← Learn

Chunking Strategies for LLM Applications

Roie Schwaber-Cohen

Jun 30, 2023

Share:

Core Components

d applications, **chunking** is the process of
:o smaller segments. It's an essential
evance of the content we get back from a
l to embed content. In this blog post, we'll
fficiency and accuracy in LLM-related

ex in Pinecone needs to be embedded first.
sure we're embedding a piece of content
still semantically relevant.

ndex a corpus of documents, with each
document containing valuable information on a specific topic. By applying an

effective chunking strategy, we can ensure our search results accurately capture the essence of the user's query. If our chunks are too small or too large, it may lead to imprecise search results or missed opportunities to surface relevant content. As a rule of thumb, if the chunk of text makes sense without the surrounding context to a human, it will make sense to the language model as well. Therefore, finding the optimal chunk size for the documents in the corpus is crucial to ensuring that the search results are accurate and relevant.

Another example is conversational agents (which we covered before using Python and Javascript). We use the embedded chunks to build the **context** for the conversational agent based on a knowledge base that **grounds** the agent in trusted information. In this situation, it's important to make the right choice about our chunking strategy for two reasons: First, it will determine whether the context is actually relevant to our prompt. Second, it will determine whether or not we'll be able to fit the retrieved text into the context before sending it to an outside model provider (e.g., OpenAI), given the limitations on the number of tokens we can send for each request. In some cases, like when using GPT-4 with a 32k context window, fitting the chunks might not be an issue. Still, we need to be mindful of when we're using very big chunks, as this may adversely affect the relevancy of the results we get back from Pinecone.

In this post, we'll explore several chunking methods and discuss the tradeoffs you should think about when choosing a chunking size and method. Finally, we'll give some recommendations for determining the best chunk size and method that will be appropriate for your application.

Embedding short and long content

When we embed our content, we can anticipate distinct behaviors depending on whether the content is short (like sentences) or long (like paragraphs or entire documents).

When a **sentence** is embedded, the resulting vector focuses on the sentence's specific meaning. The comparison would naturally be done on that level when compared to other sentence embeddings. This also implies that the embedding may miss out on broader contextual information found in a paragraph or document.

When a **full paragraph or document** is embedded, the embedding process considers both the overall context and the relationships between the sentences and phrases within the text. This can result in a more comprehensive vector representation that captures the broader meaning and themes of the text. Larger input text sizes, on the other hand, may introduce noise or dilute the significance of individual sentences or phrases, making finding precise matches when

querying the index more difficult.

en specifics and may be better suited for matching against sentence-level embeddings. A longer query that spans more than one sentence or a paragraph may be more in tune with embeddings at the paragraph or document level because it is likely looking for broader context or themes.

eous and contain embeddings for chunks of
es in terms of query result relevance, but it
ences. On the one hand, the relevance of
se of discrepancies between the semantic
ntent. On the other, a non-homogeneous
er range of context and information since
ent levels of granularity in the text. This
f queries more flexibly.

ining the best chunking strategy, and these
case. Here are some key aspects to keep in

1. **What is the nature of the content being indexed?** Are you working with long documents, such as articles or books, or shorter content, like tweets or instant messages? The answer would dictate both which model would be more suitable for your goal and, consequently, what chunking strategy to apply.
2. **Which embedding model are you using, and what chunk sizes does it perform optimally on?** For instance, sentence-transformer models work well on individual sentences, but a model like text-embedding-ada-002 performs better on chunks containing 256 or 512 tokens.
3. **What are your expectations for the length and complexity of user queries?** Will they be short and specific or long and complex? This may inform the way you choose to chunk your content as well so that there's a closer correlation between the embedded query and embedded chunks.
4. **How will the retrieved results be utilized within your specific application?** For example, will they be used for semantic search, question answering, summarization, or other purposes? For example, if your results need to be fed into another LLM with a token limit, you'll have to take that into consideration and limit the size of the chunks based on the number of chunks you'd like to fit into the request to the LLM.

Answering these questions will allow you to develop a chunking strategy that balances performance and accuracy, and this, in turn, will ensure the query results are more relevant.

Chunking methods

‹ing, and each of them might be
y examining the strengths and weaknesses
y the right scenario to apply them to.

forward approach to chunking: we simply
ıunk and, optionally, whether there should
eral, we will want to keep some overlap
e semantic context doesn't get lost
g will be the best path in most common
ıunking, fixed-sized chunking is
use since it doesn't require the use of any

Here's an example for 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])
```

"Content-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:

approach would be to split sentences by
e this may be fast and simple, this approach
possible edge cases. Here's a very simple

olkit (NLTK) is a popular Python library for
ata. It provides a sentence tokenizer that
s, 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)
```


text into smaller chunks in a hierarchical
parators. If the initial attempt at splitting
ıe desired size or structure, the method
ş chunks with a different separator or
ɔr structure is achieved. This means that
:actly the same size, they'll still "aspire" to

rsive chunking with LangChain:

```
ɔort RecursiveCharacterTextSplitter
terTextSplitter(
```

```
    # 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 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])
```


aration system and markup language often echnical documents. By parsing the LaTeX ou can create chunks that respect the ent (e.g., sections, subsections, and urate and contextually relevant results. For

```
ort LatexTextSplitter
```

```
latex_splitter = LatexTextSplitter(chunk_size=100, chunk_overlap=0)
docs = latex_splitter.create_documents([latex_text])
```

Semantic Chunking

A new experimental technique for approaching chunking was first introduced by Greg Kamradt. In his notebook, Kamradt rightfully points to the fact that a global chunking size may be too trivial of a mechanism to take into account the **meaning** of segments within the document. If we use this type of mechanism, we can't know if we're combining segments that have anything to do with one another.

Luckily, if you're building an application with LLMs, you most likely already have the ability to create **embeddings -** and embeddings can be used to extract the semantic meaning present in your data. This semantic analysis can be used to create chunks that are made up sentences that talk about the same theme or topic.

Here are the steps that make semantic chunking work:

1. Break up the document into sentences.
2. Create sentence groups: for each sentence, create a group containing some sentences before and after the given sentence. The group is essentially "anchored" by the sentence use to create it. You can decide the specific numbers before or after to include in each group - but all sentences in a

ıe "anchor" sentence.
sentence group and associate them with

ch group sequentially: When you look at the
uentially, as long as the topic or theme is
n the sentence group embedding for a
e group preceding it will be **low**. On the
tance indicates that the theme or topic has
elineate one chunk from the next.

unking splitter implemented based on
our notebook for advanced chunking

your application

Here are some pointers to help you come up with an optimal chunk size if the common chunking approaches, like fixed chunking, don't easily apply to your use case.

- **Preprocessing your Data** - You need to first pre-process your data to ensure quality before determining the best chunk size for your application. For example, if your data has been retrieved from the web, you might need to remove HTML tags or specific elements that just add noise.
- **Selecting a Range of Chunk Sizes** - Once your data is preprocessed, 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** - In order 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.

.e in most cases - but it could present some
off the beaten path. There's no one-size-
works for one use case may not work for
you get a better intuition for how to
on.

RECOMMENDED FOR YOU

Further Reading

Mar 5, 2025

Don’t be dense: Launching sparse indexes in Pinecone

9 min read

Mar 5, 2025

Unlock High-Precision Keyword Search with pinecone-sparse-english-v0

18 min read

Feb 24, 2025

:h citation highlights in Pinecone