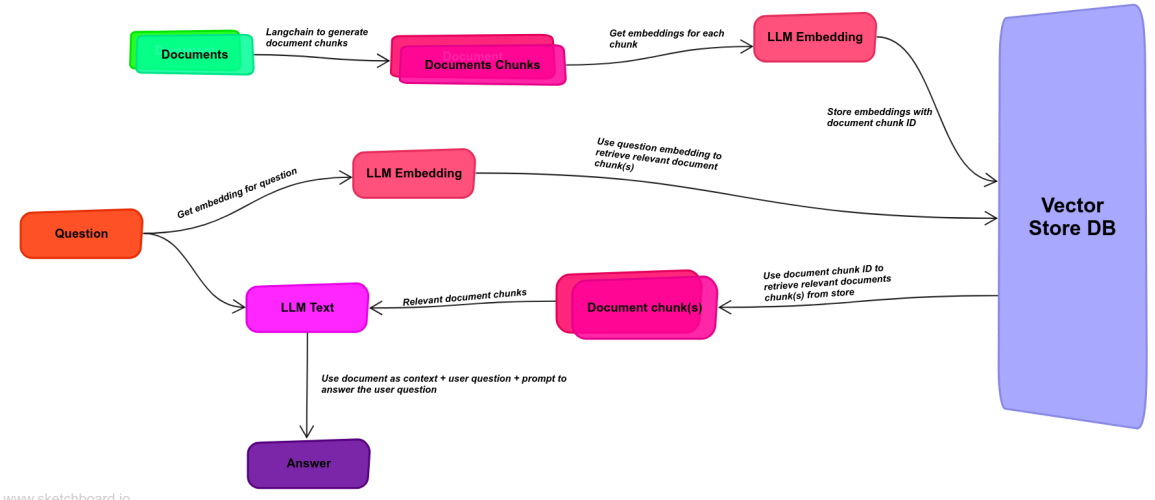
Chat With Your PDF Ultimate Guide To Retrieval Augmented Generation



Code With Prince

Load Documents

How To Load PDF Document

Document Loading

The first step in creating an ability to chat with your documents is first loading the document. Making splits of the document, creating embedding and storing these embeddings in a database that we can later on query and use to answer user questions. Let's first learn to load a PDF document. Here's how we can do this.

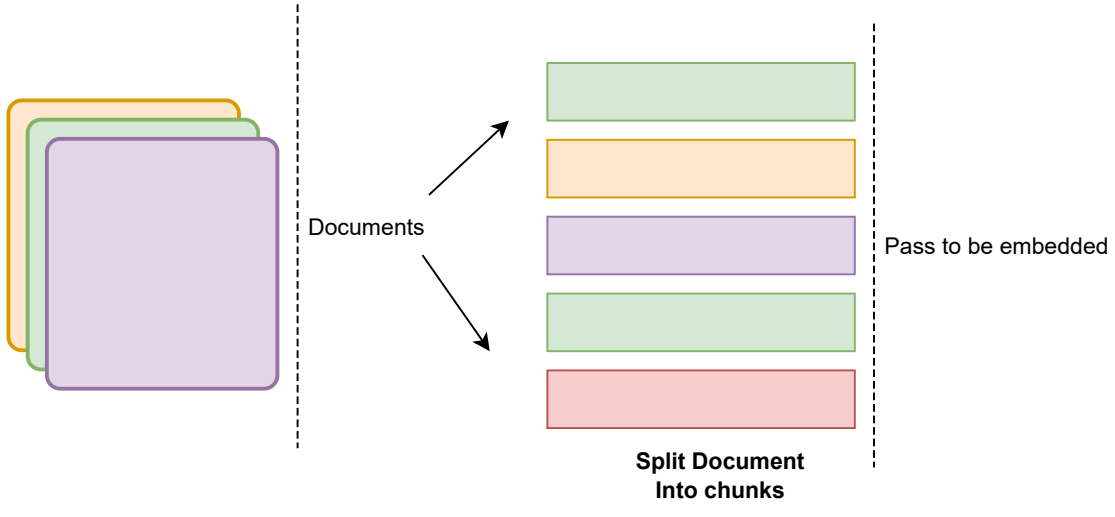
Splitting Document Into Chunks

Document Splitting

Now that we have loaded the document, it’s a whole book that we want to chat with. Imagine copying a whole book and pasting it in chatGPT. Do you honestly think that would work? No right? Of course it won’t the reason being that LLMs can only process certain about of text at a go. This limit is what is refered to as context window.

The “context window” refers to the span of text or tokens that the model uses to generate or understand a particular word or sequence of words in a given sentence or text. The context window determines how much of the surrounding text the model takes into account when making predictions or processing language.

The context window is limited in how much tokens or word count it can take. For this reason we need to split our loaded PDF file into what we call “chunks”. This chunk will be small enough to fit within a context window. Let’s do just this in Python.



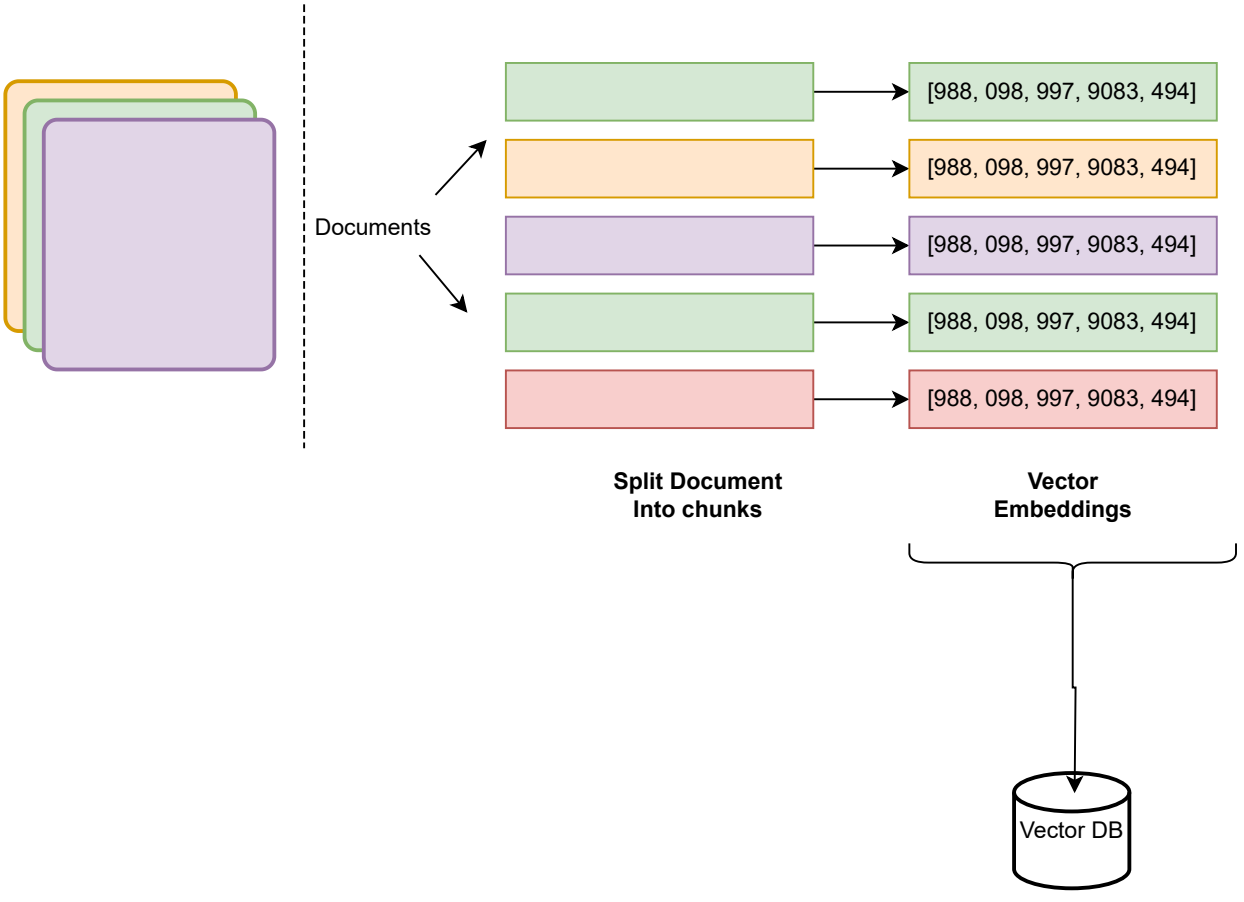
Word Embeddings And Vector Stores

Creating Embeddings

Computers do not understand natural languages that we humans speak. We humans can understand letters and a variety of symbols, on the other hand, computers can not, they only understand numeric values. So how do convert our text from the document into numeric representation that the model can understand?

Word embedding or word vector is an approach with which we represent documents and words. It is defined as a numeric vector input that allows words with similar meanings to have the same representation. It can approximate meaning and represent a word in a lower dimensional space. These can be trained much faster than the hand-built models that use graph embeddings like WordNet.

Here is where embedding comes into play. In this article I'll be using the sentence transformer which is an open source embedding model we can use. You can also pay for an OpenAI embedding model is you want to use that. The tradeoffs is not something I'll be covering. Let's install the sentence_transformers

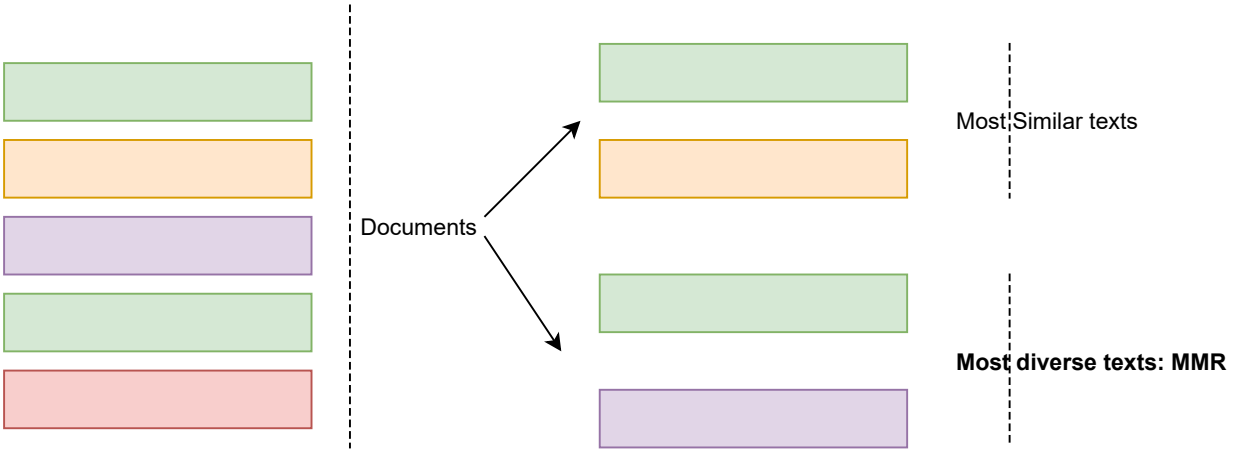


Retrieval Methods

Maximal Marginal Relevance (MMR)

Maximal Marginal Relevance (MMR)

This does not only give you the right context to work with, but also ensures diversity.



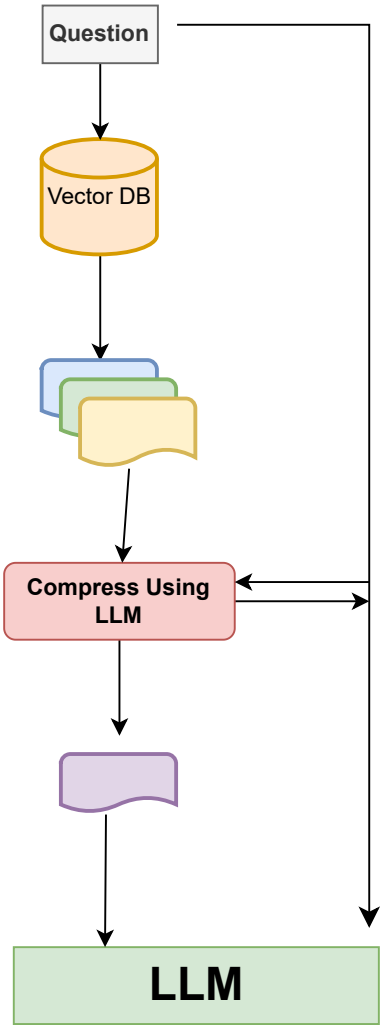
How It Works

1. Query the vector store
2. Set **fetch_k** value to get the **fetch_k** most similar search. This is basically semantic search
3. From the choosen most similar search, choose **k** most diverse context texts

Retrieval Methods

Compression Technique

Compress relevant documents into only two or three sentences containing the exact information you need with the help of an LLM. From here we can then make a final call to the LLM passing in the two or three sentences that contain the exact information needed to answer the question. Down side of this is the cost of making so many LLM calls needed to "compress" down the information we have retrieved.



Retrieval Methods

Self Query Retriever

LLM Aided Retrieval

This is used when the question asked or the text we want to search the vector store against is just not to deal with semantics but also meta-data about the semantics

Example

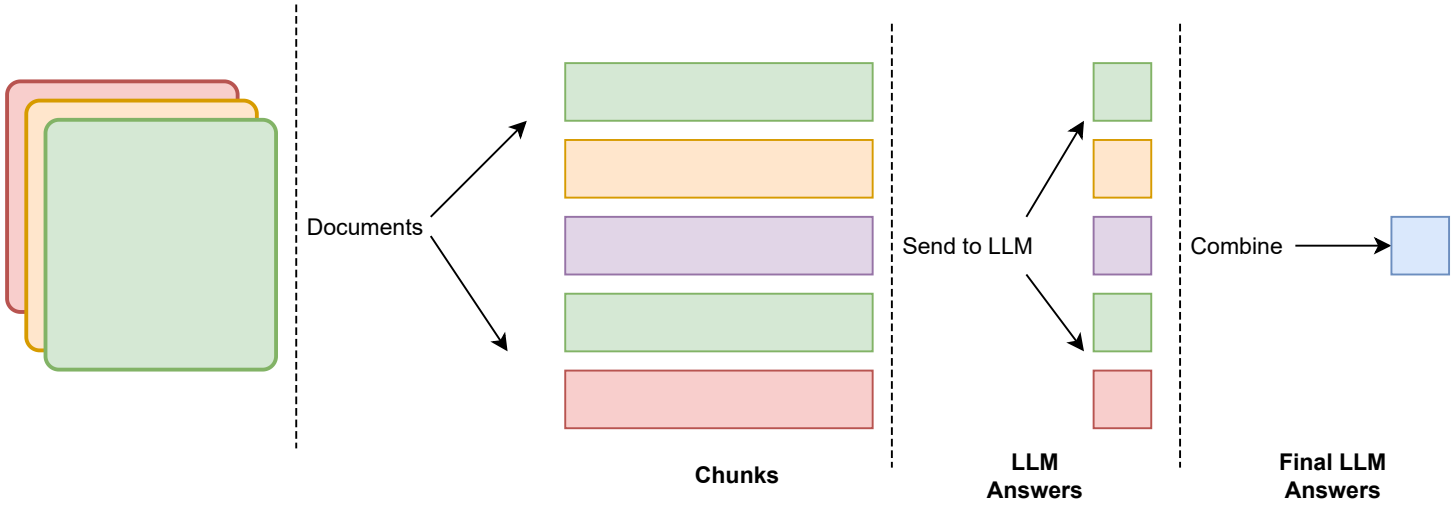
What was mentioned about puppies in the paper between pages 3 and 6?

In this question, we do not want to only answer the semantic question of what was said about puppies. We also want the meta-data information from only pages 3 and 6 specifically. Here is where LLMs help to split the question into a filter and a semantic search term. "Puppies" and "pages 3 and 6". This is what we call **SelfQuery**

Question Answering
Map Reduce

Map Reduce

In Map Reduce, each document or chunk is sent to the LLM to obtain an original answer, these original answers are then composed into one final answer. This involves more LLM calls.



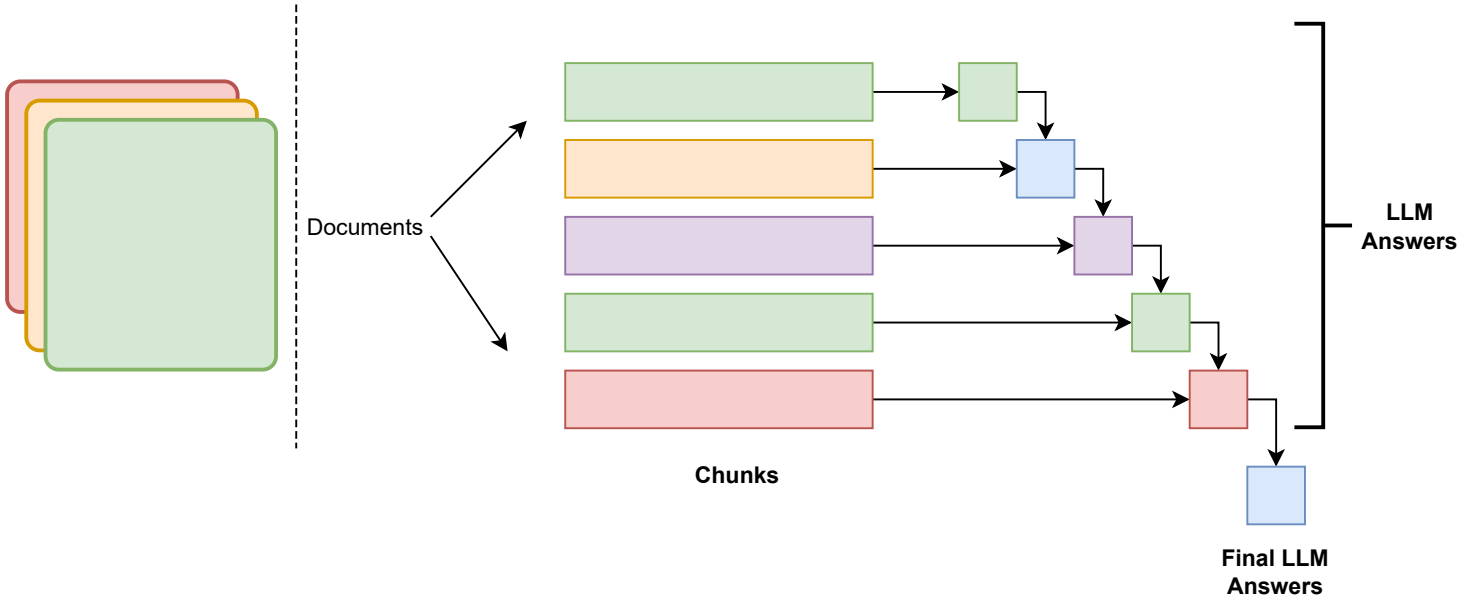
The downside of map reduce is that, sometimes the result can be worse than stuff method. This is due to the fact that the results are all spread over multiple document. Kind of like "looking for a needle in a hay-stack", the needle is there, just not easily found. Due to multiple document chunks.

The advantage of this being, if the answer or the information we are searching for is in multiple documents. We'll have a chance to look at each document and use it in answering the question at hand without running out of room in the context window.

Question Answering
Refine

Refine

This simply takes each chunk of document, passes it to the LLM along side the prompt question. An answer to that prompt question is generated. From here the next chunk or document is passed to the LLM and the first answer is refine or fine tuned based on the information from this document. This is repeated for all the other documents till a correct answer is obtained. Number of calls to the LLM is proportionate to the number of documents



The downside to this approach is that, we make calls to the LLM a number of times that is directly propotional to the number of documents we have retrieved. Hence more documents, more expensive. What is all the documents we have retrieved do not have the answer we are looking for?

Advantages of this is that it produces better results compared to the `map reduce` chain. As each time, as we iterate over each document, the final answer is fine-tuned. Example if the answer we are looking for is in document 4 then, we'll continually fine tune our answer till we arrive at that "right" answer, something map reduce technique does not provide as. All in all, this approach allows more carrying over of more information than map reduce strategy, more information more, results.