embeddings.append(response.data[0].embedding)

embeddings = np.array(embeddings)

# Calculate similarity
similarity_matrix = cosine_similarity(embeddings)
print(f"S1 vs S2: {similarity_matrix[0][1]:.3f}")
print(f"S2 vs S3: {similarity_matrix[1][2]:.3f}")
```

Key Concepts

- **Embeddings:** Vector representations of text that capture semantic meaning
- **Cosine Similarity:** Measure of similarity between two vectors (ranges from -1 to 1)
- **High Similarity:** Indicates sentences are semantically related and should be grouped
- **Low Similarity:** Suggests a topic change, indicating a good chunk boundary

Advantages

- Creates chunks based on actual meaning and topic coherence
- Automatically detects topic boundaries
- Better preserves semantic context within chunks
- Ideal for RAG (Retrieval-Augmented Generation) applications
- More intelligent than purely structural approaches

Considerations

- Requires API calls to embedding models (cost and latency)
- More computationally expensive than previous levels
- Quality depends on the embedding model used
- May need tuning of similarity thresholds for optimal results

Level 5: Agentic Chunking

Agentic chunking represents the cutting edge of text splitting technology. It uses **Large Language Models (LLMs)** to make intelligent decisions about how to chunk text based on understanding, context, and purpose.

The Concept

Instead of following predetermined rules, agentic chunking uses an LLM to:

- Understand the content and purpose of the text
- Identify logical boundaries based on topic, argument structure, or narrative flow
- Make context-aware decisions about chunk size and boundaries
- Adapt chunking strategy based on the specific document type and use case
- Generate summaries or metadata for each chunk

Basic Implementation

```
from langchain_openai import ChatOpenAI

llm = ChatOpenAI(model="gpt-4", temperature=0)

tesla_text = """
Tesla's Q3 Results
Tesla reported record revenue of $25.2B in Q3 2024.
The company exceeded analyst expectations by 15%.

Model Y Performance
The Model Y became the best-selling vehicle globally.
"""

# Ask LLM to chunk the text
prompt = f"""
Analyze the following text and divide it into logical chunks.
For each chunk, identify the main topic and create a boundary.

Text:
{tesla_text}

Return chunks with topic labels.
"""

response = llm.invoke(prompt)
```

Advanced Features

- **Dynamic Sizing:** LLM can determine optimal chunk size based on content density and complexity
- **Context Preservation:** Ensures important context isn't split across chunks inappropriately
- **Multi-Modal Understanding:** Can handle mixed content (text, code, tables) intelligently
- **Purpose-Driven:** Can optimize chunking for specific downstream tasks (summarization, QA, etc.)
- **Metadata Generation:** Automatically creates rich metadata including summaries, keywords, and topics

Use Cases

- Complex documents with varying structures (legal contracts, research papers)

- Multi-topic documents requiring topic-aware chunking
- Documents where semantic coherence is critical
- Applications requiring rich chunk metadata
- Scenarios where chunk quality is more important than processing speed

Trade-offs

- **Cost:** Most expensive approach due to LLM API calls
- **Speed:** Slowest method, not suitable for real-time applications
- **Quality:** Highest quality chunks with best context preservation
- **Consistency:** May vary between runs due to LLM non-determinism
- **Scalability:** Limited by API rate limits and cost at scale

Key Concepts Summary

Chunk Size

Definition: The target length of each text chunk, typically measured in characters or tokens.

Considerations:

- Smaller chunks: More granular, better for precise retrieval, but may lose context
- Larger chunks: More context, but may include irrelevant information
- Optimal size depends on your use case and model context window
- Typical ranges: 200-1000 characters for most applications

Chunk Overlap

Definition: The number of characters that overlap between consecutive chunks.

Benefits:

- Prevents important information from being split at chunk boundaries
- Maintains context across chunks
- Helps with retrieval when queries span chunk boundaries
- Typical overlap: 10-20% of chunk size

Separators

Definition: Characters or strings used to identify natural split points in text.

Common separators (in priority order):

- Double newline (\n\n) - paragraph breaks
- Single newline (\n) - line breaks
- Period with space (.) - sentence boundaries
- Comma with space (,) - clause boundaries
- Space ( ) - word boundaries
- Empty string () - character boundaries

Documents vs. Strings

Document Objects: LangChain wraps text in Document objects that contain:

- **page_content:** The actual text of the chunk
- **metadata:** Dictionary of additional information (source, chunk number, etc.)

This structure makes filtering, tracking, and managing chunks much easier than working with raw strings.

Best Practices

Choosing the Right Level

- **Level 1 (Character):** Quick prototyping only, not recommended for production
- **Level 2 (Recursive):** Good default for general text, blogs, articles
- **Level 3 (Document-Specific):** Use when working with code or structured formats (Markdown, HTML)
- **Level 4 (Semantic):** Best for RAG applications, QA systems, semantic search
- **Level 5 (Agentic):** Complex documents, highest quality needs, when cost is not primary concern

General Guidelines

- Start simple and increase complexity only when needed
- Always test your chunking strategy with representative data
- Monitor chunk quality in production - adjust based on performance
- Consider your downstream task when choosing chunk size
- Use overlap to maintain context, especially for QA tasks
- Document your chunking strategy for reproducibility
- Balance quality, cost, and processing time for your use case

Common Pitfalls

- **Too Small Chunks:** Lose context, require more retrieval operations, less efficient
- **Too Large Chunks:** Include irrelevant information, exceed model context limits
- **No Overlap:** Important information split across boundaries
- **Wrong Separators:** Unnatural splits that break meaning
- **Ignoring Document Type:** Generic splitting on specialized content (code, tables)

Performance Optimization

- Cache embeddings for semantic chunking to avoid redundant API calls
- Batch process documents when possible
- Use async operations for agentic chunking
- Pre-process documents to remove unnecessary whitespace and formatting
- Consider chunk size vs. embedding model limits
- Monitor and log chunk statistics for optimization

Conclusion

Text splitting is a critical component of modern LLM applications. Choosing the right chunking strategy can significantly impact the performance of your system.

Key Takeaways

- There is no one-size-fits-all solution - choose based on your specific needs
- Start with Level 2 (Recursive) as a baseline for most applications
- Upgrade to Level 4 (Semantic) for retrieval-heavy applications
- Reserve Level 5 (Agentic) for complex documents where quality is paramount
- Always consider the trade-offs between quality, cost, and speed
- Test with real data and measure performance in your actual use case

Next Steps

6. Experiment with different chunk sizes for your specific use case
7. Test chunk overlap values to find optimal context preservation
8. Evaluate retrieval quality with different chunking strategies
9. Implement monitoring to track chunk-related metrics in production
10. Stay updated with new chunking techniques and tools as they emerge

Additional Resources

For more information on text splitting and LangChain:

- • LangChain Documentation: Text Splitters
- • Visualization Tool: ChunkViz.com
- • OpenAI Embeddings: Embeddings Guide