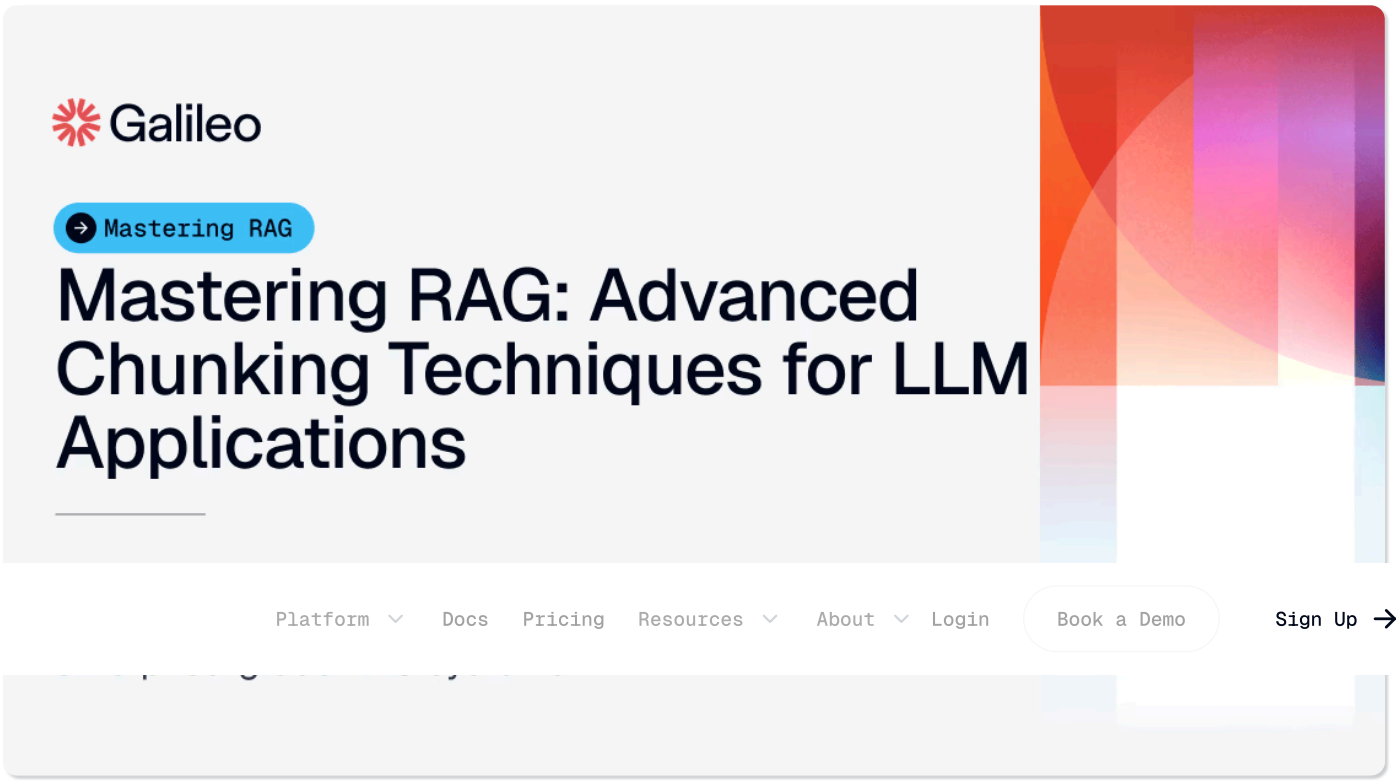
Back

Feb 23, 2024 · 21 min read

Mastering RAG: Advanced Chunking Techniques for LLM Applications

Pratik Bhavsar
Galileo Labs



Explore our research-backed evaluation metric for RAG – read our paper on Chainpoll.

CONTENTS

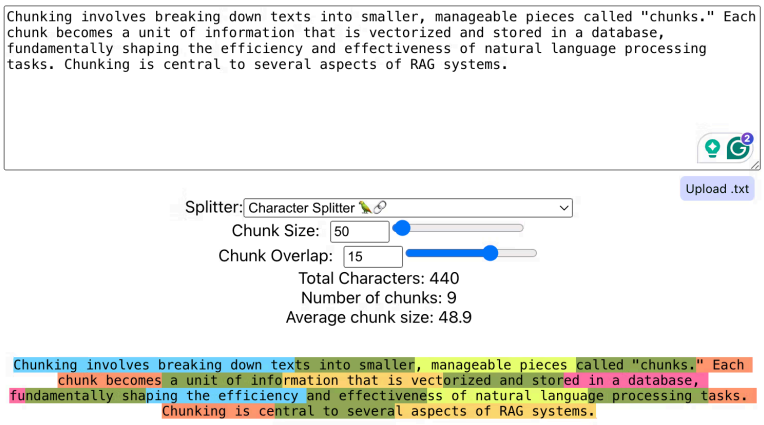
What is chunking?
Impact of chunking
Factors influencing chunking
Types of chunking
Text splitter
Character splitter
Recursive Character Splitter
Sentence splitter
Semantic splitting
LLM based chunking
Document specific splitting
How to measure chunking effectiveness
Experiment
Conclusion

SHARE

What is chunking?

Chunking involves breaking down texts into smaller, manageable pieces called "chunks." Each chunk becomes a unit of information that is vectorized and stored in a database, fundamentally shaping the efficiency and effectiveness of natural language processing tasks.

Here is an example of how the character splitter splits the above paragraph into chunks using Chunkviz.



Impact of chunking

Chunking plays a central role in various aspects of RAG systems, exerting influence not only on retrieval quality but also on response. Let's understand these aspects in more detail.

Retrieval quality

The primary objective of chunking is to enhance the retrieval quality of information from vector databases. By defining the unit of information that is stored, chunking allows for retrieval of the most relevant information needed for the task.

Vector database cost

Efficient chunking techniques help optimize storage by balancing granularity. The cost of storage grows linearly with the number of chunks, therefore chunks should be as large as possible to keep costs low.

Vector database query latency

Maintaining low latency is essential for real-time applications. Minimizing the number of chunks reduces latency.

LLM latency and cost

The mind-blowing capabilities of LLMs come at a considerable price. Improved context from larger chunk sizes increases latency and serving costs.

LLM hallucinations

While adding more context may seem better, excessive context can lead to hallucinations in LLMs. Choosing the right chunking size for the job plays a large role in determining the quality of generated content. Balancing contextual richness with retrieval precision is essential to prevent hallucinatory outputs and ensure a coherent and accurate generation process.

Factors influencing chunking

We understand the importance of taking chunking seriously, but what factors influence it? A better understanding of these parameters will enable us to select an appropriate strategy.

Text structure

The text structure, whether it's a sentence, paragraph, code, table, or transcript, significantly impacts the chunk size. Understanding how structure relates to the type of content will help influence chunking strategy.

Embedding model

The capabilities and limitations of the embedding model play a crucial role in defining chunk size. Factors such as the model's context input length and its ability to maintain high-quality embeddings guide the optimal chunking strategy.

LLM context length

LLMs have finite context windows. Chunk size directly affects how much context can be fed into the LLM. Due to context length limitations, large chunks force the user to keep the top k in retrieval as low as possible.

Type of questions

The questions that users will be asking helps determine the chunking techniques best suited for your use case. Specific factual questions, for instance, may require a different chunking approach than complex questions which will require information from multiple chunks.

Types of chunking

As you see, selecting the right chunk size involves a delicate balance of multiple factors. There is no one-size-fits-all approach, emphasizing the importance of finding a chunking technique tailored to the RAG application's specific needs.

Let's explore common chunking techniques to help AI builders optimize their RAG performance.

Text splitter

Let's first understand the base class used by all Langchain splitters. The **_merge_splits** method of the **TextSplitter** class is responsible for combining smaller pieces of text into medium-sized chunks. It takes a sequence of text splits and a separator, and then iteratively merges these splits into chunks, ensuring that the combined size of the chunks is within specified limits.

The method uses **chunk_size** and **chunk_overlap** to determine the maximum size of the resulting chunks and their allowed overlap. It also considers factors such as the length of the separator and whether to strip whitespace from the chunks.

The logic maintains a running total of the length of the chunks and the separator. As splits are added to the current chunk, the method checks if adding a new split would

exceed the specified chunk size. If so, it creates a new chunk, considering the chunk overlap, and removes splits from the beginning to meet size constraints.

This process continues until all splits are processed, resulting in a list of merged chunks. The method ensures that the chunks are within the specified size limits and handles edge cases such as chunks longer than the specified size by issuing a warning.

Source: https://github.com/langchain-ai/langchain/blob/master/libs/langchain/langchain/text_splitter.py#L99

```
from abc import ABC, abstractmethod

class TextSplitter(BaseDocumentTransformer, ABC):
    """Interface for splitting text into chunks."""

    def __init__(
        self,
        chunk_size: int = 4000,
        chunk_overlap: int = 200,
        length_function: Callable[[str], int] = len,
        keep_separator: bool = False,
        add_start_index: bool = False,
        strip_whitespace: bool = True,
    ) -> None:
        """Create a new TextSplitter.

        Args:
            chunk_size: Maximum size of chunks to return
            chunk_overlap: Overlap in characters between chunks
            length_function: Function that measures the length of giv
            keep_separator: Whether to keep the separator in the chun
            add_start_index: If True, includes chunk's start index in
            strip_whitespace: If True, strips whitespace from the sta
                              every document
        """
        if chunk_overlap > chunk_size:
            raise ValueError(
                f"Got a larger chunk overlap ({chunk_overlap}) than cl
                f"({chunk_size}), should be smaller."
            )
        self._chunk_size = chunk_size
        self._chunk_overlap = chunk_overlap
        self._length_function = length_function
        self._keep_separator = keep_separator
        self._add_start_index = add_start_index
        self._strip_whitespace = strip_whitespace

    @abstractmethod
    def split_text(self, text: str) -> List[str]:
        """Split text into multiple components."""

    def _join_docs(self, docs: List[str], separator: str) -> Optional
        text = separator.join(docs)
        if self._strip_whitespace:
            text = text.strip()
        if text == "":
            return None
        else:
            return text

    def _merge_splits(self, splits: Iterable[str], separator: str) ->
        # We now want to combine these smaller pieces into medium siz
        # chunks to send to the LLM.
        separator_len = self._length_function(separator)

        docs = []
        current_doc: List[str] = []
        total = 0
```

```
            for d in splits:
                _len = self._length_function(d)
                if (
                    total + _len + (separator_len if len(current_doc) > 0
                    > self._chunk_size
                ):
                    if total > self._chunk_size:
                        logger.warning(
                            f"Created a chunk of size {total}, "
                            f"which is longer than the specified {self._cl
                        )
                    if len(current_doc) > 0:
                        doc = self._join_docs(current_doc, separator)
                        if doc is not None:
                            docs.append(doc)
                        # Keep on popping if:
                        # - we have a larger chunk than in the chunk over
                        # - or if we still have any chunks and the length
                        while total > self._chunk_overlap or (
                            total + _len + (separator_len if len(current_
                            > self._chunk_size
                            and total > 0
                        ):
                            total -= self._length_function(current_doc[0]
                                separator_len if len(current_doc) > 1 else
                            )
                            current_doc = current_doc[1:]
                current_doc.append(d)
                total += _len + (separator_len if len(current_doc) > 1 el
            doc = self._join_docs(current_doc, separator)
            if doc is not None:
                docs.append(doc)
            return docs
```

Character splitter

Langchain's **CharacterTextSplitter** class is responsible for breaking down a given text into smaller chunks. It uses a separator such as "\n" to identify points where the text should be split.

The method first splits the text using the specified separator and then merges the resulting splits into a list of chunks. The size of these chunks is determined by parameters like chunk_size and chunk_overlap defined in the parent class TextSplitter.

Source: https://github.com/langchain-ai/langchain/blob/master/libs/langchain/langchain/text_splitter.py#L289

```
    class CharacterTextSplitter(TextSplitter):
        """Splitting text that looks at characters."""

        def __init__(
            self, separator: str = "\n\n", is_separator_regex: bool = Fal
        ) -> None:
            """Create a new TextSplitter."""
            super().__init__(**kwargs)
            self._separator = separator
            self._is_separator_regex = is_separator_regex

        def split_text(self, text: str) -> List[str]:
            """Split incoming text and return chunks."""
            # First we naively split the large input into a bunch of smal
            separator = (
                self._separator if self._is_separator_regex else re.escap
            )
            splits = _split_text_with_regex(text, separator, self._keep_s
```

```
        _separator = "" if self._keep_separator else self._separator
        return self._merge_splits(splits, _separator)
```

Let's analyze its behavior with an example text. The empty string separator “” treats each character as a splitter. Subsequently, it combines the splits according to chunk size and chunk overlap.

```
text = "This is first. This is second. This is third. This is fourth.

splitter = CharacterTextSplitter(separator="", chunk_size=40, chunk_o
splitter.split_text(text)

Output:
['This is first. This is second. This is t',
 'is second. This is third. This is fourth',
 'hird. This is fourth. This is fifth.\n\nTh',
 '. This is fifth.\n\nThis is sixth. This is',
 'is is sixth. This is seventh. This is ei',
 'seventh. This is eighth. This is ninth.',
 'ghth. This is ninth. This is tenth.']
```

Recursive Character Splitter

Langchain’s **RecursiveCharacterTextSplitter** class is designed to break down a given text into smaller chunks by recursively attempting to split it using different separators. This class is particularly useful when a single separator may not be sufficient to identify the desired chunks.

The method starts by trying to split the text using a list of potential separators specified in the _separators attribute. It iteratively checks each separator to find the one that works for the given text. If a separator is found, the text is split, and the process is repeated recursively on the resulting chunks until the chunks are of a manageable size.

The separators are listed in descending order of preference, and the method attempts to split the text using the most specific ones first. For example, in the context of the Python language, it tries to split along class definitions ("\nclass "), function definitions ("\ndef "), and other common patterns. If a separator is found, it proceeds to split the text recursively.

The resulting chunks are then merged and returned as a list. The size of the chunks is determined by parameters like chunk_size and chunk_overlap defined in the parent class TextSplitter. This approach allows for a more flexible and adaptive way of breaking down a text into meaningful sections.

Source: https://github.com/langchain-ai/langchain/blob/master/libs/langchain/langchain/text_splitter.py#L851

```
class RecursiveCharacterTextSplitter(TextSplitter):
    """Splitting text by recursively look at characters.

    Recursively tries to split by different characters to find one
    that works.
    """

    def __init__(
        self,
        separators: Optional[List[str]] = None,
        keep_separator: bool = True,
```

```
        is_separator_regex: bool = False,
        **kwargs: Any,
    ) -> None:
        """Create a new TextSplitter."""
        super().__init__(keep_separator=keep_separator, **kwargs)
        self._separators = separators or ["\n\n", "\n", " ", ""]
        self._is_separator_regex = is_separator_regex

    def _split_text(self, text: str, separators: List[str]) -> List[s
        """Split incoming text and return chunks."""
        final_chunks = []
        # Get appropriate separator to use
        separator = separators[-1]
        new_separators = []
        for i, _s in enumerate(separators):
            _separator = _s if self._is_separator_regex else re.escap
            if _s == "":
                separator = _s
                break
            if re.search(_separator, text):
                separator = _s
                new_separators = separators[i + 1 :]
                break

        _separator = separator if self._is_separator_regex else re.es
        splits = _split_text_with_regex(text, _separator, self._keep_

        # Now go merging things, recursively splitting longer texts.
        _good_splits = []
        _separator = "" if self._keep_separator else separator
        for s in splits:
            if self._length_function(s) < self._chunk_size:
                _good_splits.append(s)
            else:
                if _good_splits:
                    merged_text = self._merge_splits(_good_splits, _s
                    final_chunks.extend(merged_text)
                    _good_splits = []
                if not new_separators:
                    final_chunks.append(s)
                else:
                    other_info = self._split_text(s, new_separators)
                    final_chunks.extend(other_info)
        if _good_splits:
            merged_text = self._merge_splits(_good_splits, _separator
            final_chunks.extend(merged_text)
        return final_chunks

    def split_text(self, text: str) -> List[str]:
        return self._split_text(text, self._separators)

    @classmethod
    def from_language(
        cls, language: Language, **kwargs: Any
    ) -> RecursiveCharacterTextSplitter:
        separators = cls.get_separators_for_language(language)
        return cls(separators=separators, is_separator_regex=True, **

    @staticmethod
    def get_separators_for_language(language: Language) -> List[str]:
        if language == Language.PYTHON:
            return [
                # First, try to split along class definitions
                "\nclass ",
                "\ndef ",
                "\n\tdef ",
                # Now split by the normal type of lines
                "\n\n",
                "\n",
                " ",
                "",
```

In this example, the text is initially divided based on "\n\n" and subsequently on "" to adhere to the specified chunk size and overlap. The separator "\n\n" is removed from the text.

```
text = "This is first. This is second. This is third. This is fourth.

splitter = RecursiveCharacterTextSplitter(separators=["\n\n", ""], ch
splitter.split_text(text)


Output:
['This is first. This is second. This is t',
 'is second. This is third. This is fourth',
 'hird. This is fourth. This is fifth.',
 'This is sixth. This is seventh. This i',
 's is seventh. This is eighth. This is ni',
 's eighth. This is ninth. This is tenth.']
```

You can also employ the above method to chunk code from various languages by utilizing well-defined separators specific to each language.

Sentence splitter

Character splitting poses an issue as it tends to cut sentences midway. Despite attempts to address this using chunk size and overlap, sentences can still be cut off prematurely. Let's explore a novel approach that considers sentence boundaries instead.

The **SpacySentenceTokenizer** takes a piece of text and divides it into smaller chunks, with each chunk containing a certain number of sentences. It uses the Spacy library to analyze the input text and identify individual sentences.

The method allows you to control the size of the chunks by specifying the stride and overlap parameters. The stride determines how many sentences are skipped between consecutive chunks, and the overlap determines how many sentences from the previous chunk are included in the next one.

```
from typing import List, Optional
from langchain_core.documents import Document
import spacy


class SpacySentenceTokenizer:
    def __init__(self, spacy_model="en_core_web_sm"):
        self.nlp = spacy.load(spacy_model)

    def create_documents(
        self, documents, metadatas=None, overlap: int = 0, stride: int
    )→ List[Document]:
        chunks = []
        if not metadatas:
            metadatas = [{}] * len(documents)
        for doc, metadata in zip(documents, metadatas):
            text_chunks = self.split_text(doc, overlap, stride)
            for chunk_text in text_chunks:
                chunks.append(Document(page_content=chunk_text, metada
        return chunks

    def split_text(self, text: str, stride: int = 1, overlap: int = 1)
```

```
        sentences = list(self.nlp(text).sents)
        chunks = []
        for i in range(0, len(sentences), stride):
            chunk_text = " ".join(str(sent) for sent in sentences[i :
            chunks.append(chunk_text)
        return chunks
```

The example below shows how a text with pronouns like “they” requires the context of the previous sentence to make sense. Our brute force overlap approach helps here but is also redundant at some places and leads to longer chunks 😵

```
text = "I love dogs. They are amazing. Cats must be the easiest pets a

tokenizer = SpacySentenceTokenizer()
tokenizer.split_text(text, stride=1, overlap=2)

Output:
['I love dogs. They are amazing. Cats must be the easiest pets around.
 'They are amazing. Cats must be the easiest pets around. Tesla robots
 'Cats must be the easiest pets around. Tesla robots are advanced now
 'Tesla robots are advanced now with AI. They will take us to mars.',
 'They will take us to mars.']
```

Semantic splitting

This brings us to my favorite splitting method!

The **SimilarSentenceSplitter** takes a piece of text and divides it into groups of sentences based on their similarity. It utilizes a similarity model to measure how similar each sentence is to its neighboring sentences. The method uses a sentence splitter to break the input text into individual sentences.

The goal is to create groups of sentences where each group contains related sentences, according to the specified similarity model. The method starts with the first sentence in the first group and then iterates through the remaining sentences. It decides whether to add a sentence to the current group based on its similarity to the previous sentence.

The **group_max_sentences** parameter controls the maximum number of sentences allowed in each group. If a group reaches this limit, a new group is started. Additionally, a new group is initiated if the similarity between consecutive sentences falls below a specified similarity_threshold.

In simpler terms, this method organizes a text into clusters of sentences, where sentences within each cluster are considered similar to each other. It's useful for identifying coherent and related chunks of information within a larger body of text.

Credits: https://github.com/agamm/semantic-split

```
from typing import List

from sentence_transformers import SentenceTransformer, util
import spacy


class SentenceTransformersSimilarity:
    def __init__(self, model="all-MiniLM-L6-v2", similarity_threshold=
        self.model = SentenceTransformer(model)
        self.similarity_threshold = similarity_threshold
```

```
        def similarities(self, sentences: List[str]):
            # Encode all sentences
            embeddings = self.model.encode(sentences)

            # Calculate cosine similarities for neighboring sentences
            similarities = []
            for i in range(1, len(embeddings)):
                sim = util.pytorch_cos_sim(embeddings[i - 1], embeddings[i
                similarities.append(sim)

            return similarities


    class SpacySentenceSplitter():

        def __init__(self):
            self.nlp = spacy.load("en_core_web_sm")

        def split(self, text: str) -> List[str]:
            doc = self.nlp(text)
            return [str(sent).strip() for sent in doc.sents]


    class SimilarSentenceSplitter():

        def __init__(self, similarity_model, sentence_splitter):
            self.model = similarity_model
            self.sentence_splitter = sentence_splitter

        def split_text(self, text: str, group_max_sentences=5) -> List[str
            """
            group_max_sentences: The maximum number of sentences in a grou
            """
            sentences = self.sentence_splitter.split(text)

            if len(sentences) == 0:
                return []

            similarities = self.model.similarities(sentences)

            # The first sentence is always in the first group.
            groups = [[sentences[0]]]

            # Using the group min/max sentences contraints,
            # group together the rest of the sentences.
            for i in range(1, len(sentences)):
                if len(groups[-1]) >= group_max_sentences:
                    groups.append([sentences[i]])
                elif similarities[i - 1] >= self.model.similarity_threshol
                    groups[-1].append(sentences[i])
                else:
                    groups.append([sentences[i]])

            return [" ".join(g) for g in groups]
```

Now, let's apply semantic chunking to the text. The model appropriately connects sentences related to the dog and robot, referred to as "they" in the sentences, factoring in previous context. This approach allows us to overcome various challenges, including setting chunk size, managing overlap, and avoiding cropping in the middle.

```
    text = "I love dogs. They are amazing. Cats must be the easiest pets a

    model = SentenceTransformersSimilarity() # emb model
    sentence_splitter = SpacySentenceSplitter() # sentence tokenizer
    splitter = SimilarSentenceSplitter(model, sentence_splitter)
    splitter.split_text(text)

    Output:
```

```
['I love dogs. They are amazing.',
 'Cats must be the easiest pets around.',
 'Tesla robots are advanced now with AI. They will take us to mars.']
```

This covers all the major practical text chunking methods commonly used in production. Now let's look at one final approach to chunking for complicated documents. Or, you can download our comprehensive, end-to-end guide on building enterprise RAG systems.

LLM based chunking

These popular methods are all fine and good, but can we push them further? Let's use the power of LLMs to go beyond traditional chunking!

Propositions

Unlike the conventional use of passages or sentences, a new paper Dense X Retrieval: What Retrieval Granularity Should We Use? introduces a novel retrieval unit for dense retrieval called "propositions." Propositions are atomic expressions within text, each encapsulating a distinct factoid and presented in a concise, self-contained natural language format.

The three principles below define propositions as atomic expressions of meanings in text:

- Each proposition should represent a distinct piece of meaning in the text, collectively embodying the semantics of the entire text.
- A proposition must be minimal and cannot be further divided into separate propositions.
- A proposition should contextualize itself and be self-contained, encompassing all the necessary context from the text (e.g., coreference) to interpret its meaning.

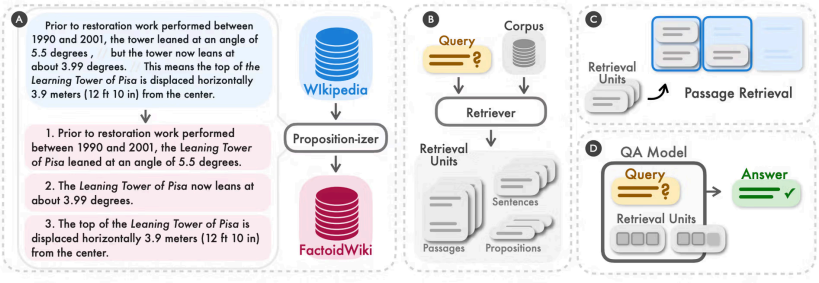


Figure 2: We discover that segmenting and indexing a retrieval corpus on the *proposition* level can be a simple yet effective strategy to increase dense retrievers' generalization performance at inference time *(A, B)*. We empirically compare the retrieval and downstream open-domain QA tasks performance when dense retrievers work with Wikipedia indexed at the level of 100-word passage, sentence or proposition *(C, D)*.

Let's put this novel approach into practice. The authors themselves trained the propositionizer model used in the code snippet below. As evident in the output, the model successfully performs coreference resolution, ensuring each sentence is self-sufficient.

```
from transformers import AutoTokenizer, AutoModelForSeq2SeqLM
import torch
import json

model_name = "chentong00/propositionizer-wiki-flan-t5-large"
device = "cuda" if torch.cuda.is_available() else "cpu"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForSeq2SeqLM.from_pretrained(model_name).to(device)

input_text = "I love dogs. They are amazing. Cats must be the easiest

input_ids = tokenizer(input_text, return_tensors="pt").input_ids
outputs = model.generate(input_ids.to(device), max_new_tokens=512).cpu

output_text = tokenizer.decode(outputs[0], skip_special_tokens=True)
try:
    prop_list = json.loads(output_text)
except:
    prop_list = []
    print("[ERROR] Failed to parse output text as JSON.")
print(json.dumps(prop_list, indent=2))

Output:
I love dogs. Dogs are amazing. Cats are the easiest pets around. Tesla
```

Multi-vector indexing

Another approach involves multi-vector indexing, where semantic search is performed for a vector derived from something other than the raw text. There are various methods to create multiple vectors per document.

Smaller chunks

Divide a document into smaller chunks and embed them (referred to as ParentDocumentRetriever).

Summary

Generate a summary for each document and embed it along with, or instead of, the document.

Hypothetical questions

Form hypothetical questions that each document would be appropriate to answer, and embed them along with, or instead of, the document.

Each of these utilizes either a text2text or an LLM with a prompt to obtain the necessary chunk. The system then indexes both the newly generated chunk and the original text, improving the recall of the retrieval system. You can find more details of these techniques in Langchain's official documentation.

Document specific splitting

Is parsing documents *really* chunking? Despite my hesitation, let's cover this related topic.

Remember the buzz around "Chat with PDF"? Things are easy with plain text, but once we introduce tables, images, and unconventional formatting things get a lot more complicated. And what about .doc, .epub, and .xml formats as well? This leads us to a very powerful library...

Unstructured, with its diverse set of document type support and flexible partitioning strategies, offers several benefits for reading documents efficiently.

Supports all major document types

Unstructured supports a wide range of document types, including .pdf, .docx, .doc, .odt, .pptx, .ppt, .xlsx, .csv, .tsv, .eml, .msg, .rtf, .epub, .html, .xml, .png, .jpg, and .txt files. This ensures that users can seamlessly work with different file formats within a unified framework.

Adaptive partitioning

The "auto" strategy in Unstructured provides an adaptive approach to partitioning. It automatically selects the most suitable partitioning strategy based on the characteristics of the document. This feature simplifies the user experience and optimizes the processing of documents without the need for manual intervention in selecting partitioning strategies.

Specialized strategies for varied use cases

Unstructured provides specific strategies for different needs. The "fast" strategy quickly extracts information using traditional NLP techniques, "hi_res" ensures precise classification using detectron2 and document layout, and "ocr_only" is designed specifically for Optical Character Recognition in image-based files. These strategies accommodate various use cases, offering users flexibility and precision in their document processing workflows.

Unstructured's comprehensive document type support, adaptive partitioning strategies, and customization options make it a powerful tool for efficiently reading and processing a diverse range of documents.

Let's run an example and see how it partitions the Gemini 1.5 technical report.

```
elements = partition_pdf(
    filename=filename,

    # Unstructured Helpers
    strategy="hi_res",
    infer_table_structure=True,
    model_name="yolox"
)

Output:
[<unstructured.documents.elements.Image at 0x2acfc24d0>,
 <unstructured.documents.elements.Title at 0x2d4562c50>,
 <unstructured.documents.elements.NarrativeText at 0x2d4563b50>,
 <unstructured.documents.elements.NarrativeText at 0x2d4563350>,
 <unstructured.documents.elements.Title at 0x2d4560b90>,
 <unstructured.documents.elements.NarrativeText at 0x2d4562350>,
 <unstructured.documents.elements.NarrativeText at 0x2d4561b10>,
 <unstructured.documents.elements.NarrativeText at 0x2d4562410>,
 <unstructured.documents.elements.NarrativeText at 0x2d45620d0>,
 <unstructured.documents.elements.Header at 0x2d4562110>,
 <unstructured.documents.elements.NarrativeText at 0x2d4560a50>,
 <unstructured.documents.elements.Title at 0x2a58e3090>,
 <unstructured.documents.elements.Image at 0x2d4563d90>,
 <unstructured.documents.elements.FigureCaption at 0x2d4563c90>,
 <unstructured.documents.elements.NarrativeText at 0x2d4563150>,
 <unstructured.documents.elements.NarrativeText at 0x2d4562290>,
 <unstructured.documents.elements.Footer at 0x2d4563e90>,
 <unstructured.documents.elements.Header at 0x2d4562790>,
```

```
<unstructured.documents.elements.Table at 0x2d4561ed0>,
<unstructured.documents.elements.Title at
```

It effectively extracted various sections from the PDF and organized them into distinct elements. Now, let's examine the data to confirm if we successfully parsed the table below.

	Context length	AutoAIS Gemini 1.5 Pro	AIS Human Evaluation	Num. Sentences per answer
Anthropic Claude 2.1	0-shot	11.1	30.2	5.7
Gemini 1.0 Pro	0-shot	85.3	79.1	2.3
Gemini 1.5 Pro	0-shot	82.1	75.5	3.4
Anthropic Claude 2.1	4k retrieved	29.1	42.2	5.1
Gemini 1.0 Pro	4k retrieved	75.3	72.1	2.6
Gemini 1.5 Pro	4k retrieved	84.8	78.2	4.9
Gemini 1.5 Pro	710k book	**91.4**	**80.0**	5.8

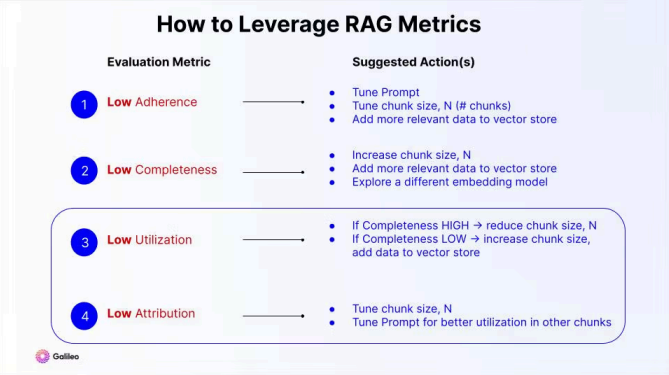
Look how similar the two tables are! It is able to identify the columns & rows to generate the table in HTML format. This makes it easier for us to do tabular QnA!

```
table = elements[149].metadata.text_as_html
from IPython.display import display, HTML
display(HTML(table))
```

[43] ✓ 0.0s

	Context length	AutoAIS Gemini 1.5 Pro	AIS Human Evaluation	Num. Sentences per answer
Anthropic Claude 2.1	0-shot	11.1	30.2	5.7
Gemini 1.0 Pro	0-shot	85.3	79.1	2.3
Gemini 1.5 Pro	0-shot	82.1	75.5	3.4
Anthropic Claude 2.1	4k retrieved	29.1	42.2	5.1
Gemini 1.0 Pro	4k retrieved	75.3	72.1	2.6
Gemini 1.5 Pro	4k retrieved	84.8	78.2	4.9
Gemini 1.5 Pro	710k book	91.4	80.0	5.8

How to measure chunking effectiveness



To optimize RAG performance, improving retrieval with effective chunking is crucial. Here are two chunk evaluation metrics to help you debug RAG faster.

Chunk Attribution

Chunk attribution evaluates whether each retrieved chunk influences the model's response. It employs a binary metric, categorizing each chunk as either Attributed or Not Attributed. Chunk Attribution is closely linked to Chunk Utilization (see below), with Attribution determining if a chunk impacted the response and Utilization measuring the extent of the chunk's text involved in the effect. Only Attributed chunks can have Utilization scores greater than zero.

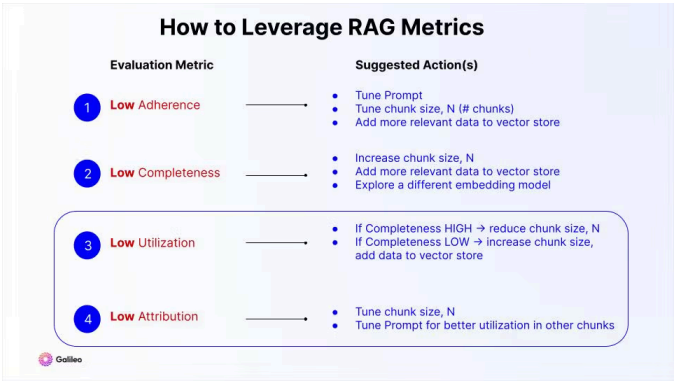
Chunk Attribution helps pinpoint areas for improvement in RAG systems, such as adjusting the number of retrieved chunks. If the system provides satisfactory responses but has many Not Attributed chunks, reducing the retrieved chunks per example may enhance system efficiency, leading to lower costs and latency.

Additionally, when investigating individual examples with unusual or unsatisfactory model behavior, Attribution helps identify the specific chunks influencing the response, facilitating quicker troubleshooting.

Chunk Utilization

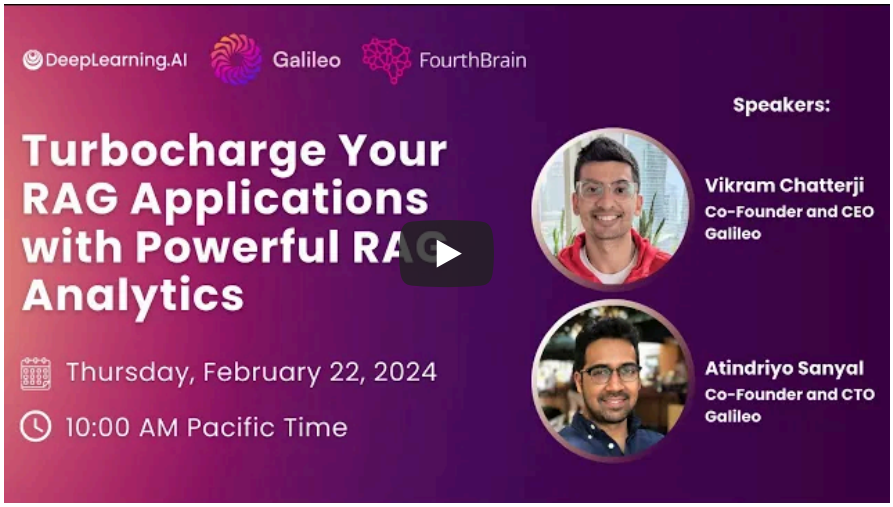
Chunk Utilization gauges the fraction of text in each retrieved chunk that impacts the model's response. This metric ranges from 0 to 1, where a value of 1 indicates the entire chunk affected the response, while a lower value, such as 0.5, signifies the presence of "extraneous" text that did not impact the response. Chunk Utilization is intricately connected to Chunk Attribution, as Attribution determines if a chunk affected the response, while Utilization measures the portion of the chunk text involved in the effect. Only Attributed chunks can have Utilization scores greater than zero.

Low Chunk Utilization scores suggest that chunks may be longer than necessary. In such cases, reducing the parameters that control the chunk size is recommended to enhance system efficiency by lowering costs and latency.



Experiment

Learn more about how to utilize metrics in this webinar where we build a Q&A over earning call transcripts using Langchain. If you are looking for an end2end example with code, have a look at this blog.



Conclusion

In summary, effective chunking is crucial for optimizing RAG systems. It ensures accurate information retrieval and influences factors like response latency and storage costs. Choosing the right chunking strategy involves considering multiple aspects but can be done easily with metrics like Chunk Attribution and Chunk Utilization.

Galileo's RAG analytics offer a transformative approach, providing unparalleled visibility into RAG systems and simplifying evaluation to improve RAG performance. Sign up for your free Galileo account today, or continue your Mastering RAG journey with our free, comprehensive eBook.