Building LLM Applications: Data Preparation (Part 2)

Vipra Singh · Follow

14 min read · Jan 8, 2024

Learn Large Language Models (LLM) through the lens of a Retrieval Augmented Generation (RAG) Application.

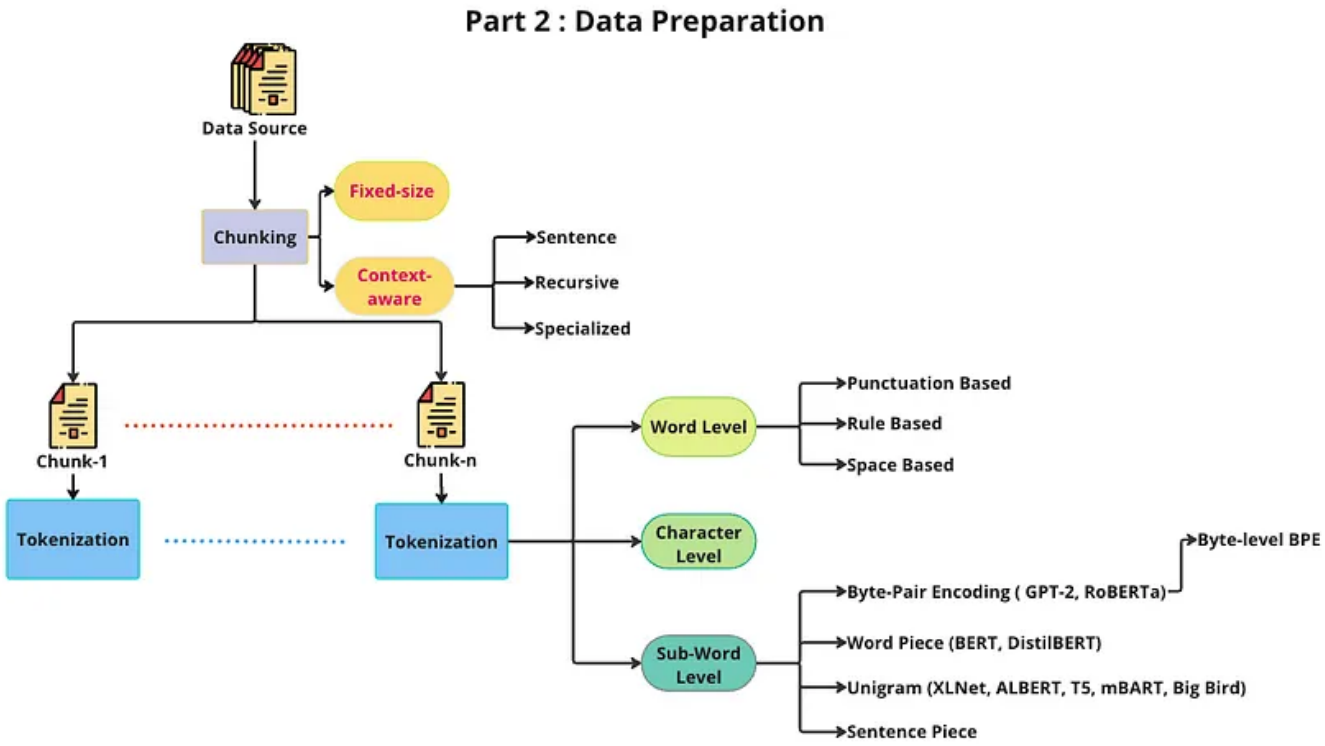
Posts in this Series

1. *Introduction*
2. *Data Preparation (This Post)*
3. *Sentence Transformers*
4. *Vector Database*
5. *Search & Retrieval*
6. *LLM*
7. *Open-Source RAG*

8. *Evaluation*
9. *Serving LLMs*
10. *Advanced RAG*

. . .

Table of Contents



Data Preparation Flow for Retrieval Augmented Generation (RAG)

Greetings,

In the previous post, we delved into the Retrieval Augmented Generation (RAG) Pipeline, gaining a comprehensive understanding of its various components.

The initial stage for any Machine Learning Application entails data preparation. This encompasses the establishment of the Data Ingestion Pipeline and the preprocessing of data to render it compatible with the Inference Pipeline.

In this post, our attention will be directed towards the Data Preparation aspect of RAG. The objective is to efficiently organize and structure the data, ensuring optimal performance in locating answers within our application.

Let’s get into the details below.

1. Step 1: Data Ingestion

Building a consumer-friendly chatbot starts with smart data choices. This blog explores how to gather, manage, and cleanse data effectively for a winning Language Model (LLM) application.

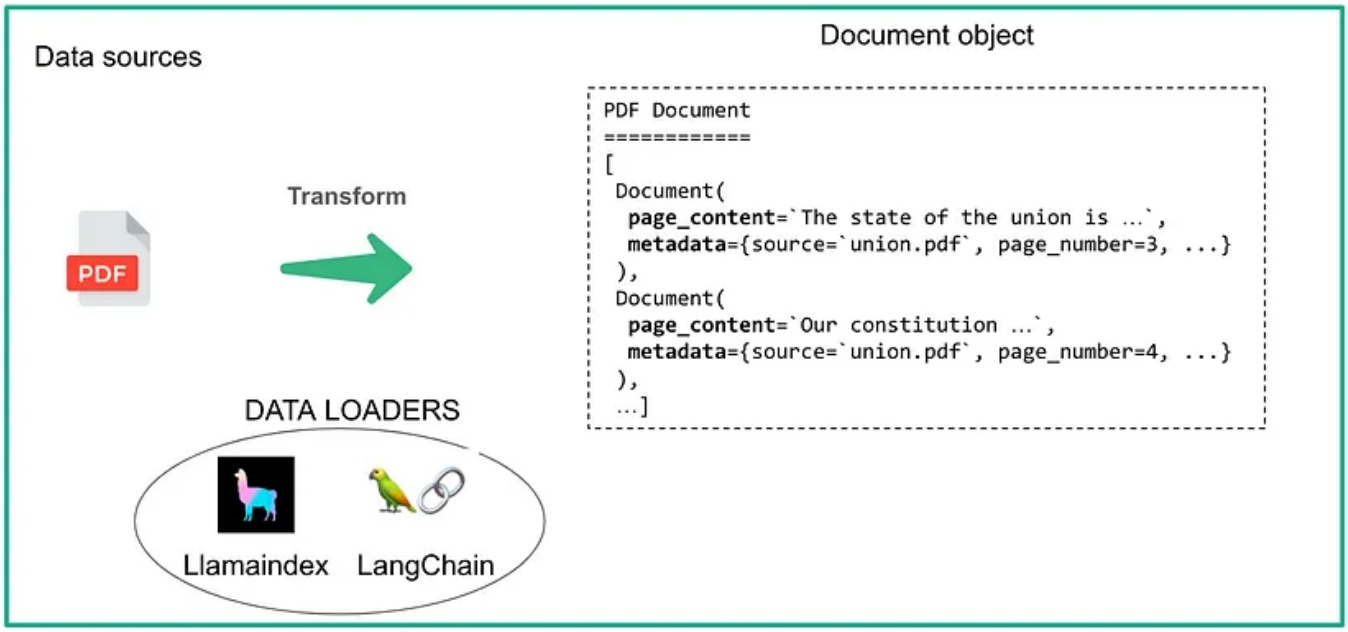
1. **Choose Wisely:** Identify data sources, from portals to APIs, and set up a push mechanism for consistent updates to your LLM app.
2. **Governance Matters:** Implement data governance policies upfront. Audit and catalog document sources, redact sensitive data and establish a foundation for context training.
3. **Quality Check:** Assess data for diversity, size, and noise levels. Lower-quality datasets dilute responses, necessitating an early classification mechanism.
4. **Stay Ahead:** Adhere to data governance even in fast-paced LLM development. It reduces risks and ensures scalable, robust data extraction.
5. **Real-Time Cleansing:** Pulling data from platforms like Slack? Filter out noise, typos, and sensitive content in real-time for a clean, effective LLM app.

2. Step 2: Data Cleaning

Every page of our files transforms into a Document object and has two essential components: **page_content** and **metadata**.

The **page_content** unveils the textual content extracted straight from the document page.

The **metadata** is a vital ensemble of additional details, like the document's source (the file it originates from), the page number, file type, and other nuggets of information. The metadata keeps tabs on the specific sources it taps into when weaving its magic and generating insightful answers.



Data Loading

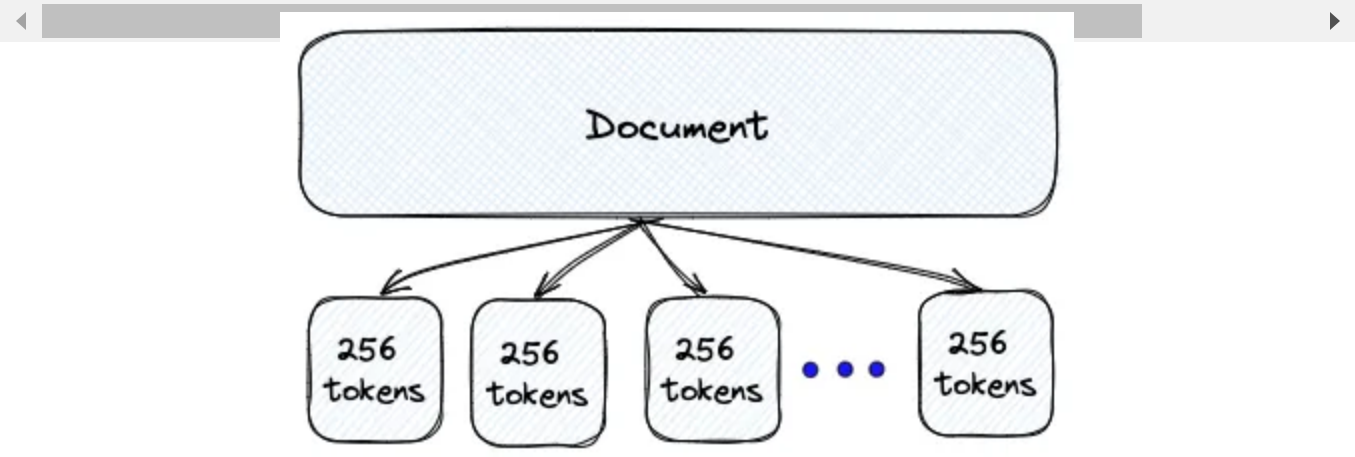
To accomplish this, we leverage robust tools such as Data Loaders, which are offered by open-source libraries like LangChain and Llamaindex. These libraries support various formats, ranging from PDF and CSV to HTML, Markdown, and even databases.

```
!pip install pypdf
!pip install langchain

#for PDF file we need to import PyPDFLoader from langchain framework
from langchain_community.document_loaders import PyPDFLoader

# for CSV file we need to import csv_loader
# for Doc we need to import UnstructuredWordDocumentLoader
# for Text document we need to import TextLoader
```

```
filePath = "/content/A_miniature_version_of_the_course_can_be_found_here__170174
loader = PyPDFLoader(filePath)
#Load document
pages = loader.load_and_split()
print(pages[0].page_content)
```

An advantage of this approach is that documents can be retrieved with page numbers.

3. Step 3: Chunking



Chunking

3.1. Why Chunk?

In the realm of applications, the game-changer lies in how you mold your data — be it markdown, PDFs, or other textual files. Picture this: you've got a hefty PDF, and you're eager to fire questions about its content. The catch? Traditional methods of tossing the entire document and your question at the model fall flat. Why? Well, let's talk about the limitations of the model's context window.

Enter GPT-3.5 and its kind. Picture the context window as a peek into the document, often limited to just a page or a few. Now, sharing your entire document at once? Not so realistic. But fear not!

The magic trick lies in chunking your data. Break it down into digestible portions, sending only the most relevant chunks to the model. This way, you're not overwhelming it, and you get the precise insights you crave.

By breaking down our structured documents into manageable chunks, we empower our LLM to process information with unparalleled efficiency. No longer limited by page constraints, this approach ensures that crucial details aren't lost in the shuffle.

3.2. Considerations before Chunking

Structure & Length of Documentation:

- Long Documents: Books, academic articles, etc.
- Short Documents: Social media posts, customer reviews, etc.

Embedding Model: Chunk size determines what embedding models should be used.

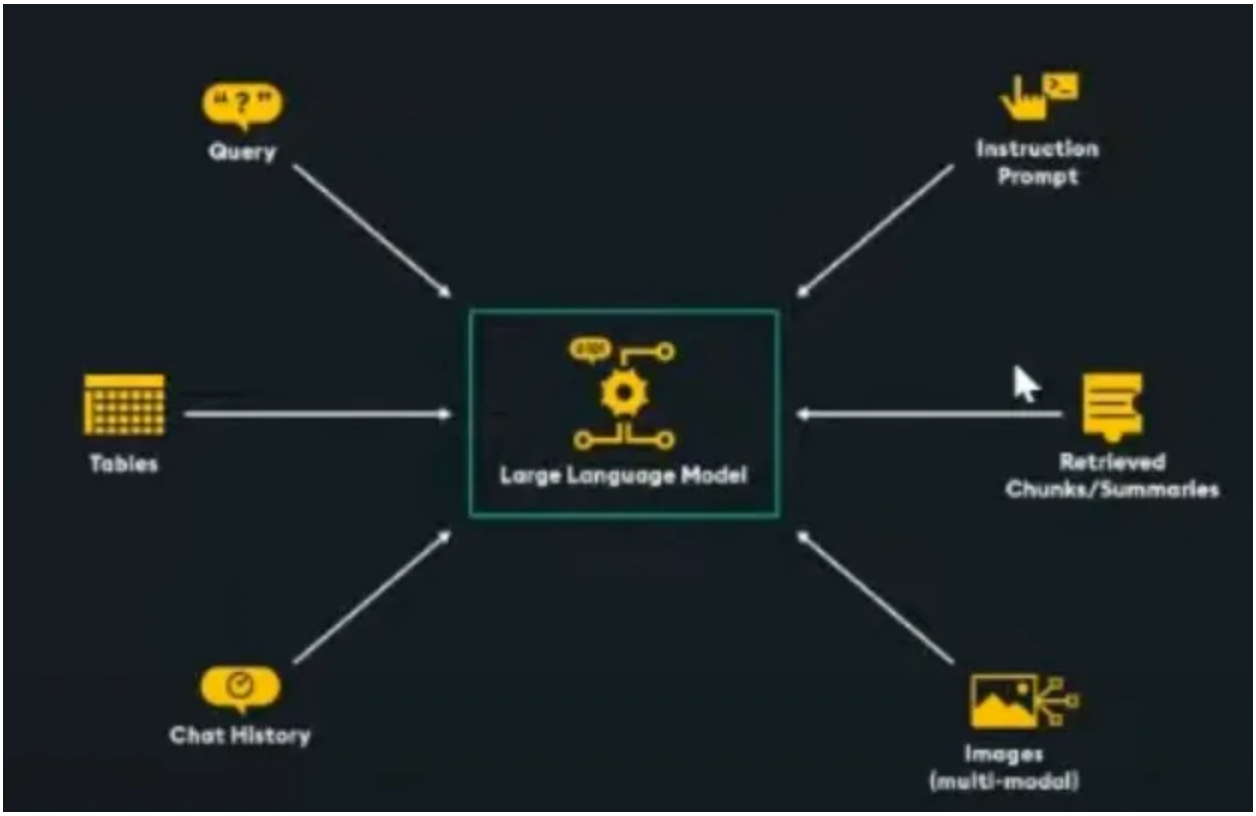
Expected Queries: What is the use case?

3.3. Chunk Size

- **Small chunk size :** Example: Single Sentence → Low contextual information for generation.
- **Large chunk size :** Example: Full Page, multiple paragraphs, full document. In this case, chunks cover more information, which could

increase the effectiveness of generation with more context.

3.3.1. Choosing Chunk Size



LLM Inputs

LLM Context Window

- Limit on how much data you can input to an LLM.
- **Top-K Retrieved Chunks:** Say the LLM has a 10,000 tokens context window size and we preserve about 1000 of that for a given user query, let's reserve 2000 of it for the instruction prompt and the chat history, which only leaves us with 7000 tokens left for any other information. Suppose, we intend to pass K = 10, Top-10 chunks into the LLM, that means we take the remaining 7000 tokens, divide that by 10 total chunks and I would have a maximum chunk size of about 700 tokens per chunk.

- **Range of Chunk Sizes:** The next step is to choose a range of potential chunk sizes to test. As mentioned previously, the choice should take into account the nature of the content (e.g., short messages or lengthy documents), the embedding model you'll use, and its capabilities (e.g., token limits). The objective is to find a balance between preserving context and maintaining accuracy. Start by exploring a variety of chunk sizes, including smaller chunks (e.g., 128 or 256 tokens) for capturing more granular semantic information and larger chunks (e.g., 512 or 1024 tokens) for retaining more context.

Evaluating the Performance of Each Chunk Size — To test various chunk sizes, you can either use multiple indices or a single index with multiple namespaces. With a representative dataset, create the embeddings for the chunk sizes you want to test and save them in your index (or indices). You can then run a series of queries for which you can evaluate quality, and compare the performance of the various chunk sizes. This is most likely to be an iterative process, where you test different chunk sizes against different queries until you can determine the best-performing chunk size for your content and expected queries.

Limitation of high context length :

- High context length can cause a quadratic increase in time & memory due to the transformer model's self-attention mechanism.

In this example published by LlamaIndex, you can see in the table below as the chunk size increases, there is a minor uptick in the Average Response Time. Interestingly, the Average Faithfulness seems to reach its zenith at chunk_size of 1024, whereas Average Relevancy shows a consistent improvement with larger chunk sizes, also peaking at 1024. This suggests

that a chunk size of 1024 might strike an optimal balance between response time and the quality of the responses, measured in terms of faithfulness and relevancy

Chunk Size	Average Response Time (s)	Average Faithfulness	Average Relevancy
128	1.55	0.85	0.78
256	1.57	0.90	0.78
512	1.68	0.85	0.85
1024	1.68	0.93	0.90
2048	1.72	0.90	0.89

3.4. Chunking Methods

There are different methods for chunking, and each of them might be appropriate for different situations. By examining the strengths and weaknesses of each method, our goal is to identify the right scenario to apply them to.

3.4.1. Fixed-size chunking

We decide the number of tokens in each chunk, throwing in optional overlaps for good measure. Why the overlap? To ensure that the richness of semantic context remains intact across the chunks.

Why go fixed-sized? It's the golden path for most scenarios. Not only is it computationally cheap, saving you processing power, but it's also a breeze to use. No need for complex NLP libraries; just the elegance of fixed-sized chunks seamlessly breaking down your data.

Here's an example of performing fixed-sized chunking with LangChain:

```
text = "..." # your text
from langchain.text_splitter import CharacterTextSplitter
text_splitter = CharacterTextSplitter(
    separator = "\n\n",
    chunk_size = 256,
    chunk_overlap  = 20
)
docs = text_splitter.create_documents([text])
```

3.4.2. “Context-aware” Chunking

These are a set of methods for taking advantage of the nature of the content we’re chunking and applying more sophisticated chunking to it. Here are some examples:

3.4.2.1. Sentence splitting

As we mentioned before, many models are optimized for embedding sentence-level content. Naturally, we would use sentence chunking, and there are several approaches and tools available to do this, including:

- **Naive splitting:** The most naive approach would be to split sentences by periods (“.”) and new lines. While this may be fast and simple, this approach would not take into account all possible edge cases. Here’s a very simple example:

```
text = "..." # your text
docs = text.split(".")
```

- NLTK: The Natural Language Toolkit (NLTK) is a popular Python library for working with human language data. It provides a sentence tokenizer

that can split the text into sentences, helping to create more meaningful chunks. For example, to use NLTK with LangChain, you can do the following:

```
text = "..." # your text
from langchain.text_splitter import NLTKTextSplitter
text_splitter = NLTKTextSplitter()
docs = text_splitter.split_text(text)
```

- **spaCy**: spaCy is another powerful Python library for NLP tasks. It offers a sophisticated sentence segmentation feature that can efficiently divide the text into separate sentences, enabling better context preservation in the resulting chunks. For example, to use spaCy with LangChain, you can do the following:

```
text = "..." # your text
from langchain.text_splitter import SpacyTextSplitter
text_splitter = SpaCyTextSplitter()
docs = text_splitter.split_text(text)
```

3.4.2.2. Recursive Chunking

Meet our secret weapon: the **RecursiveCharacterTextSplitter** from LangChain. This versatile tool gracefully splits text based on chosen characters, preserving semantic context. Think double newlines, single newlines, and spaces — it's like sculpting information into bite-sized, meaningful portions.

How does it work? Simple. Just pass the Document and specify your desired chunk length (let's say 1000 words). You can even fine-tune the overlap between chunks.

Here's an example of how to use recursive chunking with LangChain:

```
text = "..." # your text
from langchain.text_splitter import RecursiveCharacterTextSplitter
text_splitter = RecursiveCharacterTextSplitter(
    # Set a really small chunk size, just to show.
    chunk_size = 256,
    chunk_overlap  = 20
)
```

```
docs = text_splitter.create_documents([text])
```

3.4.2.3. Specialized chunking

Markdown and LaTeX are two examples of structured and formatted content you might run into. In these cases, you can use specialized chunking methods to preserve the original structure of the content during the chunking process.

- **Markdown**: Markdown is a lightweight markup language commonly used for formatting text. By recognizing the Markdown syntax (e.g., headings, lists, and code blocks), you can intelligently divide the content based on its structure and hierarchy, resulting in more semantically coherent chunks. For example:

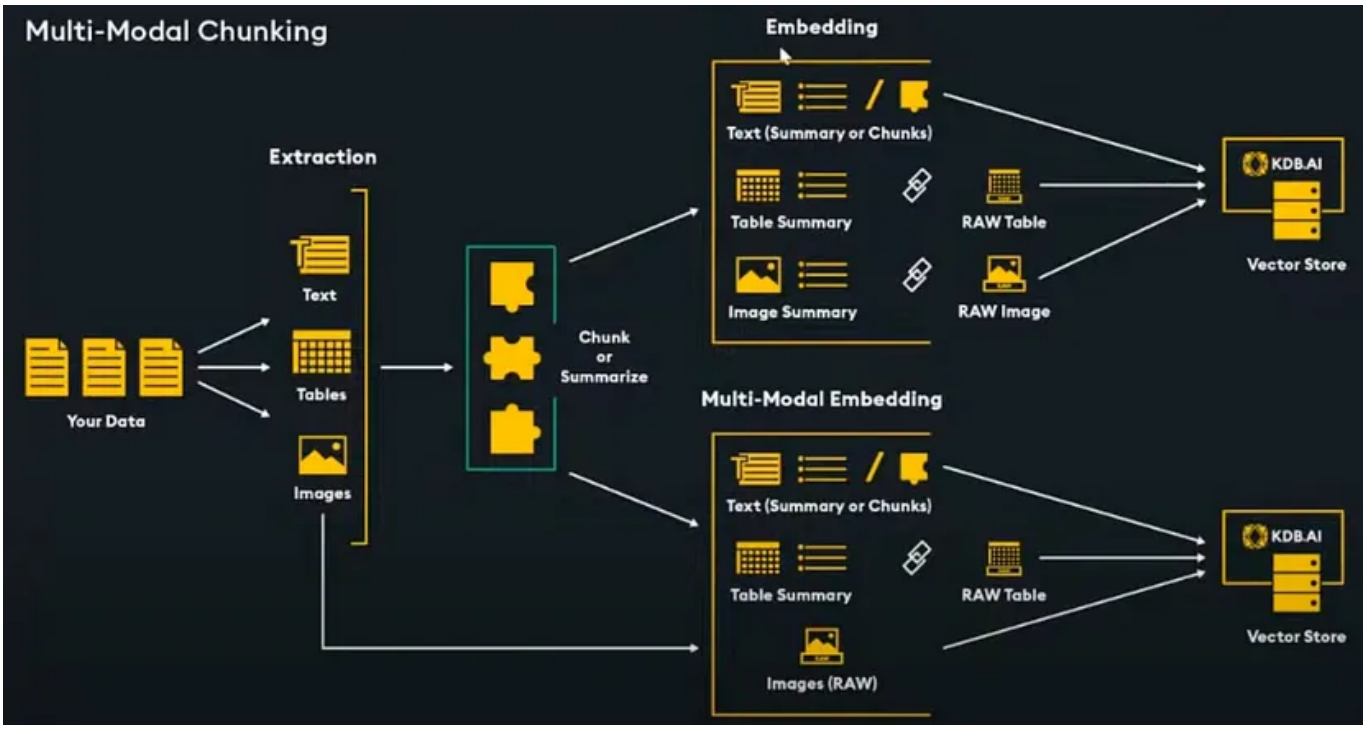
```
from langchain.text_splitter import MarkdownTextSplitter
markdown_text = "..."
```

```
markdown_splitter = MarkdownTextSplitter(chunk_size=100,
chunk_overlap=0)
docs =
markdown_splitter.create_documents([markdown_text])markdown_splitter
= MarkdownTextSplitter(chunk_size=100, chunk_overlap=0)
docs = markdown_splitter.create_documents([markdown_text])
```

- **LaTex**: LaTeX is a document preparation system and markup language often used for academic papers and technical documents. By parsing the LaTeX commands and environments, you can create chunks that respect the logical organization of the content (e.g., sections, subsections, and equations), leading to more accurate and contextually relevant results. For example:

```
from langchain.text_splitter import LatexTextSplitter
latex_text = "..."
latex_splitter = LatexTextSplitter(chunk_size=100, chunk_overlap=0)
docs = latex_splitter.create_documents([latex_text])
```

3.5. Multi-Modal Chunking



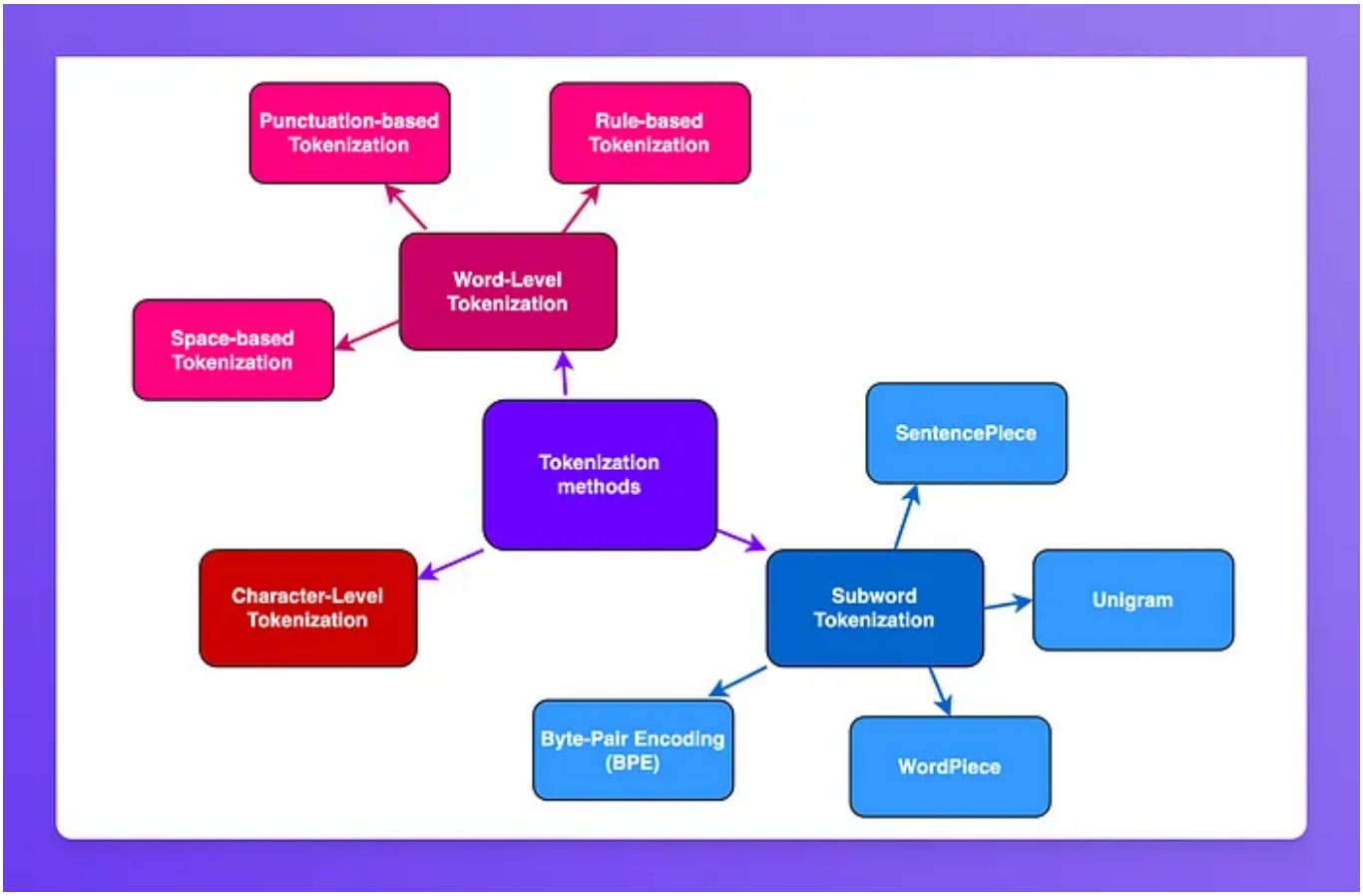
Multimodal Chunking

Extract tables and images from documents: LayoutPDFReader, Unstructured. Tables and images are extracted and can be tagged with metadata such as titles, descriptions & summaries.

Multi-Modal Methods:

- **Text Embedding:** Summarize images and tables
- **Multi-Modal Embeddings:** Use an embedding model that can process raw images

4. Step 4: Tokenization



Summary of the most used tokenization methods

Tokenization consists of splitting a phrase, sentence, paragraph, or entire text document into smaller units, such as individual words or terms. In this article, we'll see what are the main tokenization methods and where they are currently used. I suggest that you also have a look at this summary of tokenizers made by Hugging Face for a more in-depth guide.

4.1. Word-Level Tokenization

Word-level tokenization consists of splitting the text into units which are words. To do it properly, there are some precautions to consider.

Space and Punctuation Tokenization

Splitting text into smaller chunks is harder than it looks, and there are multiple ways of doing so. For example, let's look at the following sentence:

```
"Don't you like science? We sure do."
```

A simple way of tokenizing this text is to split it by spaces, which would give:

```
["Don't", "you", "like", "science?", "We", "sure", "do."]
```

If we look at the tokens `"science?"` and `"do."`, we notice that the punctuation is attached to the words `"science"` and `"do"`, which is suboptimal. We should take punctuation into account so that a model does not have to learn a different representation of a word and every possible punctuation symbol that could follow it, which would explode the number of representations the model has to learn.

Taking punctuation into account, tokenizing our text would give:

```
["Don", "'", "t", "you", "like", "science", "?", "We", "sure", "do", "."]
```

Rule-based Tokenization

The previous tokenization is somewhat better than pure space-based tokenization. However, we can further improve how the tokenization deals with the word `"Don't"`. `"Don't"` stands for `"do not"`, so it would be better tokenized with something like `["Do", "n't"]`. Other several ad-hoc rules can further improve tokenization.

However, depending on the rules we apply for tokenizing a text, a different tokenized output is generated for the same text. As a consequence, a pre-trained model only performs properly if you feed it an input that was tokenized with the same rules that were used to tokenize its training data.

The Problem with Word-Level Tokenization

Word-level tokenization can lead to problems for massive text corpora, as it generates a very big vocabulary. For example, the Transformer XL language model uses space and punctuation tokenization, resulting in a vocabulary size of 267k.

As a result of such a large vocabulary size, the model has a huge embedding matrix as the input and output layer, which increases both the memory and time complexity. To give a reference value, transformer models rarely have vocabulary sizes greater than 50,000.

4.2. Character-Level Tokenization

So if word-level tokenization is not ok, why not simply tokenize on characters?

Even though character tokenization would greatly reduce memory and time complexity, it makes it much more difficult for the model to learn meaningful input representations. E.g. learning a meaningful context-independent representation for the letter `"t"` is much harder than learning a context-independent representation for the word `"today"`.

Therefore, character tokenization often leads to a loss of performance. To get the best of both worlds, transformer models often use a hybrid between word-level and character-level tokenization called subword tokenization.

4.3. Subword Tokenization

Subword tokenization algorithms rely on the principle that frequently used words should not be split into smaller subwords, but rare words should be decomposed into meaningful subwords.

For instance `"annoyingly"` might be considered a rare word and could be decomposed into `"annoying"` and `"ly"`. Both `"annoying"` and `"ly"` as stand-alone subwords would appear more frequently while at the same time the meaning of `"annoyingly"` is kept by the composite meaning of `"annoying"` and `"ly"`.

In addition to allowing the model's vocabulary size to be reasonable, subword tokenization allows it to learn meaningful context-independent representations. Moreover, subword tokenization can be used to process words the model has never seen before, by breaking them down into known subwords.

Let's see now several different ways of doing subword tokenization.

Byte-Pair Encoding (BPE)

Byte-Pair Encoding (BPE) relies on a pre-tokenizer that splits the training data into words (such as with space tokenization, used in GPT-2 and Roberta).

After pre-tokenization, BPE creates a base vocabulary consisting of all symbols that occur in the set of unique words of the corpus and learns merge rules to form a new symbol from two symbols of the base vocabulary. This process iterates until the vocabulary has attained the desired vocabulary size.

WordPiece

WordPiece, used for BERT, DistilBERT, and Electra, is very similar to BPE. WordPiece first initializes the vocabulary to include every character present in the training data and progressively learns a given number of merge rules. In contrast to BPE, WordPiece does not choose the most frequent symbol pair, but the one that maximizes the likelihood of the training data once added to the vocabulary.

Intuitively, WordPiece is slightly different from BPE in that it evaluates what it loses by merging two symbols to ensure it's worth it.

Unigram

In contrast to BPE or WordPiece, Unigram initializes its base vocabulary to a large number of symbols and progressively trims down each symbol to obtain a smaller vocabulary. The base vocabulary could for instance correspond to all pre-tokenized words and the most common substrings. Unigram is often used in conjunction with SentencePiece.

SentencePiece

All tokenization algorithms described so far have the same problem: it is assumed that the input text uses spaces to separate words. However, not all languages use spaces to separate words.

To solve this problem generally, SentencePiece treats the input as a raw input stream, thus including the space in the set of characters to use. It then uses the BPE or Unigram algorithm to construct the appropriate vocabulary.

Examples of models using SentencePiece are ALBERT, XLNet, Marian, and T5.

OpenAI tokenization visuals: https://platform.openai.com/tokenizer

Conclusion

In this blog we explored the data preparation process for Retrieval Augmented Generation (RAG) applications, emphasizing efficient structuring for optimal performance. It covers transforming raw data into structured documents, creating relevant chunks, and tokenization methods like subword tokenization. The importance of choosing the right chunk size and considerations for each tokenization method are highlighted. The post provides insights into tailoring data preparation for specific application needs.

Credits

In this blog post, we have compiled information from various sources, including research papers, technical blogs, official documentations, YouTube videos, and more. Each source has been appropriately credited beneath the corresponding images, with source links provided.

Below is a consolidated list of references:

1. https://dataroots.io/blog/aiden-data-ingestion
2. https://www.pinecone.io/learn/chunking-strategies/
3. https://www.youtube.com/watch?v=uhVMFZjUOJI&t=1209s
4. https://medium.com/nlplanet/two-minutes-nlp-a-taxonomy-of-tokenization-methods-60e330aacad3

Thank you for reading!

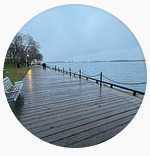
If this guide has enhanced your understanding of Python and Machine Learning:

- Please show your support with a clap 👏 or several claps!
- Your claps help me create more valuable content for our vibrant Python or ML community.
- Feel free to share this guide with fellow Python or AI / ML enthusiasts.
- Your feedback is invaluable — it inspires and guides our future posts.

Connect with me!

Vipra

Chunking Tokenization Large Language Models Data Preprocessing Langchain

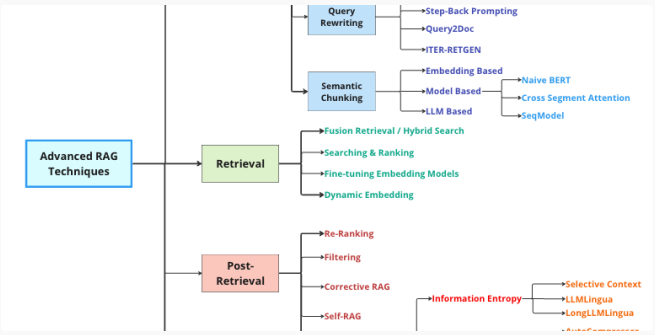


Written by Vipra Singh

Follow

1K Followers

More from Vipra Singh



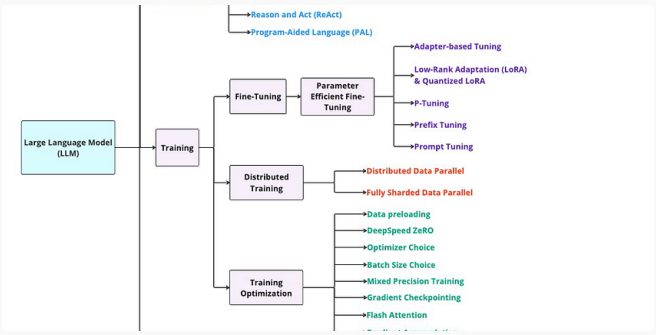
Vipra Singh

Building LLM Applications: Advanced RAG (Part 10)

Learn Large Language Models (LLM) through the lens of a Retrieval Augmented...

✨ · 48 min read · Apr 27, 2024

384 2



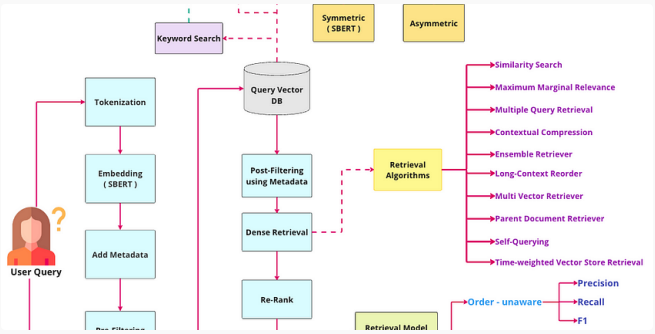
Vipra Singh

Building LLM Applications: Large Language Models (Part 6)

Learn Large Language Models (LLM) through the lens of a Retrieval Augmented...

38 min read · Feb 10, 2024

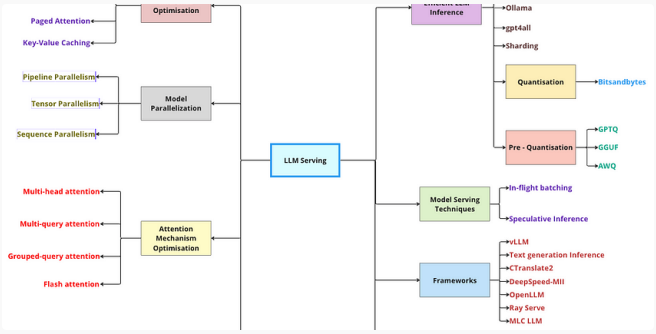
390 1



Vipra Singh

Building LLM Applications: Search & Retrieval (Part 5)

Learn Large Language Models (LLM) through the lens of a Retrieval Augmented...



Vipra Singh

Building LLM Applications: Serving LLMs (Part 9)

Learn Large Language Models (LLM) through the lens of a Retrieval Augmented...

26 min read · Jan 27, 2024

405 1

✦ · 50 min read · Apr 17, 2024

546 3

See all from Vipra Singh

Recommended from Medium

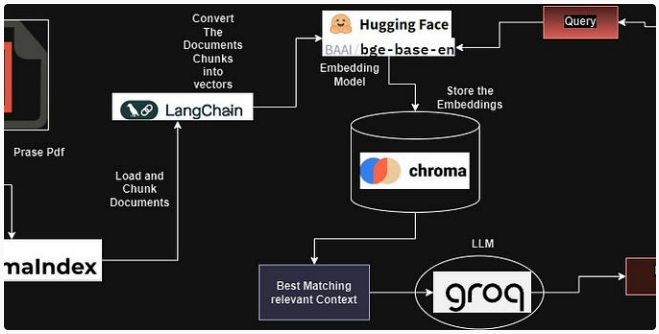
Paul Iusztin in Decoding ML

The 4 Advanced RAG Algorithms You Must Know to Implement

Implement from scratch 4 advanced RAG methods to optimize your retrieval and post-...

15 min read · May 4, 2024

1.4K 10



Plaban Nayak in The AI Forum

RAG on Complex PDF using LlamaParse, Langchain and Groq

Retrieval-Augmented Generation (RAG) is a new approach that leverages Large Languag...

13 min read · Apr 7, 2024

678 9

Lists

Natural Language Processing

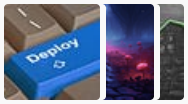
1494 stories · 1011 saves

AI Regulation

6 stories · 473 saves

ChatGPT prompts

47 stories · 1642 saves



Predictive Modeling w/ Python

20 stories · 1254 saves

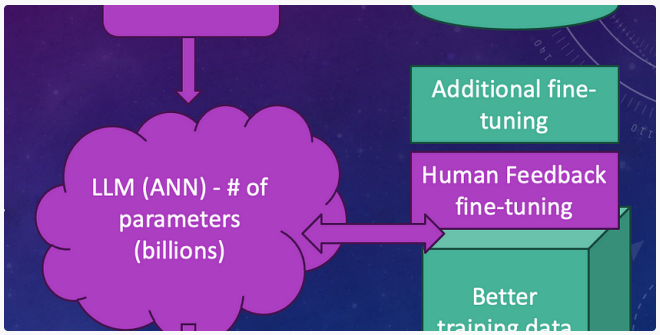
Mahesh

How to Productionize Large Language Models (LLMs)

Understand LLMOps, architectural patterns, how to evaluate, fine tune & deploy...

94 min read · Mar 27, 2024

186 2



Anton Antich in Superstring Theory

Retrieval Augmented Generation (RAG) In 2 Minutes

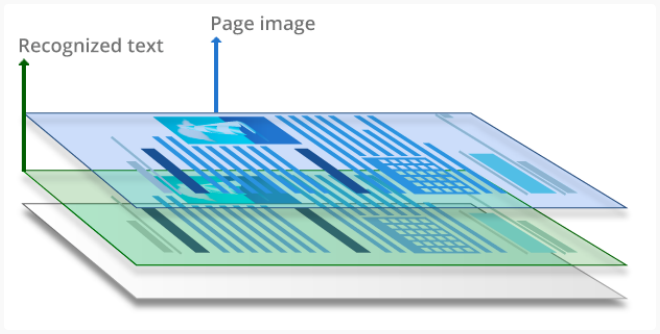
What's the fuzz all about and how to build my custom agent or "chatbot" easy and fast

4 min read · May 2, 2024

35 1

Ian Kelk

RAG Detective: Retrieval Augmented Generation with...



Sasha Korovkina in Dev Genius

Building a High Precision Financial PDF Extraction Tool. Part 1.

This article was produced as part of the final project for Harvard's AC215 Fall 2023 course.

16 min read · Dec 10, 2023

207 2

Parsing Text from PDF Files

10 min read · Apr 29, 2024

209

See more recommendations