**Retrieval-Augmented Generation (RAG)**

**Introduction**

Retrieval-Augmented Generation (RAG) is a framework that combines the strengths of retrieval-based and generation-based models to enhance the performance of natural language processing tasks. RAG leverages external knowledge sources or custom data to retrieve relevant information, which is then used to generate more accurate and contextually appropriate responses.

RAG works by combining LLMs with a retrieval system. When a query is made, instead of solely relying on the LLM's training, RAG first consults a content store, which can be either an open source like the internet or a closed collection of documents. This process helps in retrieving the most current and relevant information. This method not only provides up-to-date answers but also includes source information, reducing the likelihood of "hallucinating" responses or giving incorrect information.

**GenAI challenges to solve:**

Generative AI is [the most impactful technology of the last decade, according to Gartner](https://snorkel.ai/genai-most-impactful-tech-of-the-decade-gartner-ai-hype-cycle-2023/), but the field is still in its infancy. Generative models available today present a number of challenges—some of them significant.

1. **Out-of-date information.** At the time of this writing, ChatGPT warned users that its pre-training data contains no information after September 2021. Updating a pre-training corpus requires a lot of time and effort, and likely won’t be a regular process for the foreseeable future.
2. **Lack of domain knowledge.** Generative applications built atop [foundation models](https://snorkel.ai/foundation-models/) (FMs[)](https://snorkel.ai/foundation-models/) often contain a wealth of general knowledge but struggle with tasks focussed on narrow domains or specializations.
3. **Hallucinations.** Generative models respond confidently at all times, even when their pre-training data does not cover the topic at hand. The models sometimes fabricate plausible nonsense, which can lead users astray.
4. **Poor performance on specific tasks.** Generalist models handle an impressive spread of tasks surprisingly well. But they may struggle on specific tasks important to particular data teams.

**RAG Architecture**

RAG typically consists of two main components:

**1.** **Retriever**: A model that searches for relevant documents or pieces of information from a large corpus based on the input query.

**2. Generator**: A sequence-to-sequence model that generates responses by conditioning on both the input query and the retrieved documents.

**RAG Implementation Approach**

**1. Corpus Preparation:** Ensure the knowledge corpus is comprehensive and well-organized.

- Regularly update the corpus to maintain the relevance of retrieved information.

**2. Retrieval Efficiency:** Optimize retrieval algorithms for speed and accuracy.

- Consider using indexing techniques to improve retrieval times.

**3. Integration with Downstream Tasks:** Tailor the RAG system to specific tasks, such as question answering, document summarization, or conversational agents.

- Fine-tune the generator on task-specific data to enhance performance.

**4. Evaluation Metrics:** Use appropriate metrics to evaluate both retrieval and generation components.

- Common metrics include precision, recall, F1-score for retrieval, and BLEU, ROUGE, and METEOR for generation.

**Use Cases**

Question and answer chatbots, Content Creation, Search augmentation, Language Translation, Knowledge engine

**RAG Types**

**Naïve RAG:**Naive RAG follows the traditional process that includes indexing, retrieving, augmenting and generation of response. A user input is used to query relevant documents which are then combined with a prompt and passed to the model to generate a final response. Conversational history can be integrated into the prompt if the application involves multi-turn dialogue interactions. Naive RAG has limitations such as low precision (misaligned retrieved chunks) and low recall (failure to retrieve all relevant chunks).

**Advanced RAG:** Advanced RAG models incorporate additional techniques to further enhance the performance such as improving retrieval quality that could involve optimizing the pre-retrieval, retrieval, and post-retrieval processes.

The pre-retrieval process involves optimizing data indexing which aims to enhance the quality of the data being indexed through five stages: enhancing data granularity, optimizing index structures, adding metadata, alignment optimization, and mixed retrieval.

The retrieval stage can be further improved by optimizing the embedding model itself which directly impacts the quality of the chunks that make up the context. This can be done by fine-tuning the embedding to optimize retrieval relevance or employing dynamic embeddings that better capture contextual understanding (e.g., OpenAI’s embeddings-ada-02 model).

Optimizing post-retrieval focuses on avoiding context window limits by prompt compression and dealing with noisy or potentially distracting information. A common approach to address these issues is re-ranking which could involve approaches such as relocation of relevant context to the edges of the prompt or recalculating the semantic similarity between the query and relevant text chunks.

**Modular RAG:** Modular RAG enhances functional modules such as incorporating a search module for similarity retrieval and applying fine-tuning in the retriever. Both Naive RAG and Advanced RAG are special cases of Modular RAG and are made up of fixed modules.

Extended RAG modules include search, memory, fusion, routing, predict, and task adapter which solve different problems. These modules can be rearranged to suit specific problem contexts. Therefore, Modular RAG benefits from greater diversity and flexibility in that you can add or replace modules or adjust the flow between modules based on task requirements.

Some optimization techniques include Hybrid Search, Recursive Retrieval and Querying, Step-back approach, Sub-queries, Hypothetical Document Embeddings.

**Retrievers**

Retrievers are critical in RAG systems, as they determine the quality of information fed into the generator.

**1. Sparse Retrievers:** BM25, TF-IDF.

- Operate on exact keyword matching and term frequencies.

- Fast and interpretable but may miss semantically relevant documents.

**2. Dense Retrievers:** Dense Passage Retrieval (DPR), Sentence-BERT.

- Use neural embeddings to capture semantic similarities.

- More effective in finding contextually relevant documents but require significant computational resources.

**3. Hybrid Retrievers:** Combines sparse and dense retrieval methods.

- Aim to leverage the strengths of both approaches to improve retrieval performance.

**Fine-tuning**

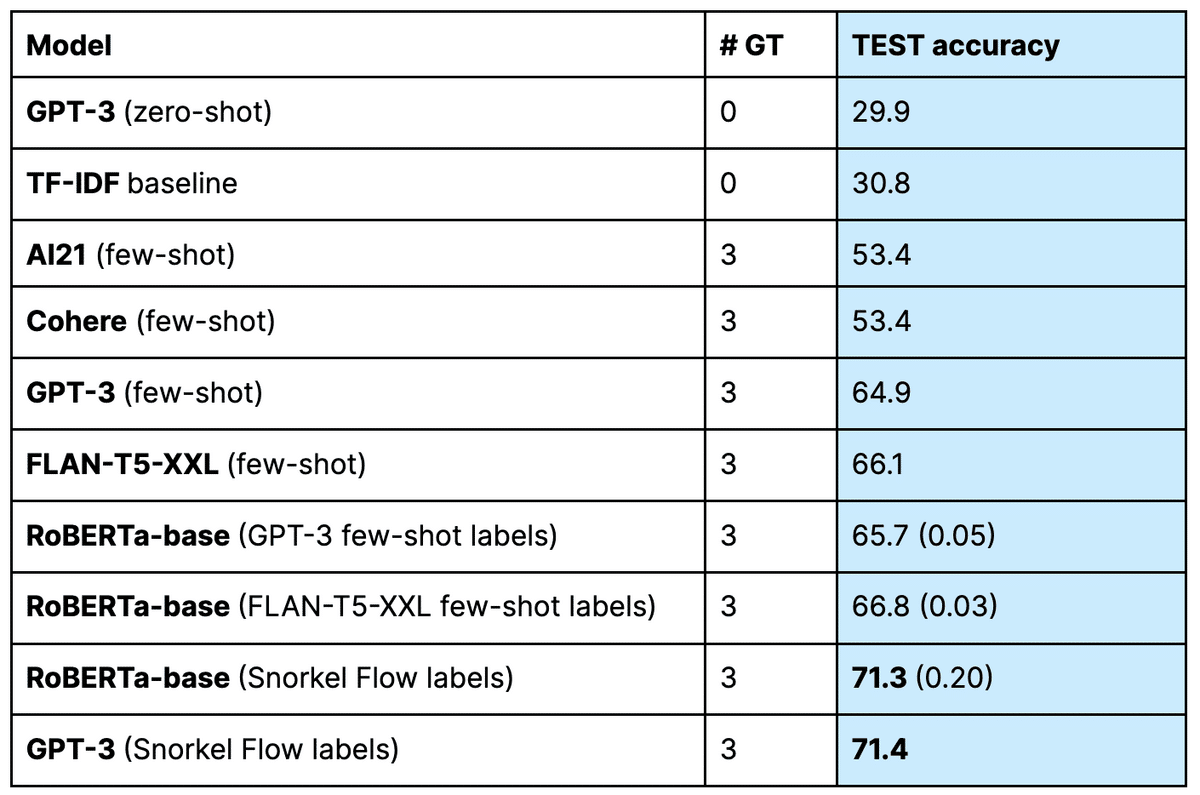
Fine-tuning adapts the LLM’s weights to custom domains and tasks.

Data scientists feed the model a collection of prompts and expected responses. The model learns the gaps between what it currently produces and what the training pipeline expected and adjusts its “attention” to specific features and patterns.

For example, if a data team wants to use an LLM to examine financial documents—something the model may perform poorly on out of the box—the team can fine-tune it on something like the [Financial Documents Clustering data set](https://www.kaggle.com/datasets/drcrabkg/financial-statements-clustering).

Fine-tuning mitigates underperformance for the target domain and task(s). It can also help the model overcome some of its biases and limitations, such as hallucination, repetition, or inconsistency.

In a [past experiment](https://snorkel.ai/better-not-bigger-how-to-get-gpt-3-quality-at-0-1-the-cost/), Snorkel researchers working with academic partners at Stanford and Brown found that a fine-tuned RoBERTa model significantly outperformed zero-shot prompts on an off-the-shelf version of GPT-3.

Baselines and language model labeler accuracy on the test set, with standard error (n=3) in parentheses where available. The GPT-3 (Zero-shot) results are included only for illustration; they were not included in the Foundation Model Warm Start stage. The RoBERTa models report 3 GT because they are trained with labels from few-shot models that used 3 GT labels.

However, fine-tuning requires a large amount of labeled data, which may be scarce, noisy, or expensive to obtain. It also requires significant computational resources, which could present a significant hurdle.

**Retrieval augmented generation (RAG)**

Retrieval augmented generation inserts an additional step between users’ requests and the generative model.

In this step, the pipeline finds information relevant to the user’s request and injects it as context. For example, Google’s Bard and Microsoft’s Bing Chat perform traditional search queries relevant to the user’s prompt before feeding the search results as additional context to the LLM.

This information could come from:

A vector database such as [FAISS](https://github.com/facebookresearch/faiss) or [Pinecone](https://www.pinecone.io/).

Traditional databases such as SQL or MongoDB.

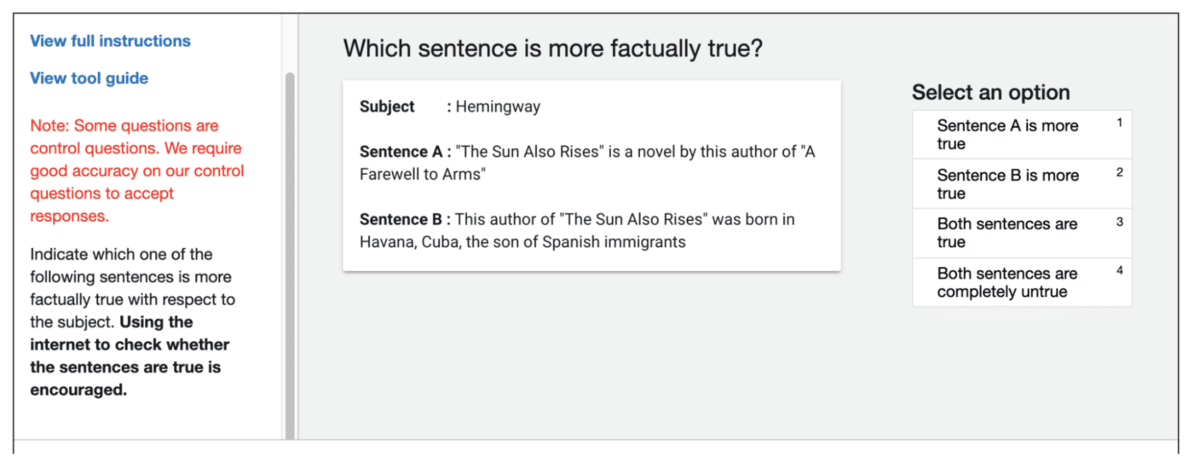
APIs such as those for Google Maps or IMDB.

A search engine such as Google or Bing.

RAG mitigates the challenge of out-of-date pre-training data by providing up-to-date information at inference time.

While quantifiable real-world examples are hard to come by, the results from the original paper the proposed RAG ([Retrieval-Augmented Generation fo Knowledge-Intensive NLP Tasks, Piktus Et al.](https://arxiv.org/pdf/2005.11401.pdf)) remain compelling.

In a head-to-head comparison, human testers were shown a subject and presented with two Jeopardy-style questions that the subject would answer. The interface asked the testers which question was more factual. They rated answers from the RAG-enhanced model as factual 54.4% of the time compared to just 18.8% of the time for the non-RAG model.

The annotation interface for human evaluation of factuality used in the original RAG paper, Retrieval-Augmented Generation fo Knowledge-Intensive NLP Tasks, Piktus Et al

However, RAG presents its own pitfalls. Effective RAG implementations require an efficient and effective mechanism to retrieve the correct context. Improperly implemented RAG tools can negatively impact responses by injecting irrelevant information—or, worse, it could surface sensitive information that should have been kept confidential.

**Fine-tuning vs. RAG**

Retrieval augmentation and fine-tuning address different aspects of LLMs’ limitations.

Fine-tuning outperforms RAG when addressing slow-to-change challenges, such as adapting the model to a particular domain or set of long-term tasks. RAG outperforms fine-tuning on quick-to-change challenges, such as keeping up with incremental documentation updates or records of customer intersections.

These approaches are neither mutually exclusive nor incompatible. In fact, they can be combined to achieve better results.

For example, imagine a customer support chat copilot. When agents require guidance during live interactions, they prompt the copilot, which triggers the following process:

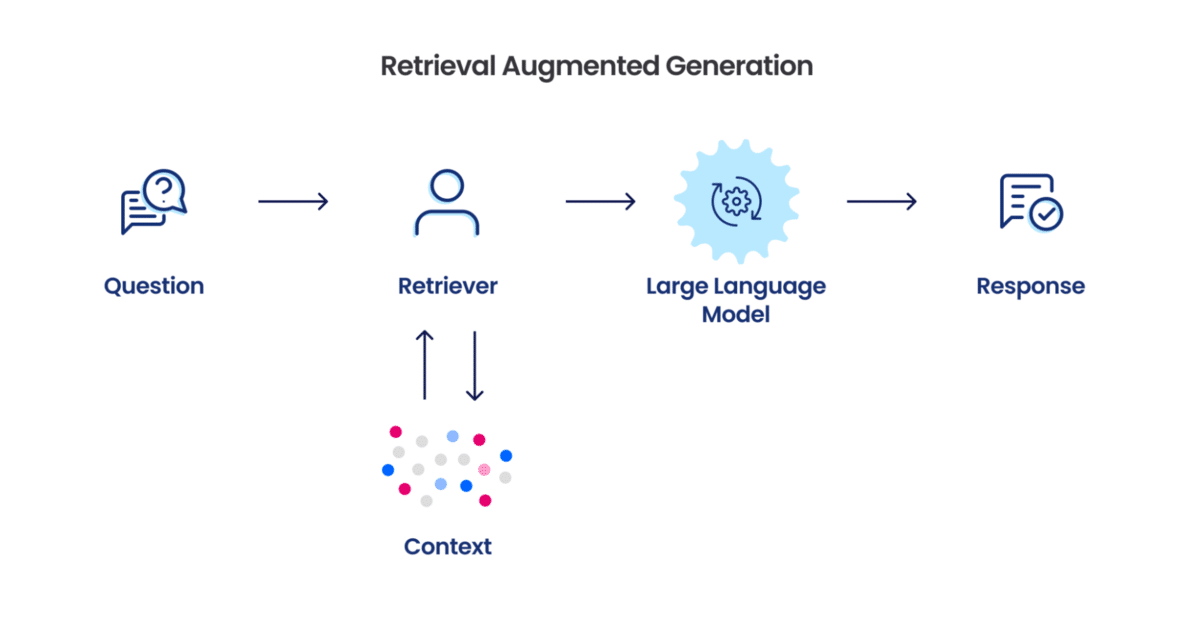
The pipeline retrieves essential customer information, including the customer ID.

The pipeline queries the customer’s history, issues, policies, special circumstances (e.g., outages), current support team availability, and other dynamic external variables.

The pipeline consolidates the gathered context and the original query into a final prompt.

The model generates a response grounded in the provided context.

While this pipeline could use an “off-the-shelf” model, the style of the generated response may vary and deviate from internal policies or end-user requirements. To improve the likelihood of consistent, helpful responses, the data team can train a “specialized” copilot model.



The data team would start with an appropriate foundation model, such as Llama 2 or GPT-3.5. Then, they would build or [curate a corpus](https://snorkel.ai/how-we-built-better-genai-with-programmatic-data-development/) of examples of likely inputs (context + prompt) and ideal outputs (including desired formatting) for each specific task type the model will regularly encounter. This could include separate approaches for tasks that customer service agents would handle themselves (such as reactivating an account) and tasks that involve guiding the user through actions on their side (such as troubleshooting a problem).

In this instance, fine-tuning and retrieval augmentation work hand-in-hand to deliver the right solution, faster.

**Fine-tuning and retrieval augmentation: a powerful pair**

Fine-tuning and RAG are not rivals, but complementary techniques that collaboratively enhance LLMs. Each has its own goals, mechanisms, advantages, and disadvantages, but they also synergize and benefit from each other.

**Chunking Strategies for LLM Applications**

In the context of building LLM-related applications, **chunking** is the process of breaking down large pieces of text into smaller segments. It’s an essential technique that helps optimize the relevance of the content we get back from a [vector database](https://www.pinecone.io/learn/vector-database/)once we use the LLM to embed content. In this blog post, we’ll explore if and how it helps improve efficiency and accuracy in LLM-related applications.

As we know, any content that we index in Pinecone needs to be [embedded](https://www.pinecone.io/learn/vector-embeddings-for-developers/) first. The main reason for chunking is to ensure we’re embedding a piece of content with as little noise as possible that is still semantically relevant.

For example, in semantic search, we index 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.

The length of the query also influences how the embeddings relate to one another. A shorter query, such as a single sentence or phrase, will concentrate on 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.

The index may also be non-homogeneous and contain embeddings for chunks of *varying* sizes. This may pose challenges in terms of query result relevance, but it may also have some positive consequences. On the one hand, the relevance of the query result may fluctuate because of discrepancies between the semantic representations of long and short content. On the other, a non-homogeneous index could potentially capture a wider range of context and information since different chunk sizes represent different levels of granularity in the text. This could accommodate different types of queries more flexibly.

**Chunking Considerations**

Several variables play a role in determining the best chunking strategy, and these variables vary depending on the use case. Here are some key aspects to keep in mind:

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](https://huggingface.co/sentence-transformers) models work well on individual sentences, but a model like [text-embedding-ada-002](https://openai.com/blog/new-and-improved-embedding-model) 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**

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**

This is the most common and straightforward approach to chunking: we simply decide the number of tokens in our chunk and, optionally, whether there should be any overlap between them. In general, we will want to keep some overlap between chunks to make sure that the semantic context doesn’t get lost between chunks. Fixed-sized chunking will be the best path in most common cases. Compared to other forms of chunking, fixed-sized chunking is computationally cheap and simple to use since it doesn’t require the use of any NLP libraries.

Here’s an example for performing fixed-sized chunking with [LangChain](https://api.python.langchain.com/en/latest/api_reference.html):

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:

* **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**](https://www.nltk.org/): 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**](https://spacy.io/): 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)

**Recursive Chunking**

Recursive chunking divides the input text into smaller chunks in a hierarchical and iterative manner using a set of separators. If the initial attempt at splitting the text doesn’t produce chunks of the desired size or structure, the method recursively calls itself on the resulting chunks with a different separator or criterion until the desired chunk size or structure is achieved. This means that while the chunks aren’t going to be exactly the same size, they’ll still “aspire” to be of a similar size.

Here’s an example of how to use recursive chunking with [LangChain](https://api.python.langchain.com/en/latest/text_splitter/langchain.text_splitter.RecursiveCharacterTextSplitter.html?highlight=recursive%20text%20splitter#langchain.text_splitter.RecursiveCharacterTextSplitter):

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**](https://www.markdownguide.org/): 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])

* [**LaTex**](https://www.latex-project.org/): 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])

**Semantic Chunking**

A new experimental technique for approaching chunking was first introduced by [Greg Kamradt](https://www.linkedin.com/in/gregkamradt/). [In his notebook](https://github.com/FullStackRetrieval-com/RetrievalTutorials/blob/main/tutorials/LevelsOfTextSplitting/5_Levels_Of_Text_Splitting.ipynb), 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 group will be associated with **one** “anchor” sentence.
3. Generate embeddings for each sentence group and associate them with their “anchor” sentence.
4. Compare distances between each group sequentially: When you look at the sentences in the document sequentially, as long as the topic or theme is the same - the distance between the sentence group embedding for a given sentence and the sentence group preceding it will be **low**. On the other hand, **higher** semantic distance indicates that the theme or topic has changed. This can effectively delineate one chunk from the next.

[LangChain](https://python.langchain.com/docs/get_started/introduction) has created a [semantic chunking splitter](https://python.langchain.com/docs/modules/data_connection/document_transformers/semantic-chunker/) implemented based on Kamradt’s work. You can also try out [our notebook for advanced chunking methods for RAG](https://github.com/pinecone-io/examples/blob/master/learn/generation/better-rag/02b-semantic-chunking.ipynb).

Figuring out the best chunk size for 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](https://docs.pinecone.io/docs/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.

**Conclusion**

Chunking your content is pretty simple in most cases - but it could present some challenges when you start wandering off the beaten path. There’s no one-size-fits-all solution to chunking, so what works for one use case may not work for another. Hopefully, this post will help you get a better intuition for how to approach chunking for your application.

**Advanced Chunking Methods for RAG**

Semantic chunking takes the idea of chunking documents (usually for RAG) to optimize for their end state of vector embeddings. Vector embeddings are retrieved based on semantic similarity, and semantic chunking focuses on building chunks using the exact same mechanism.

That means that we optimize our chunks for ideal retrieval performance. In essence, we are doing this by identifying the optimal chunk size that maintains a concise semantic meaning. A concise semantic meaning is important because we are compressing our chunk into a single vector embedding, so if the meaning of that chunk is not concise we would, in theory, produce suboptimal embeddings that are attempting to capture multiple meanings into a single vector, which just isn't possible — at best, we produce a type of average over the multiple meanings.

In this example, we'll explore semantic chunking and see the full pipeline from raw data through to chunking and embedding our data, ready for RAG.

<https://github.com/pinecone-io/examples/blob/master/learn/generation/better-rag/02b-semantic-chunking.ipynb>

<https://python.langchain.com/v0.1/docs/modules/data_connection/document_transformers/semantic-chunker/>

**New and improved embedding model**

The new model, text-embedding-ada-002, replaces five separate models for text search, text similarity, and code search, and outperforms our previous most capable model, Davinci, at most tasks, while being priced 99.8% lower.

Embeddings are numerical representations of concepts converted to number sequences, which make it easy for computers to understand the relationships between those concepts. Since the [initial launch](https://openai.com/index/introducing-text-and-code-embeddings/) of the OpenAI [/embeddings(opens in a new window)](https://beta.openai.com/docs/api-reference/embeddings) endpoint, many applications have incorporated embeddings to personalize, recommend, and search content.

<https://openai.com/index/new-and-improved-embedding-model/>

**5 Levels Of Text Splitting**

In this tutorial we are reviewing the 5 Levels Of Text Splitting. This is an unofficial list put together for fun and educational purposes.

Ever try to put a long piece of text into ChatGPT but it tells you it’s too long? Or you're trying to give your application better long term memory, but it’s still just not quite working.

One of the most effective strategies to improve performance of your language model applications is to split your large data into smaller pieces. This is call splitting or chunking (we'll use these terms interchangeably). In the world of multi-modal, splitting also applies to images.

We are going to cover a lot, but if you make it to the end, I guarantee you’ll have a solid grasp on chunking theory, strategies, and resources to learn more.

**Levels Of Text Splitting**

* **Level 1:**[**Character Splitting**](https://github.com/FullStackRetrieval-com/RetrievalTutorials/blob/8a30b5710b3dd99ef2239fb60c7b54bc38d3613d/tutorials/LevelsOfTextSplitting/#CharacterSplitting) - Simple static character chunks of data
* **Level 2:**[**Recursive Character Text Splitting**](https://github.com/FullStackRetrieval-com/RetrievalTutorials/blob/8a30b5710b3dd99ef2239fb60c7b54bc38d3613d/tutorials/LevelsOfTextSplitting/#RecursiveCharacterSplitting) - Recursive chunking based on a list of separators
* **Level 3:**[**Document Specific Splitting**](https://github.com/FullStackRetrieval-com/RetrievalTutorials/blob/8a30b5710b3dd99ef2239fb60c7b54bc38d3613d/tutorials/LevelsOfTextSplitting/#DocumentSpecific) - Various chunking methods for different document types (PDF, Python, Markdown)
* **Level 4:**[**Semantic Splitting**](https://github.com/FullStackRetrieval-com/RetrievalTutorials/blob/8a30b5710b3dd99ef2239fb60c7b54bc38d3613d/tutorials/LevelsOfTextSplitting/#SemanticChunking) - Embedding walk based chunking
* **Level 5:**[**Agentic Splitting**](https://github.com/FullStackRetrieval-com/RetrievalTutorials/blob/8a30b5710b3dd99ef2239fb60c7b54bc38d3613d/tutorials/LevelsOfTextSplitting/#AgenticChunking) - Experimental method of splitting text with an agent-like system. Good for if you believe that token cost will trend to $0.00
* **\*Bonus Level:\*** [**Alternative Representation Chunking + Indexing**](https://github.com/FullStackRetrieval-com/RetrievalTutorials/blob/8a30b5710b3dd99ef2239fb60c7b54bc38d3613d/tutorials/LevelsOfTextSplitting/#BonusLevel) - Derivative representations of your raw text that will aid in retrieval and indexing

**Notebook resources:**

* [Video Overview](https://notebooks.githubusercontent.com/view/ipynb?browser=chrome&bypass_fastly=true&color_mode=auto&commit=8a30b5710b3dd99ef2239fb60c7b54bc38d3613d&device=unknown_device&docs_host=https%3A%2F%2Fdocs.github.com&enc_url=68747470733a2f2f7261772e67697468756275736572636f6e74656e742e636f6d2f46756c6c537461636b52657472696576616c2d636f6d2f52657472696576616c5475746f7269616c732f386133306235373130623364643939656632323339666236306337623534626333386433363133642f7475746f7269616c732f4c6576656c734f665465787453706c697474696e672f355f4c6576656c735f4f665f546578745f53706c697474696e672e6970796e62&logged_in=false&nwo=FullStackRetrieval-com%2FRetrievalTutorials&path=tutorials%2FLevelsOfTextSplitting%2F5_Levels_Of_Text_Splitting.ipynb&platform=windows&repository_id=733569108&repository_type=Repository&version=125) - Walkthrough of this code with commentary
* [ChunkViz.com](https://www.chunkviz.com/) - Visual representation of chunk splitting methods
* [RAGAS](https://github.com/explodinggradients/ragas) - Retrieval evaluation framework

This tutorial was created with ❤️ by [Greg Kamradt](https://twitter.com/GregKamradt). MIT license, attribution is always welcome.

This tutorial will use code from LangChain (pip install langchain) & Llama Index (pip install llama-index)

**Evaluations**

It's important to test your chunking strategies in retrieval evals. It doesn't matter how you chunk if the performance of your application isn't great.

Eval Frameworks:

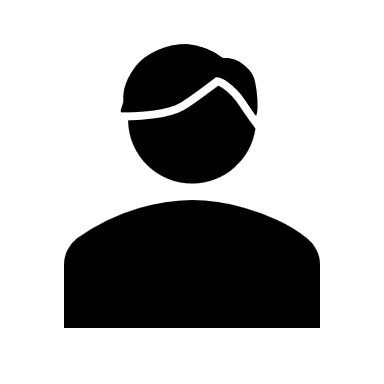
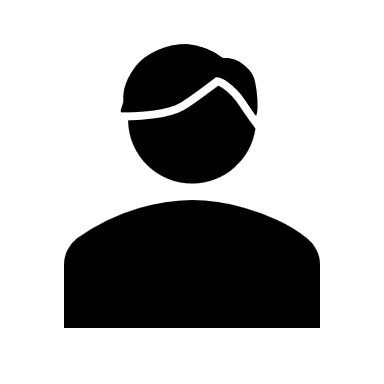
* [LangChain Evals](https://python.langchain.com/docs/guides/evaluation/)
* [Llama Index Evals](https://docs.llamaindex.ai/en/stable/module_guides/evaluating/root.html)
* [RAGAS Evals](https://github.com/explodinggradients/ragas)

I'm not going to demo evals for each method because success is domain specific. The arbitrary eval that I pick may not be suitable for your data. If anyone is interested in collaborating on a rigorous evaluation of different chunking strategies, please reach out (contact@dataindependent.com).

If you only walk away from this tutorial with one thing have it be the **The Chunking Commandment**

**The Chunking Commandment:** Your goal is not to chunk for chunking sake, our goal is to get our data in a format where it can be retrieved for value later.

<https://github.com/FullStackRetrieval-com/RetrievalTutorials/blob/main/tutorials/LevelsOfTextSplitting/5_Levels_Of_Text_Splitting.ipynb>



SME

Validation

Python conversion & testing

Logic validation using SM,

Syntax validation

Using pylint, python agent

to validate output

Use generated document

as part of

the prompt

to create python code

Create dynamic prompt

based on a data dictionary,

network graph & SME input

Chunking

Reverse Engineering

SME

Validation

Prompt creation

Identify functions &

perform chunking

Identify user-created

libraries &

Create network graph

Based on which programs

are calling other R programs

R scripts

Select program

Program/scripts level network graph

Identify R scripts

Identify program with

maximum outbound

Nodes

Used above approach to convert code from R to python.