Retrieval Augmented Generation

Indexing Pipeline: Data Chunking Strategies and Methods

Abhinav Kimothi

with

LlamaIndex LangChain OpenAI Hugging Face

Download as pdf for clickable links

Indexing Pipeline

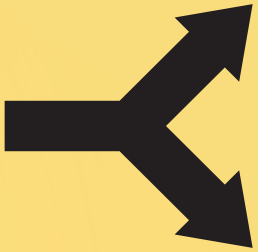
The indexing pipeline sets up the knowledge source for the RAG system. It is generally considered an offline process. However, information can also be fetched in real time. It involves four primary steps.

1

Loading

This step involves extracting information from different knowledge sources a loading them into documents.

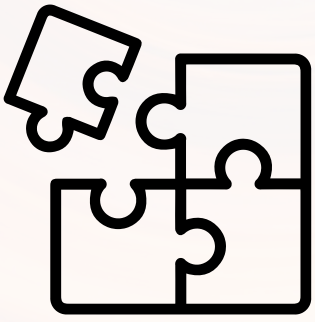
2



Splitting

This step involves splitting documents into smaller manageable chunks. Smaller chunks are easier to search and to use in LLM context windows.

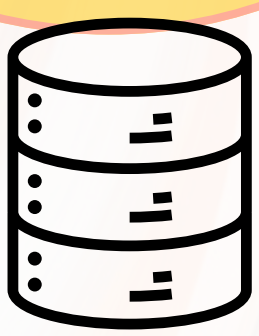
3



Embedding

This step involves converting text documents into numerical vectors. ML models are mathematical models and therefore require numerical data.

4



Storing

This step involves storing the embeddings vectors. Vectors are typically stored in Vector Databases which are best suited for searching.

Abhinav Kimothi

Document Splitting

Once the data is loaded, the next step in the indexing pipeline is splitting the documents into manageable chunks. The question arises around the need of this step. Why is splitting of documents necessary. There are two reasons for that -

Ease of Search

Large chunks of data are harder to search over. Splitting data into smaller chunks therefore helps in better indexation.

Context Window Size

LLMs allow only a finite number of tokens in prompts and completions. The context therefore cannot be larger than what the context window permits.

Chunking Strategies

While splitting documents into chunks might sound a simple concept, there are certain best practices that researchers have discovered. There are a few considerations that may influence the overall chunking strategy.

1 Nature of Content

Consider whether you are working with lengthy documents, such as articles or books, or shorter content like tweets or instant messages. The chosen model for your goal and, consequently, the appropriate chunking strategy depend on your response.

2 Embedding Model being Used

We will discuss embeddings in detail in the next section but the choice of embedding model also dictates the chunking strategy. Some models perform better with chunks of specific length

3 Expected Length and Complexity of User Queries

Determine whether the content will be short and specific or long and complex. This factor will influence the approach to chunking the content, ensuring a closer correlation between the embedded query and the embedded chunks

4 Application Specific Requirements

The application use case, such as semantic search, question answering, summarization, or other purposes will also determine how text should be chunked. If the results need to be input into another language model with a token limit, it is crucial to factor this into your decision-making process.

Abhinav Kimothi

Chunking Methods

Depending on the aforementioned considerations, a number of text splitters are available. At a broad level, text splitters operate in the following manner:

- Divide the text into compact, semantically meaningful units, often sentences.
- Merge these smaller units into larger chunks until a specific size is achieved, measured by a length function.
- Upon reaching the predetermined size, treat that chunk as an independent segment of text. Thereafter, start creating a new text chunk with some degree of overlap to maintain contextual continuity between chunks.

Two areas to focus on, therefore are -

How the text is split?

How the chunk size is measured?

A very common approach is where we pre-determine the size of the text chunks.

Additionally, we can specify the overlap between chunks (Remember, overlap is preferred to maintain contextual continuity between chunks).

This approach is simple and cheap and is, therefore, widely used. Let's look at some examples -

Split by Character

In this approach, the text is split based on a character and the chunk size is measured by the number of characters.

Example text : **alice_in_wonderland.txt** (the book in .txt format)

using **LangChain's CharacterTextSplitter**

```
with open('../Data/alice_in_wonderland.txt') as f:
    AliceInWonderland = f.read()

from langchain.text_splitter import CharacterTextSplitter
text_splitter = CharacterTextSplitter(
    separator = "\n\n",
    chunk_size = 2000,
    chunk_overlap  = 100,
    length_function = len,
    is_separator_regex = False,
)

texts = text_splitter.create_documents([AliceInWonderland])
print(texts[0])
print(texts[1])
```

texts[0]

"TITLE: Alice's Adventures in Wonderland\nAUTHOR: Lewis Carroll\n\n\n CHAPTER I \n(Down the Rabbit-Hole)\n\n Alice was beginning to get very tired of sitting by her sister\non the bank, and of having nothing to do: once or twice she had\npeeped into the book her sister was reading, but it had no\npictures or conversations in it, `and what is the use of a book,'\nthought Alice `without pictures or conversation?'\n\n So she was considering in her own mind (as well as she could,\nfor the hot day made her feel very sleepy and stupid), whether\nthe pleasure of making a daisy-chain would be worth the trouble\nof getting up and picking the daisies, when suddenly a White\nRabbit with pink eyes ran close by her.\n\n There was nothing so VERY remarkable in that; nor did Alice\nthink it so VERY much out of the way to hear the Rabbit say to\nitself, `Oh dear! Oh dear! I shall be late!' (when she thought\nit over afterwards, it occurred to her that she ought to have\nwondered at this, but at the time it all seemed quite natural);\nbut when the Rabbit actually TOOK A WATCH OUT OF ITS WAISTCOAT-\nPOCKET, and looked at it, and then hurried on, Alice started to\nher feet, for it flashed across her mind that she had never\nbefore seen a rabbit with either a waistcoat-pocket, or a watch to\ntake out of it, and burning with curiosity, she ran across the\nfield after it, and fortunately was just in time to see it pop\ndown a large rabbit-hole under the hedge.\n\n In another moment down went Alice after it, never once\nconsidering how in the world she was to get out again.\n\n The rabbit-hole went straight on like a tunnel for some way,\nand then dipped suddenly down, so suddenly that Alice had not a\nmoment to think about stopping herself before she found herself\nfalling down a very deep well."

Chunk 1

Overlap

texts[1]

"In another moment down went Alice after it, never once\nconsidering how in the world she was to get out again.\n\n The rabbit-hole went straight on like a tunnel for some way,\nand then dipped suddenly down, so suddenly that Alice had not a\nmoment to think about stopping herself before she found herself\nfalling down a very deep well.\n\n Either the well was very deep, or she fell very slowly, for she\nhad plenty of time as she went down to look about her and to\nwonder what was going to happen next. First, she tried to look\ndown and make out what she was coming to, but it was too dark to\nsee anything; then she looked at the sides of the well, and\nnoticed that they were filled with cupboards and book-shelves;\nhere and there she saw maps and pictures hung upon pegs. She\ntook down a jar from one of the shelves as she passed; it was\nlabelled `ORANGE MARMALADE', but to her great disappointment it\nwas empty: she did not like to drop the jar for fear of killing\nsomebody, so managed to put it into one of the cupboards as she\nfell past it."

Chunk 2

Let's find out how many chunks were created

```
print(f"Total Number of Chunks Created => {len(texts)}")

print(f"Length of the First Chunk is => {len(texts[0].page_content)} characters")

print(f"Length of the Last Chunk is => {len(texts[-1].page_content)} characters")
```

Total Number of Chunks Created => 93

Length of the First Chunk is => 1777 characters

Length of the Last Chunk is => 816 characters

Recursive Split by Character

A subtle variation to splitting by character is Recursive Split. The only difference is that instead of a single character used for splitting, this technique uses a list of characters and tries to split hierarchically till the chunk sizes are small enough. This technique is generally recommended for generic text.

Example text : **AK_BusyPersonIntroLLM.txt**
(Transcript of a YouTube video by Andrej Karpathy titled [1hr Talk] Intro to Large Language Models - https://www.youtube.com/watch?v=zjkBMFhNj_g&t=9s)

using **LangChain's RecursiveCharacterTextSplitter**

This is a generic text that is not formatted. Let's compare the two strategies.

with **CharacterTextSplitter**

```
with open('../Data/AK_BusyPersonIntroLLM.txt') as f:
    IntroToLLM = f.read()

from langchain.text_splitter import CharacterTextSplitter
text_splitter = CharacterTextSplitter(
    separator = "\n\n",
    chunk_size = 2000,
    chunk_overlap  = 400,
    length_function = len,
    is_separator_regex = False,
)

texts = text_splitter.create_documents([IntroToLLM])

print(f"Total Number of Chunks Created => {len(texts)}")

print(f"Length of the First Chunk is => {len(texts[0].page_content)} characters")

print(f"Length of the Last Chunk is => {len(texts[-1].page_content)} characters")
```

Total Number of Chunks Created => 1
Length of the First Chunk is => 64383 characters
Length of the Last Chunk is => 64383 characters

Text splitter fails to convert the text into chunks since there are no '\n\n' character present in the raw transcript

with **RecursiveCharacterTextSplitter**

```
with open('../Data/AK_BusyPersonIntroLLM.txt') as f:
    IntroToLLM = f.read()

from langchain.text_splitter import RecursiveCharacterTextSplitter

text_splitter = RecursiveCharacterTextSplitter(
    chunk_size = 2000,
    chunk_overlap  = 400,
    length_function = len,
    is_separator_regex = False,
)

texts = text_splitter.create_documents([IntroToLLM])

print(f"Total Number of Chunks Created => {len(texts)}")

print(f"Length of the First Chunk is => {len(texts[0].page_content)} characters")

print(f"Length of the Last Chunk is => {len(texts[-1].page_content)} characters")
```

Total Number of Chunks Created => 40
Length of the First Chunk is => 1998 characters
Length of the Last Chunk is => 1967 characters

Recursive text splitter performs well in dealing with generic text

Split by Tokens

For those well versed with Large Language Models, tokens is not a new concept. All LLMs have a token limit in their respective context windows which we cannot exceed. It is therefore a good idea to count the tokens while creating chunks. All LLMs also have their tokenizers.

Tiktoken Tokenizer

OpenAI

Tiktoken tokenizer has been created by OpenAI for their family of models. Using this strategy, the split still happens based on the character. However, the length of the chunk is determined by the number of tokens.

Example text : **AK_BusyPersonIntroLLM.txt**
(Transcript of a YouTube video by Andrej Karpathy titled [1hr Talk] Intro to Large Language Models - https://www.youtube.com/watch?v=zjkBMFhNj_g&t=9s)

using **LangChain's TokenTextSplitter**

```
with open('../Data/AK_BusyPersonIntroLLM.txt') as f:
    IntroToLLM = f.read()

from langchain.text_splitter import TokenTextSplitter
import tiktoken

text_splitter = TokenTextSplitter(
    chunk_size = 1000,
    chunk_overlap  = 20,
    length_function = len,

)

texts = text_splitter.create_documents([IntroToLLM])

encoding = tiktoken.get_encoding("cl100k_base")

print(f"Total Number of Chunks Created => {len(texts)}")

print(f"Total Number of Tokens in the document => {len(encoding.encode(IntroToLLM))} tokens")

print(f"Length of the First Chunk is => {len(encoding.encode(texts[0].page_content))} tokens")

print(f"Length of the Last Chunk is => {len(encoding.encode(texts[0].page_content))} tokens")
```

Total Number of Chunks Created => 14
Total Number of Tokens in the document => 12865 tokens
Length of the First Chunk is => 1014 tokens
Length of the Last Chunk is => 1014 tokens

Tokenizers are helpful in creating chunks that sit well in the context window of an LLM

Hugging Face Tokenizer

Hugging Face

Hugging Face has become the go-to platform for anyone building apps using LLMs or even other models. All models available via Hugging Face are also accompanied by their tokenizers.

Example text : **AK_BusyPersonIntroLLM.txt**
(Transcript of a YouTube video by Andrej Karpathy titled [1hr Talk] Intro to Large Language Models - https://www.youtube.com/watch?v=zjkBMFhNj_g&t=9s)

using **Transformers** and **LangChain's RecursiveCharacterTextSplitter**

Example tokenizer : GPT2TokenizerFast

```
with open('../Data/AK_BusyPersonIntroLLM.txt') as f:
    IntroToLLM = f.read()

from transformers import GPT2TokenizerFast
from langchain.text_splitter import RecursiveCharacterTextSplitter


tokenizer = GPT2TokenizerFast.from_pretrained("gpt2")

text_splitter = RecursiveCharacterTextSplitter.from_huggingface_tokenizer(
    tokenizer, chunk_size=100, chunk_overlap=0
)

texts = text_splitter.split_text(IntroToLLM)

print(texts[0])
print(texts[1])
```

texts[0]
"hi everyone so recently I gave a 30-minute talk on large language models just kind of like an intro talk um unfortunately that talk was not recorded but a lot of people came to me after the talk and they told me that uh they really liked the talk so I would just I thought I would just re-record it and basically put it up on YouTube so here we go the busy person's intro to large language models director Scott okay so let's begin first of all what is a large language model

Chunk 1

No Overlap as specified

texts[1]
really well a large language model is just two files right um there be two files in this hypothetical directory so for example work with the specific example of the Llama 270b model this is a large language model released by meta Ai and this is basically the Llama series of language models the second iteration of it and this is the 70 billion parameter model of uh of this series so there's multiple models uh belonging to the Lama 2 Series uh 7 billion um 13 billion 34 billion and 70 billion is the the

Chunk 2

Do take a look at Hugging Face documents on Tokenizers

Hugging Face

https://huggingface.co/docs/transformers/tokenizer_summary

Abhinav Kimothi

Other Tokenizer

Other libraries like Spacy, NLTK and SentenceTransformers also provide splitters

```python
from langchain.text_splitter import NLTKTextSplitter

text_splitter = NLTKTextSplitter(chunk_size=1000)

texts = text_splitter.split_text(IntroToLLM)
print(texts[0])
```

```python
from langchain.text_splitter import SpacyTextSplitter

text_splitter = SpacyTextSplitter(chunk_size=1000)

texts = text_splitter.split_text(IntroToLLM)
print(texts[0])
```

Specialized Chunking

Chunking often aims to keep text with common context together. With this in mind, we might want to specifically honour the structure of the document itself for example HTML, Markdown, Latex or even code.

Example : https://medium.com/p/29a7e8610843

Example HTML : “Context is Key: The Significance of RAG in Language Models”
(A blog on Medium - https://medium.com/p/29a7e8610843)

using **LangChain’s HTMLHeaderTextSplitter** & **RecursiveCharacterTextSplitter**

```python
from langchain.text_splitter import HTMLHeaderTextSplitter

from langchain.text_splitter import RecursiveCharacterTextSplitter

url = "https://medium.com/p/29a7e8610843"

headers_to_split_on = [
    ("h1", "Header 1"),
    ("h2", "Header 2"),
    ("h3", "Header 3"),
    ("h4", "Header 4"),
]

html_splitter = HTMLHeaderTextSplitter(headers_to_split_on=headers_to_split_on)

# for local file use html_splitter.split_text_from_file(<path_to_file>)
html_header_splits = html_splitter.split_text_from_url(url)

chunk_size = 5000
chunk_overlap = 300
text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=chunk_size, chunk_overlap=chunk_overlap
)

# Split
splits = text_splitter.split_documents(html_header_splits)
len(splits)
```

LangChain All LangChain Splitters

Things to Keep in Mind

- Ensure data quality by preprocessing it before determining the optimal chunk size. Examples include removing HTML tags or eliminating specific elements that contribute noise, particularly when data is sourced from the web.

- Consider factors such as content nature (e.g., short messages or lengthy documents), embedding model characteristics, and capabilities like token limits in choosing chunk sizes. Aim for a balance between preserving context and maintaining accuracy.

- Test different chunk sizes. Create embeddings for the chosen chunk sizes and store them in your index or indices. Run a series of queries to evaluate quality and compare the performance of different chunk sizes.

Acknowledgements

LangChain https://python.langchain.com/docs

LlamaIndex https://docs.llamaindex.ai/en/stable/

Pinecone https://www.pinecone.io/learn/