Agentic AI LAB

Submitted By:

Prodip Barman(2023800192)

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING SHARDA SCHOOL OF ENGINEERING & TECHNOLOGY

SHARDA UNIVERSITY, GREATER NOIDA

5 Levels Of Text Splitting: Complete Code Explanation

Introduction

Text splitting (also called chunking) is one of the most effective strategies to improve the performance of language model applications. When working with large documents that exceed model context windows, splitting data into smaller, manageable pieces becomes essential. This guide covers all five levels of text splitting techniques from basic to advanced approaches.

Main Problem Addressed:

- Text too long for ChatGPT or language models
- Poor long-term memory in applications
- Need for better context management in RAG systems

Key Concept: Splitting large data into smaller pieces while maintaining semantic meaning and context.

Level 1: Character Splitting

What It Does

Character splitting divides text into fixed-size chunks regardless of content or structure. It's the simplest approach but not recommended for production applications.

Key Concepts

Chunk Size: The number of characters in each chunk (e.g., 35, 100, 1000)

Chunk Overlap: The amount of overlap between sequential chunks to avoid cutting context in half. Creates duplicate data across chunks.

Pros and Cons

Pros:

- Simple and easy to implement
- Predictable chunk sizes

Cons:

- Very rigid - ignores document structure
- Can split sentences or important concepts
- Poor for semantic understanding

Code Walkthrough

Step 1: Define your text

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

Step 2: Create a list to hold chunks

chunks = []

Step 3: Set chunk size

chunk_size = 35 # Characters

Step 4: Iterate through text with sliding window

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

Result:

['This is the text I would like to ch',

'unk up. It is the example text for ',

'this exercise']

How It Works:

1. `range(0, len(text), chunk_size)` creates starting positions every 35 characters
2. `text[i:i + chunk_size]` extracts 35 characters starting at position i
3. Each chunk is appended to the list

Using LangChain's CharacterTextSplitter

from langchain_text_splitters import CharacterTextSplitter

Initialize the splitter

text_splitter = CharacterTextSplitter(
chunk_size=35, # Size of each chunk
chunk_overlap=0, # No overlap between chunks

separator='', # Split on any character
strip_whitespace=False # Keep whitespace
)

Create documents with metadata

documents = text_splitter.create_documents([text])

Note: LangChain removes trailing whitespace by default. Use `strip_whitespace=False` to preserve it.

Chunk Overlap Example

With `chunk_overlap=4`, chunks share their last 4 characters with the next chunk's first 4 characters:

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

Result with overlap:

['This is the text I would like to ch',

'o chunk up. It is the example text', <- 'o ch' overlaps with previous

'ext for this exercise']

Level 2: Recursive Character Text Splitting

What It Does

Recursively splits text based on a series of separators, respecting document structure better than Level 1.

The Algorithm

The splitter tries separators in this order (for English text):

1. `\\n\\n` - Double newlines (paragraph breaks) - Most important

2. `\\n` - Single newlines
3. `\\\" \\\"` - Spaces
4. `\\"\\\"` - Individual characters

Key Concept: If a chunk is too large, it moves to the next separator and tries again until the chunk fits the size requirement.

Code Example

from langchain_text_splitters import RecursiveCharacterTextSplitter

Initialize with larger chunk size

text_splitter = RecursiveCharacterTextSplitter(
chunk_size=65, # Target chunk size
chunk_overlap=0 # No overlap
)

Sample text with paragraphs

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. If your product is only half as good as your competitor's, you don't get half as many customers. You get no customers, and you go out of business.

It's obviously true that the returns for performance are superlinear in business."""

documents = text_splitter.create_documents([text])

How It Works Step-by-Step

1. **First Pass:** Try splitting on `\\n\\n` (paragraph breaks)
2. **Check Size:** If any chunk > chunk_size, recursively split that chunk
3. **Second Pass:** Try splitting on `\\n`
4. **Check Size Again:** If still too large, try spaces
5. **Last Resort:** Split on characters

With Metadata

docs = text_splitter.create_documents([text])

Add metadata to each chunk

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 logical chunks
- Better for semantic understanding
- Chunks more likely to end with periods

Level 3: Document-Specific Splitting

What It Does

Uses different separators optimized for specific file types (Markdown, Python, JavaScript, etc.).

Three Main Types

3A. Markdown Splitting

Separators (in order):

1. `\\n#{1,6}` - Headers (H1-H6)
2. ``` ``` ``` - Code blocks
3. `\\n\\***\\n` - Horizontal lines
4. `\\n\\n` - Double newlines
5. `\\n` - Single newlines
6. `\\\" \\\"` - Spaces
7. `\\\"\\\"` - Characters

from langchain_text_splitters import MarkdownTextSplitter

splitter = MarkdownTextSplitter(
chunk_size=40,
chunk_overlap=0
)

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
"""

chunks = splitter.create_documents([markdown_text])

Result: Chunks preserve markdown structure, grouping by headers.

3B. Python Code Splitting

Separators:

1. `\\nclass` - Class definitions
2. `\\ndef` - Function definitions
3. `\\n\\tdef` - Indented functions
4. `\\n\\n` - Double newlines
5. `\\n` - Single newlines
6. `\\\" \\\"` - Spaces
7. `\\\"\\\"` - Characters

from langchain_text_splitters import PythonCodeTextSplitter

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

python_code = """
class Person:
def **init**(self, name, age):
self.name = name
self.age = age

p1 = Person("John", 36)

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

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

Result: Keeps class and function definitions together.

3C. JavaScript Code Splitting

Key Separators:

- `\\nfunction` - Function declarations
- `\\nconst`, `\\nlet`, `\\nvar` - Variable declarations
- `\\nclass` - Class definitions
- `\\nif`, `\\nfor`, `\\nwhile`, `\\nswitch` - Control flow
- `\\n\\n` - Double newlines

```
from langchain_text_splitters import RecursiveCharacterTextSplitter, Language

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

javascript_code = """
let x = myFunction(4, 3);

function myFunction(a, b) {
return a * b;
}
"""

chunks = js_splitter.create_documents([javascript_code])
```

Level 4: Semantic Chunking

What It Does

Uses embeddings to group semantically similar content together, regardless of format or structure.

Key Concept

Instead of splitting based on characters or structure, semantic chunking:

1. Converts sentences to embeddings (vector representations)
2. Calculates similarity between consecutive sentences

3. Creates chunks where similarity drops below a threshold

How It Works

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

# Initialize semantic chunker
semantic_splitter = SemanticChunker(
embeddings=OpenAIEmbeddings(),
breakpoint_threshold_type="percentile",
breakpoint_threshold_amount=70 # Use 70th percentile as break threshold
)

# Split text based on semantic meaning
chunks = semantic_splitter.split_text(text)
```

Similarity Calculation Example

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

client = OpenAI()

sentences = [
"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.",
"The Model Y became the best-selling vehicle globally.",
"Customer satisfaction ratings reached an all-time high of 96%."
]

# Step 1: Get embeddings for each sentence
embeddings = []
for sentence in sentences:
response = client.embeddings.create(
model="text-embedding-3-small",
input=sentence
```

)
embeddings.append(response.data[0].embedding)

embeddings = np.array(embeddings)

Step 2: Calculate similarity between consecutive sentences

similarity_matrix = cosine_similarity(embeddings)

print(f"Similarity S1 vs S2: {similarity_matrix[0][1]:.3f}") # 0.403
print(f"Similarity S2 vs S3: {similarity_matrix[1][2]:.3f}") # 0.443
print(f"Similarity S3 vs S4: {similarity_matrix[2][3]:.3f}") # 0.414
print(f"Similarity S4 vs S5: {similarity_matrix[3][4]:.3f}") # 0.261

Advantages

- Chunks by **meaning**, not just structure
- Better for RAG and retrieval systems
- Maintains semantic coherence
- Works across different document types

Disadvantages

- Requires embedding API calls (slower, costs money)
- More complex to implement
- Higher latency

Level 5: Agentic Chunking

What It Does

Uses an AI agent (like GPT) to intelligently decide where to split text based on semantic boundaries.

The Approach

from langchain_openai import ChatOpenAI

Initialize LLM

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

Create prompt for chunking

prompt = """
You are a text chunking expert. Split this text into logical chunks.

Rules:

- Each chunk should be around 200 characters or less
- Split at natural topic boundaries
- Keep related information together
- Put "<<>>" between chunks

Text:
{text}

Return the text with <<>> markers where you want to split:
"""

Get AI's chunking decision

response = llm.invoke(prompt.format(text=text))
marked_text = response.content

Split on the markers

chunks = marked_text.split("<<>>")

Clean chunks

clean_chunks = [chunk.strip() for chunk in chunks if chunk.strip()]

Example Output

Input 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..."

AI's Decision with <<>> markers:
"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..."

Result:

- Chunk 1: Q3 Results overview
- Chunk 2: Revenue drivers
- Chunk 3: Model Y Performance
- ...

Advantages

- Most intelligent chunking strategy
- Understands context and meaning
- Flexible for any content type
- Can apply complex business logic

Disadvantages

- Slowest approach (LLM calls)
- Most expensive (API costs)
- Experimental/not production-ready yet
- Requires good prompt engineering

Summary Comparison

Level	Method	Speed	Quality	Best Use Case
1	Character	Very Fast	Poor	Baseline only
2	Recursive Character	Fast	Good	General text documents
3	Document-Specific	Fast	Very Good	Code and Markdown
4	Semantic	Slow	Excellent	RAG and retrieval systems
5	Agentic	Very Slow	Excellent	Complex documents

Table 1: Text Splitting Methods Comparison

Implementation Guide

Step 1: Assess Your Data

Determine data type

if is_code(data):
use_language_splitter()
elif is_markdown(data):
use_markdown_splitter()
else:
use_recursive_splitter()

Step 2: Choose Appropriate Level

Data Type	Recommended Level
General prose/articles	Level 2: Recursive
Python code	Level 3: Python Splitter
JavaScript code	Level 3: JS Splitter
Markdown docs	Level 3: Markdown
RAG queries	Level 4: Semantic
Complex documents	Level 5: Agentic

Table 2: Data Type to Splitting Level Mapping

Step 3: Configure Parameters

Key configuration

chunk_size = 1000 # Adjust based on model context
chunk_overlap = 100 # Usually 10-20% of chunk_size
separators = [...] # Choose based on document type
strip_whitespace = True # Clean up whitespace

Step 4: Process Documents

chunks = splitter.create_documents(
[text], # List of texts to split
metadatas=[metadata] # Optional: add metadata
)

Step 5: Evaluate Results

Check chunk quality

for i, chunk in enumerate(chunks):
print(f"Chunk {i} ({len(chunk.page_content)} chars)")
print(chunk.page_content[:100]) # Preview first 100 chars

Important Concepts Summary

Chunk Overlap

Prevents cutting important context in half. If chunk_overlap=100 and chunk_size=1000:

- Chunk 1: Characters 0-1000
- Chunk 2: Characters 900-1900 (100 chars overlap with Chunk 1)

When to use:

- Use 10-20% of chunk_size
- Helps preserve context at boundaries
- Creates duplicate data (acceptable trade-off)

Separator Strategy

The order of separators matters. Try separators in order of importance:

1. Most important first (e.g., paragraph breaks)
2. Less important next (e.g., sentences)
3. Least important last (e.g., words, characters)

Metadata

Always add metadata for tracking:
doc.metadata = {
"source": "filename.txt",
"chunk_index": 0,
"page": 1
}

Evaluation Frameworks

Test your chunking strategy with evaluation frameworks:

- **LangChain Evals**
- **Llama Index Evals**
- **RAGAS Evals**

Remember: Chunking strategy directly affects RAG retrieval quality. Always test with your actual use case.

Best Practices

1. **Start with Level 2** (Recursive) for most use cases
2. **Use Level 3** if your data is code or structured (Markdown, JSON)
3. **Use Level 4** (Semantic) for RAG systems where retrieval quality is critical
4. **Use Level 5** (Agentic) only for complex, expensive operations
5. **Always evaluate** chunking strategy on your specific task
6. **Iterate** on chunk size and overlap based on model performance
7. **Monitor** chunk quality in production
8. **Consider costs** - API calls for semantic/agentic chunking add up

---

Quick Reference Code

Level 2: Most common use case

```
from langchain_text_splitters import RecursiveCharacterTextSplitter

splitter = RecursiveCharacterTextSplitter(
chunk_size=1000,
chunk_overlap=100
)
chunks = splitter.create_documents([text])
```

Level 3: For Python code

```
from langchain_text_splitters import PythonCodeTextSplitter

python_splitter = PythonCodeTextSplitter(chunk_size=1000)
chunks = python_splitter.create_documents([code])
```

Level 4: For RAG systems

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

semantic_splitter = SemanticChunker(embeddings=OpenAIEmbeddings())
chunks = semantic_splitter.split_text(text)
```

This comprehensive guide covers all five levels of text splitting with complete code examples and explanations. Use this knowledge to optimize your LLM applications and RAG systems!