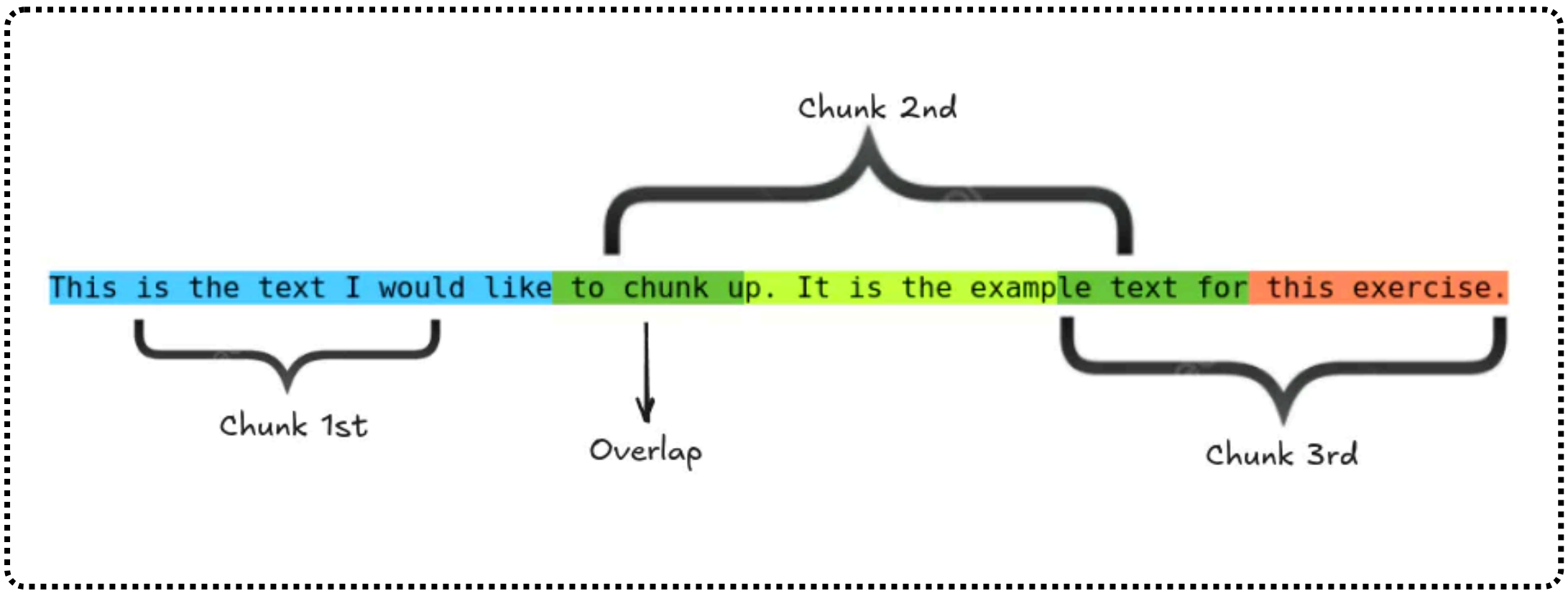
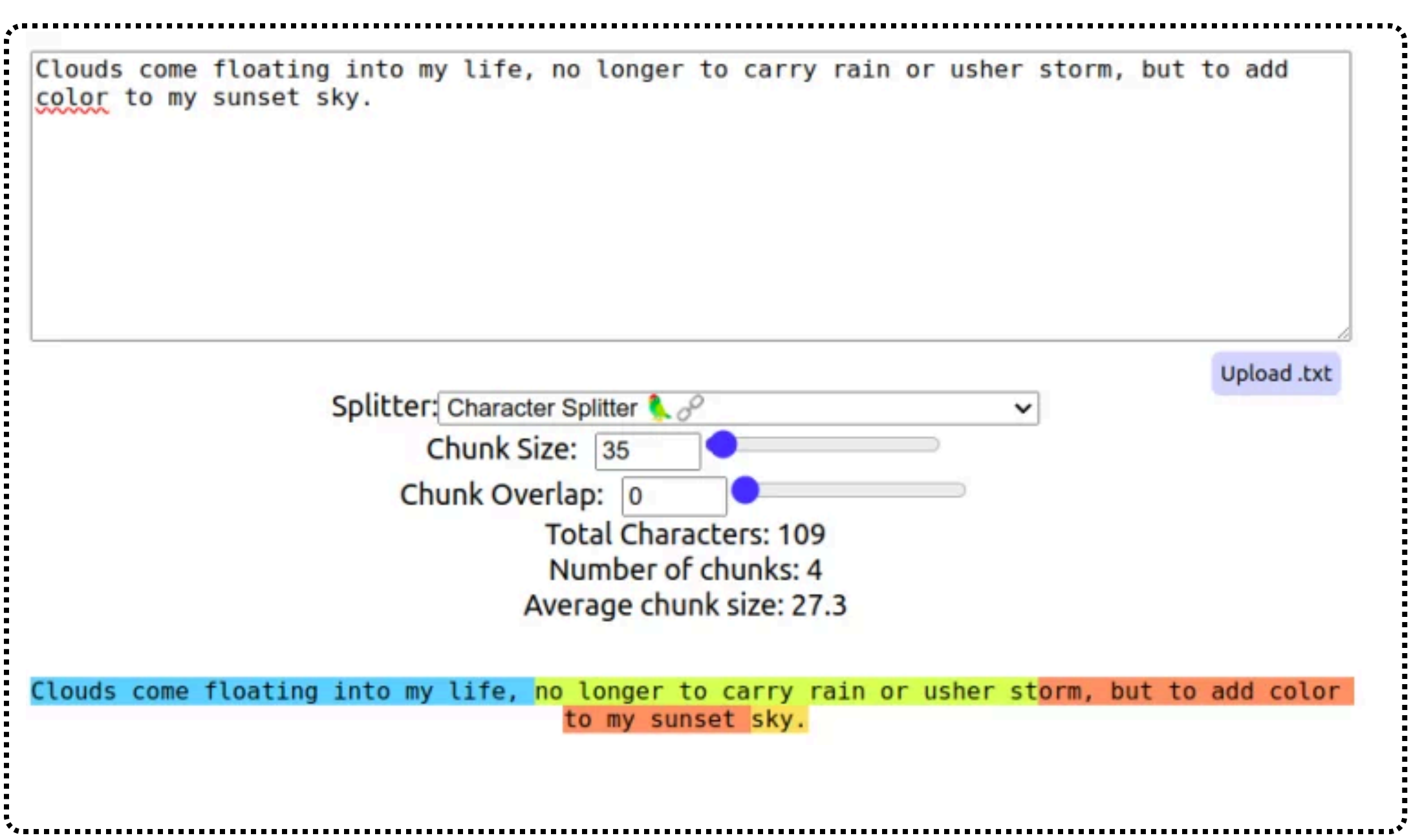
Analytics Vidhya

Mastering RAG

Different Types of chunking in RAG





Introduction

In Retrieval-Augmented Generation (RAG), chunking is the process of dividing large documents into smaller segments to optimize retrieval and improve response generation. Choosing the right chunking strategy significantly impacts retrieval efficiency, contextual accuracy, and overall system performance.

This document explores different types of chunking techniques used in RAG and provides their corresponding Python implementations.

1. Fixed-Length Chunking

Overview

Fixed-length chunking splits text into segments of a predefined number of words or tokens, ensuring uniform chunk sizes.

Pros

- Simple and easy to implement.
- Works well with structured text like tables and logs.

Cons

- Can cut off sentences, leading to incomplete context.
- Lacks adaptability to different document structures.

```python
from transformers import AutoTokenizer


def fixed_length_chunking(text, chunk_size=512):
    tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased")
    tokens = tokenizer.tokenize(text)
    chunks = [tokens[i: i + chunk_size] for i in range(0, len(tokens),
chunk_size)]
    return [tokenizer.convert_tokens_to_string(chunk) for chunk in chunks]
```

2. Sentence-Based Chunking

Overview

This method segments text based on sentence boundaries,
ensuring that each chunk contains semantically meaningful information.

Pros

- Preserves sentence integrity.
- Prevents information loss due to cut-off sentences.

Cons

- Chunk sizes may vary significantly.
- May require additional processing for optimal retrieval.

Analytics Vidhya

Implementation

```
import spacy


def semantic_chunking(text, chunk_size=512):
    nlp = spacy.load("en_core_web_sm")
    doc = nlp(text)
    chunks, current_chunk = [], ""

    for sent in doc.sents:
        if len(current_chunk.split()) +
len(sent.text.split()) > chunk_size:
            chunks.append(current_chunk)
            current_chunk = sent.text
        else:
            current_chunk += " " + sent.text

    if current_chunk:
        chunks.append(current_chunk)
    return chunks
```

3. Overlapping (Sliding Window) Chunking

Overview

Creates overlapping chunks to maintain context continuity across adjacent segments. This helps reduce loss of key information during retrieval.

Pros

- Preserves context better than fixed-length chunking.
- Useful for handling long-form content.

Cons

- Increases storage and computation requirements.
- May introduce redundancy in retrieval.

Analytics Vidhya

Implementation

```
def sliding_window_chunking(text,
chunk_size=512, overlap=256):
    tokenizer =
AutoTokenizer.from_pretrained("bert-base-
uncased")
    tokens = tokenizer.tokenize(text)
    chunks = []

    for i in range(0, len(tokens),
chunk_size - overlap):
        chunk = tokens[i: i + chunk_size]

 chunks.append(tokenizer.convert_tokens_to_s
tring(chunk))

    return chunks
```

Topic-Based Chunking

Overview

Uses natural language understanding (NLU) techniques to group text into topic-based segments.

Pros

- Ensures that each chunk contains a self-contained topic.
- Works well for knowledge-intensive tasks.

Cons

- Requires topic modeling algorithms like LDA or BERT.
- Can be computationally expensive.

```
from sklearn.feature_extraction.text import
TfidfVectorizer
from sklearn.cluster import KMeans


def topic_based_chunking(text, n_clusters=5):
    sentences = text.split(". ")
    vectorizer = TfidfVectorizer(stop_words="english")
    X = vectorizer.fit_transform(sentences)

    kmeans = KMeans(n_clusters=n_clusters,
random_state=42)
    labels = kmeans.fit_predict(X)

    chunks = [[] for _ in range(n_clusters)]
    for i, label in enumerate(labels):
        chunks[label].append(sentences[i])

    return [". ".join(chunk) for chunk in chunks]
```