**Building LLM Apps**

**Data Preparation**

**STEP1: DATA INGESTION**

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 foundation for context training.
3. **Quality Check:** Asses data from 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.

**STEP 2: DATA CLEANING**

* Every page of our files transforms into a Document object and has two essential components: **page\_content** and **metadata.**
* **Page\_content:** unveils the textual content extracted straight from the document page.
* **Metadata:** a vital ensemble of additional details, like the document 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.A screenshot of a document

  Description automatically generated
* 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\_\_1701743458.pdf"  
  loader = PyPDFLoader(filePath)   
  #Load document   
  pages = loader.load\_and\_split()  
  print(pages[0].page\_content)

**A diagram of a document

Description automatically generatedSTEP 3: CHUNKING**

* **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?
  + 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.
  + 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.
* **Considerations before Chunking**
  + **Structure & Length of Document**
    - **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?
* **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.
* **A diagram of a large language model

  Description automatically generatedChoosing Chunk Size\**
  + **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 maintain 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.
    - By LlamaIndex, you can see 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.
* **A graph with numbers and text

  Description automatically generated with medium confidence**
* **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.
  + **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])
* **“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:
* **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:
  + **Native splitting:** The most naïve 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 SpacyTextSplitter  
      text\_splitter = SpaCyTextSplitter()  
      docs = text\_splitter.split\_text(text)
  + **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])**
  + **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 usef 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:
  + **A screenshot of a computer code

    Description automatically generated**
    - **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:
  + **A screen shot of a computer

    Description automatically generated**
  + **Multi-Modal Chunking**
* **A diagram of a software development

  Description automatically generated with medium confidence**
  + - **Extract tables and images from documents:** LayoutPDFReader, Unstructured. Tables and images are extracted and can be tagged with metadata such as titles, descriptiions & summaries.
    - **MultiModal Methods:**
      * **Text Embeddings:** Summarize images and tables
      * **Multi-Modal Embeddings:** Use an embedding model that can process raw images

**STEP 4: TOKENIZATION**

**A diagram of a system

Description automatically generated**

* 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.
* **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 memory and time complexity. To give a reference value, transformer models rarely have vocabulary sizes greater than 50,000.
* **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.
* **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 “annonying” 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 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

**Wrap-Up:**

* 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 methods are highlighted. The post provides insights into tailoring data preparation for specific application needs.