10

TYPES OF

CHUNKING TECHNIQUES FOR TEXT PROCESSING WITH PRACTICAL IMPLEMENTATION

In [ ]:

```
pip install PyPDF2
```

In [ ]:

```
from PyPDF2 import PdfReader

def extract_text_from_pdf(pdf_path):
    reader = PdfReader(pdf_path)
    text = ""
    for page in reader.pages:
        text += page.extract_text()
    return text

pdf_path = "C:\\Users\\nares\\OneDrive\\Desktop\\ml.pdf"
text = extract_text_from_pdf(pdf_path)
print(text)
```

1. Fixed Chunking

Splits text into equal-sized chunks (by word count).

In [ ]:

```
def fixed_chunking(text, chunk_size=100):
    words = text.split()
    chunks = [' '.join(words[i:i + chunk_size]) for i in range(0, len(words)
    return chunks

chunks = fixed_chunking(text, chunk_size=10)
for i, chunk in enumerate(chunks):
    print(f"Chunk {i+1}: {chunk}")
```

2. Overlapping Chunking

Splits text into chunks with an overlap for smoother transitions.

In [ ]:

```
def overlapping_chunking(text, chunk_size=100, overlap=50):
    words = text.split()
    chunks = [' '.join(words[i:i + chunk_size]) for i in range(0, len(words)
    return chunks

chunks = overlapping_chunking(text, chunk_size=20, overlap=5)
for i, chunk in enumerate(chunks):
    print(f"Chunk {i+1}: {chunk}")
```

3. Semantic Chunking

Uses spaCy to split text into meaningful sentences.

In [ ]:

```
pip install spacy
```

In [ ]:

```python
import spacy.cli
spacy.cli.download("en_core_web_sm")

def semantic_chunking(text):
    nlp = spacy.load("en_core_web_sm")
    doc = nlp(text)
    chunks = [sent.text for sent in doc.sents]
    return chunks

chunks = semantic_chunking(text)
for i, chunk in enumerate(chunks):
    print(f"Chunk {i+1}: {chunk}")
```

4. Recursive Character Chunking

Splits text recursively based on character count, prioritizing word boundaries.

In [ ]:

```python
def recursive_chunk(text, max_size):
    if len(text) <= max_size:
        return [text]
    split_point = text.rfind(" ", 0, max_size)
    if split_point == -1:  # No space found, force split
        split_point = max_size
    chunk = text[:split_point]
    remaining_text = text[split_point:].strip()  # Remove leading spaces
    return [chunk] + recursive_chunk(remaining_text, max_size)

chunks = recursive_chunk(text, 100)
for i, chunk in enumerate(chunks):
    print(f"Chunk {i+1}: {chunk}")
```

5. Agentic Chunking

Uses an AI agent (via Groq API) to split text meaningfully for a given task.

In [ ]:

```
pip install groq
```

In [ ]:

```python
from groq import Groq

def agentic_chunking(text, task="summarize"):
    client = Groq(api_key="your_api_key")
    prompt = f"Split the following text into meaningful chunks for the task
    response = client.chat.completions.create(
        model="llama3-70b-8192",
        messages=[{"role": "user", "content": prompt}]
    )
    chunks = response.choices[0].message.content.split('\n')
    return chunks

chunks = agentic_chunking(text, task="summarize")
```

```
for i, chunk in enumerate(chunks):
    print(f"Chunk {i+1}: {chunk}")
```

6. Advanced Semantic Chunking

Uses SentenceTransformer and KMeans clustering to group semantically similar sentences.

In [ ]:

```
!pip install sentence-transformers scikit-learn numpy
```

In [ ]:

```
from sentence_transformers import SentenceTransformer
from sklearn.cluster import KMeans
import numpy as np

def advanced_semantic_chunking(text, num_chunks=15):
    model = SentenceTransformer('all-MiniLM-L6-v2')
    sentences = text.split('. ')
    embeddings = model.encode(sentences)
    kmeans = KMeans(n_clusters=num_chunks)
    kmeans.fit(embeddings)
    clusters = kmeans.labels_
    chunks = [[] for _ in range(num_chunks)]
    for i, cluster in enumerate(clusters):
        chunks[cluster].append(sentences[i])
    return [' '.join(chunk) for chunk in chunks]

chunks = advanced_semantic_chunking(text, num_chunks=5)
for i, chunk in enumerate(chunks):
    print(f"Chunk {i+1}: {chunk}")
```

7. Context Enriched Chunking

Combines surrounding sentences to add context to each chunk.

In [ ]:

```
def context_enriched_chunking(text, window_size=2):
    sentences = text.split('. ')
    chunks = []
    for i in range(len(sentences)):
        start = max(0, i - window_size)
        end = min(len(sentences), i + window_size + 1)
        chunk = '. '.join(sentences[start:end])
        chunks.append(chunk)
    return chunks

chunks = context_enriched_chunking(text, window_size=1)
for i, chunk in enumerate(chunks):
    print(f"Chunk {i+1}: {chunk}")
```

8. Paragraph Chunking

Splits text based on paragraphs (using double line breaks).

In [ ]:

```
def paragraph_chunking(text):
    paragraphs = text.split('\n\n')
    return paragraphs

chunks = paragraph_chunking(text)
for i, chunk in enumerate(chunks):
    print(f"Chunk {i+1}: {chunk}")
```

9. Recursive Sentence Chunking

Recursively splits text into chunks based on a set number of sentences.

In [ ]:

```
def recursive_sentence_chunking(text, max_sentences=3):
    sentences = text.split('. ')
    if len(sentences) <= max_sentences:
        return ['. '.join(sentences)]
    chunk = '. '.join(sentences[:max_sentences])
    remaining = '. '.join(sentences[max_sentences:])
    return [chunk] + recursive_sentence_chunking(remaining, max_sentences)

chunks = recursive_sentence_chunking(text, max_sentences=3)
for i, chunk in enumerate(chunks):
    print(f"Chunk {i+1}: {chunk}")
```

10. Token Based Chunking

Splits text into chunks based on a specific token count.

In [ ]:

```
def token_based_chunking(text, token_limit=50):
    tokens = text.split()  # Basic tokenization by whitespace
    chunks = [' '.join(tokens[i:i+token_limit]) for i in range(0, len(tokens
    return chunks

chunks = token_based_chunking(text, token_limit=50)
for i, chunk in enumerate(chunks):
    print(f"Chunk {i+1}: {chunk}")
```