Practical Lab File for
Agentic AI

Sharda School of Engineering and Technology

Name- Kunwar Utkarsh Kant Mishra
Sys ID- 2023438539
Semester/Year- 6th sem, 3rd Year

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

SHARDA UNIVERSITY
Beyond Boundaries

5 Levels of Text Splitting

for NLP & LLM Pipelines

Abstract

This document explores five hierarchical strategies for text splitting, a critical preprocessing step in Natural Language Processing (NLP) and Large Language Model (LLM) pipelines. Proper text chunking significantly improves embedding quality, retrieval accuracy, and context preservation in downstream tasks. We progress from simple character-based splitting to advanced agentic methods, providing implementation details and best practices for each approach.

Contents

1 Project Overview

Text splitting, also known as chunking, is a fundamental preprocessing technique that divides large text documents into smaller, manageable pieces. This process is essential for:

Token Limit Management: Working within context window constraints of LLMs

Improved Retrieval: Enhancing semantic search and retrieval accuracy

Better Embeddings: Creating more focused vector representations

Context Preservation: Maintaining semantic coherence within chunks

1.1 Why Text Splitting Matters

Have you ever tried to input a long document into ChatGPT only to receive a "text too long" error? Or struggled to give your application e ective long-term memory? Text splitting addresses these challenges by intelligently dividing content while preserving context and meaning.

2 Architecture & Processing Flow

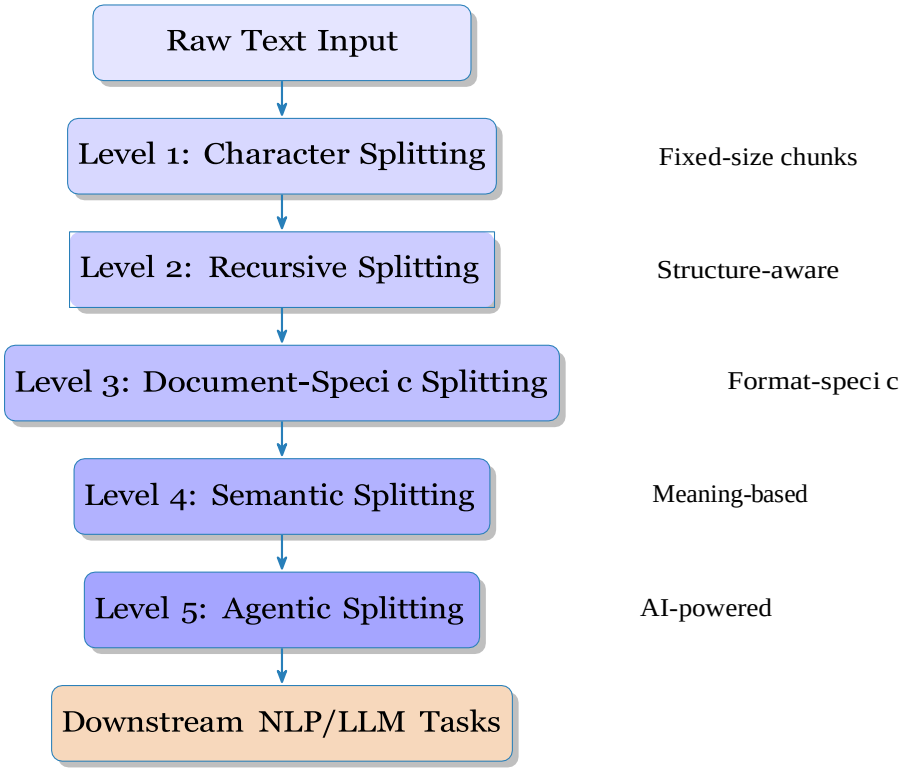

Figure 1: Text Splitting Pipeline Architecture

3 The Five Levels

3.1 Overview

1. **Level 1: Character Splitting** Simple static character chunks

2. **Level 2: Recursive Character Splitting** Recursive chunking based on separators
3. **Level 3: Document-Speci c Splitting** Format-aware chunking (PDF, Python, Markdown)
4. **Level 4: Semantic Splitting** Embedding-based chunking
5. **Level 5: Agentic Splitting** LLM-powered intelligent chunking

3.2 Installation Requirements

This tutorial uses code from LangChain. Install the required packages:

Code Cell 0

```
pip install langchain langchain-text-splitters
pip install langchain_experimental langchain_openai
```

4 Level 1: Character Splitting

4.1 Introduction

Character splitting is the most basic form of text division. It divides text into xed-size chunks of N characters, regardless of content structure or semantic boundaries.

Advantages & Disadvantages

Pros:

- Simple to implement
- Predictable chunk sizes
- Fast processing

Cons:

- Ignores document structure
- May split words or sentences awkwardly
- Poor context preservation

4.2 Key Concepts

Key Concept

Chunk Size
The number of characters per chunk (e.g., 50, 100, 1000)

Chunk Overlap
The number of characters that overlap between consecutive chunks to preserve context

4.3 Implementation

4.3.1 Manual Implementation

Code Cell 1

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

Code Cell 2

```
# Create a list to hold chunks
chunks = []
chunk_size = 35  # Characters

# Iterate through text with step size equal to chunk_size
for i in range(0, len(text), chunk_size):
    chunk = text[i:i + chunk_size]
    chunks.append(chunk)

print(chunks)
# Output: ['This is the text I would like to ',
#          'chunk up. It is the example text',
#          ' for this exercise']
```

4.3.2 Using LangChain

In production systems, we work with `Document` objects that contain both text and metadata for easier manipulation and ltering.

Code Cell 3

```
from langchain_text_splitters import CharacterTextSplitter

# Initialize the splitter
text_splitter = CharacterTextSplitter(
    chunk_size=35,
    chunk_overlap=0,
    separator='',
    strip_whitespace=False
)

# Create documents
text = "This is the text I would like to chunk up. It is the
    example text for this exercise"
documents = text_splitter.create_documents([text])

for doc in documents:
    print(f"Content: {doc.page_content}")
    print(f"Metadata: {doc.metadata}\n")
```

4.4 Adding Chunk Overlap

Chunk overlap helps preserve context by ensuring that adjacent chunks share some content.

Code Cell 4

```
# Configure splitter with overlap
text_splitter = CharacterTextSplitter(
    chunk_size=35,
    chunk_overlap=4,
    separator=''
)

documents = text_splitter.create_documents([text])

# Notice how chunks now overlap:
# Chunk 1: "This is the text I would like to "
# Chunk 2: "o chunk up. It is the example text"
#           ^^^^  <- overlaps with end of Chunk 1
```

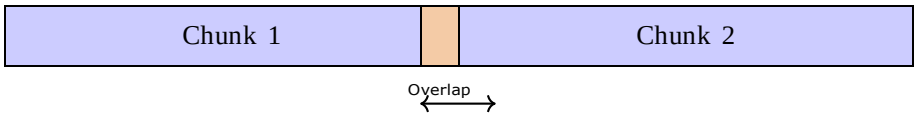

Figure 2: Visualization of chunk overlap

5 Level 2: Recursive Character Splitting

5.1 Introduction

The recursive character text splitter addresses the limitations of Level 1 by respecting document structure. It uses a hierarchy of separators to intelligently split text at natural boundaries.

5.2 Separator Hierarchy

The default separators (in order of priority):

1. "\n\n" Paragraph breaks (double newlines)
2. "\n" Single newlines
3. " " Spaces
4. "" Individual characters

5.3 Implementation

Code Cell 5

```
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."

It's obviously true that the returns for performance are
    superlinear
in business. Some think this is a flaw of capitalism.
"""

# Initialize the splitter
text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=65,
    chunk_overlap=0
)

documents = text_splitter.create_documents([text])

for i, doc in enumerate(documents):
    print(f"Chunk {i+1}: {doc.page_content}\n")
```

5.4 Adding Metadata

Code Cell 6

```
# Create documents with metadata
docs = text_splitter.create_documents([text])

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

# Access metadata
for doc in docs:
    print(f"Chunk {doc.metadata['chunk_no']}: {doc.metadata}")
```

5.5 Optimized Chunking

The recursive splitter includes intelligent "snapping" to nearest separators:

Code Cell 7

```
# Larger chunk size for paragraph-level splits
text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=450,
    chunk_overlap=0
)

documents = text_splitter.create_documents([text])

# With chunk_size=450 (or even 469), the splitter creates
# perfect paragraph breaks due to separator snapping
```

6 Level 3: Document-Speci c Splitting

6.1 Introduction

Di erent document types require specialized splitting strategies. This level provides format-aware splitters for Markdown, Python, JavaScript, and other formats.

6.2 Markdown Splitting

6.2.1 Markdown Separator Hierarchy

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

6.2.2 Implementation

Code Cell 8

```
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
)

documents = splitter.create_documents([markdown_text])

# Chunks will align with markdown sections
for doc in documents:
    print(f"Chunk: {doc.page_content}\n")
```

6.3 Python Code Splitting

6.3.1 Python Separator Hierarchy

1. `\nclass` Class de nitions
2. `\ndef` Function de nitions
3. `\n\tdef` Indented functions (methods)
4. `\n\n` Double newlines
5. `\n` Single newlines
6. `" "` Spaces
7. `""` Characters

6.3.2 Implementation

Code Cell 9

```python
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
)

documents = python_splitter.create_documents([python_text])

# Chunks preserve class and function boundaries
for i, doc in enumerate(documents):
    print(f"Chunk {i+1}:\n{doc.page_content}\n")
```

6.4 JavaScript Code Splitting

6.4.1 JavaScript Separator Hierarchy

Separator	Purpose
\nfunction	Function declarations
\nconst	Constant variable declarations
\nlet	Block-scoped variable declarations
\nvar	Variable declarations
\nclass	Class de nitions
\nif	Conditional statements
\nfor	For loops
\nwhile	While loops
\nswitch	Switch statements
\ncase	Case clauses in switch
\ndefault	Default clause in switch
\n\n	Double newlines
\n	Single newlines
" "	Spaces
""	Characters

Table 1: JavaScript separator hierarchy

6.4.2 Implementation

Code Cell 10

```
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
)

documents = js_splitter.create_documents([javascript_text])

for doc in documents:
    print(f"Chunk: {doc.page_content}\n")
```

7 Level 4: Semantic Splitting

7.1 Introduction

Semantic chunking uses embeddings to group text based on meaning rather than structure. This approach identi es natural topic boundaries by analyzing semantic similarity between sentences.

7.2 How It Works

1. Split text into sentences
2. Generate embeddings for each sentence
3. Calculate similarity between consecutive sentences
4. Create chunk boundaries where similarity drops below a threshold

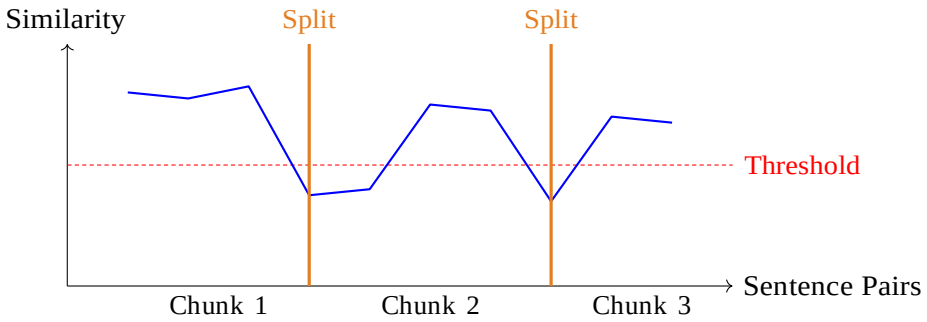

Figure 3: Semantic chunking based on similarity threshold

7.3 Implementation

Code Cell 11

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

# Set up API key
os.environ['OPENAI_API_KEY'] = 'your-api-key-here'

# Sample text with distinct topics
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,
with 350,000 units sold.
Customer satisfaction ratings reached an all-time high of 96%.
Model Y now represents 60% of Tesla's total vehicle sales.

Production Challenges
Supply chain issues caused a 12% increase in production costs.
Tesla is working to diversify its supplier base.
New manufacturing techniques are being implemented to reduce
    costs."""

# Initialize semantic chunker
semantic_splitter = SemanticChunker(
    embeddings=OpenAIEmbeddings(),
    breakpoint_threshold_type="percentile",
    breakpoint_threshold_amount=70
)

# Split text semantically
chunks = semantic_splitter.split_text(tesla_text)

print("SEMANTIC CHUNKING RESULTS:")
print("=" * 50)
for i, chunk in enumerate(chunks, 1):
    print(f"Chunk {i}: ({len(chunk)} chars)")
    print(f'"{chunk}"\n')
```

7.4 Understanding Embeddings and Similarity

Code Cell 12

```python
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%."
]

# Generate embeddings
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 between consecutive sentences
similarity_matrix = cosine_similarity(embeddings)

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

# High similarity (>0.8) = keep together
# Low similarity (<0.6) = create boundary
```

8 Level 5: Agentic Splitting

8.1 Introduction

Agentic splitting leverages LLMs to intelligently determine chunk boundaries. The AI analyzes content, identi es logical topics, and creates chunks that preserve semantic coherence.

8.2 Advantages

Context-aware: Understands content meaning and structure

Flexible: Adapts to di erent document types automatically

Intent-driven: Can optimize for speci c use cases

Natural boundaries: Creates semantically coherent chunks

8.3 Implementation

Code Cell 13

```python
from langchain_openai import ChatOpenAI

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

# Text to chunk
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,
with 350,000 units sold.
Customer satisfaction ratings reached an all-time high of 96%.
Model Y now represents 60% of Tesla's total vehicle sales.

Production Challenges
Supply chain issues caused a 12% increase in production costs.
Tesla is working to diversify its supplier base.
New manufacturing techniques are being implemented to reduce
    costs."""

# Create the prompt
prompt = f"""
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 "<<<SPLIT>>>" between chunks

Text:
{tesla_text}

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

# Get AI response
print("Asking AI to chunk the text...")
response = llm.invoke(prompt)
marked_text = response.content
print(marked_text)

# Split the text at the markers
chunks = marked_text.split("<<<SPLIT>>>")

# Clean up the chunks
clean_chunks = [chunk.strip() for chunk in chunks if chunk.strip
    ()]

# Show results
print("\nAGENTIC CHUNKING RESULTS:")
print("=" * 50)
for i, chunk in enumerate(clean_chunks, 1):
    print(f"Chunk {i}: ({len(chunk)} chars)")
    print(f'"{chunk}"\n')
```

8.4 Advanced Agentic Chunking

Code Cell 14

```
# More sophisticated prompt for domain-specific chunking
advanced_prompt = f"""
You are an expert at chunking technical documents for RAG systems.

Context: This text will be used in a question-answering system
about Tesla's financial performance.

Requirements:
1. Each chunk must be self-contained and understandable
2. Target chunk size: 150-250 characters
3. Preserve numerical data with context
4. Keep section headers with their content
5. Mark splits with <<<SPLIT>>>

Optimization goals:
- Maximize retrieval accuracy
- Preserve causal relationships
- Maintain temporal context

Text:
{tesla_text}

Provide the chunked text:
"""

# Process with advanced prompt
response = llm.invoke(advanced_prompt)
# ... (same processing as before)
```

9 Evaluation & Best Practices

9.1 Importance of Evaluation

Chunking strategy signi cantly impacts downstream performance. Always evaluate your chunks using retrieval metrics:

Retrieval Accuracy: Are the right chunks retrieved?

Context Completeness: Do chunks contain su cient context?

Answer Quality: Does chunking improve nal outputs?

9.2 Evaluation Frameworks

Framework	Description
LangChain Evals	Comprehensive evaluation tools for RAG systems https://python.langchain.com/docs/guides/evaluation/
Llama Index Evals	Specialized retrieval evaluation metrics https://docs.llamaindex.ai/en/stable/module_guides/evaluating/
RAGAS	Advanced RAG assessment framework https://github.com/explodinggradients/ragas

Table 2: Evaluation frameworks for text chunking

9.3 Choosing the Right Level

Level	Best For	Avoid For
Character	Simple prototypes, uniform text	Production systems, varied content
Recursive	General-purpose text, mixed content	Code, structured documents
Document-Specific	Code files, Markdown, specific formats	General prose
Semantic	Topic-based retrieval, coherent sections	Performance-critical systems
Agentic	Complex documents, custom requirements	High-volume processing

Table 3: Chunking strategy selection guide

9.4 Best Practices

1. **Start Simple:** Begin with recursive splitting before advanced methods
2. **Measure Impact:** Always benchmark against retrieval performance
3. **Consider Cost:** Semantic and agentic methods require API calls
4. **Use Overlap:** 10-20% overlap helps preserve context
5. **Test Multiple Sizes:** Optimal chunk size varies by use case (typically 200-1000 chars)
6. **Add Metadata:** Include source, chunk number, and context in metadata
7. **Preserve Structure:** Maintain headings, sections, and formatting
8. **Iterate:** Chunking is not one-size- ts-all; experiment and re ne

10 Conclusion

Text splitting is a critical preprocessing step that directly impacts the performance of NLP and LLM applications. By understanding and applying these ve levels from simple character splitting to sophisticated agentic methods you can signi cantly improve embedding quality, retrieval accuracy, and overall system performance.

10.1 Key Takeaways

Choose the splitting strategy that matches your document type and use case

Always evaluate chunking decisions against downstream task performance

Consider the trade-o s between simplicity, accuracy, and computational cost

Use overlap to preserve context across chunk boundaries

Iterate and re ne based on real-world performance metrics

10.2 Further Resources

LangChain Text Splitters Documentation: https://python.langchain.com/docs/modules/data_connection/document_transformers/

ChunkViz Visualization Tool: https://www.chunkviz.com

OpenAI Embeddings Guide: https://platform.openai.com/docs/guides/embeddings

Appendix: Quick Reference

Chunk Size Guidelines

Use Case	Recommended Size	Overlap
Question Answering	200-500 chars	50-100 chars
Summarization	500-1000 chars	100-200 chars
Code Analysis	100-300 chars	20-50 chars
Long-form Retrieval	1000-2000 chars	200-400 chars

Table 4: Chunk size recommendations by use case

Common Pitfalls

1. **Too Small:** Chunks lack su cient context
2. **Too Large:** Exceeds model context windows, poor retrieval precision
3. **No Overlap:** Important context lost at boundaries
4. **Excessive Overlap:** Increased storage and processing costs
5. **Ignoring Structure:** Character splitting on formatted documents
6. **No Evaluation:** Optimizing without measuring impact