Day67_NLP_5_Chunking_in_NLP_and_LLMs

August 19, 2025

Chunking in NLP & LLMs

What is Chunking?

- **Chunking** is the process of dividing text into **meaningful segments** called **chunks**.
- Unlike POS tagging (which assigns a tag to each word), **chunking groups words together** to form higher-level units like Noun Phrases, Verb Phrases, and Prepositional Phrases.
- Example:
  - "The black dog" → **Noun Phrase (NP)**
  - "is running" → **Verb Phrase (VP)**
  - "in the park" → **Prepositional Phrase (PP)**

Why is Chunking Important?

- Helps extract meaningful **phrases** instead of individual words.
- Useful in **Information Extraction, Named Entity Recognition, Question Answering, LLM preprocessing**.
- Interview Tip → *"Explain Chunking in NLP and in LLMs"* is a common question.

1 Chunking in NLP using NLTK

We'll use **NLTK** to create and visualize chunks.

[1]:
```
import nltk
from nltk import pos_tag, word_tokenize, RegexpParser

# Download required models only onces
nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')
```

```
[nltk_data] Downloading package punkt to
[nltk_data]     C:\Users\Lenovo\AppData\Roaming\nltk_data…
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package averaged_perceptron_tagger to
```

```
[nltk_data]     C:\Users\Lenovo\AppData\Roaming\nltk_data…
[nltk_data]   Package averaged_perceptron_tagger is already up-to-
[nltk_data]       date!
```

[1]:
```
True
```

[2]:
```python
# Sample text
text = "full stack datascience, generative ai, agenti ai, llm model keep␣
↪increase by different compnay"
```

[3]:
```python
# Tokenize
tokens = word_tokenize(text)
```

[4]:
```python
# POS Tagging
tagged_tokens = pos_tag(tokens)
```

[5]:
```python
print("Tagged Tokens:\n", tagged_tokens)
```

```
Tagged Tokens:
 [('full', 'JJ'), ('stack', 'NN'), ('datascience', 'NN'), (',', ','),
('generative', 'JJ'), ('ai', 'NN'), (',', ','), ('agenti', 'NN'), ('ai', 'NN'),
(',', ','), ('llm', 'JJ'), ('model', 'NN'), ('keep', 'VB'), ('increase', 'NN'),
('by', 'IN'), ('different', 'JJ'), ('compnay', 'NN')]
```

[6]:
```python
# Define chunk grammar
chunk_grammar = r"""
NP: {<DT>?<JJ>*<NN>}          # Noun Phrase
VP: {<VB.*><NP|PP>*}          # Verb Phrase
PP: {<IN><NP>}                # Prepositional Phrase
"""
```

[7]:
```python
# Create a chunk parser
chunk_parser = RegexpParser(chunk_grammar)
```

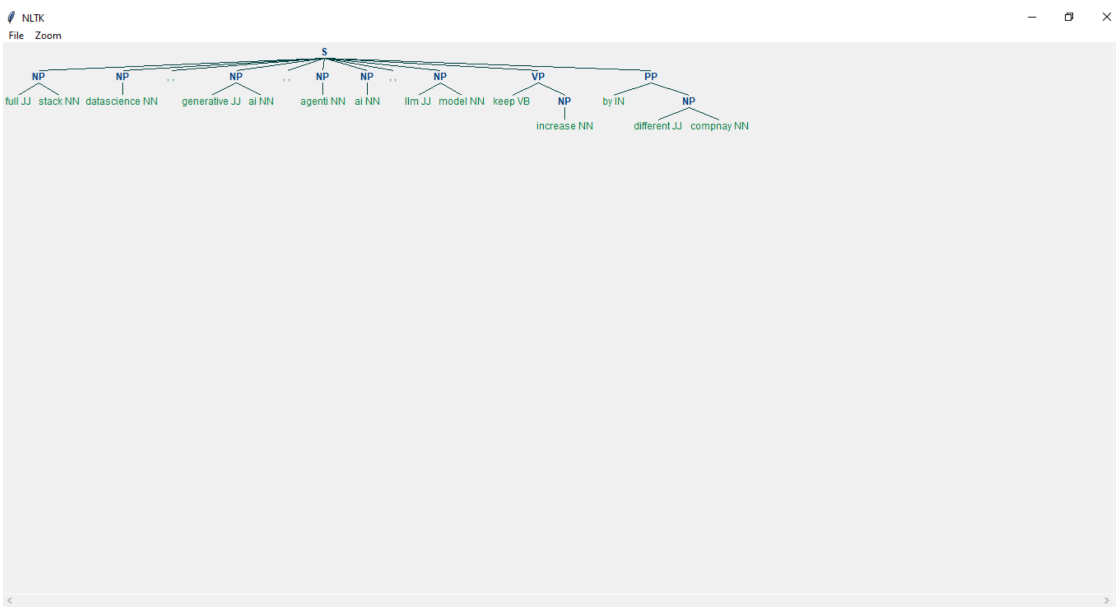
[8]:
```python
# Parse the tagged tokens
chunked = chunk_parser.parse(tagged_tokens)
```

[9]:
```python
# Print and visualize
print("\nChunked Output:\n", chunked)
chunked.draw()
```

```
Chunked Output:
 (S
  (NP full/JJ stack/NN)
  (NP datascience/NN)
  ,/,
  (NP generative/JJ ai/NN)
  ,/,
```

```
(NP agenti/NN)
(NP ai/NN)
,/,
(NP llm/JJ model/NN)
(VP keep/VB (NP increase/NN))
(PP by/IN (NP different/JJ compnay/NN)))
```

2 Chunking Visualization (NLTK Output)



3 Chunking in LLMs

- In **NLP with NLTK**, chunking = grouping tokens into phrases.
- In **LLMs (Large Language Models)**, chunking = splitting **long text into smaller parts (chunks)** so that models like GPT can process them within their **context window**.

Why?

- LLMs (like GPT-2, GPT-3, BERT, T5) have a **fixed token limit**.
- Chunking long documents ensures the model can process text without losing information.

[ ]:
```
## Note on Running LLM Chunking on CPU

- GPT-2 and similar models are large, so running them on CPU may cause **out-of-memory** or **very slow performance**.
- To avoid errors:
  - Use smaller models like **distilgpt2**.
  - Explicitly load the model on CPU.
  - Keep `max_length` small (like 50-100).
```

3.1 Running Larger LLMs on Google Colab (GPU) — Documentation Only

> **Note:** This section is for **documentation** inside your notebook.
> We **haven't executed** these cells here or shown outputs.
> To run them, open **Google Colab**, set **Runtime → GPU**, then **copy-paste each cell** below into separate Colab cells **in order**.

Step 0 — Switch Colab to GPU

Colab menu: `Runtime` → `Change runtime type` → **Hardware accelerator:** `GPU` → **Save**

Step 1 — Verify GPU

```
# Check that Colab gave you a GPU
!nvidia-smi

import torch
print("CUDA available:", torch.cuda.is_available())
if torch.cuda.is_available():
    print("GPU name:", torch.cuda.get_device_name(0))
```

Step 2 — Install libraries (install transformers** first)**

```
# Install/upgrade essential libraries for HF models on GPU
!pip -q install --upgrade transformers accelerate safetensors
```

Step 3 — (Optional) Mount Google Drive

```
from google.colab import drive
drive.mount('/content/drive')  # Skip if you don't need Drive
```

Step 4 — Load Tokenizer & Model on GPU (GPT-2)

```
from transformers import AutoTokenizer, AutoModelForCausalLM
import torch

model_name = "gpt2"            # If you hit memory issues, try: "distilgpt2"

tokenizer = AutoTokenizer.from_pretrained(model_name)
# Some GPT-2 variants do not have a pad token; align it with EOS for safety
if tokenizer.pad_token is None:
    tokenizer.pad_token = tokenizer.eos_token

# Choose dtype: fp16 on GPU saves VRAM, fp32 on CPU
dtype = torch.float16 if torch.cuda.is_available() else torch.float32
device = "cuda" if torch.cuda.is_available() else "cpu"

model = AutoModelForCausalLM.from_pretrained(
    model_name,
    torch_dtype=dtype
).to(device)

print(f"Loaded {model_name} on {device} with dtype {dtype}.")
```

Step 5 — Generate Text (your "indian cricket" example)

```
prompt = "indian cricket"
inputs = tokenizer(prompt, return_tensors='pt').to(model.device)

with torch.inference_mode():
    output_ids = model.generate(
        **inputs,
        max_length=50,                # keep modest for speed/VRAM
        do_sample=True,               # sampling makes output more natural
        top_p=0.9,
        temperature=0.8,
        pad_token_id=tokenizer.eos_token_id
    )

generated_text = tokenizer.decode(output_ids[0], skip_special_tokens=True)
print(generated_text)
```

> If you want to re-run the exact same thing again (like your original snippet did twice), just **re-run this cell** to generate a fresh sample.

Step 6 — If You Still Hit Memory Errors (OOM)

```
# Try a smaller model first:
#   model_name = "distilgpt2"

# Or let Transformers shard layers across devices automatically (needs accelerate):
from transformers import AutoTokenizer, AutoModelForCausalLM
import torch

model_name = "gpt2-medium"     # Larger than gpt2; may still OOM on T4; try cautiously
tokenizer = AutoTokenizer.from_pretrained(model_name)
if tokenizer.pad_token is None:
    tokenizer.pad_token = tokenizer.eos_token

model = AutoModelForCausalLM.from_pretrained(
    model_name,
    device_map="auto",           # places layers on GPU/CPU as needed
    torch_dtype=torch.float16    # reduce VRAM
)

prompt = "indian cricket"
inputs = tokenizer(prompt, return_tensors='pt').to(model.device)

with torch.inference_mode():
    output_ids = model.generate(
        **inputs,
        max_length=50,
        do_sample=True,
        top_p=0.9,
```

```
        temperature=0.8,
        pad_token_id=tokenizer.eos_token_id
    )

print(tokenizer.decode(output_ids[0], skip_special_tokens=True))
```

Other tips

- Reduce `max_length`
- Use `gpt2` → `distilgpt2` (smaller)
- Clear VRAM: restart runtime if memory gets fragmented

4 (Optional) LLM Chunking** Demo (when input text is long)**

> Split long text into chunks so it fits in the model's context window.

```
from transformers import AutoTokenizer, AutoModelForCausalLM
import torch

model_name = "gpt2"  # or "distilgpt2" if you need lighter model
tokenizer = AutoTokenizer.from_pretrained(model_name)
if tokenizer.pad_token is None:
    tokenizer.pad_token = tokenizer.eos_token

device = "cuda" if torch.cuda.is_available() else "cpu"
dtype = torch.float16 if device == "cuda" else torch.float32
model = AutoModelForCausalLM.from_pretrained(model_name, torch_dtype=dtype).to(device)

def chunk_text(text, max_tokens=256):
    """Split text into token chunks that fit max_tokens."""
    ids = tokenizer.encode(text, return_tensors='pt')[0]
    chunks = [ids[i:i+max_tokens] for i in range(0, len(ids), max_tokens)]
    return chunks

def generate_for_chunks(chunks, gen_len=80):
    outputs = []
    with torch.inference_mode():
        for ch in chunks:
            ch = ch.unsqueeze(0).to(device)
            out = model.generate(
                ch,
                max_length=min(ch.shape[-1] + gen_len, 1024),  # GPT-2 max length
                do_sample=True,
                top_p=0.9,
                temperature=0.8,
                pad_token_id=tokenizer.eos_token_id
            )
            outputs.append(tokenizer.decode(out[0], skip_special_tokens=True))
    return outputs
```

```python
long_text = "Artificial Intelligence and NLP are growing rapidly in healthcare, finance, and ed
chunks = chunk_text(long_text, max_tokens=256)
responses = generate_for_chunks(chunks, gen_len=60)

for i, r in enumerate(responses, 1):
    print(f"\n--- Response for chunk {i} ---\n{r}\n")
```

Why the same code often fails on CPU

- Large models + long sequences → **slow** or **out-of-memory** on CPU
- Missing tokenizer import/usage (you must tokenize before `.generate`)
- No `pad_token_id` (some GPT-2 configs need it for generation)

CPU fallback: use `distilgpt2`, **shorter `max_length`**, and keep inputs small.

Summary

1. Set **Colab → GPU**
2. Install **transformers** (first), **accelerate**, **safetensors**
3. Verify GPU with **`!nvidia-smi`**
4. Load **tokenizer + model** to **GPU** (use fp16)
5. Generate text (your **"indian cricket"** prompt)
6. If OOM → **smaller model**, **shorter sequences**, or **device_map="auto"**

   You can now copy each block into **separate Colab cells** and run top to bottom.

[ ]: