Retrieval-Augmented Generation (RAG) using LangChain and LlamaIndex

Step-by-step implementation

Prasad Mahamulkar · Follow

9 min read · Feb 9, 2024

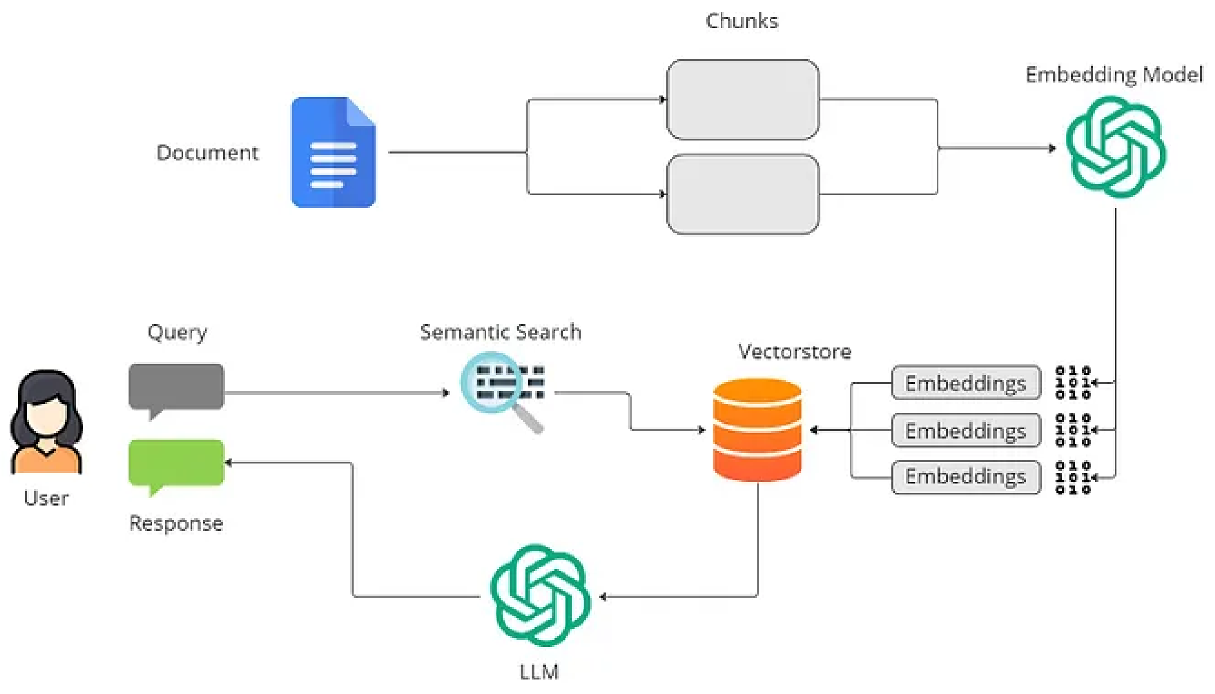


Image by Author

Large Language Models (LLMs) demonstrate significant capabilities but sometimes generate incorrect but believable responses when they lack information, and this is known as “hallucination.” It means they confidently provide information that may sound accurate but could be incorrect due to outdated knowledge.

Retrieval-Augmented Generation or RAG framework solves this problem by integrating an information retrieval system into the LLM pipeline. Instead of relying on pre-trained knowledge, RAG allows the model to dynamically fetch information from external knowledge sources when generating responses. This dynamic retrieval mechanism ensures that the information provided by the LLM is not only contextually relevant but also accurate and up-to-date.

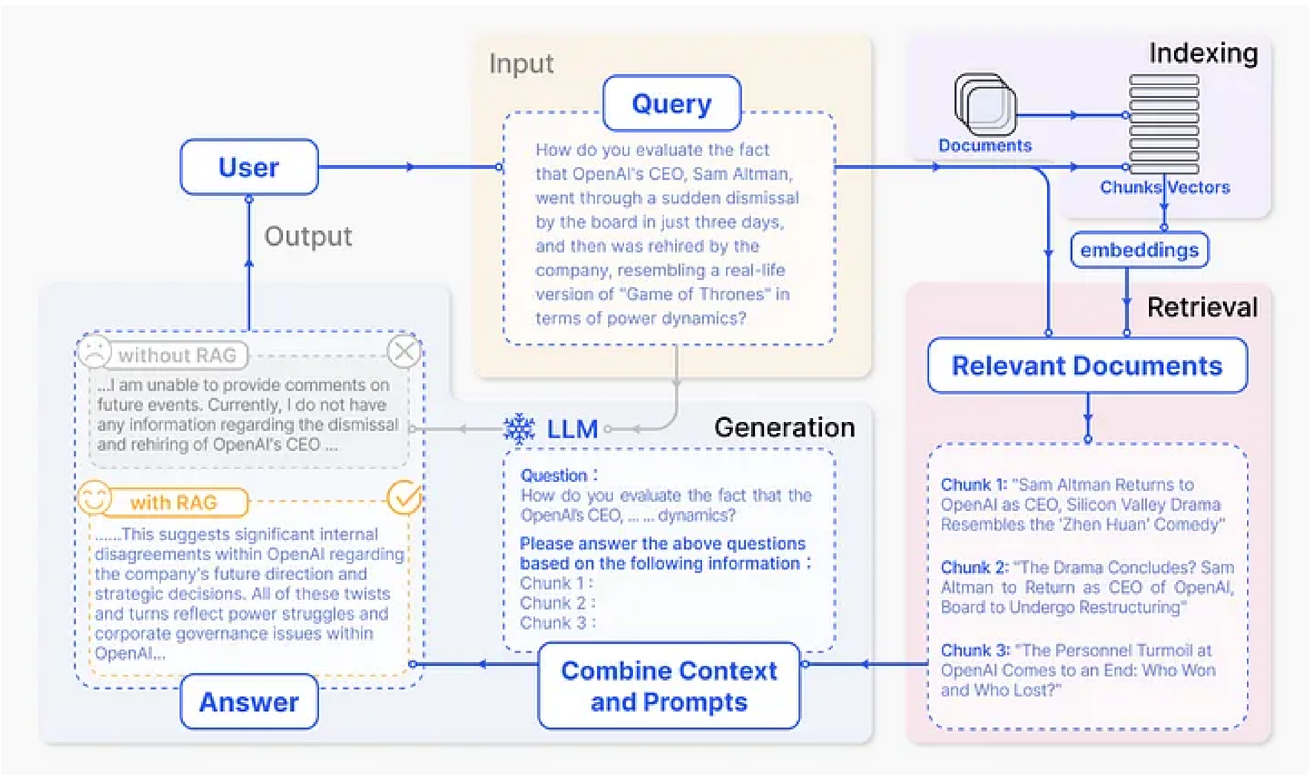
It is a more efficient way to provide additional or domain-specific information using an external database, rather than repeatedly retraining or fine-tuning the model on updated data.

In this article, we will understand how RAG works and create our own basic and Advanced RAG systems using LangChain and LlamaIndex.

Now let's start with understanding how RAG works.

How does RAG work?

The basic RAG workflow is illustrated below:



How do you evaluate the fact that OpenAI's CEO, Sam Altman, went through a sudden dismissal by the board in just three days, and then was rehired by the company, resembling a real-life version of "Game of Thrones" in terms of power dynamics?

...I am unable to provide comments on future events. Currently, I do not have any information regarding the dismissal and rehiring of OpenAI's CEO ...

......This suggests significant internal disagreements within OpenAI regarding the company's future direction and strategic decisions. All of these twists and turns reflect power struggles and corporate governance issues within OpenAI...

Question :
How do you evaluate the fact that the OpenAI's CEO, ... dynamics?
Please answer the above questions based on the following information :
Chunk 1 :
Chunk 2 :
Chunk 3 :

Chunk 1: "Sam Altman Returns to OpenAI as CEO, Silicon Valley Drama Resembles the 'Zhen Huan' Comedy"

Chunk 2: "The Drama Concludes? Sam Altman to Return as CEO of OpenAI, Board to Undergo Restructuring"

Chunk 3: "The Personnel Turmoil at OpenAI Comes to an End: Who Won and Who Lost?"

Image Source: RAG for Large Language Models: A Survey (paper)

Indexing

The indexing process is a crucial first step in data preparation for language models. Original data is cleaned, converted into standardized plain text, and segmented into smaller chunks for efficient processing. These chunks are transformed into vector representations through an embedding model, facilitating similarity comparisons during retrieval. The final index stores these text chunks and their vector embeddings, enabling efficient and scalable search capabilities.

Retrieval

When a user asks a question, the system uses the encoding model from the indexing phase to transcode it. Next, it calculates similarity scores between the query vector and vectorized chunks within the indexed corpus. The system prioritizes and retrieves the top K chunks showing the highest similarity, using them as an expanded contextual basis to address the user's request.

Generation

The user's question and chosen documents are combined into a clear prompt for a large language model. Then model crafts a response, adapting its approach based on task-specific criteria.

Now let's see how to implement the basic RAG technique using LangChain and LlamaIndex.

Basic RAG Implementation using LangChain and LlamaIndex

Google Colab Notebook

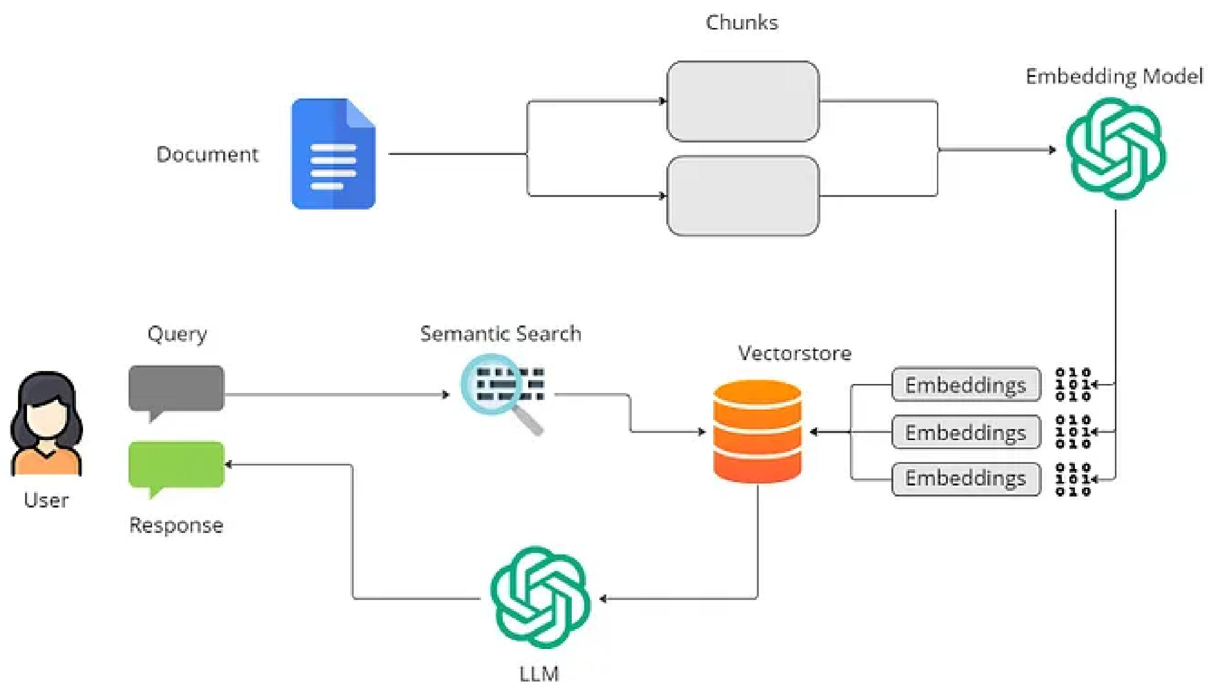


Image by Author

1. LangChain

Step 1: Start by installing and loading all the necessary libraries.

```
!pip install sentence_transformers pypdf faiss-gpu
!pip install langchain langchain-openai

from langchain_community.document_loaders import PyPDFLoader
from langchain.text_splitter import RecursiveCharacterTextSplitter
from langchain.embeddings import HuggingFaceEmbeddings
from langchain.vectorstores import FAISS
from langchain.chat_models import ChatOpenAI
from langchain.chains.combine_documents import create_stuff_documents_chain
from langchain_core.prompts import ChatPromptTemplate
from langchain.chains import create_retrieval_chain

# For openai key
import os
os.environ["OPENAI_API_KEY"] = "Your Key"
```

Step 2: Next, load a PDF document using PyPDFLoader to extract text from PDF files.

```
# load a PDF
loader = PyPDFLoader("/content/qlora_paper.pdf")
documents = loader.load()
```

Step 3: Once the PDF is loaded, use the TextSplitter to split the document into chunks.

```
# Split text
text = RecursiveCharacterTextSplitter().split_documents(documents)
```

Step 4: Now, load an embedding model to convert text into numerical embeddings. Here, we are using the "BAAI/bge-small-en-v1.5" embedding model, but you can choose any model from the Hugging Face embedding models.

```
# Load embedding model
embeddings = HuggingFaceEmbeddings(model_name="BAAI/bge-small-en-v1.5",
encode_kwargs={"normalize_embeddings": True})
```

Step 5: Create a Vector Store using FAISS to store embeddings and text chunks. If you want you can save these embeddings for later use.

```
# Create a vectorstore
vectorstore = FAISS.from_documents(text, embeddings)

# Save the documents and embeddings
vectorstore.save_local("vectorstore.db")
```

Step 6: Now, create a retriever using the vector store. This step establishes the foundation for information retrieval based on vector similarities.

```
# Create retriever
retriever = vectorstore.as_retriever()
```

Step 7: Load the Language Model (LLM) that you want to use for retrieval and create a document chain.

```
# Load the llm
llm = ChatOpenAI(model_name="gpt-3.5-turbo")

# Define prompt template
template = """
You are an assistant for question-answering tasks.
Use the provided context only to answer the following question:

<context>
{context}
</context>
```

```
Question: {input}
"""

# Create a prompt template
prompt = ChatPromptTemplate.from_template(template)

# Create a chain
doc_chain = create_stuff_documents_chain(llm, prompt)
chain = create_retrieval_chain(retriever, doc_chain)
```

Step 8: Finally, create a retrieval chain by combining the retriever and document chain. Invoke the chain with a user query to get a relevant response.

```
# User query
response = chain.invoke({"input": "what is Qlora?"})

# Get the Answer only
response['answer']
```

```
Output:
QLoRA is an efficient finetuning approach that allows for the finetuning of
quantized language models without any performance degradation.
It reduces memory usage enough to finetune a 65B parameter model on a single
48GB GPU while preserving full 16-bit finetuning task performance.
```

Now let's move on to basic RAG with LlamaIndex

2. LlamaIndex

Step 1: Start by installing and loading all the necessary libraries from llamaIndex.

```
! pip install -U llama_hub llama_index pypdf

from llama_index import SimpleDirectoryReader
from llama_index import Document
from llama_index.node_parser import SimpleNodeParser
from llama_index.schema import IndexNode
from llama_index.llms import OpenAI
from llama_index import ServiceContext
from llama_index import VectorStoreIndex
from llama_index.query_engine import RetrieverQueryEngine

# For openai key
import os
os.environ["OPENAI_API_KEY"] = "Your Key"
```

Step 2: Load a PDF document and combine each page of the document into one document object.

```
# load pdf
documents = SimpleDirectoryReader(
input_files=["/content/qlora_paper.pdf"]).load_data()

# combine documents into one
doc_text = "\n\n".join([d.get_content() for d in documents])
text= [Document(text=doc_text)]
```

Step 3: Now, split the document into text chunks. These chunks are known as “Nodes” in LlamaIndex. Also, reset default node IDs for better understanding.

```
# set up text chunk
node_parser = SimpleNodeParser.from_defaults()

# split doc
base_nodes = node_parser.get_nodes_from_documents(text)

# reset node ids
for idx, node in enumerate(base_nodes):
    node.id_ = f"node-{idx}"
```

Step 4: Load an embedding model and language model (LLM). We are using the same models that we used for LangChain.

```
# load embedding model
embed_model = resolve_embed_model("local:BAAI/bge-small-en")

# load llm
llm = OpenAI(model="gpt-3.5-turbo")
```

Step 5: Create a service by bundling LLM and embedding model for the indexing and querying stage.

```
# set up service
service_context = ServiceContext.from_defaults(llm=llm, embed_model=embed_moc
```

Step 6: Create and store embeddings of nodes (chunks) and store them in the vector store index.

```
# create & store in embeddings vectorstore index
index = VectorStoreIndex(base_nodes, service_context=service_context)
```

Step 7: Create a retriever using the vector store index to retrieve relevant information for user queries.

```
# create retriever
retriever = index.as_retriever()
```

Step 8: Finally, set up a query engine by combining the retriever and service context, and add a user query to get a relevant response.

```
# set up query engine
query_engine = RetrieverQueryEngine.from_args(retriever,
 service_context=service_context)

# query
response = query_engine.query("What is Qlora?")
print(str(response))
```

```
Output:
QLORA is an efficient finetuning approach that allows for the reduction of
memory usage while preserving the performance of a pretrained language model.
It achieves this by backpropagating gradients through a frozen, 4-bit
quantized pretrained language model into Low Rank Adapters (LoRA). QLORA
introduces several innovations to save memory, including the use of a new
data type called 4-bit NormalFloat (NF4), Double Quantization to reduce
memory footprint, and Paged Optimizers to manage memory spikes. QLORA has
been used to finetune more than 1,000 models and has shown state-of-the-art
results in chatbot performance.
```

Now let’s take a look at how to improve LLM response using Advanced RAG.

Advanced RAG Implementation using LangChain and LlamaIndex

The problem with the basic RAG technique is that, as document size increases, embeddings become larger and more complex, which can reduce the specificity and contextual meaning of a document. To solve this problem, we use the advanced RAG technique called Parent Document Retriever.

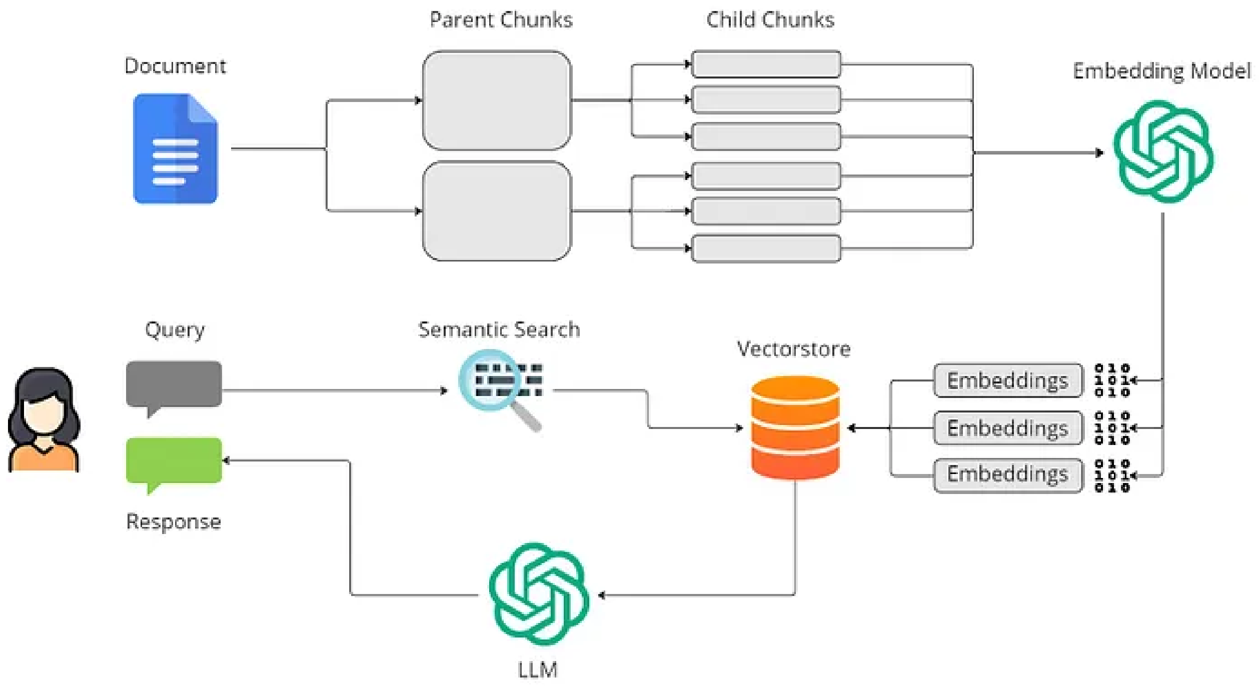


Image by Author

Parent Document Retriever creates small and more accurate embeddings while retaining the contextual meaning of large documents. Parent document retriever helps LLM by using details from child documents for accurate retrieval and gaining additional context from parent documents during generation. This makes language models better at providing detailed and comprehensive answers.

Let's see how to implement the Parent Document Retriever technique using LangChain and LlamaIndex.

1. LangChain

Parent Document Retriever LangChain Documentation.

Update the following steps in the basic RAG process.

Step 3: Use the TextSplitter to split the document into parent and child chunks.

```
# split pages content
from langchain.text_splitter import RecursiveCharacterTextSplitter

# create the parent documents - The big chunks
parent_splitter = RecursiveCharacterTextSplitter(chunk_size=2000)
```

```
# create the child documents - The small chunks
child_splitter = RecursiveCharacterTextSplitter(chunk_size=400)

# The storage layer for the parent chunks
from langchain.storage import InMemoryStore
store = InMemoryStore()
```

Step 5: Create a Vector Store using Chromadb to store new embeddings and text chunks.

```
# create vectorstore using Chromadb
from langchain.vectorstores import Chroma
vectorstore = Chroma(collection_name="split_parents",
embedding_function=embeddings)
```

Step 6: Create a Parent doc retriever then, add a document to the retriever.

```
# create retriever
from langchain.retrievers import ParentDocumentRetriever
retriever = ParentDocumentRetriever(
    vectorstore=vectorstore,
    docstore=store,
    child_splitter=child_splitter,
    parent_splitter=parent_splitter,
)

# add documents to vectorstore
retriever.add_documents(documents)
```

Step 8: Finally, create a retrieval chain, similar to the previous chain, and invoke it with a user query to get a response.

```
# User query
response = chain.invoke({"input": "what is Qlora?"})
```

```
    # Get the Answer only
    response['answer']
```

```
Output:
QLORA is a method that achieves high-fidelity 4-bit finetuning by using two
techniques: 4-bit NormalFloat (NF4) quantization and Double Quantization.
It also introduces Paged Optimizers to prevent memory spikes during gradient
checkpointing. QLORA has one low-precision storage data type (usually 4-bit)
 and one computation data type (usually BFloat16). It dequantizes the
low-precision tensor to BFloat16 and performs matrix multiplication in 16-bit
QLORA is a method that achieves high-fidelity 4-bit finetuning by using two
techniques: 4-bit NormalFloat (NF4) quantization and Double Quantization.
It also introduces Paged Optimizers to prevent memory spikes during gradient
checkpointing. QLORA has one low-precision storage data type (usually 4-bit)
and one computation data type (usually BFloat16). It dequantizes the
low-precision tensor to BFloat16 and performs matrix multiplication in 16-bit
```

As you can see in the output, we get a more detailed response compared to the basic RAG method.

2. LlamaIndex

Parent Document Reteriver LlamaIndex Documentation.

Update the following steps in the basic RAG.

Step 3:

In this section, we set child chunk sizes (128, 256, 512) in `sub_chunk_sizes` and create parsers (`sub_node_parsers`) for child chunks. These parsers, with specific sizes, will be used to parse child chunks from the original document.

We then iterate over each original document chunk (`base_node`). For each base node, we use parsers to generate child nodes, linked to their parent node. Parent nodes and child nodes are added to the `all_nodes` list.

Finally, a dictionary (`all_nodes_dict`) is created for easy access to nodes based on their IDs. This helps efficiently retrieve information.

```
# add this code after base node

# set up child chunk
sub_chunk_sizes = [128, 256, 512]
sub_node_parsers = [
    SimpleNodeParser.from_defaults(chunk_size=c,chunk_overlap=20) for c in su
]

all_nodes = []
for base_node in base_nodes:
    for n in sub_node_parsers:
        sub_nodes = n.get_nodes_from_documents([base_node])
        sub_inodes = [
            IndexNode.from_text_node(sn, base_node.node_id) for sn in sub_noc
        ]
        all_nodes.extend(sub_inodes)

    # also add original node to node
    original_node = IndexNode.from_text_node(base_node, base_node.node_id)
    all_nodes.append(original_node)

all_nodes_dict = {n.node_id: n for n in all_nodes}
```

Step 6: Create embeddings of all nodes (which contain both parent and child nodes) and store them in the vector store index.

```
# create and store embedding in vectorstore
index = VectorStoreIndex(all_nodes, service_context=service_context)
```

Step 7: In the Retriever process, we use a RecursiveRetriever that navigates node connections based on "references." This retriever explores links from nodes to other retrievers. If the retrieved nodes include IndexNodes, it further explores linked retrievers and conducts queries. This recursive approach helps gather information effectively.

```
# create retriever
vector_retriever_chunk = index.as_retriever()
```

```python
from llama_index.retrievers import RecursiveRetriever
retriever_chunk = RecursiveRetriever(
    "vector",
    retriever_dict={"vector": vector_retriever_chunk},
    node_dict=all_nodes_dict,
    verbose=True,
)
```

Step 8: Finally, set up a query engine by combining the retriever and service context, and add a user query.

```python
# create retriever query
from llama_index.query_engine import RetrieverQueryEngine
query_engine_chunk = RetrieverQueryEngine.from_args(retriever_chunk,
service_context=service_context)

# query
response = query_engine_chunk.query("What is Qlora?")
print(str(response))
```

```
Output:
QLORA is an efficient finetuning approach that allows for the finetuning of
quantized language models without any performance degradation. It reduces
memory usage and enables the finetuning of large models on a single GPU.
QLORA uses a novel technique to quantize a pretrained model to 4-bit and
incorporates Low-rank Adapters (LoRA) for backpropagating gradients through
the quantized weights. It introduces several innovations to save memory
without sacrificing performance, such as a new data type called 4-bit
NormalFloat (NF4), Double Quantization to reduce memory footprint, and
Paged Optimizers to manage memory spikes. QLORA has been used to finetune
more than 1,000 models and has achieved state-of-the-art results on various
benchmarks.
```

Here also, we get a more detailed response compared to basic RAG.

Summary

In this article, we explored the fundamentals of RAG and successfully developed both basic and Advanced RAG systems using LangChain and LlamaIndex. I hope you found this article useful. I recommend exploring other Advanced RAG techniques and working with different data types (like CSV) to gain more experience.

. . .

Reference:

Retrieval-Augmented Generation for LLMs: A Survey

Parent Document Retriever langChain Documentation

Parent Document Reteriver LlamaIndex Documentation

Machine Learning Data Science Chatbots Large Language Models Artificial Intelligence

Written by Prasad Mahamulkar

34 Followers

MSc. Data Science and Analytics.

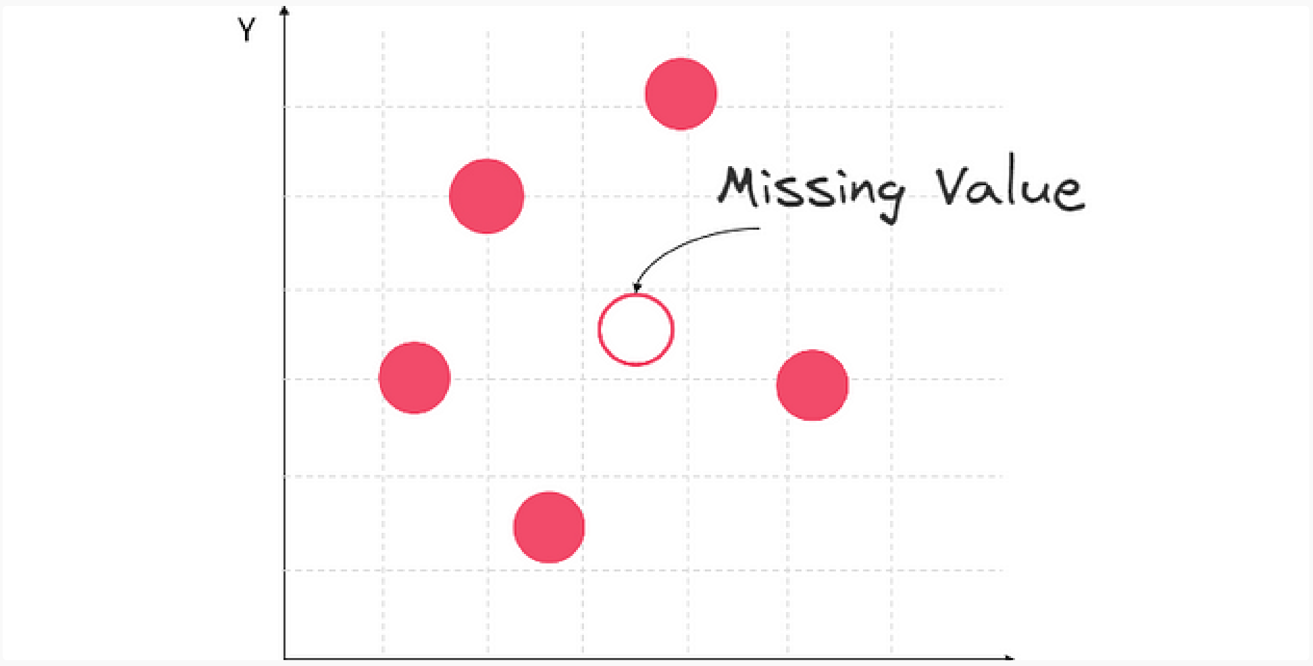
More from Prasad Mahamulkar

Prasad Mahamulkar

Fine-tune a Large Language Model

A Step-by-Step Phi-2 Model Fine-tuning

Jan 22

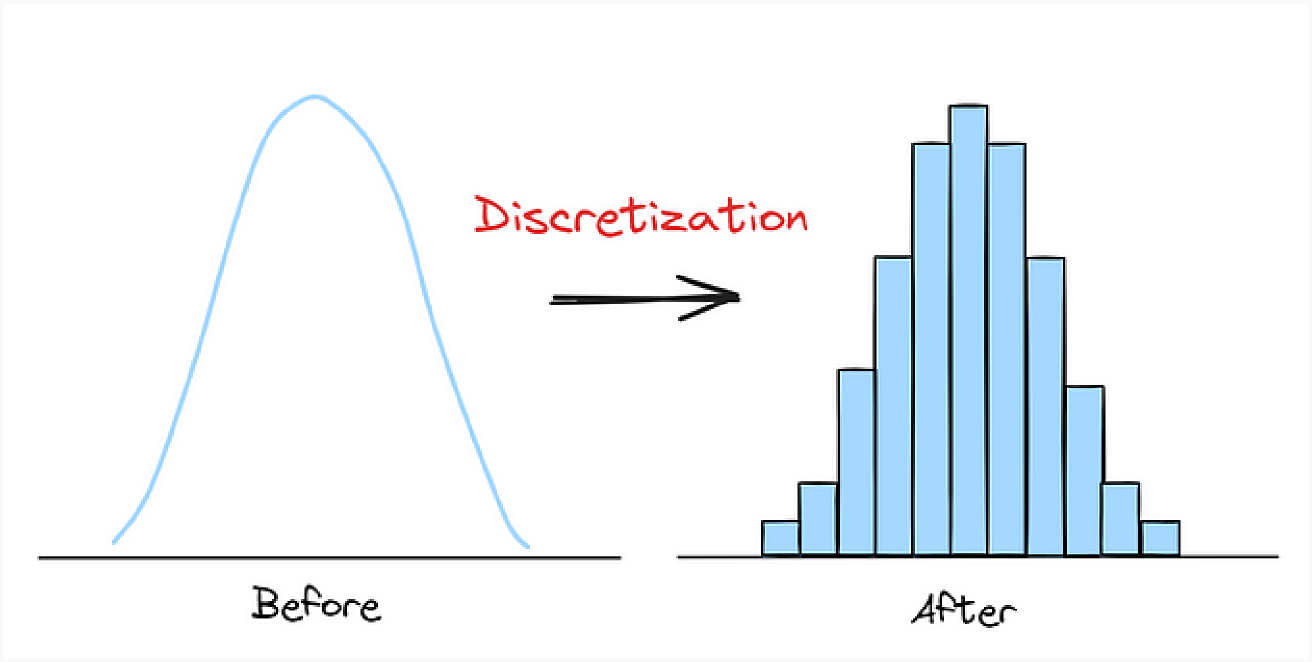


Prasad Mahamulkar

Univariate and Multivariate Imputation Techniques in Machine Learning

Imputing missing values (Null) is an important step in feature engineering because many machine-learning algorithms do not support missing...

Jan 1

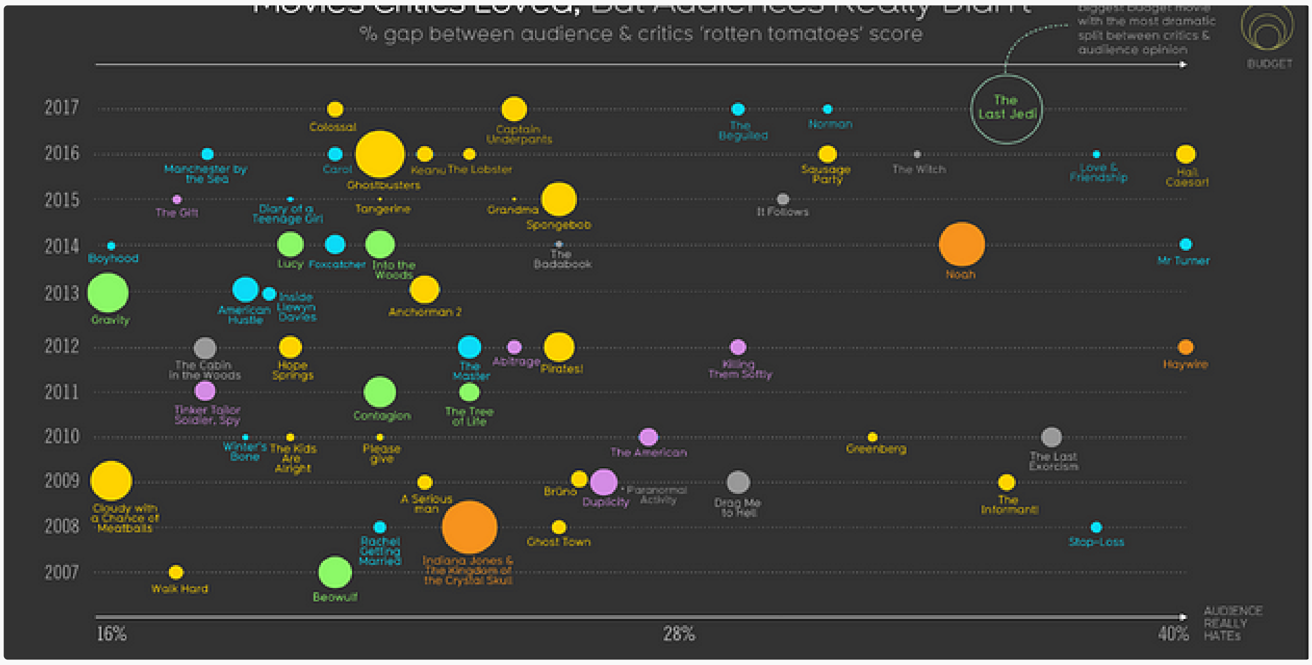


Prasad Mahamulkar

Discretization or Numerical Feature Encoding

Discretization or binning is the process that transforms a numerical feature into a discrete feature. In simpler terms, it creates bins...

Nov 21, 2023



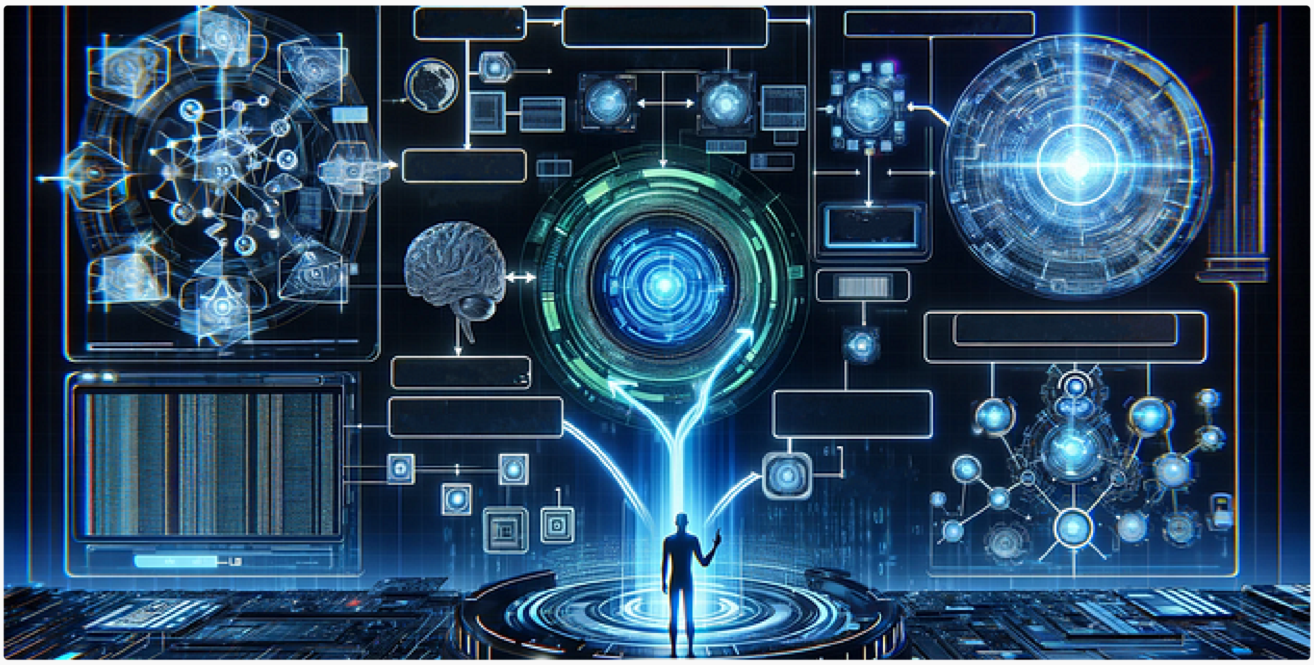
Prasad Mahamulkar

Data Visualization Best Practices

Let's understand Data Visualization Best Practices using the following visualization. The visualization is taken from...

Jun 13, 2023

Recommended from Medium

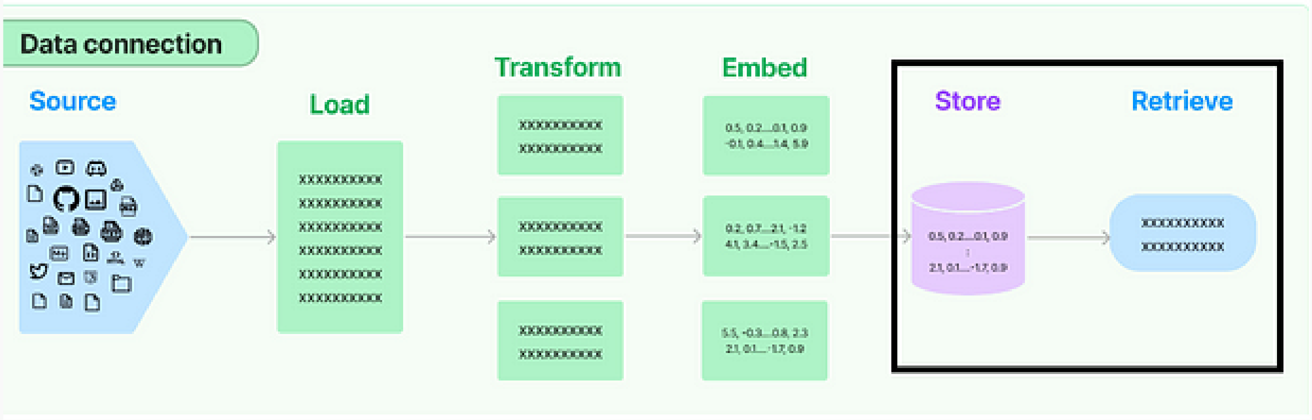


Necati Demir

Hands-On with RAG: Step-by-Step Guide to Integrating Retrieval Augmented Generation in LLMs

With the help of Large Language Models (LLMs), finally, we have a system that can understand(!) our questions and answer them. But, we know...

Dec 5, 2023



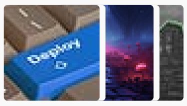
Jayant Pal

RAG using LangChain : Part 3- Vector Stores and Retrievers

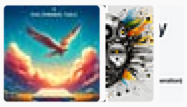
Welcome to the third article in this series of RAG where we explore the different components in the Retrieval process in RAG. So far we...

Mar 31

Lists

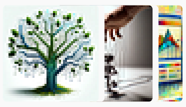


Predictive Modeling w/ Python
20 stories · 1242 saves

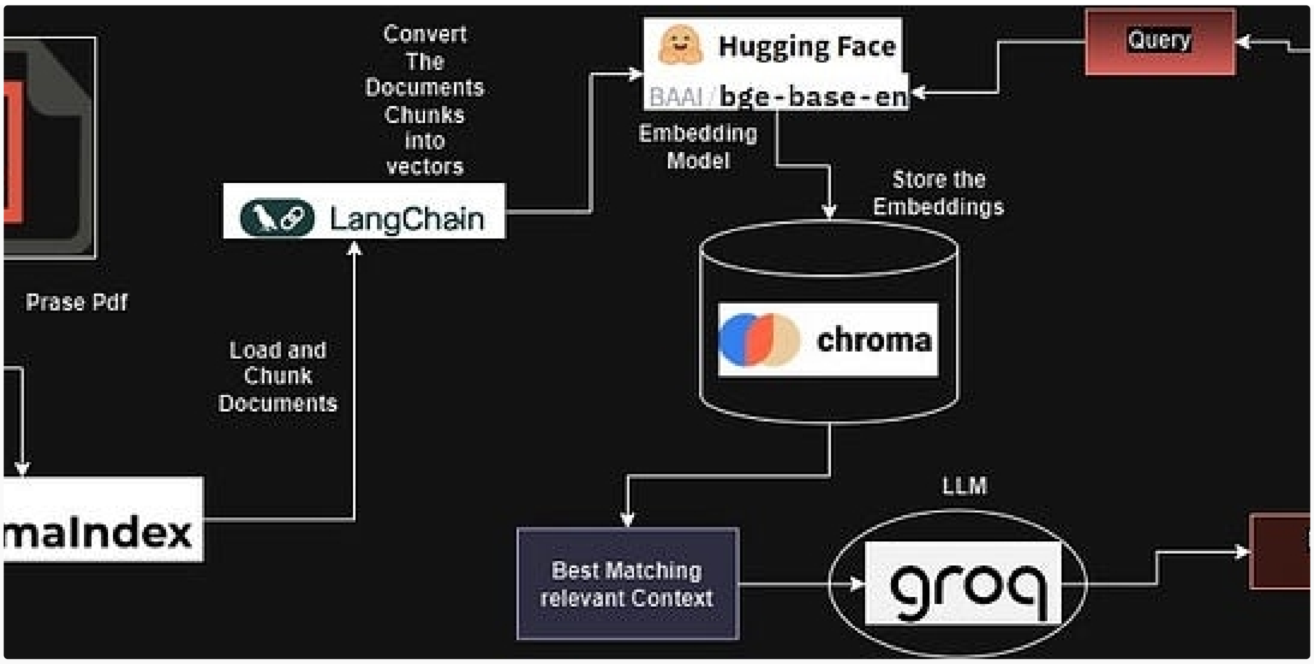


Natural Language Processing
1488 stories · 999 saves

Practical Guides to Machine Learning
10 stories · 1498 saves



ChatGPT prompts
47 stories · 1624 saves



Plaban Nayak in The AI Forum

RAG on Complex PDF using LlamaParse, Langchain and Groq

Retrieval-Augmented Generation (RAG) is a new approach that leverages Large Language Models (LLMs) to automate knowledge search, synthesis...

Apr 7

Saurabh Tiwari

RAG App using LLama and LLama_Index

In this article, I'll guide you through building a Retrieval-Augmented Generation (RAG) system using the open-source LLama2 model from...

Feb 26

Jayita Bhattacharyya in GoPenAI

Building a RAG Chatbot using Llamaindex, Groq with Llama3 & Chainlit

Retrieval Augmented Generation (RAG) is a language model that combines the strengths of retrieval-based and generation-based approaches

May 16

Ming

Comparing LangChain and LlamaIndex with 4 tasks

LangChain v.s. LlamaIndex — How do they compare? Show me the code!

Jan 11