Module 2 Introduction –

Advanced Retrieval Augmented Generation

The "Advanced Retrieval Augmented Generation" module offers an in-depth exploration into optimizing large language models (LLMs) with advanced Retrieval-Augmented Generation (RAG) techniques. Across four lessons, it encompasses a range of topics including query transformation, re-ranking, optimization techniques like fine-tuning, the implementation of Activeloop's Deep Memory, and other advanced strategies using LlamaIndex. Students will gain practical experience in enhancing RAG system performance, from query refinement to production deployment and iterative optimization. The module is designed to provide a comprehensive understanding of building, refining, and deploying efficient RAG systems, integrating hands-on examples with theoretical knowledge to prepare students for real-world applications.

Fine-tuning vs RAG; Introduction to Activeloop's Deep Memory;

In this lesson, students will explore various **optimization techniques** to maximize the performance of large language models (LLMs), such as **prompt engineering, fine-tuning**, and retrieval-augmented generation (**RAG**). The lesson begins with identifying the benefits and challenges of each method. It further examines the limitations of RAG systems and introduces Activeloop's Deep Memory as a solution to these challenges, particularly in improving retrieval precision for user queries. Students will see a step-by-step guide on how to implement **Deep Memory in experimental workflows**, including creating a synthetic training dataset and running inference with the trained Deep Memory model. A significant portion of the lesson is dedicated to hands-on examples using code to demonstrate the increased recall rates when Deep Memory is applied in a RAG system. The lesson concludes with a comparison of empirical data, highlighting the advantages of Deep Memory over traditional methods and emphasizing its role in advancing the capabilities of LLMs.

Mastering Advanced RAG Techniques with LlamaIndex

In this lesson, students will learn about the advanced techniques and strategies that enhance the performance of Retrieval-Augmented Generation (RAG) systems, using LlamaIndex as a framework. They will explore the concepts of **query construction, query expansion, and query transformation** to refine the information retrieval process. Students will also be introduced to advanced strategies like **reranking** with Cohere Reranker, **recursive retrieval, and small-to-big retrieval** to further improve the quality and relevance of search results. The lesson includes hands-on examples of setting up a query engine from indexing to querying, as well as creating custom retrievers and utilizing reranking. The conclusion underlines the importance of these techniques and strategies in developing more efficient RAG-based applications.

Production-Ready RAG Solutions with LlamaIndex

In this lesson, students will learn about the challenges, optimization strategies, and best practices for Retrieval-Augmented Generation (RAG) systems in production. The discussion includes dealing with dynamic data management, diverse representation in latent space, regulatory compliance, and model selections for system efficiency. The lesson emphasizes the importance of f**ine-tuning both the embedding models and** the Language Large Models **(LLMs)** to improve retrieval metrics and generate more accurate responses. Additionally, students will explore the role of Intel® technologies **in optimizing neural network models on CPUs**, and they will acquire knowledge on **utilizing generative feedback loops**, **hybrid searches**, and the **continuous evaluation of RAG** system performance. Practical use cases, data management tools, and integration of metadata in retrieval steps are also highlighted, with LlamaIndex being presented as a comprehensive framework for building data-driven LLM applications.

Iterative Optimization of LlamaIndex RAG Pipeline: A Step-by-Step Approach

In this lesson, you will learn the process of iteratively optimizing a LlamaIndex Retrieval-Augmented Generation (RAG) pipeline to enhance its performance in information retrieval and generating relevant answers. The lesson guides you through establishing a baseline pipeline, experimenting with retrieval values and embedding models like "text-embedding-ada-002" and "cohere/embed-english-v3.0," and incorporating techniques like **reranking and deep memory** to refine document selection. Additionally, you will learn about performance metrics, such as **Hit Rate and Mean Reciprocal Rank (MRR), and evaluate faithfulness and relevancy of answers** using GPT-4 as a judge. The lesson provides hands-on code examples for each optimization step and concludes with the overall enhancement observed in the RAG pipeline's accuracy.

Use Deep Memory to Boost RAG Apps' Accuracy by up to +22%

In this lesson, you will be introduced to a practical example of Deep Memory. Students will learn about the limitations of current RAG systems, such as suboptimal retrieval accuracy, and explore the benefits of implementing Deep Memory. The lesson explains how Deep Memory provides a significant accuracy boost by optimizing the vector search process using a tailored index from labeled queries. Throughout the lesson, students will be guided through hands-on examples for adopting Deep Memory in their applications, including data loading, creating a relevance dataset, training, and evaluating methods. The lesson emphasizes the practical advantages of this approach, like higher retrieval quality, cost savings from reduced context size needs and compatibility with existing workflows.

How to Use Deep Memory with LlamaIndex to Get +15% RAG hit_rate Improvement for Question Answering on Docs?

In this comprehensive tutorial, students will learn about improving the hit rate of Retrieval-Augmented Generators (RAGs) when answering questions from documentation by up to 15% or more using Activeloop's Deep Memory. The lesson covers dataset creation and ingestion using BeautifulSoup and LlamaIndex, training deep memory with synthetic queries, evaluating the performance improvement, and leveraging deep memory for real-world inference. By integrating a small neural network layer into the retrieval process, the tutorial demonstrates how to precisely match user queries with relevant data, significantly boosting the accuracy of returned information while maintaining minimal search latency. Students will get hands-on experience with Python libraries and AI models such as OpenAI's GPT-4 and vector store operations to create a more efficient and accurate RAG system.

Use Deep Memory with LangChain to Get Up to +22% Increase in Accurate Questions Answers to LangChain Code DB

In this lesson, students will learn how to utilize Activeloop Deep Memory with Langchain to enhance the efficiency and accuracy of Retrieval-Augmented Generation (RAG) systems by parsing documentation, creating datasets, generating synthetic queries, training a retrieval model, evaluating performance, and ultimately integrating Deep Memory into RAG-powered Language Learning Model (LLM) applications. They'll be guided through the practical steps involved in setting up this system, including library installation, data scraping and transformation, model training and evaluation, and even cost-saving measures, all while focusing on the balance between recall, cost, and latency in AI retrieval tasks.

Deep Memory for RAG Applications Across Legal, Financial, and Biomedical Industries

In this comprehensive lesson, students will learn how to enhance RAG systems using Deep Memory in conjunction with LLMs for applications within legal, financial, and biomedical sectors. Students will be guided through the process of preparing datasets, including gathering and chunking data as well as question and relevance score generation using LLMs. The lesson emphasizes the significant performance improvements offered by Deep Memory, such as an increase in retrieval accuracy without compromising search time, and demonstrates how to integrate and test this feature with real datasets—Legalbench, FinQA, and CORD-19. Additionally, students will gain insight into the practical implementation of Deep Memory through code examples and explore the advantages of Deep Memory over classic retrieval methods.

Fine-tuning vs RAG;

Introduction to Activeloop’s Deep Memory

Introduction

In this lesson, we will explore **optimization techniques** that maximize large language model performance. We will learn about the appropriate use of **prompt engineering**, retrieval augmented generation (**RAG**), and **fine-tuning**, distinguishing how each method contributes and their specific challenges.
A significant portion of the lesson will be dedicated to addressing the limitations of RAG systems in real-world applications. These mainly include maintaining high retrieval accuracy and ensuring accurate responses from LLMs. Much of our discussion will include Activeloop's Deep Memory, a technique designed to augment the retrieval precision of embeddings for user queries.
We will also perform a detailed comparison of empirical data, analyzing the differences in retrieval recall rates between systems employing Deep Memory and those that do not.

Overview of RAG Enhancement Techniques

Expanding on the discussion surrounding fine-tuning, retrieval-augmented generation, and prompt engineering, it's essential to understand each approach's distinct strengths, weaknesses, and most suitable applications.

Prompt engineering

Prompt engineering is often the **first step** in enhancing the performance of an LLM for specific tasks. This approach alone can be sufficient, especially for simpler or well-defined tasks.
Techniques like **few-shot prompting** can notably improve task performance. This method involves providing small task-specific examples to guide the LLM.
Chain of Thought (CoT) prompting can also improve reasoning capabilities and encourage the model to generate more detailed responses.
Combining Few-shot with RAG—using a tailored dataset of examples to retrieve the most relevant information for each query—can be more effective.

Fine-tuning

Fine-tuning enhances LLM’s capabilities in the following areas:

1. Modifying the **structure** or **tone** of responses.
2. Teaching the model to follow complex instructions.

For example, fine-tuning enables models to perform tasks like extracting JSON-formatted data from text, translating natural language into SQL queries, or adopting a specific writing style.
Fine-tuning demands a large, high-quality, task-specific dataset for effective training. You can start with a small dataset and training to see if the method works for your task.
Fine-tuning is less effective in adapting to new, rapidly changing data or unfamiliar queries beyond the training dataset. It's also not the best choice for incorporating new information into the model. Alternative methods, such as Retrieval-Augmented Generation, are more suitable.

Retrieval-Augmented Generation

RAG specializes in incorporating **external knowledge**, enabling the model to access current and varied information.
Real-Time Updates: It is more adept at dealing with evolving datasets and can provide more up-to-date responses.
Complexity in Integration: Setting up a RAG system is more complex than basic prompting, requiring extra components like a Vector Database and retrieval algorithms.

Data Management: Managing and updating the external data sources is crucial for maintaining the accuracy and relevance of its outputs.
Retrieval accuracy: Ensuring precise embedding retrieval is crucial in RAG systems to guarantee reliable and comprehensive responses to user queries. For that, we will demonstrate how Activeloop's Deep Memory method can greatly increase the recall of embedding retrieval.

RAG + Fine-tuning

Fine-tuning and RAGs are not mutually exclusive techniques.
Fine-tuning brings the advantage of customizing models for a specific style or format, which can be useful when using LLMs for specific domains such as medical, financial, or legal, requiring a highly specialized tone of writing.
When combined with RAG, the model becomes **adept in its specialized area** and gains access to a vast range of external information. The resulting model provides accurate responses in the niche area.
Implementing these two methods can demand considerable resources for setup and ongoing upkeep. It involves multiple training runs of fine-tuning with the data handling requirements inherent to RAG.

Enhanced RAG with Deep Memory

Deep Memory is a method developed by Activeloop to **boost the accuracy** of embedding retrieval for RAG systems integrated into the Deep Lake vector store database.
Central to its functionality is an **embedding transformation process**. Deep Memory trains a model that transforms embeddings into a space optimized for your use case. This reconfiguration significantly improves vector search accuracy.

Deep Memory is effective where **query reformulation, query transformation, or document re-ranking might cause latency and increased token usage**. It boosts retrieval capabilities without negatively impacting the system's performance.

The figure below shows the recall performance for different algorithms compared to Deep Memory.
Recall@1: This measures whether the top result (i.e., the first result) returned by the retrieval system is relevant to the query.
Recall@10: This metric assesses whether the relevant document is within the top 10 results returned by the retrieval system.

Comparison to Lexical search
BM25 is considered a state-of-the-art approach for "**lexical search**," based on the explicit presence of words (or lexicons) from the query in the documents. It's particularly effective for applications where the relevance of documents depends heavily on the presence of specific terms, such as in traditional search engines. However, **BM25 does not account for the semantic** relationships between words, where more advanced techniques like vector search with neural embeddings and semantic search come into play.

Overview of Deep Memory

In the figure above, we see the Inference and Training workflow:

1. **Embeddings**: Vector representation of a text sentence or set of words. We can create them using embedding models such as OpenAI's text-embedding-ada-002 or open-source models.
2. **Deep Memory Training**: A dataset of **query and context pairs** trains the Deep Memory model. This training process runs in Deep Lake Cloud, which provides the computational resources and infrastructure for handling the training.

3. **Deep Memory Inference**: The model enters the inference phase after training, which transforms query embeddings. We can use the Tensor Query Language (TQL) when running an inference/querying in the Vector Store.
4. **Transformed Embeddings**: The result of the inference process is a set of transformed embeddings optimized for a specific use case. This optimization means that the embeddings are now in a more conducive space for returning accurate results.
5. **Vector Search**: These optimized embeddings are used in a vector search, utilizing standard similarity search techniques (e.g., cosine similarity). The vector search is retrieving information, leveraging the refined embeddings to find and retrieve the most relevant data points for a given query.

Step by Step - Training a Deep Memory Model

Moving forward in our lesson, let's implement Deep Memory within our workflow to see firsthand how it impacts retrieval recall.
You can follow along with this Colab notebook.
As Step 0, please note that Deep Memory is a premium feature in Activeloop paid plans. As a reminder, you are able to redeem a free trial. As a part of the course, all course takers can redeem a free extended trial of one month for the Activeloop Growth plan by redeeming GENAI360 promo code at checkout. To redeem the plan, please create a Deep Lake Account, and on the following screen on account creation, please watch the following video.

1. Install the required libraries

```
!pip3 install deeplake langchain openai tiktoken llama-index
```

2. Set your ACTIVELOOP_TOKEN and OPENAI_API_KEY

```
import os, getpass

os.environ['ACTIVELOOP_TOKEN'] = getpass.getpass()

os.environ['OPENAI_API_KEY'] = getpass.getpass()
```

3. Download the data or use your own. Here, we download a text file hosted on GitHub.

```
!mkdir -p 'data/paul_graham/'

!curl 'https://raw.githubusercontent.com/run-llama/llama_index/main/docs/examples/data/paul_graham/paul_graham_essay.txt' -o 'data/paul_graham/paul_graham_essay.txt'
```

4. Create the Llama-index nodes/chunks

```
from llama_index.node_parser import SimpleNodeParser

from llama_index import SimpleDirectoryReader

documents = SimpleDirectoryReader("./data/paul_graham/").load_data()

node_parser = SimpleNodeParser.from_defaults(chunk_size=512)

nodes = node_parser.get_nodes_from_documents(documents)

# By default, the node/chunks ids are set to random uuids. To ensure same id's per run, we manually set them.

for idx, node in enumerate(nodes):

    node.id_ = f"node_{idx}"
```

```
print(f"Number of Documents: {len(documents)}")
print(f"Number of nodes: {len(nodes)} with the current chunk size of {node_parser.chunk_size}")
```

```
Number of Documents: 1
Number of nodes: 58 with the current chunk size of 512
```

The output.

5. Create a local Deep Lake vector store

```
from llama_index import VectorStoreIndex, ServiceContext, StorageContext
from llama_index.vector_stores import DeepLakeVectorStore
from llama_index.embeddings.openai import OpenAIEmbedding
from llama_index.llms import OpenAI
# Create a DeepLakeVectorStore locally to store the vectors
dataset_path = "./data/paul_graham/deep_lake_db"
vector_store = DeepLakeVectorStore(dataset_path=dataset_path, overwrite=True)
# LLM that will answer questions with the retrieved context
llm = OpenAI(model="gpt-3.5-turbo-1106")
embed_model = OpenAIEmbedding()
service_context = ServiceContext.from_defaults(embed_model=embed_model, llm=llm,)
storage_context = StorageContext.from_defaults(vector_store=vector_store)
vector_index = VectorStoreIndex(nodes, service_context=service_context, storage_context=storage_context, show_progress=True)
```

```
Uploading data to deeplake dataset.
100%|██████████| 58/58 [00:00<00:00, 274.94it/s]Dataset(path='./data/paul_graham/deep_lake_db', tensors=['text', 'metadata', 'embedding', 'id'])

  tensor      htype       shape      dtype  compression
  -------    -------     -------    -------  -------
   text       text       (58, 1)      str     None
 metadata     json       (58, 1)      str     None
 embedding  embedding  (58, 1536)  float32    None
```

```
id         text       (58, 1)      str      None
```

The output.

6. Now, let's upload the local Vectore Store to Activeloop's platform and convert it into a managed database.

```
import deeplake
local = "./data/paul_graham/deep_lake_db"
hub_path = "hub://genai360/optimization_paul_graham"
hub_managed_path = "hub://genai360/optimization_paul_graham_managed"
# First upload our local vector store
deeplake.deepcopy(local, hub_path, overwrite=True)
# Create a managed vector store under a different name
deeplake.deepcopy(hub_path, hub_managed_path, overwrite=True, runtime={"tensor_db": True})
```

7. Instantiate a Vector Store with the managed dataset that we just created.

```
db = DeepLakeVectorStore(dataset_path=hub_managed_path,
                         overwrite=False, read_only =True)
```

Now, let’s generate a dataset of Queries and Documents

4. Fetching our `docs` and `ids` from the vector store.

```
# Fetch dataset docs and ids
docs = db.vectorstore.dataset.text.data(fetch_chunks=True, aslist=True)['value']
ids = db.vectorstore.dataset.id.data(fetch_chunks=True, aslist=True)['value']
print(len(docs))
```

5. Generating a synthetic training dataset.
We need labeled data (`query` and `document_id` pairs) to train a Deep Memory model. Sometimes, it can be difficult to get labeled data when you are starting from scratch. This tutorial generates queries/questions using `gpt-3.5-turbo` from our existing documents.

```
from openai import OpenAI
client = OpenAI()


def generate_question(text):
    try:
        response = client.chat.completions.create(
            model="gpt-3.5-turbo-1106",
            messages=[
                {"role": "system", "content": "You are a world class expert for
generating questions based on provided context. \
                        You make sure the question can be answered by the text."},
                {
                    "role": "user",
                    "content": text,
                },
            ],
        )
        return response.choices[0].message.content
    except:
        question_string = "No question generated"
        return question_string


import random
from tqdm import tqdm
def generate_queries(docs: list[str], ids: list[str], n: int):
    questions = []
    relevances = []
    pbar = tqdm(total=n)
    while len(questions) < n:
        # 1. randomly draw a piece of text and relevance id
        r = random.randint(0, len(docs)-1)
        text, label = docs[r], ids[r]
        # 2. generate queries and assign and relevance id
        generated_qs = [generate_question(text)]
        if generated_qs == ["No question generated"]:
            print("No question generated")
            continue
        questions.extend(generated_qs)
        relevances.extend([[(label, 1)] for _ in generated_qs])
        pbar.update(len(generated_qs))

    return questions[:n], relevances[:n]
```

5.1 Launch the query generation process with a desired size of 40 queries/questions.

```
questions, relevances = generate_queries(docs, ids, n=40)
print(len(questions)) #40
print(questions[0])
```

You will have a list of generated questions and the associated contexts by running the two cells above.

6. Launch Deep Memory Training

```
from langchain.embeddings.openai import OpenAIEmbeddings
openai_embeddings = OpenAIEmbeddings()
job_id = db.vectorstore.deep_memory.train(
    queries=questions,
    relevance=relevances,
    embedding_function=openai_embeddings.embed_documents)
```

7. Starting DeepMemory training job

```
Your Deep Lake dataset has been successfully created!
Preparing training data for DeepMemory: Creating 20 embeddings in 1 batches of size 20:: 100%|██████████| 1/1 [06:36<00:00, 396.77s/it] DeepMemory training job started. Job ID: 657b3083d528b0fd224173c6
```

```
# During training you can check the status of the training run
db.vectorstore.deep_memory.status(job_id="657b3083d528b0fd224173c6")
```

```
--------------------------------------------------------------
|                  657b3083d528b0fd224173c6                  |
--------------------------------------------------------------
| status                      | completed                    |
--------------------------------------------------------------
```

```
| progress                    | eta: 0.9 seconds           |
|                             | recall@10: 60.00% (+25.00%) |
-------------------------------------------------------------
| results                     | recall@10: 60.00% (+25.00%) |
-------------------------------------------------------------
```

Output
We see an increase of 25% in recall@10 after finetuning.

8. Run a Deep Memory-enabled inference by setting `deep_memory=True`.

```
from llama_index.llms import OpenAI

query = "What are the main things Paul worked on before college?"


llm = OpenAI(model="gpt-3.5-turbo-1106")

embed_model = OpenAIEmbedding()


service_context = ServiceContext.from_defaults(embed_model=embed_model, llm=llm,)

storage_context = StorageContext.from_defaults(vector_store=vector_store)


db = DeepLakeVectorStore(dataset_path=hub_managed_path, overwrite=False,
read_only=True,)

vector_index = VectorStoreIndex.from_vector_store(db,
service_context=service_context, storage_context=storage_context, show_progress=True)


query_engine = vector_index.as_query_engine(similarity_top_k=3,
vector_store_kwargs={"deep_memory": True})

response_vector = query_engine.query(query)

print(response_vector.response)
```

9. Now, let's run a quantitative evaluation on another set of synthetically generated test queries.

```
# Generate validation queries

validation_questions, validation_relevances = generate_queries(docs, ids, n=40)

# Launch the evaluation function

recalls = db.vectorstore.deep_memory.evaluate(

    queries=validation_questions,
    relevance=validation_relevances,
    embedding_function=openai_embeddings.embed_documents,)
```

```
Embedding queries took 0.82 seconds
---- Evaluating without Deep Memory ----
Recall@1:         27.0%
Recall@3:         42.0%
Recall@5:         42.0%
Recall@10:        50.0%
Recall@50:        67.0%
Recall@100:       72.0%
---- Evaluating with Deep Memory ----
Recall@1:         32.0%
Recall@3:         45.0%
Recall@5:         48.0%
Recall@10:        55.0%
Recall@50:        69.0%
Recall@100:       73.0%
```

Output

Even with our new test dataset, we notice higher recall values using Deep Memory. Comparing these results with the training dataset highlights how a **query-context dataset** has better quality and represents your use case.

Conclusion

In this lesson, we explored the optimization techniques for large language models, covering prompt engineering as a first way to maximize LLM performance, fine-tuning, and Retrieval-Augmented Generation (RAG) for integrating external, up-to-date knowledge.

We also discussed combining fine-tuning with RAG for complex, domain-specific applications requiring considerable resources. A significant focus was on Activeloop's Deep Memory, which was integrated into RAG systems to enhance embedding retrieval accuracy. Deep Memory outperforms traditional methods like BM25 using lexical search and vector search using cosine similarity. We demonstrated it by getting higher recall values. It also efficiently reduces token usage in LLM prompts compared to query reformulation or transformation.

This approach addresses key embedding retrieval challenges and signals a promising future for increasingly capable and versatile LLMs.

Mastering Advanced RAG Techniques with LlamaIndex

Introduction

The Retrieval-Augmented Generation (RAG) pipeline heavily relies on retrieval performance guided by the adoption of various techniques and advanced strategies. Methods like query expansion, query transformations, and query construction each play a distinct role in refining the search process. These techniques enhance the scope of search queries and the overall result quality.

In addition to core methods, strategies such as **reranking** (with the Cohere Reranker), **recursive retrieval**, and **small-to-big retrieval** further enhance the retrieval process.

Together, these techniques create a comprehensive and efficient approach to information retrieval, ensuring that searches are wide-ranging, highly relevant, and accurate.

Querying in LlamaIndex

As mentioned in a previous lesson, the process of querying an index in LlamaIndex is structured around several key components.

- **Retrievers**: These classes are designed to retrieve a set of nodes from an index based on a given query. Retrievers source the relevant data from the index.
- **Query Engine**: It is the central class that processes a query and returns a response object. Query Engine leverages the retrievers and the response synthesizer modules to curate the final output.
- **Query Transform**: It is a class that enhances a raw query string with various transformations to improve the retrieval efficiency. It can be used in conjunction with a Retriever and a Query Engine.

Incorporating the above components can lead to the development of an effective retrieval engine, complementing the functionality of any RAG-based application. However, the relevance of search results can noticeably improve with more advanced techniques like query construction, query expansion, and query transformations.

Query Construction

Query construction in RAG converts user queries to a format that aligns with various data sources. This process involves transforming questions into vector formats for unstructured data, facilitating their comparison with vector representations of source documents to identify the most relevant ones. It also applies to structured data, such as databases where queries are formatted in a compatible language like SQL, enabling effective data retrieval.

The core idea is to answer user queries by leveraging the inherent structure of the data. For instance, a query like "movies about aliens in the year 1980" combines a semantic component like "aliens" (which will get better results if retrieved through vector storage) with a structured component like "year == 1980". The process involves translating a natural language query into the query language of a specific database, such as SQL for relational databases or Cypher for graph databases.

Incorporating different approaches to perform query construction depends on the specific use case. The first category includes the **MetadataFilter** classes for vector stores with metadata filtering, an auto-retriever that translates natural language into unstructured queries. This involves defining the data source, interpreting the user query, extracting logical conditions, and forming an unstructured request.

The other approach is **Text-to-SQL** for relational databases; converting natural language into SQL requests poses challenges like hallucination (creating fictitious tables or fields) and user errors (mis-spellings or irregularities). This is addressed by providing the LLM with an accurate database description and using few-shot examples to guide query generation.

Query Construction improves RAG answer quality with logical filter conditions inferred directly from user questions, and the retrieved text chunks passed to the LLM are refined before final answer synthesis.

💡

Query Construction is a process that translates natural language queries into structured or unstructured database queries, enhancing the accuracy of data retrieval.

Query Expansion

Query expansion works by extending the original query with additional terms or phrases that are related or synonymous.
For instance, if the original query is too narrow or uses specific terminology, query expansion can include broader or more commonly used terms relevant to the topic. Suppose the original query is "*climate change effects.*" Query expansion would involve adding related terms or synonyms to this query, such as "*global warming impact*," "*environmental consequences*," or "*temperature rise implications*."
One approach to do it is utilizing the `synonym_expand_policy` from the `KnowledgeGraphRAGRetriever` class. In the context of LlamaIndex, the effectiveness of query expansion is usually enhanced when combined with the Query Transform class.

Query Transformation

Query transformations modify the original query to make it more effective in retrieving relevant information. Transformations can include changes in the query's structure, the use of synonyms, or the inclusion of contextual information.
Consider a user query like "*What were Microsoft's revenues in 2021?*" To enhance this query through transformations, the original query could be modified to be more like *"Microsoft revenues 2021", which is more optimized for search engines and vector DBs.*
Query transformations involve **changing the structure** of a query to improve its performance.

Query Engine

A Query engine is a sophisticated interface designed to interact with data through natural language queries. It's a system that **processes queries and delivers responses**. As mentioned in previous lessons, multiple query engines can be combined for enhanced functionality, catering to complex data interrogation needs.
For a more interactive experience resembling a back-and-forth conversation, a Chat Engine can be used in scenarios requiring multiple queries and responses, providing a more dynamic and engaging interaction with data.

A basic usage of query engines is to call the `.as_query_engine()` method on the created Index. This section will include a step-by-step example of creating indexes from text files and utilizing query engines to interact with the dataset.

The first step is installing the required packages using Python package manager (PIP), followed by setting the API key environment variables.

```
pip install -q llama-index==0.9.14.post3 deeplake==3.8.8 openai==1.3.8 cohere==4.37

import os

os.environ['OPENAI_API_KEY'] = '<YOUR_OPENAI_API_KEY>'

os.environ['ACTIVELOOP_TOKEN'] = '<YOUR_ACTIVELOOP_KEY>'Copy
```

The next step is downloading the text file that serves as our source document. This file is a compilation of all the essays Paul Graham wrote on his blog, merged into a single text file. You have the option to download the file from the provided URL, or you can execute these commands in your terminal to create a directory and store the file.

```
mkdir -p './paul_graham/'

wget 'https://raw.githubusercontent.com/run-
llama/llama_index/main/docs/examples/data/paul_graham/paul_graham_essay.txt' -O
'./paul_graham/paul_graham_essay.txt'
```

Now, use the `SimpleDirectoryReader` within the LlamaIndex framework to read all files from a specified directory. This class will automatically cycle through the files, reading them as `Document` objects.

```
from llama_index import SimpleDirectoryReader

# load documents

documents = SimpleDirectoryReader("./paul_graham").load_data()
```

We can now employ the `ServiceContext` to divide the lengthy single document into several smaller chunks with some overlap. Following this, we can proceed to create the nodes out of the generated documents.

```
from llama_index import ServiceContext

service_context = ServiceContext.from_defaults(chunk_size=512, chunk_overlap=64)

node_parser = service_context.node_parser

nodes = node_parser.get_nodes_from_documents(documents)
```

The nodes must be stored in a vector store database to enable easy access. The `DeepLakeVectorStore` class can create an empty dataset when given a path. You can use `genai360` to access the processed dataset or alter the organization ID to your Activeloop username to store the data in your workspace.

```
from llama_index.vector_stores import DeepLakeVectorStore

my_activeloop_org_id = "genai360"

my_activeloop_dataset_name = "LlamaIndex_paulgraham_essays"
```

```
dataset_path = f"hub://{my_activeloop_org_id}/{my_activeloop_dataset_name}"

# Create an index over the documnts

vector_store = DeepLakeVectorStore(dataset_path=dataset_path, overwrite=False)
```

```
Your Deep Lake dataset has been successfully created!
```

The output.

The new database will be wrapped as a `StorageContext` object, which accepts nodes to provide the necessary context for establishing relationships if needed. Finally, the `VectorStoreIndex` takes in the nodes along with links to the database and uploads the data to the cloud. Essentially, it constructs the index and generates embeddings for each segment.

```
from llama_index.storage.storage_context import StorageContext

from llama_index import VectorStoreIndex

storage_context = StorageContext.from_defaults(vector_store=vector_store)

storage_context.docstore.add_documents(nodes)

vector_index = VectorStoreIndex(nodes, storage_context=storage_context)
```

```
Uploading data to deeplake dataset.

100%|██████████| 40/40 [00:00<00:00, 40.60it/s]

|Dataset(path='hub://genai360/LlamaIndex_paulgraham_essays', tensors=['text',
'metadata', 'embedding', 'id'])


  tensor      htype       shape      dtype  compression

  -------    -------     -------    -------  -------

   text       text       (40, 1)      str     None

 metadata     json       (40, 1)      str     None

 embedding  embedding  (40, 1536)  float32    None

    id        text       (40, 1)      str     None
```

The output.

The created index serves as the basis for defining the query engine. We initiate a query engine by using the vector index object and executing the `.as_query_engine()` method. The following code sets the `streaming` flag to True, which reduces idle waiting time for the end user (more details on this will follow). Additionally, it employs the `similarity_top_k` flag to specify the number of source documents it can consult to respond to each query.

```
query_engine = vector_index.as_query_engine(streaming=True, similarity_top_k=10)
```

The final step involves utilizing the `.query()` method to engage with the source data. We can pose questions and receive answers. As mentioned, the query engine employs retrievers and a response synthesizer to formulate an answer.

```
streaming_response = query_engine.query(     "What does Paul Graham do?")

streaming_response.print_response_stream()
```

```
Paul Graham is an artist and entrepreneur. He is passionate about creating paintings
that can stand the test of time. He has also co-founded Y Combinator, a startup
accelerator, and is actively involved in the startup ecosystem. While he has a
background in computer science and has worked on software development projects, his
primary focus is on his artistic pursuits and supporting startups.
```

The output.

The query engine can be configured into a streaming mode, providing a real-time response stream to enhance continuity and interactivity. This feature is beneficial in reducing idle time for end users. It allows users to view each word as generated, meaning they don't have to wait for the model to produce the entire text. To observe the impact of this feature, use the `print_response_stream` method on the response object of the query engine.

Sub Question Query Engine

Sub Question Query Engine, a more sophisticated querying method, can be employed to address the challenge of responding to complex queries. This engine can generate several sub-questions from the user's main question, answer each separately, and then compile the responses to construct the final answer. First, we must modify the previous query engine by removing the streaming flag, which conflicts with this technique.

```
query_engine = vector_index.as_query_engine(similarity_top_k=10)
```

We register the created `query_engine` as a tool by employing the `QueryEngineTool` class and compose metadata (description) for it. It is done to inform the framework about this tool's function and enable it to select the most suitable tool for a given task, especially when multiple tools are available. Then, the combination of the tools we declared earlier and the service context, which was previously defined, can be used to initialize the `SubQuestionQueryEngine` object.

```
from llama_index.tools import QueryEngineTool, ToolMetadata

from llama_index.query_engine import SubQuestionQueryEngine


query_engine_tools = [

    QueryEngineTool(

        query_engine=query_engine,

        metadata=ToolMetadata(

            name="pg_essay",
            description="Paul Graham essay on What I Worked On",
        ),
    ),
]

query_engine = SubQuestionQueryEngine.from_defaults(
    query_engine_tools=query_engine_tools,
    service_context=service_context,
    use_async=True )
```

The setup is ready to ask a question using the same `query` method. As observed, it formulates three questions, each responding to a part of the query, and attempts to find their answers individually. A response synthesizer then processes these answers to create the final output.

```
response = query_engine.query(
    "How was Paul Grahams life different before, during, and after YC?")
print( ">>> The final response:\n", response )
```

```
Generated 3 sub questions.
[pg_essay] Q: What did Paul Graham work on before YC?
[pg_essay] Q: What did Paul Graham work on during YC?
[pg_essay] Q: What did Paul Graham work on after YC?
[pg_essay] A: During YC, Paul Graham worked on writing essays and working on YC
itself.
[pg_essay] A: Before YC, Paul Graham worked on a variety of projects. He wrote
essays, worked on YC's internal software in Arc, and also worked on a new version of
Arc. Additionally, he started Hacker News, which was originally meant to be a news
aggregator for startup founders.
[pg_essay] A: After Y Combinator (YC), Paul Graham worked on various projects. He
focused on writing essays and also worked on a programming language called Arc.
However, he gradually reduced his work on Arc due to time constraints and the
infrastructure dependency on it. Additionally, he engaged in painting for a period of
time. Later, he worked on a new version of Arc called Bel, which he worked on
intensively and found satisfying. He also continued writing essays and exploring
other potential projects.


>>> The final response:
 Paul Graham's life was different before, during, and after YC. Before YC, he worked
on a variety of projects including writing essays, developing YC's internal software
in Arc, and creating Hacker News. During YC, his focus shifted to writing essays and
```

```
working on YC itself. After YC, he continued writing essays but also worked on
various projects such as developing the programming language Arc and later its new
version called Bel. He also explored other potential projects and engaged in painting
for a period of time. Overall, his work and interests evolved throughout these
different phases of his life.
```

The output.

Custom Retriever Engine

As you might have noticed, the choice of retriever and its parameters (e.g., the number of returned documents) influences the quality and relevance of the outcomes generated by the `QueryEngine`. LlamaIndex supports the creation of custom retrievers. Custom retrievers are a combination of different retriever styles, creating more nuanced retrieval strategies that adapt to distinct individual queries. The `RetrieverQueryEngine` operates with a designated retriever, which is specified at the time of its initialization. The choice of this retriever is vital as it significantly impacts the query results' outcome.

There are two main types of `RetrieverQueryEngine`:

1. **VectorIndexRetriever** fetches the top-k nodes that are most similar to the query. It focuses on relevance and similarity, ensuring the results closely align with the query's intent. It is the approach we used in previous subsections.
   Use Case: It is ideal for situations where precision and relevance to the specific query are paramount, like in detailed research or topic-specific inquiries.

2. **SummaryIndexRetriever** retrieves all nodes related to the query without prioritizing their relevance. This approach is less concerned with aligning closely to the specific context of the question and more about providing a broad overview.
   Use Case: Useful in scenarios where a comprehensive sweep of information is needed, regardless of the direct relevance to the specific terms of the query, like in exploratory searches or general overviews.

💡

You can read the following tutorial for a usage example:

Building and Advanced Fusion Retriever from Scratch.

Reranking

While any retrieval mechanism capable of extracting multiple chunks from a large document can be efficient to an extent, there is always a likelihood that it will select some irrelevant candidates among the results. Reranking is re-evaluating and re-ordering search results to present the most relevant options. By eliminating the chunks with lower scores, the final context given to the LLM boosts overall efficiency as the LLM gets more concentrated information.

The **Cohere Reranker** improves the performance of retrieving close content. While the semantic search component is already highly capable of retrieving relevant documents, the Rerank endpoint boosts the quality of the search results, especially for complex and domain-specific queries. It sorts the search results according to their relevance to the query. It is important to note that Rerank **is not a replacement for a search engine but a supplementary tool** for sorting search results in the most effective way possible for the user.

The process begins with grouping documents into batches, after which the LLM evaluates each batch, attributing relevance scores to them. The final step in the reranking process involves aggregating the most relevant documents from all these batches to form the final retrieval response. This method guarantees that the most pertinent information is highlighted and becomes the focal point of the search outcomes.

The necessary dependencies have already been installed; the only remaining step is to obtain your API key from Cohere.com and substitute it for the placeholder provided.

```
import cohere

import os

os.environ['COHERE_API_KEY'] = "<YOUR_COHERE_API_KEY>"

# Get your cohere API key on: www.cohere.com

co = cohere.Client(os.environ['COHERE_API_KEY'])

# Example query and passages

query = "What is the capital of the United States?"

documents = [

   "Carson City is the capital city of the American state of Nevada. At the  2010 United States Census, Carson City had a population of 55,274.",

   "The Commonwealth of the Northern Mariana Islands is a group of islands in the Pacific Ocean that are a political division controlled by the United States. Its capital is Saipan.",

   "Charlotte Amalie is the capital and largest city of the United States Virgin Islands. It has about 20,000 people. The city is on the island of Saint Thomas.",

   "Washington, D.C. (also known as simply Washington or D.C., and officially as the District of Columbia) is the capital of the United States. It is a federal district. ",

   "Capital punishment (the death penalty) has existed in the United States since before the United States was a country. As of 2017, capital punishment is legal in 30 of the 50 states.",

   "North Dakota is a state in the United States. 672,591 people lived in North Dakota in the year 2010. The capital and seat of government is Bismarck."]
```

We define a rerank object by passing both the query and the documents. We also set the `rerank_top_k` argument to 3; we specifically instruct the system to retrieve the top three highest-scored candidates by the model. In this case, the model employed for reranking is `rerank-multilingual-v2.0`.

```
results = co.rerank(query=query, documents=docs, top_n=3, model='rerank-english-v2.0')

# Change top_n to change the number of results returned. If top_n is not passed, all results will be returned.
```

```
for idx, r in enumerate(results):
  print(f"Document Rank: {idx + 1}, Document Index: {r.index}")
  print(f"Document: {r.document['text']}")
  print(f"Relevance Score: {r.relevance_score:.2f}")
  print("\n")
```

```
Document Rank: 1, Document Index: 3

Document: Washington, D.C. (also known as simply Washington or D.C., and officially
as the District of Columbia) is the capital of the United States. It is a federal
district. The President of the USA and many major national government offices are in
the territory. This makes it the political center of the United States of America.

Relevance Score: 0.99


Document Rank: 2, Document Index: 1

Document: The Commonwealth of the Northern Mariana Islands is a group of islands in
the Pacific Ocean that are a political division controlled by the United States. Its
capital is Saipan.

Relevance Score: 0.30


Document Rank: 3, Document Index: 5

Document: Capital punishment (the death penalty) has existed in the United States
since before the United States was a country. As of 2017, capital punishment is legal
in 30 of the 50 states. The federal government (including the United States military)
also uses capital punishment.

Relevance Score: 0.27
```

The output.

This can be accomplished using LlamaIndex in conjunction with Cohere Rerank. The rerank object can be integrated into a query engine, allowing it to manage the reranking process seamlessly in the background. We will use the same vector index defined earlier to prevent writing repetitive codes and integrate the rerank object with it. The `CohereRerank` class initiates a rerank object by taking in the API key and specifying the number of documents to be returned following the scoring process.

```
import os

from llama_index.postprocessor.cohere_rerank import CohereRerank

cohere_rerank = CohereRerank(api_key=os.environ['COHERE_API_KEY'], top_n=2)
```

The sample code.

Now, we can employ the same `as_query_engine` method and utilize the `node_postprocessing` argument to incorporate the reranker object. The retriever initially selects the top 10 documents based on semantic similarity, and then the reranker reduces this number to 2.

```
query_engine = vector_index.as_query_engine(
    similarity_top_k=10,
    node_postprocessors=[cohere_rerank] )

response = query_engine.query(
    "What did Sam Altman do in this essay?" )
print(response)Copy
```

```
Sam Altman was asked if he wanted to be the president of Y Combinator (YC) and
initially said no. However, after persistent persuasion, he eventually agreed to take
over as president starting with the winter 2014 batch.
```

The output.

Rerank computes a relevance score for the query and each document and returns a sorted list from the most to the least relevant document.

The reranking process in search systems offers numerous advantages, including practicality, enhanced performance, simplicity, and integration capabilities. It allows for augmenting existing systems without requiring complete overhauls, making it a cost-effective solution for improving search functionality. Reranking elevates search systems, which is particularly useful for complex, domain-specific queries in embedding-based systems.
The Cohere Rerank has proven to be effective in improving search quality across various embeddings, making it a reliable option for enhancing search results.

Advanced Retrievals

An alternative method for retrieving relevant documents involves using document summaries instead of extracting fragmented snippets or brief text chunks to respond to queries. This technique ensures that the answers reflect the entire context or topic being examined, offering a more thorough grasp of the subject.

Recursive Retrieval

The recursive retrieval method is particularly effective for **documents** with a **hierarchical** structure, allowing them to **form relationships and connections between the nodes**. According to Jerry Liu, founder of LlamaIndex, this is evident in cases like a PDF, which may contain "sub-data" such as tables and diagrams, alongside references to other documents. This technique can precisely navigate through the graph of connected nodes to locate information. This technique is versatile and can be applied in various scenarios, such as with node references, document agents, or even the query engine. For practical applications, including processing a PDF file and utilizing data from tables, you can refer to the tutorials in the LlamaIndex documentation here.

Small-to-Big retrieval

The small-to-big retrieval approach is a strategic method for information search, starting with concise, focused sentences to pinpoint the most relevant section of content with a question. It then passes a longer text to the model, allowing for a broader understanding of the context preceding and following the targeted area. This technique is particularly useful in situations where the initial query may not encompass the entirety of relevant information or where the data's relationships are intricate and multi-layered.

The LlamaIndex framework employs the Sentence Window Retrieval technique, which involves using the `SentenceWindowNodeParser` class to break down documents into individual sentences per node. Each node includes a "window" that encompasses the sentences surrounding the main node sentence. (It is 5 sentences before and after each node by default) During retrieval, the single sentences initially retrieved are substituted with their respective windows, including the adjacent sentences, through the `MetadataReplacementNodePostProcessor`. This substitution ensures that the Large Language Model receives a comprehensive view of the context surrounding each sentence.

💡

The small-to-big retrieval method begins by extracting small, targeted text segments and then presents the larger text chunks from which these segments were derived to the Large Language Model, thereby offering a more complete scope of information.

You can follow a hands-on tutorial to implement this technique from the documentation here.

Conclusion

Effective information retrieval involves mastering techniques such as query expansion, query transformations, and query construction, coupled with advanced strategies like reranking, recursive retrieval, and small-to-big retrieval. Together, these techniques enhance the search process by increasing accuracy and broadening the range of results. By incorporating these methods, information retrieval systems become more proficient in providing precise results, essential for improving the performance of RAG-based applications.

>> Notebook.

Production-Ready RAG Solutions with LlamaIndex

Introduction

LlamaIndex is a framework for developing data-driven LLM applications, offering data ingestion, indexing, and querying tools. It plays a key role in incorporating additional data sources into LLMs, which is essential for RAG systems.
In this lesson, we will explore how RAG-based applications can be improved by focusing on building production-ready code with a focus on data considerations. We'll discuss how to improve RAG retrieval performance through clear data definition and state management. Additionally, we will cover how to use LLMs to extract metadata to boost retrieval efficiency.
The lesson also covers the concerns about how embedding references and summaries in text chunks can significantly improve retrieval performance and the capability of LLMs to infer metadata filters for structured retrieval. We'll also discuss fine-tuning embedding representations in LLM applications to achieve optimal retrieval performance.

Challenges of RAG Systems

Retrieval-Augmented Generation (RAG) applications present unique challenges crucial for their successful implementation. In this section, we explore the dynamic management of data, ensuring varied and effective data representation and adhering to regulatory standards, highlighting the intricate balance required in RAG systems.

Document Updates and Stored Vectors

A significant challenge in RAG systems is **keeping up with changes in documents** and ensuring these updates are accurately reflected in the stored vectors. When documents are modified, added, or removed, the corresponding vectors need to be updated to maintain the accuracy and relevance of the retrieval system. Not addressing this can lead to outdated or irrelevant data retrieval, negatively impacting the system's effectiveness.
Implementing dynamic updating mechanisms for vectors can greatly improve the system's ability to provide relevant and current information, enhancing its overall performance.

Chunking and Data Distribution

The granularity level is vital in achieving accurate retrieval results. If the chunk size is too large, important details might be missed; if it's too small, the system might get bogged down in details and miss the bigger picture. This setting requires testing and refinement tailored to the specific characteristics of the data and its application.

Diverse Representations in Latent Space

The presence of different representations in the same latent space can be challenging (e.g., for **representing a paragraph of text versus representing a table or an image**). These diverse representations can cause conflicts or inconsistencies when retrieving information, leading to less accurate results.

Compliance

Compliance is another critical issue, especially when implementing RAG systems in regulated industries or environments with strict data handling requirements, particularly for private documents with limited access. Non-compliance can lead to **legal issues** (think about a finance application), data breaches, or misuse of sensitive information. Ensuring the system adheres to relevant laws, regulations, and ethical guidelines prevents these risks. It increases the system's reliability and trustworthiness, vital for its successful deployment.

Optimization

Understanding the intricacies of challenges in RAG systems and their solutions is crucial for boosting their overall effectiveness. We will explore several optimization strategies that can contribute to performance enhancement.

Model Selection and Hybrid Retrieval

Selecting appropriate models for the embedding and generation phases is critical. Choosing efficient and cheap embedding models can minimize costs while maintaining performance levels, but not in the generation process where an LLM is needed. Different options are available for both phases, including proprietary models with API access, such as OpenAI or Cohere, as well as open-source alternatives like LLaMA-2 and Mistral, which offer the flexibility of self-hosting or using third-party APIs. This choice should be based on the unique needs and resources of the application.

It’s worth noting that, in some retrieval systems**, balancing latency with quality** is essential. Combining different methods, like **keyword and embedding retrieval with reranking**, ensures that the system is **fast enough** to meet user expectations while still providing accurate results.

LlamaIndex also offers extensive integration options with various platforms, allowing for easy selection and comparison between different providers. This facilitates finding the optimal balance between cost and performance for specific needs.

CPU-Based Inference

In production, relying on GPU-based inference can incur substantial costs. Investigating options like better hardware or refining the inference code can lower the costs in large-scale applications where small inefficiencies can accumulate into considerable expenses. This approach is particularly important when using open-source models from sources such as HuggingFace hub.

Intel®'s **advanced optimization** technologies help with the **efficient fine-tuning** and **inference** of neural network models on **CPUs**. The 4th Gen Intel® Xeon® Scalable processors come with Intel® Advanced Matrix Extensions (Intel® AMX), an AI-enhanced acceleration feature. Each core of these processors includes integrated BF16 and INT8 accelerators, contributing to the acceleration of deep learning fine-tuning and inference speed. Additionally, libraries such as Intel Extension for PyTorch and Intel® Extension for Transformers further optimize the performance of neural network models demanding computations on CPUs.

Retrieval Performance

In RAG applications, the primary method involves dividing the data into smaller, independent units and housing them within a vector dataset. However, this often leads to failures during document retrieval, as individual segments may lack the broader context necessary to answer specific queries. LlamaIndex offers features designed to construct a **network of interlinked chunks** (**nodes**), along with retrieval tools. These tools **improve search capabilities** by augmenting user queries, extracting key terms, or navigating through the connected nodes to locate the necessary information for answering queries.

Advanced data management tools can help organize, index, and retrieve data more effectively. New tooling can also assist in handling large volumes of data and complex queries, which are common in RAG systems.

The Role of the Retrieval Step

While the role of the retrieval step is frequently underestimated, it is vital for the effectiveness of the RAG pipeline. The techniques employed in this phase significantly influence the relevance and contextuality of the output. The LlamaIndex framework provides a variety of retrieval methods, complete with practical examples for different use cases, including the following examples, to name a few.

- Combining **keyword + embedding search** in a hybrid approach can enhance retrieval of specific queries. [link]
- **Metadata filtering** can provide additional context and improve the performance of the RAG pipeline. [link]
- **Re-ranking orders** the search results by considering the recency of data to the user's input query. [link]
- **Indexing documents by summaries** and retrieving relevant information within the document. [link]

Additionally, augmenting chunks with metadata will provide more context and enhance retrieval accuracy by defining node relationships between chunks for retrieval algorithms. Language models can help extract page numbers and other annotations from text chunks. Decouple embeddings from raw text chunks to avoid biases and improve context capture. Embedding references, summaries in text chunks, and text at the sentence level improves retrieval performance by fetching granular pieces of information. Organizing data with metadata filters helps with structured retrieval by ensuring relevant chunks are fetched.

RAG Best Practices

Here are some good practices for dealing with RAG:

Fine-Tuning the Embedding Model

Fine-tuning the embedding model involves several key steps (like the creation of the training set) to enhance the embedding performance.
Initially, it's necessary to get the training set, which can be done by generating synthetic questions/answers from random documents. The next phase is fine-tuning the model, where adjustments are made to optimize its functioning. Following this, the model can optionally undergo an evaluation process to assess its improvements. The reported numbers from LlamaIndex show that the fine-tuning process can yield a 5-10% improvement in retrieval metrics, enabling the enhanced model to be effectively integrated into RAG applications.
LlamaIndex offers capabilities for various fine-tuning types, including adjustments to embedding models, adaptors, and even routers, to boost the overall efficiency of the pipeline. This method supports the model by improving its capacity to develop more impactful embedding representations, extracting deeper and more significant insights from the data.
You can read here for more information.

LLM Fine-Tuning

Fine-tuning the LLM creates a model that effectively grasps the overall style of the dataset, leading to the generation of more precise responses. Fine-tuning the generative model brings several advantages, such as reducing hallucinations during output formation, which are typically challenging to eliminate through prompt engineering. Moreover, the refined model has a deeper understanding of the dataset, enhancing performance even in smaller models. This means achieving performance comparable to GPT-4 while employing more cost-effective alternatives like GPT-3.5.
LlamaIndex offers a variety of fine-tuning schemas tailored to specific goals. It enhances model capabilities for use cases such as following a predetermined output structure, boosting its

proficiency in converting natural language into SQL queries or augmenting its capacity for memorizing new knowledge. The documentation section has several examples.

Evaluation

Regularly monitoring the performance of your RAG pipeline is a recommended practice, as it allows for assessing changes and their impact on the overall results. While evaluating a model's response, which can be highly subjective, is challenging, there are several methods available to track progress effectively.

LlamaIndex provides modules for assessing the quality of the generated results and the retrieval process. Response evaluation focuses on whether the response aligns with the retrieved context and the initial query and if it adheres to the reference answer or set guidelines. For retrieval evaluation, the emphasis is on the relevance of the sources retrieved in relation to the query.

A common method for assessing responses involves employing a proficient LLM, such as GPT-4, to evaluate the generated responses against various criteria. This evaluation can encompass aspects like correctness, semantic similarity, and faithfulness, among others. Please refer to the following tutorial for more information on the evaluation process and techniques.

Generative Feedback Loops

A key aspect of generative feedback loops is injecting data into prompts. This process involves feeding specific data points into the RAG system to generate contextualized outputs. Once the RAG system generates descriptions or vector embeddings, these outputs can be stored in the database. The creation of a loop where generated data is continually used to enrich and update the database can improve the system's ability to produce better outputs.

Hybrid Search

It is essential to keep in mind that embedding-based retrieval is not always practical for entity lookup. Implementing a hybrid search that combines the benefits of keyword lookup with additional context from embeddings can yield better results, offering a balanced approach between specificity and context.

Conclusion

In this lesson, we covered the challenges and optimization strategies of Retrieval-Augmented Generation (RAG) systems, emphasizing the importance of effective data management, diverse representations in latent space, and compliance in complex environments.

We highlighted techniques like dynamic updating of vectors, chunk size optimization, and hybrid retrieval approaches. We also explored the role of **LlamaIndex** in enhancing retrieval performance through data organization and the significance of fine-tuning embedding and LLM models for optimal RAG applications.

Lastly, we recommended regular evaluation and the use of generative feedback loops and hybrid searches for maintaining and improving RAG systems.

Iterative Optimization of LlamaIndex RAG Pipeline: A Step-by-Step Approach

Introduction

In previous lessons, we learned about advanced techniques and evaluation metrics for LlamaIndex Retrieval-Augmented Generation (RAG) pipelines. Building on this knowledge, we now focus on optimizing a LlamaIndex RAG pipeline through a series of iterative evaluations. We aim to enhance the system's ability to retrieve and generate accurate and relevant information.
Here's our step-by-step plan:

1. **Baseline Evaluation**: Construct a standard LlamaIndex RAG pipeline and establish an initial performance baseline.
   a. Adjusting TOP_K Retrieval Values: Experiment with different values of k (1, 3, 5, 7) to understand their effect on the accuracy of retrieved information and the relevance of generated answers.
2. **Testing Different Embedding Models**: Evaluate models such as "text-embedding-ada-002" and "cohere/embed-english-v3.0" to identify the most effective one for our pipeline.
3. **Incorporating a Reranker**: Implement a reranking mechanism to refine the document selection process of the retriever.
4. **Employing a Deep Memory Approach**: Investigate the impact of a deep memory component on the accuracy of information retrieval.

Through these steps, we aim to refine our RAG system systematically, enhancing its performance by providing accurate and relevant information.
The code for this lesson is also available through a Colab notebook, where you can follow along.

1. Baseline evaluation

The first step is installing the required Python packages.

```
!pip3 install deeplake llama_index langchain openai tiktoken cohere pandas torch sentence-
transformers
```

Here, you can set our API keys. You can skip this step if you plan to use other services.

```
import os

os.environ['OPENAI_API_KEY'] = '<YOUR_OPENAI_API_KEY>'

os.environ['ACTIVELOOP_TOKEN'] = '<YOUR_ACTIVELOOP_KEY>'

os.environ['COHERE_API_KEY'] = '<COHERE_API_KEY>'
```

We download the data, which is a single text file. You can use this or replace it with your own data.

```
!mkdir -p 'data/paul_graham/'
```

```
!curl 'https://raw.githubusercontent.com/run-llama/llama_index/main/docs/examples/data/paul_graham/paul_graham_essay.txt' -o 'data/paul_graham/paul_graham_essay.txt'
```

Let’s load the Data and build LlamaIndex nodes/chunks.

```python
from llama_index.node_parser import SimpleNodeParser

from llama_index import SimpleDirectoryReader


# First we create Document LlamaIndex objects from the text data

documents = SimpleDirectoryReader("./data/paul_graham/").load_data()

node_parser = SimpleNodeParser.from_defaults(chunk_size=512)

nodes = node_parser.get_nodes_from_documents(documents)


# By default, the node/chunks ids are set to random uuids. To ensure same id's per run, we manually set them.

for idx, node in enumerate(nodes):

    node.id_ = f"node_{idx}"


print(f"Number of Documents: {len(documents)}")

print(f"Number of nodes: {len(nodes)} with the current chunk size of {node_parser.chunk_size}")
```

```
Number of Documents: 1

Number of nodes: 58 with the current chunk size of 512
```

The output.

The next step is to create a LlamaIndex `VectorStoreIndex` object and use a `DeepLakeVectorStore` to store the vector embeddings.
We also choose `gpt-3.5-turbo-1106` as our LLM and OpenAI’s embedding model `text-embedding-ada-002`

```python
from llama_index import VectorStoreIndex, ServiceContext, StorageContext

from llama_index.vector_stores import DeepLakeVectorStore

from llama_index.embeddings.openai import OpenAIEmbedding

from llama_index.llms import OpenAI


# Create a local Deep Lake VectorStore
```

```
dataset_path = "./data/paul_graham/deep_lake_db"
vector_store = DeepLakeVectorStore(dataset_path=dataset_path, overwrite=True)

# LLM that will answer questions with the retrieved context
llm = OpenAI(model="gpt-3.5-turbo-1106")
# We use OpenAI's embedding model "text-embedding-ada-002"
embed_model = OpenAIEmbedding()

service_context = ServiceContext.from_defaults(embed_model=embed_model, llm=llm,)
storage_context = StorageContext.from_defaults(vector_store=vector_store)

vector_index = VectorStoreIndex(nodes, service_context=service_context,
storage_context=storage_context, show_progress=True)
```

```
Generating embeddings: 100%
58/58 [00:06<00:00, 8.75it/s]
Uploading data to deeplake dataset.
100%|██████████| 58/58 [00:00<00:00,
169.79it/s]Dataset(path='./data/paul_graham/deep_lake_db', tensors=['text',
'metadata', 'embedding', 'id'])

  tensor      htype       shape      dtype  compression
  -------    -------     -------    -------  -------
   text       text       (58, 1)      str     None
 metadata     json       (58, 1)      str     None
 embedding  embedding  (58, 1536)  float32    None
    id        text       (58, 1)      str     None
```

The output.

With the vector index, we can now build a `QueryEngine`, which generates answers with the LLM and the retrieved chunks of text.

```
query_engine = vector_index.as_query_engine(similarity_top_k=10)
response_vector = query_engine.query("What are the main things Paul worked on before
college?")
print(response_vector.response)
```

```
Before college, Paul worked on writing and programming.
```

The output.

Now that we have a simple RAG pipeline, we can evaluate it. For that, we need a dataset. Since we don’t have one, we will generate one. `LlamaIndex` offers a `generate_question_context_pairs` module specifically for generating questions and context pairs. We will use that dataset to assess the RAG chunk retrieval and response capabilities.
Let’s also save the generated dataset in JSON format for later use. In this case we only generate **58 question and context pairs**, but you can increase the number of samples in the dataset for a more thorough evaluation.

```python
from llama_index.evaluation import generate_question_context_pairs

qc_dataset = generate_question_context_pairs(

    nodes,

    llm=llm,

    num_questions_per_chunk=1 )

# We can save the dataset as a json file for later use.

qc_dataset.save_json("qc_dataset.json")
```

```
100%|██████████| 58/58 [01:30<00:00,  1.56s/it]
```

The output.

You can load the dataset from your local disk if you have already generated it.

```python
from llama_index.finetuning.embeddings.common import ( EmbeddingQAFinetuneDataset,)

qc_dataset = EmbeddingQAFinetuneDataset.from_json("qc_dataset.json")
```

We now have a synthetic dataset, but here, you must take the time to review it or even consider building it manually. Doing so will increase the accuracy of your RAG pipeline evaluations.

If you want more control over the quality of the generated dataset, you can look into modifying the prompt. This is the current default for the `generate_question_context_pairs` function.

```python
DEFAULT_QA_GENERATE_PROMPT_TMPL = """\

Context information is below.
```

```
---------------------
{context_str}
---------------------

Given the context information and not prior knowledge.
generate only questions based on the below query.

You are a Teacher/ Professor. Your task is to setup \
{num_questions_per_chunk} questions for an upcoming \
quiz/examination. The questions should be diverse in nature \
across the document. Restrict the questions to the \
context information provided."
"""
```

With the generated dataset, we can first start with the retrieval evaluations.
We will use the `RetrieverEvaluator` class available in LlamaIndex to measure the Hit Rate and Mean Reciprocal Rank (MRR).

Hit Rate:
Think of the Hit Rate as playing a game of guessing. You're given a question and need to guess the correct answer from a list of options. The Hit Rate measures how often you guess the correct answer by only looking at your top few guesses. You have a high Hit Rate if you often find the right answer in your first few guesses.
So, in a retrieval system, it's about how frequently the system finds the correct document within its top 'k' picks (where 'k' is a number you decide, like top 5 or top 10).

Mean Reciprocal Rank (MRR):
MRR is like measuring how quickly you can find a treasure in a list of boxes. Imagine you have a row of boxes, and only one has a treasure. The MRR calculates how close to the start of the row the treasure box is, on average.
If the treasure is always in the first box you open, you're doing great and have an MRR of 1. If it's in the second box, the score is 1/2, since you took two tries to find it. If it's in the third box, your score is 1/3, and so on. MRR averages these scores across all your searches. So, for a retrieval system, MRR looks at where the correct document ranks in the system's guesses. If it's usually near the top, the MRR will be high, indicating good performance.

In summary, **Hit Rate tells you how often the system gets it right in its top guesses**, and **MRR tells you how close to the top the right answer usually is**. Both metrics are useful for evaluating the effectiveness of a retrieval system, like how well a search engine or a recommendation system works.

First, we define a function to display the Retrieval evaluation results in table format.

```python
import pandas as pd
def display_results_retriever(name, eval_results):
    """Display results from evaluate."""
    metric_dicts = []
    for eval_result in eval_results:
        metric_dict = eval_result.metric_vals_dict
        metric_dicts.append(metric_dict)

    full_df = pd.DataFrame(metric_dicts)

    hit_rate = full_df["hit_rate"].mean()
    mrr = full_df["mrr"].mean()

    metric_df = pd.DataFrame(
        {"Retriever Name": [name], "Hit Rate": [hit_rate], "MRR": [mrr]})
    return metric_df
```

Then, Run the evaluation procedure.

```python
from llama_index.evaluation import RetrieverEvaluator
# We can evaluate the retievers with different top_k values.
for i in [2, 4, 6, 8, 10]:
    retriever = vector_index.as_retriever(similarity_top_k=i)
    retriever_evaluator = RetrieverEvaluator.from_metric_names(
        ["mrr", "hit_rate"], retriever=retriever)
    eval_results = await retriever_evaluator.aevaluate_dataset(qc_dataset)
    print(display_results_retriever(f"Retriever top_{i}", eval_results))
```

```
Retriever Name  Hit Rate       MRR
0  Retriever top_2  0.827586  0.702586
    Retriever Name  Hit Rate       MRR
0  Retriever top_4  0.913793  0.729167
    Retriever Name  Hit Rate       MRR
0  Retriever top_6  0.922414  0.730891
    Retriever Name  Hit Rate       MRR
```

```
0  Retriever top_8  0.956897  0.735509
     Retriever Name  Hit Rate       MRR
0  Retriever top_10  0.982759  0.738407
```

The output.

We notice that the Hit Rate increases as the top_k value increases, which is what we can expect. We're increasing the probability of the correct answer being included in the returned set.
But how does that impact the quality of the generated answers?

Evaluation for Relevancy and Faithfulness metrics.

Relevancy evaluates whether the retrieved context and answer are relevant to the query.
Faithfulness evaluates if the answer is faithful to the retrieved contexts or, in other words, whether there’s a **hallucination**.
LlamaIndex includes functions that evaluate both metrics using an LLM as the judge. **GPT4 will be used as the judge.**
Now, let's see how the top_k value affects these two metrics.

```
from llama_index.evaluation import RelevancyEvaluator, FaithfulnessEvaluator,
BatchEvalRunner


for i in [2, 4, 6, 8, 10]:
    # Set Faithfulness and Relevancy evaluators
    query_engine = vector_index.as_query_engine(similarity_top_k=i)
    # While we use GPT3.5-Turbo to answer questions
    # we can use GPT4 to evaluate the answers.
    llm_gpt4 = OpenAI(temperature=0, model="gpt-4-1106-preview")
    service_context_gpt4 = ServiceContext.from_defaults(llm=llm_gpt4)
    faithfulness_evaluator =
FaithfulnessEvaluator(service_context=service_context_gpt4)
    relevancy_evaluator = RelevancyEvaluator(service_context=service_context_gpt4)


    # Run evaluation
    queries = list(qc_dataset.queries.values())
    batch_eval_queries = queries[:20]


    runner = BatchEvalRunner(
    {"faithfulness": faithfulness_evaluator, "relevancy": relevancy_evaluator},
    workers=8)
    eval_results = await runner.aevaluate_queries(query_engine,
queries=batch_eval_queries)

    faithfulness_score = sum(result.passing for result in
eval_results['faithfulness']) / len(eval_results['faithfulness'])

    print(f"top_{i} faithfulness_score: {faithfulness_score}")


    relevancy_score = sum(result.passing for result in eval_results['faithfulness'])
/ len(eval_results['relevancy'])

    print(f"top_{i} relevancy_score: {relevancy_score}")Copy
```

```
top_2 faithfulness_score: 0.95

top_2 relevancy_score: 0.95

top_4 faithfulness_score: 0.95

top_4 relevancy_score: 0.95

top_6 faithfulness_score: 0.95

top_6 relevancy_score: 0.95

top_8 faithfulness_score: 1.0

top_8 relevancy_score: 1.0

top_10 faithfulness_score: 1.0

top_10 relevancy_score: 1.0
```

The output.

We can notice the relevancy and faithfulness scores increase as the Top_k value increases. We also get a perfect score using eight retrieved chunks as context.

Remember that these scores come from another LLM, such as GPT4. Since LLM outputs are not deterministic, you can expect different results if you run the evaluations repeatedly.

This is **the LlamaIndex Relevancy prompt default template**.

DEFAULT_EVAL_TEMPLATE = **PromptTemplate**("Your task is to evaluate if the response for the query \ is in line with the context information provided.\n" "You have two options to answer. Either YES/ NO.\n" "Answer - YES, if the response for the query \ is in line with context information otherwise NO.\n" "Query and Response: \n {query_str}\n" "Context: \n {context_str}\n" "Answer: ")

2. Changing the embedding model

Now that we have the baseline evaluation score, we can start changing some modules of our LlamaIndex RAG pipeline.
We can start by changing the embedding model. Here, we will be testing the cohere embedding model `embed-english-v3.0` instead of OpenAI's `text-embedding-ada-002`.

```python
import os
from llama_index import VectorStoreIndex, ServiceContext, StorageContext
from llama_index.vector_stores import DeepLakeVectorStore
from llama_index.embeddings.cohereai import CohereEmbedding
from llama_index.llms import OpenAI

# Create another local DeepLakeVectorStore to store the embeddings
dataset_path = "./data/paul_graham/deep_lake_db_1"
vector_store = DeepLakeVectorStore(dataset_path=dataset_path, overwrite=False)

llm = OpenAI(model="gpt-3.5-turbo-1106")
embed_model = CohereEmbedding(
    cohere_api_key=os.getenv('COHERE_API_KEY'),
    model_name="embed-english-v3.0",
    input_type="search_document",)

service_context = ServiceContext.from_defaults(embed_model=embed_model, llm=llm,)
storage_context = StorageContext.from_defaults(vector_store=vector_store)
vector_index = VectorStoreIndex(nodes, service_context=service_context,
storage_context=storage_context, show_progress=True)
```

```
Generating embeddings: 100%
58/58 [00:02<00:00, 23.68it/s]
Uploading data to deeplake dataset.
100%|██████████| 58/58 [00:00<00:00,
315.69it/s]Dataset(path='./data/paul_graham/deep_lake_db_1', tensors=['text',
'metadata', 'embedding', 'id'])

  tensor      htype      shape      dtype  compression
  -------    -------    -------    -------  -------
   text       text      (58, 1)     str     None
 metadata     json      (58, 1)     str     None
 embedding  embedding  (58, 1024)  float32  None
    id        text      (58, 1)     str     None
```

The output.

We run the retrieval evaluation using these new embeddings.

```
from llama_index.evaluation import RetrieverEvaluator
embed_model.input_type = "search_query"
retriever = vector_index.as_retriever(similarity_top_k=10, embed_model=embed_model)

retriever_evaluator = RetrieverEvaluator.from_metric_names(
    ["mrr", "hit_rate"], retriever=retriever)
eval_results = await retriever_evaluator.aevaluate_dataset(qc_dataset)
print(display_results_retriever(f"Retriever_cohere_embeds", eval_results))
```

```
Retriever Name  Hit Rate       MRR
0  Retriever_cohere_embeds  0.965517  0.754823
```

The output.

These embeddings show a lower Hit Rate but a better MRR value.

In this tutorial, we test the cohere embeddings, but you can also try any embedding models from the Hugging Face hub. The models at the top of the mteb/leaderboard are a good choice to try.

If pre-trained embedding models do not perform well on your data, consider fine-tuning your own embedding model.

3. Incorporating a Reranker

Here, we will be testing three different Rerankers that we learned about in previous lessons.

- `cross-encoder/ms-marco-MiniLM-L-6-v2` from the Hugging Face hub.
- LlamaIndex's `LLMRerank`
- Cohere's `CohereRerank`.

```
from llama_index.postprocessor.cohere_rerank import CohereRerank
from llama_index.indices.postprocessor import SentenceTransformerRerank, LLMRerank

st_reranker = SentenceTransformerRerank(    top_n=5, model="cross-encoder/ms-marco-
MiniLM-L-6-v2")
llm_reranker = LLMRerank(    choice_batch_size=4, top_n=5,)
```

```
cohere_rerank = CohereRerank(api_key=os.getenv('COHERE_API_KEY'), top_n=10)


for reranker in [cohere_rerank, st_reranker, llm_reranker]:
    retriever_with_reranker = vector_index.as_retriever(similarity_top_k=10,
postprocessor=reranker, embed_model=embed_model)
    retriever_evaluator_1 = RetrieverEvaluator.from_metric_names(
        ["mrr", "hit_rate"], retriever=retriever_with_reranker )
    eval_results1 = await retriever_evaluator_1.aevaluate_dataset(qc_dataset)
    print(display_results_retriever("Retriever with added Reranker", eval_results1))
```

```
config.json: 100%
794/794 [00:00<00:00, 23.6kB/s]
pytorch_model.bin: 100%
90.9M/90.9M [00:00<00:00, 145MB/s]
tokenizer_config.json: 100%
316/316 [00:00<00:00, 11.0kB/s]
vocab.txt: 100%
232k/232k [00:00<00:00, 3.79MB/s]
special_tokens_map.json: 100%
112/112 [00:00<00:00, 3.88kB/s]
                 Retriever Name  Hit Rate       MRR
0  Retriever with added Reranker  0.965517  0.754823
                 Retriever Name  Hit Rate       MRR
0  Retriever with added Reranker  0.965517  0.754823
                 Retriever Name  Hit Rate       MRR
0  Retriever with added Reranker  0.965517  0.754823
```

The output.

Here, we unfortunately don't see a significant improvement in the retriever's performance. We suspect it is mainly caused by the evaluation dataset we've built. Rerankers can nonetheless offer great benefits depending on your application and are easy to implement.

4. Employing Deep Memory

Activeloop's Deep Memory is a feature that introduces a tiny neural network layer trained to match user queries with relevant data from a corpus. While this addition incurs minimal latency during search, it can **boost retrieval accuracy by up to 27%.**

First, let's reuse and convert our generated dataset into a format Deep Memory expects. We need queries and relevant IDs.

```
def create_query_relevance(qa_dataset):
    """Function for converting LlamaIndex dataset to correct format for deep memory
training"""
    queries = [text for _, text in qa_dataset.queries.items()]
    relevant_docs = qa_dataset.relevant_docs
    relevance = []
    for doc in relevant_docs:
        relevance.append([(relevant_docs[doc][0], 1)])
    return queries, relevance
train_queries, train_relevance = create_query_relevance(qc_dataset)
print(len(train_queries))
```

Now, let's upload our baseline Vectore Store on Activeloop's cloud platform and convert it into a managed database.

```
import deeplake
local = "./data/paul_graham/deep_lake_db"
hub_path = "hub://genai360/optimization_paul_graham"
hub_managed_path = "hub://genai360/optimization_paul_graham_managed"


# First upload our local vector store
deeplake.deepcopy(local, hub_path, overwrite=True)
# Create a managed vector store
deeplake.deepcopy(hub_path, hub_managed_path, overwrite=True, runtime={"tensor_db":
True})
```

You can replace the paths using your organization name and database name.
Let's create a LlamaIndex RAG pipeline using our new managed vector store.

```
import os
from llama_index import VectorStoreIndex, ServiceContext, StorageContext
from llama_index.vector_stores import DeepLakeVectorStore
from llama_index.embeddings.openai import OpenAIEmbedding
from llama_index.llms import OpenAI
```

```python
vector_store = DeepLakeVectorStore(dataset_path=hub_managed_path, overwrite=False,
runtime={"tensor_db": True}, read_only=True)

llm = OpenAI(model="gpt-3.5-turbo-1106")

embed_model = OpenAIEmbedding()


service_context = ServiceContext.from_defaults(embed_model=embed_model, llm=llm,)

storage_context = StorageContext.from_defaults(vector_store=vector_store)


vector_index =
VectorStoreIndex.from_vector_store(vector_store,service_context=service_context,
storage_context=storage_context, use_async=False, show_progress=True)
```

```
Deep Lake Dataset in hub://genai360/optimization_paul_graham_managed already exists,
loading from the storage
```

The output.

And now we can launch the Deep Memory training.

```python
from langchain.embeddings.openai import OpenAIEmbeddings

embeddings = OpenAIEmbeddings()


job_id = vector_store.vectorstore.deep_memory.train(

    queries=train_queries,

    relevance=train_relevance,

    embedding_function=embeddings.embed_documents,)
```

```
Your Deep Lake dataset has been successfully created!

creating embeddings: 100%|██████████| 1/1 [00:02<00:00,  2.27s/it]

100%|██████████| 100/100 [00:00<00:00, 158.16it/s]

Dataset(path='hub://genai360/optimization_paul_graham_managed_queries',
tensors=['text', 'metadata', 'embedding', 'id'])


  tensor      htype       shape      dtype  compression

  -------    -------     -------    -------  -------

   text       text      (100, 1)      str     None

 metadata     json      (100, 1)      str     None

 embedding  embedding  (100, 1536)  float32   None
```

```
    id         text      (100, 1)      str     None
DeepMemory training job started. Job ID: 652dceeed7d1579bf6abf3df
```

The output.

With the job_id, you can keep track of deep memory training.

```
vector_store.vectorstore.deep_memory.status('652dceeed7d1579bf6abf3df')
```

To evaluate our Deep Memory-enabled vector store, we can generate a test dataset. Here, we only send 20 chunks to make things fast, but a bigger dataset size would be recommended for a stronger evaluation.

```
from llama_index.evaluation import generate_question_context_pairs
# Generate test dataset
test_dataset = generate_question_context_pairs(
    nodes[:20],
    llm=llm,
    num_questions_per_chunk=1 )
test_dataset.save_json("test_dataset.json")

# We can also load the dataset from a json file if already done previously.
from llama_index.finetuning.embeddings.common import (EmbeddingQAFinetuneDataset)
test_dataset = EmbeddingQAFinetuneDataset.from_json("test_dataset.json")
test_queries, test_relevance = create_query_relevance(test_dataset)
```

```
100%|██████████| 20/20 [00:29<00:00,  1.49s/it]
```

The output.

Let’s evaluate the recall on the generated test dataset using the Deep Lakes evaluation Python function.

Recall measures the proportion of relevant items successfully retrieved by the system from all relevant items available in the dataset.

Formula: Recall is calculated as:

Recall = (Number of Relevant Items RetrievedTotal Number of Relevant Items in the Dataset)/
(Total Number of Relevant Items in the DatasetNumber of Relevant Items Retrieved)

It focuses on the system's ability to find all relevant items. A high recall means the system is good at not missing relevant items.

Compared to Hit Rate: Recall is about the system's thoroughness in retrieving all relevant items, and Hit Rate is about its effectiveness in ensuring that each query retrieves something relevant.

```
# Evaluate recall on the generated test dataset
recalls = vector_store.vectorstore.deep_memory.evaluate(
    queries=test_queries,
    relevance=test_relevance,
    embedding_function=embeddings.embed_documents )
```

```
Embedding queries took 1.24 seconds
---- Evaluating without Deep Memory ----
Recall@1:          55.2%
Recall@3:          87.1%
Recall@5:          90.5%
Recall@10:         97.4%
Recall@50:         100.0%
Recall@100:        100.0%
---- Evaluating with Deep Memory ----
Recall@1:          56.0%
Recall@3:          87.1%
Recall@5:          92.2%
Recall@10:         99.1%
Recall@50:         100.0%
Recall@100:        100.0%
```

The output.

Now, let’s get the Hit Rate and MRR scores of our Deep Memory enabled vector store.
We start measuring the Hit Rate and MRR of our base vector store:

```
import os
from llama_index.postprocessor.cohere_rerank import CohereRerank
from llama_index.evaluation import RetrieverEvaluator
```

```
base_retriever = vector_index.as_retriever(similarity_top_k=10)
deep_memory_retriever = vector_index.as_retriever(
similarity_top_k=10, vector_store_kwargs={"deep_memory": True})

base_retriever_evaluator = RetrieverEvaluator.from_metric_names(
    ["mrr", "hit_rate"], retriever=base_retriever )
eval_results = await base_retriever_evaluator.aevaluate_dataset(test_dataset)
print(display_results_retriever("Retriever Results", eval_results))
```

```
Retriever Name  Hit Rate       MRR
0  Retriever Results  0.974138  0.717809
```

The output.

Now, the same evaluation for the Deep Memory Vector Store

```
deep_memory_retriever = vector_index.as_retriever(
similarity_top_k=10, vector_store_kwargs={"deep_memory": True})

dm_retriever_evaluator = RetrieverEvaluator.from_metric_names(
    ["mrr", "hit_rate"], retriever=deep_memory_retriever)
dm_eval_results = await dm_retriever_evaluator.aevaluate_dataset(test_dataset)
print(display_results_retriever("Retriever Results", dm_eval_results))
```

```
Retriever Name  Hit Rate       MRR
0  Retriever Results  0.991379  0.72865
```

The output.

We can see a small increase in the MRR score compared to the baseline RAG pipeline while our Hit Rate stays the same. Note that this is again mainly due to our evaluation test set and the fact that we only chose 20 chunks. You can experiment with more chunks or manually build a different test set for improved results, especially in your application!

Conclusion

In this lesson, optimizing a LlamaIndex RAG pipeline involved a structured approach to improve information retrieval and generation quality.
We adjusted retrieval top_k values, evaluated two embedding models, introduced reranking mechanisms, and integrated Active Loop’s Deep Memory, some leading to performance enhancements. The improvements were somewhat negligible in this short demo. Still, it is crucial

to try these more advanced improvements as they could have high impacts on a real application and improved evaluation dataset. We also highlight the importance of a good evaluation set of tools, such as a well-curated and large enough evaluation dataset.

RESOURCES

- Colab notebook for the lesson:
  colab.research.google.com

- LlamaIndex and Deep Memory integration:

  How to get +15% RAG hit_rate improvement for question answering on documentation? - LlamaIndex 🦙 0.9.15.post2

This lesson is based on the Llamaindex AI-engineer-workshop posted by Disiok.

GitHub - run-llama/ai-engineer-workshop
Contribute to run-llama/ai-engineer-workshop development by creating an account on GitHub.

Activeloop Deep Memory

How do we get +15% RAG hit_rate improvement for question answering on documentation?

Retrieval-Augmented Generators (RAGs) have recently gained significant attention. As advanced RAG techniques and agents emerge, they expand the potential of what RAGs can accomplish. However, several challenges may limit the integration of RAGs into production. The primary factors to consider when implementing RAGs in production settings are accuracy (recall), cost, and latency. For basic use cases, OpenAI's Ada model paired with a naive similarity search can produce satisfactory results. Yet, for higher accuracy or recall during searches, one might need to employ advanced retrieval techniques. These methods might involve varying data chunk sizes, rewriting queries multiple times, and more, potentially increasing latency and costs. Activeloop's Deep Memory a feature available to Activeloop Deep Lake users, addresses these issuea by introducing a tiny neural network layer trained to match user queries with relevant data from a corpus. While this addition incurs minimal latency during search, it can boost retrieval accuracy by up to 27 % and remains cost-effective and simple to use, without requiring any additional advanced rag techniques.

```
%pip install llama-index-vector-stores-deeplake
%pip install llama-index-llms-openai
import nest_asyncio
import os
import getpass

nest_asyncio.apply()
!pip install deeplake beautifulsoup4 html2text tiktoken openai llama-index python-dotenv
```

For this tutorial we will parse deeplake documentation, and create a RAG system that could answer the question from the docs.

The tutorial can be divided into several parts:

1. Dataset creation and uploading
2. Generating synthetic queries and training deep_memory
3. Evaluating deep memory performance
4. Deep Memory inference

1. Dataset Creation and ingestion

Let me parse all of the links using BeautifulSoup and convert them into LlamaIndex documents:

```
import requests
from bs4 import BeautifulSoup
from urllib.parse import urljoin


def get_all_links(url):
    response = requests.get(url)
    if response.status_code != 200:
        print(f"Failed to retrieve the page: {url}")
        return []

    soup = BeautifulSoup(response.content, "html.parser")

    # Finding all 'a' tags which typically contain href attribute for links
    links = [
        urljoin(url, a["href"])
        for a in soup.find_all("a", href=True)
        if a["href"]
    ]

    return links
from langchain.document_loaders import AsyncHtmlLoader
from langchain.document_transformers import Html2TextTransformer
from llama_index.core import Document


def load_documents(url):
    all_links = get_all_links(url)
    loader = AsyncHtmlLoader(all_links)
    docs = loader.load()

    html2text = Html2TextTransformer()
    docs_transformed = html2text.transform_documents(docs)
    docs = [Document.from_langchain_format(doc) for doc in docs_transformed]
    return docs


docs = load_documents("https://docs.deeplake.ai/en/latest/")
```

```
Fetching pages: 100%|##########| 120/120 [00:13<00:00,  8.70it/s]
```

```
len(docs)
```

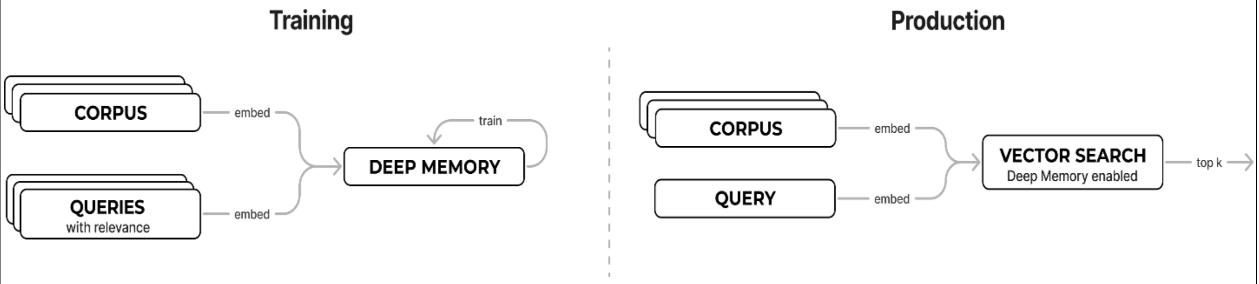
```
120
```

```
from llama_index.core.evaluation import generate_question_context_pairs
from llama_index.core import (
    VectorStoreIndex,
    SimpleDirectoryReader,
    StorageContext,)
from llama_index.vector_stores.deeplake import DeepLakeVectorStore
from llama_index.core.node_parser import SimpleNodeParser
from llama_index.llms.openai import OpenAI


os.environ["OPENAI_API_KEY"] = getpass.getpass("Enter your OpenAI API token: ")
# # activeloop token is needed if you are not signed in using CLI: `activeloop login -u <USERNAME> -p <PASSWORD>`
os.environ["ACTIVELOOP_TOKEN"] = getpass.getpass(    "Enter your ActiveLoop API token: ")
 # Get your API token from https://app.activeloop.ai, click on your profile picture in the top right corner, and select "API Tokens"
token = os.getenv("ACTIVELOOP_TOKEN")

vector_store = DeepLakeVectorStore(
    dataset_path="hub://activeloop-test/deeplake_docs_deepmemory2",
    overwrite=False,  # set to True to overwrite the existing dataset
    runtime={"tensor_db": True},
    token=token,)
```

```
Deep Lake Dataset in hub://activeloop-test/deeplake_docs_deepmemory2 already exists, loading from the storage
```

```
def create_modules(vector_store, docs=[], populate_vector_store=True):
    if populate_vector_store:
        node_parser = SimpleNodeParser.from_defaults(chunk_size=512)
        nodes = node_parser.get_nodes_from_documents(docs)
    else:
        nodes = []

    # by default, the node ids are set to random uuids. To ensure same id's per run, we manually set them.
    for idx, node in enumerate(nodes):
        node.id_ = f"node_{idx}"

    llm = OpenAI(model="gpt-4")
    storage_context = StorageContext.from_defaults(vector_store=vector_store)
    return storage_context, nodes, llm
(
    storage_context,
    nodes,
    llm,
) = create_modules(
    docs=docs,
    vector_store=vector_store,
    # populate_vector_store=False, # uncomment this line to skip populating the vector store
)
vector_index = VectorStoreIndex(nodes, storage_context=storage_context)
deep_memory_retriever = vector_index.as_retriever(
    similarity_top_k=4, deep_memory=True
)
```

2. Training Deep Memory



Here above, we showed the overall schema of how deep_memory works. So as you can see, in order to train it, you need relevance, queries together with corpus data (data that we want to query). The corpus data was already populated in the previous section; here, we will be generating questions and relevance.

1. `questions` - is a text of strings, where each string represents a query.

2. relevance - contains links to the ground truth for each question. There might be several docs that contain an answer to the given question. Because of this, relevance is List[List[tuple[str, float]]], where the outer list represents queries and the inner list relevant documents. The tuple contains a str, float pair where the string represents the id of the source doc (corresponds to the id tensor in the dataset), while the float corresponds to how much the current document is related to the question.

```
from llama_index.core.evaluation import (
    generate_question_context_pairs,
    EmbeddingQAFinetuneDataset )
import random


def create_train_test_datasets( number_of_samples=600, llm=None, nodes=None, save=False ):

    random_indices = random.sample(range(len(nodes)), number_of_samples)
    ratio = int(len(random_indices) * 0.8)

    train_indices = random_indices[:ratio]
    test_indices = random_indices[ratio:]

    train_nodes = [nodes[i] for i in train_indices]
    test_nodes = [nodes[i] for i in test_indices]

    train_qa_dataset = generate_question_context_pairs(
        train_nodes, llm=llm, num_questions_per_chunk=1)

    test_qa_dataset = generate_question_context_pairs(
        test_nodes, llm=llm, num_questions_per_chunk=1)

    # [optional] save
    if save:
        train_qa_dataset.save_json(
            f"deeplake_docs_{number_of_samples}_train.json")
        test_qa_dataset.save_json(
            f"deeplake_docs_{number_of_samples}_test.json")
    return train_qa_dataset, test_qa_dataset

train_qa_dataset, test_qa_dataset = create_train_test_datasets(
    number_of_samples=600, llm=llm, nodes=nodes, save=True)
```

```
  4%|▌         | 19/480 [02:25<1:04:00,  8.33s/it]
```

```
train_qa_dataset = EmbeddingQAFinetuneDataset.from_json(
    "deeplake_docs_600_train.json")
test_qa_dataset = EmbeddingQAFinetuneDataset.from_json(
    "deeplake_docs_600_test.json")

def create_query_relevance(qa_dataset):
    """Function for converting llama-index dataset to correct format for deep memory training"""
    queries = [text for _, text in qa_dataset.queries.items()]
    relevant_docs = qa_dataset.relevant_docs
    relevance = []
    for doc in relevant_docs:
        relevance.append([(relevant_docs[doc][0], 1)])
    return queries, relevance
train_queries, train_relevance = create_query_relevance(train_qa_dataset)
test_queries, test_relevance = create_query_relevance(test_qa_dataset)
train_queries[:3]
```

```
['In the context of creating a bounding box tensor in a dataset, explain the significance of the "coords" argument and
its keys "type" and "mode". What does the "type" key specify about the bounding box coordinates?',
 'Explain the process of creating an intrinsics tensor and appending intrinsics matrices in the context of computer visi
on. What are the dimensions of the intrinsics parameters and what do they represent? Also, describe the concept of a
Segmentation Mask Htype and its role in image processing.',
 'In the context of querying for images in the MNIST Train Dataset using `ds.query`, what does the command "select
* where labels == 0" signify and what is the expected output?']
```

```
train_relevance[:3]
```

```
[[('node_788', 1)], [('node_861', 1)], [('node_82', 1)]]
```

```
test_queries[:3]
```

```
['What are the steps to update the information of keypoints and connections in a tensor, and what types of data can b
e appended to keypoints?',
 'What is the command to create a mesh tensor in DeepLake and what are the supported compressions? Also, explain
how to append a ply file containing mesh data to this tensor.',
 'What is a Sequence htype in the context of tensors and how does it function as a wrapper for other htypes? Provide
examples.']
```

```
test_relevance[:3]
```

```
[[('node_933', 1)], [('node_671', 1)], [('node_471', 1)]]
```

```
from langchain.embeddings.openai import OpenAIEmbeddings

embeddings = OpenAIEmbeddings()

job_id = vector_store.vectorstore.deep_memory.train(
    queries=train_queries,
    relevance=train_relevance,
    embedding_function=embeddings.embed_documents,)
```

```
Starting DeepMemory training job
Your Deep Lake dataset has been successfully created!
Preparing training data for deepmemory:
Creating 483 embeddings in 1 batches of size 483:: 100%|██████████| 1/1 [00:03<00:00,  3.67s/it]
DeepMemory training job started. Job ID: 65421a5003888c9ca36c72e8
```

```
vector_store.vectorstore.deep_memory.status(job_id)
```

```
This dataset can be visualized in Jupyter Notebook by ds.visualize() or at https://app.activeloop.ai/adilkhan/deeplake
_docs_deepmemory2
--------------------------------------------------------------
|                  65421a5003888c9ca36c72e8                  |
--------------------------------------------------------------
| status                     | completed                     |
--------------------------------------------------------------
| progress                   | eta: 12.2 seconds             |
|                            | recall@10: 67.01% (+18.56%)   |
--------------------------------------------------------------
| results                    | recall@10: 67.01% (+18.56%)   |
--------------------------------------------------------------
```

3. DeepMemory Evaluation

Fantastic! The training has led to some remarkable improvements! Now, let's assess its performance on a test set.

```
recalls = vector_store.vectorstore.deep_memory.evaluate(
    queries=test_queries,
    relevance=test_relevance,
    embedding_function=embeddings.embed_documents,)
```

```
info Wed Nov  1 09:32:44 2023 GMT        Added distance metric `deepmemory_distance`.
Embedding queries took 0.95 seconds
---- Evaluating without Deep Memory ----
Recall@1:   12.5%
Recall@3:   23.3%
Recall@5:   30.8%
Recall@10:              50.8%
Recall@50:              94.2%
Recall@100:             95.8%
---- Evaluating with Deep Memory ----
Recall@1:   11.7%
Recall@3:   27.5%
Recall@5:   40.8%
Recall@10:              65.0%
Recall@50:              96.7%
Recall@100:             98.3%
```

Impressive! We've observed a 15% increase in recall on the test set. Next, let's employ the RetrieverEvaluator to examine the MRR (Mean Reciprocal Rank) and hit rates.

```
import pandas as pd


def display_results(eval_results):
    """Display results from evaluate."""
    hit_rates = []
    mrrs = []
    names = []
    for name, eval_result in eval_results.items():
        metric_dicts = []
        for er in eval_result:
            metric_dict = er.metric_vals_dict
            metric_dicts.append(metric_dict)

        full_df = pd.DataFrame(metric_dicts)

        hit_rate = full_df["hit_rate"].mean()
        mrr = full_df["mrr"].mean()

        hit_rates.append(hit_rate)
        mrrs.append(mrr)
        names.append(name)

    metric_df = pd.DataFrame(
        [
            {"retrievers": names[i], "hit_rate": hit_rates[i], "mrr": mrrs[i]}
            for i in range(2)
        ],
    )

    return metric_df
```

Evaluating performance of retrieval with deep memory:

```
from llama_index.core.evaluation import RetrieverEvaluator

deep_memory_retriever = vector_index.as_retriever(
    similarity_top_k=10, vector_store_kwargs={"deep_memory": True}
)
dm_retriever_evaluator = RetrieverEvaluator.from_metric_names(
    ["mrr", "hit_rate"], retriever=deep_memory_retriever
)

dm_eval_results = await dm_retriever_evaluator.aevaluate_dataset(
    test_qa_dataset, retriever=dm_retriever_evaluator
)
from llama_index.core.evaluation import RetrieverEvaluator

naive_retriever = vector_index.as_retriever(similarity_top_k=10)
naive_retriever_evaluator = RetrieverEvaluator.from_metric_names(
    ["mrr", "hit_rate"], retriever=naive_retriever
)

naive_eval_results = await naive_retriever_evaluator.aevaluate_dataset(
    test_qa_dataset, retriever=naive_retriever
)
eval_results = {
    f"{mode} with Deep Memory top-10 eval": eval_result
    for mode, eval_result in zip(
        ["with", "without"], [dm_eval_results, naive_eval_results]
    )
}

display_results(eval_results)
```

	retrievers	hit_rate	mrr
0	with with Deep Memory top-10 eval	0.650000	0.244775
1	without with Deep Memory top-10 eval	0.508333	0.215129

Not only hit_rate has increased but also MRR

4. Deep Memory Inference

```
query_engine = vector_index.as_query_engine(
    vector_store_kwargs={"deep_memory": True}, llm=llm
)
response = query_engine.query(
    "How can you connect your own storage to the deeplake?"
)
print(response)
```

```
info Wed Nov  1 11:37:33 2023 GMT        Can't find any metric in the dataset.
You can connect your own storage to deeplake by using the `connect()` function in the deeplake API.
```

```
query_engine = vector_index.as_query_engine(
    vector_store_kwargs={"deep_memory": False}, llm=llm
)
response = query_engine.query(
    "How can you connect your own storage to the deeplake?"
)
print(response)
```

```
The context does not provide information on how to connect your own storage to Deep Lake.
```

From our observations, without "deep memory", our model tends to produce inaccuracies because it retrieves the wrong context.

Reference:

https://docs.llamaindex.ai/en/stable/examples/retrievers/deep_memory/

Use Deep Memory with LangChain to Get Up to +27% Increase in Accurate Questions Answers to LangChain Code DB

Activeloop Deep Memory

Activeloop Deep Memory is a suite of tools that enables you to optimize your Vector Store for your use-case and achieve higher accuracy in your LLM apps.

`Retrieval-Augmented Generatation` (`RAG`) has recently gained significant attention. As advanced RAG techniques and agents emerge, they expand the potential of what RAGs can accomplish. However, several challenges may limit the integration of RAGs into production. The primary factors to consider when implementing RAGs in production settings are accuracy (recall), cost, and latency. For basic use cases, OpenAI's Ada model paired with a naive similarity search can produce satisfactory results. Yet, for higher accuracy or recall during searches, one might need to employ advanced retrieval techniques. These methods might involve varying data chunk sizes, rewriting queries multiple times, and more, potentially increasing latency and costs. Activeloop's Deep Memory a feature available to `Activeloop Deep Lake` users, addresses these issuea by introducing a tiny neural network layer trained to match user queries with relevant data from a corpus. While this addition incurs minimal latency during search, it can boost retrieval accuracy by up to 27 % and remains cost-effective and simple to use, without requiring any additional advanced rag techniques.

For this tutorial we will parse `DeepLake` documentation, and create a RAG system that could answer the question from the docs.

1. Dataset Creation

We will parse activeloop's docs for this tutorial using `BeautifulSoup` library and LangChain's document parsers like `Html2TextTransformer`, `AsyncHtmlLoader`. So we will need to install the following libraries:

```
%pip install --upgrade --quiet  tiktoken langchain-openai python-dotenv
datasets langchain deeplake beautifulsoup4 html2text ragas
```

Also you'll need to create a Activeloop account.

```
ORG_ID = "..."

from langchain.chains import RetrievalQA
from langchain_community.vectorstores import DeepLake
from langchain_openai import ChatOpenAI, OpenAIEmbeddings

import getpass
import os

os.environ["OPENAI_API_KEY"] = getpass.getpass("Enter OpenAI API token: ")
# # activeloop token is needed if you are not signed in using CLI:
`activeloop login -u <USERNAME> -p <PASSWORD>`
os.environ["ACTIVELOOP_TOKEN"] = getpass.getpass("Enter ActiveLoopAPI token")
```

```
# Get your API token from https://app.activeloop.ai, click on your profile
picture in the top right corner, and select "API Tokens"

token = os.getenv("ACTIVELOOP_TOKEN")
openai_embeddings = OpenAIEmbeddings()

db = DeepLake(
    dataset_path=f"hub://{ORG_ID}/deeplake-docs-deepmemory",  # org_id stands
for your username or organization from activeloop
    embedding=openai_embeddings,
    runtime={"tensor_db": True},
    token=token,
    # overwrite=True, # user overwrite flag if you want to overwrite the full
dataset
    read_only=False)
```

parsing all links in the webpage using `BeautifulSoup`

```
from urllib.parse import urljoin
import requests
from bs4 import BeautifulSoup

def get_all_links(url):
    response = requests.get(url)
    if response.status_code != 200:
        print(f"Failed to retrieve the page: {url}")
        return []

    soup = BeautifulSoup(response.content, "html.parser")

    # Finding all 'a' tags which typically contain href attribute for links
    links = [
        urljoin(url, a["href"]) for a in soup.find_all("a", href=True) if
a["href"] ]

    return links

base_url = "https://docs.deeplake.ai/en/latest/"
all_links = get_all_links(base_url)
```

Loading data:

```
from langchain_community.document_loaders.async_html import AsyncHtmlLoader

loader = AsyncHtmlLoader(all_links)
docs = loader.load()
```

Converting data into user readable format:

```
from langchain_community.document_transformers import Html2TextTransformer

html2text = Html2TextTransformer()
```

```
docs_transformed = html2text.transform_documents(docs)
```

Now, let us chunk further the documents as some of the contain too much text:

```
from langchain_text_splitters import RecursiveCharacterTextSplitter

chunk_size = 4096
docs_new = []

text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=chunk_size,
)

for doc in docs_transformed:
    if len(doc.page_content) < chunk_size:
        docs_new.append(doc)
    else:
        docs = text_splitter.create_documents([doc.page_content])
        docs_new.extend(docs)
```

Populating VectorStore:

```
docs = db.add_documents(docs_new)
```

2. Generating synthetic queries and training Deep Memory

Next step would be to train a deep_memory model that will align your users queries with the dataset that you already have. If you don't have any user queries yet, no worries, we will generate them using LLM!

TODO: Add image

Here above we showed the overall schema how deep_memory works. So as you can see, in order to train it you need relevance, queries together with corpus data (data that we want to query). Corpus data was already populated in the previous section, here we will be generating questions and relevance.

1. `questions` - is a text of strings, where each string represents a query
2. `relevance` - contains links to the ground truth for each question. There might be several docs that contain answer to the given question. Because of this relevenve is `List[List[tuple[str, float]]]`, where outer list represents queries and inner list relevant documents. Tuple contains str, float pair where string represent the id of the source doc (corresponds to the `id` tensor in the dataset), while float corresponds to how much current document is related to the question.

Now, let us generate synthetic questions and relevance:

```
from typing import List
from langchain.chains.openai_functions import create_structured_output_chain
from langchain_core.messages import HumanMessage, SystemMessage
from langchain_core.prompts import ChatPromptTemplate,
HumanMessagePromptTemplate
from langchain_openai import ChatOpenAI
from pydantic import BaseModel, Field

# fetch dataset docs and ids if they exist (optional you can also ingest)
docs = db.vectorstore.dataset.text.data(fetch_chunks=True,
aslist=True)["value"]
ids = db.vectorstore.dataset.id.data(fetch_chunks=True, aslist=True)["value"]

# If we pass in a model explicitly, we need to make sure it supports the
OpenAI function-calling API.
llm = ChatOpenAI(model="gpt-3.5-turbo", temperature=0)


class Questions(BaseModel):
    """Identifying information about a person."""
    question: str = Field(..., description="Questions about text")


prompt_msgs = [
    SystemMessage(
        content="You are a world class expert for generating questions based
on provided context. \
            You make sure the question can be answered by the text."),
    HumanMessagePromptTemplate.from_template(
        "Use the given text to generate a question from the following input:
{input}"
    ),
    HumanMessage(content="Tips: Make sure to answer in the correct format"),
]
prompt = ChatPromptTemplate(messages=prompt_msgs)
chain = create_structured_output_chain(Questions, llm, prompt, verbose=True)

text = "# Understanding Hallucinations and Bias ## **Introduction** In this
lesson, we'll cover the concept of **hallucinations** in LLMs, highlighting
their influence on AI applications and demonstrating how to mitigate them
using techniques like the retriever's architectures. We'll also explore
**bias** within LLMs with examples."
questions = chain.run(input=text)
print(questions)

import random

from langchain_openai import OpenAIEmbeddings
from tqdm import tqdm


def generate_queries(docs: List[str], ids: List[str], n: int = 100):
    questions = []
    relevances = []
    pbar = tqdm(total=n)
    while len(questions) < n:
        # 1. randomly draw a piece of text and relevance id
        r = random.randint(0, len(docs) - 1)
        text, label = docs[r], ids[r]
        # 2. generate queries and assign and relevance id
        generated_qs = [chain.run(input=text).question]
        questions.extend(generated_qs)
        relevances.extend([[(label, 1)] for _ in generated_qs])
        pbar.update(len(generated_qs))
        if len(questions) % 10 == 0:
            print(f"q: {len(questions)}")
    return questions[:n], relevances[:n]


chain = create_structured_output_chain(Questions, llm, prompt, verbose=False)
questions, relevances = generate_queries(docs, ids, n=200)

train_questions, train_relevances = questions[:100], relevances[:100]
test_questions, test_relevances = questions[100:], relevances[100:]
```

Now we created 100 training queries as well as 100 queries for testing. Now let us train the deep_memory:

```
job_id = db.vectorstore.deep_memory.train(
    queries=train_questions,
    relevance=train_relevances,)
```

Let us track the training progress:

```
db.vectorstore.deep_memory.status("6538939ca0b69a9ca45c528c")
```

```
--------------------------------------------------------------
|                  6538e02ecda4691033a51c5b                  |
--------------------------------------------------------------
| status                     | completed                     |
--------------------------------------------------------------
| progress                   | eta: 1.4 seconds              |
|                            | recall@10: 79.00% (+34.00%)   |
--------------------------------------------------------------
| results                    | recall@10: 79.00% (+34.00%)   |
--------------------------------------------------------------
```

3. Evaluating Deep Memory performance

Great we've trained the model! It's showing some substantial improvement in recall, but how can we use it now and evaluate on unseen new data? In this section we will delve into model evaluation and inference part and see how it can be used with LangChain in order to increase retrieval accuracy

3.1 Deep Memory evaluation

For the beginning we can use deep_memory's builtin evaluation method. It calculates several `recall` metrics. It can be done easily in a few lines of code.

```
recall = db.vectorstore.deep_memory.evaluate(
    queries=test_questions,
    relevance=test_relevances,)
```

```
Embedding queries took 0.81 seconds
---- Evaluating without model ----
Recall@1:     9.0%
Recall@3:     19.0%
Recall@5:     24.0%
Recall@10:    42.0%
Recall@50:    93.0%
Recall@100:   98.0%
---- Evaluating with model ----
Recall@1:     19.0%
Recall@3:     42.0%
Recall@5:     49.0%
Recall@10:    69.0%
Recall@50:    97.0%
Recall@100:   97.0%
```

It is showing quite substatntial improvement on an unseen test dataset too!!!

3.2 Deep Memory + RAGas

```
from ragas.langchain import RagasEvaluatorChain
from ragas.metrics import (
    context_recall,
)
```

Let us convert recall into ground truths:

```
def convert_relevance_to_ground_truth(docs, relevance):
    ground_truths = []

    for rel in relevance:
        ground_truth = []
        for doc_id, _ in rel:
            ground_truth.append(docs[doc_id])
        ground_truths.append(ground_truth)
    return ground_truths

ground_truths = convert_relevance_to_ground_truth(docs, test_relevances)

for deep_memory in [False, True]:
    print("\nEvaluating with deep_memory =", deep_memory)
    print("===================================")
    retriever = db.as_retriever()
    retriever.search_kwargs["deep_memory"] = deep_memory

    qa_chain = RetrievalQA.from_chain_type(
        llm=ChatOpenAI(model="gpt-3.5-turbo"),
        chain_type="stuff",
        retriever=retriever,
        return_source_documents=True,
    )

    metrics = {
        "context_recall_score": 0,
    }

    eval_chains = {m.name: RagasEvaluatorChain(metric=m) for m in
[context_recall]}

    for question, ground_truth in zip(test_questions, ground_truths):
        result = qa_chain({"query": question})
        result["ground_truths"] = ground_truth
        for name, eval_chain in eval_chains.items():
            score_name = f"{name}_score"
            metrics[score_name] += eval_chain(result)[score_name]

    for metric in metrics:
        metrics[metric] /= len(test_questions)
        print(f"{metric}: {metrics[metric]}")
    print("===================================")
```

```
Evaluating with deep_memory = False
===================================
context_recall_score = 0.3763423145
===================================

Evaluating with deep_memory = True
===================================
context_recall_score = 0.5634545323
===================================
```

3.3 Deep Memory Inference

TODO: Add image

with deep_memory

```
retriever = db.as_retriever()
retriever.search_kwargs["deep_memory"] = True
retriever.search_kwargs["k"] = 10

query = "Deamination of cytidine to uridine on the minus strand of viral DNA
results in catastrophic G-to-A mutations in the viral genome."
qa = RetrievalQA.from_chain_type(
    llm=ChatOpenAI(model="gpt-4"), chain_type="stuff", retriever=retriever
)
```

```
print(qa.run(query))
```

```
The base htype of the 'video_seq' tensor is 'video'.
```

without deep_memory

```
retriever = db.as_retriever()
retriever.search_kwargs["deep_memory"] = False
retriever.search_kwargs["k"] = 10

query = "Deamination of cytidine to uridine on the minus strand of viral DNA
results in catastrophic G-to-A mutations in the viral genome."
qa = RetrievalQA.from_chain_type(
    llm=ChatOpenAI(model="gpt-4"), chain_type="stuff", retriever=retriever
)
qa.run(query)
```

```
The text does not provide information on the base htype of the 'video_seq'
tensor.
```

3.4 Deep Memory cost savings

Deep Memory increases retrieval accuracy without altering your existing workflow. Additionally, by reducing the top_k input into the LLM, you can significantly cut inference costs via lower token usage.

Reference:

https://python.langchain.com/docs/integrations/retrievers/activeloop/

Deep Memory for RAG Applications Across Legal, Financial, and Biomedical Industries

Build a RAG System with LLM and Deep Memory. Use three different datasets and test the quality of response with and without the Deep Memory feature to improve retrieval!

Traditionally, retrieval methods have relied on standard techniques like RAG or query-based document retrieval. However, often the results are not fully satisfying as the retrieved chunks of text are not exactly what we expect. To solve this problem, many techniques tested every day.

In this guide, we will explore how to enhance the storage of indices in the retrieval pipeline in a more effective way through finetuning techniques.

Unlike other techniques, **Deep Memory** enables an automatic and convenient **finetuning** of the **retrieval step** on the chunks of data provided, thus improving the solution when compared to a classic and agnostic RAG application.

Deep Memory emerges as a fundamental solution in addressing the critical need for accurate retrieval in generating high-quality results. It is crucial to increase the accuracy of Deep Lake's vector search by up to 22%, achieved through learning an index from labeled queries tailored to specific applications. Importantly, this improvement is achieved without compromising research time, demonstrating the efficacy of Deep Memory in fine-tuning the retrieval process.

The enterprise application landscape, particularly the development of "chat with your data" solutions, highlights the importance of accurate retrieval. Current practices involve the integration of Retrieval Augmented Generation (RAG) systems with Large Language Models (LLMs) like GPT-4.

The role of Deep Memory in significantly improving the accuracy of vector search becomes fundamental in this context, as it offers a potential solution to increase the reliability of these applications. By emphasizing the importance of accurate retrieval, the integration of technologies like Deep Memory becomes a focal point in achieving the desired level of consistency and precision in generating results.

Let's build a Deep Memory RAG application!

We need three main components: A **dataset** containing the text chunks we want to retrieve, an **LLM model** to generate the text embeddings and the **Deep Memory** library.

Let's delve into the details.

Get Deep Memory Access

As Step 0, please note that Deep Memory is a premium feature in Activeloop paid plans. As a reminder, you can redeem a free trial. As a part of the course, all course takers can redeem a free extended trial of one month for the Activeloop Growth plan by redeeming GENAI360 promo code at checkout. To redeem the plan, please create a Deep Lake Account, and on the following screen on account creation, please watch the following video.

Preparing the Dataset

In this guide, we have prepared three different datasets that could be downloaded and tested with and without the Deep Memory feature.

To find out more about them, you can follow the links:

1. **Finance**: we chose the FinQA Dataset that contains text explaining the economy, acquisitions, etc. It also is a QA dataset, making it easier for us to embed as we already have questions and related answers without generating them! This work focuses on answering deep questions about financial data, aiming to automate the analysis of a large corpus of financial documents. In

contrast to existing tasks in the general domain, the finance domain includes complex numerical reasoning and an understanding of heterogeneous representations. Source: https://github.com/czyssrs/FinQA

2. **Legal**: the Legalbench Dataset contains questions and answers about legal subjects like a company's legal rights, policies, and such. That is a very tedious and specific topic that is not readable by everyone, so retrieving the right information for this task is very welcome! LegalBench tasks span multiple types (binary classification, multi-class classification, extraction, generation, entailment), multiple types of text (statutes, judicial opinions, contracts, etc.), and multiple areas of law (evidence, contracts, civil procedure, etc.). It is a benchmark consisting of different legal reasoning tasks.
   Source: https://huggingface.co/datasets/nguha/legalbench?clone=true

3. **Biomedical**: To address a biomedical topic we chose the Cord19 Dataset, which is about Covid. As it is a very discussed topic, retrieving every possible information is crucial and so we wanted to test it. CORD-19 is a corpus of academic papers about COVID-19 and related coronavirus research. It's curated and maintained by the Semantic Scholar team at the Allen Institute for AI to support text mining and NLP research.
   Source: https://github.com/allenai/cord19

These three datasets are in the Activeloop organization space, so you need to load them in a Tensor Database format to be able to take advantage of the Deep Memory functionality.
In the code above the variable `user_hub` will be equal to the Organization name, in our case `"activeloop"`, and `name_db` will be the `<dataset_name>`.

```
def load_vector_store(user_hub, name_db):
    vector_store_db = DeepLakeVectorStore(
        f"hub://{user_hub}/{name_db}",
        embedding_function=embeddings_function.embed_documents,
        runtime={"tensor_db": True},)
    return vector_store_db
```

The datasets were created with a preprocessing consisting of 3 different steps:

- **Gather** the data
- **Divide** the data into **chunks**
- **Create sample questions**

The last point listed must be applied for every chunk. Specifically, it creates a relevance score that represents how relevant the question is when compared to the chunk of text (this is necessary for the most critical part: the Deep Memory Finetuning).

Gather the data

The simplest solution is to download a QA Dataset on whatever is our topic of interest so that it possesses everything we need. We can easily do it with the following command:

```
wget <source_link>
```

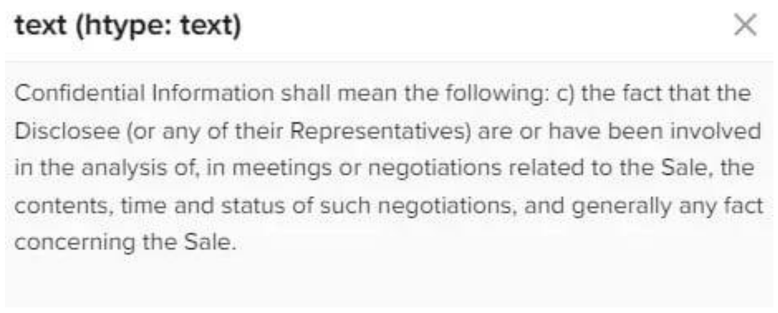
But what if our data is just a long text?

Chunk Generation

To generate the chunks, we can use libraries like **Langchain**, which provide methods to divide our text into chunks automatically.

Below are some examples of the generated chunks:

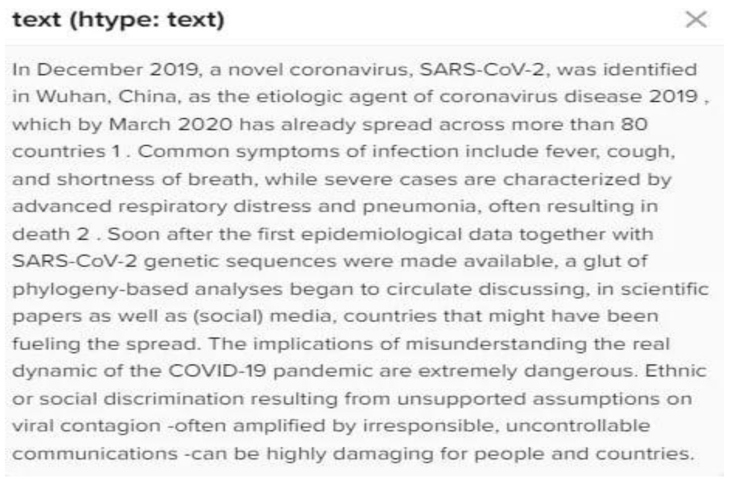
- Legal:



text (htype: text)

Confidential Information shall mean the following: c) the fact that the Disclosee (or any of their Representatives) are or have been involved in the analysis of, in meetings or negotiations related to the Sale, the contents, time and status of such negotiations, and generally any fact concerning the Sale.

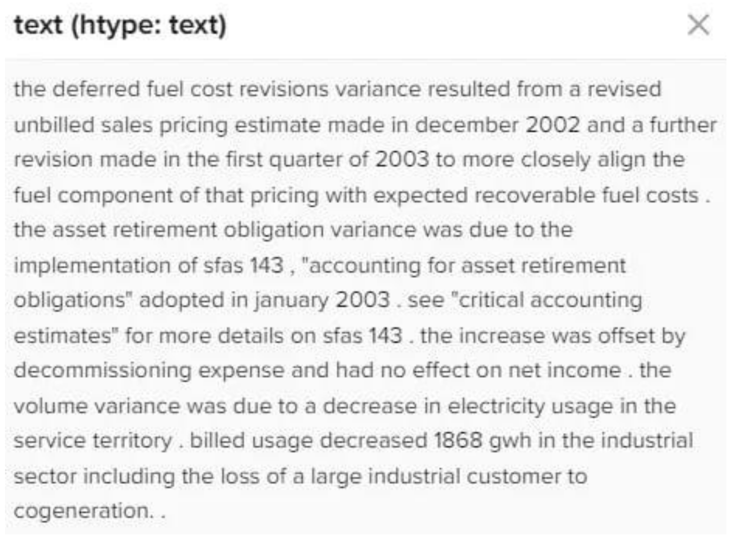
- Biomedical:



text (htype: text)

In December 2019, a novel coronavirus, SARS-CoV-2, was identified in Wuhan, China, as the etiologic agent of coronavirus disease 2019 , which by March 2020 has already spread across more than 80 countries 1 . Common symptoms of infection include fever, cough, and shortness of breath, while severe cases are characterized by advanced respiratory distress and pneumonia, often resulting in death 2 . Soon after the first epidemiological data together with SARS-CoV-2 genetic sequences were made available, a glut of phylogeny-based analyses began to circulate discussing, in scientific papers as well as (social) media, countries that might have been fueling the spread. The implications of misunderstanding the real dynamic of the COVID-19 pandemic are extremely dangerous. Ethnic or social discrimination resulting from unsupported assumptions on viral contagion -often amplified by irresponsible, uncontrollable communications -can be highly damaging for people and countries.

- Finance:



text (htype: text)

the deferred fuel cost revisions variance resulted from a revised unbilled sales pricing estimate made in december 2002 and a further revision made in the first quarter of 2003 to more closely align the fuel component of that pricing with expected recoverable fuel costs . the asset retirement obligation variance was due to the implementation of sfas 143 , "accounting for asset retirement obligations" adopted in january 2003 . see "critical accounting estimates" for more details on sfas 143 . the increase was offset by decommissioning expense and had no effect on net income . the volume variance was due to a decrease in electricity usage in the service territory . billed usage decreased 1868 gwh in the industrial sector including the loss of a large industrial customer to cogeneration. .

There are multiple strategies to do this step most efficiently. For instance, we can use "." as a separator character, or define the length of each chunk as a standard, or a combination of those!

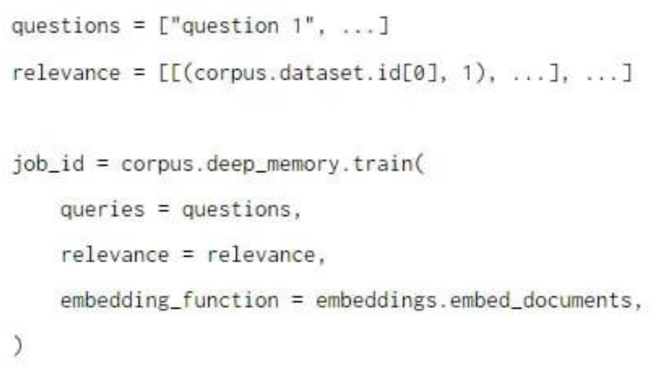
We suggest creating chunks that are not too short and overlapping them to keep relevant information intact.
The main disadvantage of not overlapping chunks is the potential loss of information; depending on the nature of the data and the requirements of the analysis or modeling task, it's often beneficial to experiment with overlapping chunks.

```
from langchain.text_splitter import RecursiveCharacterTextSplitter

# Divide text in chunks

def create_chunks(context, chunk_size=300, chunk_overlap=50):

    # Initialize the text splitter with custom parameters

    custom_text_splitter = RecursiveCharacterTextSplitter(

        # Set custom chunk size

        chunk_size = chunk_size,

        chunk_overlap  = chunk_overlap,

        # Use length of the text as the size measure

        length_function = len )

    chunks = custom_text_splitter.split_text(context)

    return chunks
```

Questions and Relevance Generation

This is the most subtle step: how do we generate information like questions or scores without a properly trained model?

The good (not so old) LLMs come to the rescue to accomplish this task! We can use prompt engineering on a LLM to generate a **question** for each chunk and at the same time generate the **relevance score** as a **classification task**. We just need to call the model and then parse the text output to obtain the necessary data. To do this we construct a dataset of questions and relevance. Relevance is a set of pairs (corpus.id: str, significance: str) that provides information where the answer is inside the corpus. Sometimes an answer can be found in multiple locations or have different significance. Relevance enables Deep Memory training to optimize the embedding space for higher accuracy.



```
questions = ["question 1", ...]

relevance = [[(corpus.dataset.id[0], 1), ...], ...]


job_id = corpus.deep_memory.train(

    queries = questions,

    relevance = relevance,

    embedding_function = embeddings.embed_documents,

)
```

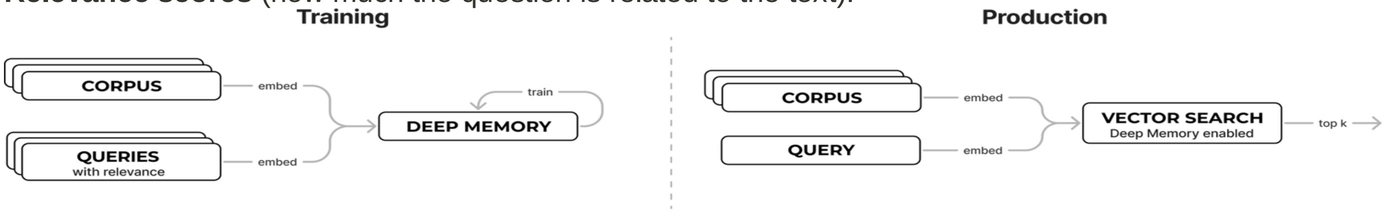
An example of how to generate questions and relevance scores is the following:

```
#Sample Prompt message to generate Question and Answer for the provided context

system_message = """

Generate a question related to the context and provide a relevance score on a scale
of 0 to 1, where 0 is not relevant at all and 1 is highly relevant.

The input is provided in the following format:

Context: [The context that for the generated question]

The output is in the following format:

#Question#: [Text of the question]

#Relevance#: [score number between 0 and 1]

The context is: {context}

"""

def get_chunk_qa_data(context):

    # Generate the Question and Relevance Text with LLM

    llm = OpenAI(temperature=0)

    llm_chain = LLMChain(llm=llm,
prompt=PromptTemplate.from_template(system_message))

    output = llm_chain(context)


    # CHECK THE RELEVANCE STRING IN THE OUTPUT

    check_relevance = None

    relevance_strings = ["#Relevance#: ", "Relevance#: ", "Relevance: ", "Relevance"]

    for rel_str in relevance_strings:

        if rel_str in output["text"]:

            check_relevance = rel_str

            break


    if check_relevance is None:

        raise ValueError("Relevance not found in the output")


    messages = output["text"].split(check_relevance)

    relevance = messages[1]


    # CHECK THE QUESTION STRING IN THE OUTPUT

    question = None
    question_strings = ["#Question#: ", "Question#: ", "Question: ", "Question"]
    for qst_str in question_strings:
        if qst_str in messages[0]:
            question = messages[0].split(qst_str)[1]
            break

    if question is None:
        raise ValueError("Question not found in the output")

    return question, relevance
```

Now we are all set for the fun part!

Deep Memory Tool

Now we dive into the core of this tutorial: **Deep Memory**.
Deep Memory is one of the tools included into the High Performance Features of **Deep Lake**. It is very effective in improving the retrieval accuracy of an LLM model by optimizing your vector store for your use-case, **enhancing the performance** of you overall LLM app.
This is possible by performing a **fine-tuning of the embeddings** of your embedding model using your own Dataset enriched by QA additional information that consists in **Questions** and **Relevance scores** (how much the question is related to the text).



Creating the Deep Memory Vector Store

Deep Memory leads an index from labeled queries tailored to your Dataset, without impacting search time. These results can be achieved with only a few hundred example pairs of prompt embeddings and the most relevant answers from the vector store.
As we can see, Deep Memory uses the dataset text (corpus) along with the questions (queries) and relevance score we generated to train an enhanced retrieval model that can be used without any other modification and adds no latency while boosting the retrieval quality.
Deep Memory increases retrieval accuracy without altering your existing workflow.

```
def load_vector_store(user_hub, name_db):
    vector_store_db = DeepLakeVectorStore(
        f"hub://{user_hub}/{name_db}",
        embedding_function=embeddings_function.embed_documents,
        runtime={"tensor_db": True},)
    return vector_store_db
```

In order to create a Deep Memory Dataset, we just need 2 things: the chunks of text

1. An **embedding function** to generate the text embeddings
2. The metadata we prepared (**questions and relevance scores**)

For the first point, embeddings can still be computed using a model of your choice such as Open AI ada-002 or other OSS models BGE by BAAI. Furthermore, search results from Deep Memory can be further improved by combining them with lexical search or reranker. For the second point, we explained how to generate questions and relevance scores in the previous paragraphs.
If you want to go deeper into details, there are some questions and relevance computed during the training phase:

Legal Dataset:

- Chunk: "*Confidential Information means all confidential information relating to the Purpose which the Disclosing Party or any of its Affiliates, discloses or makes available, to the Receiving Party or any of its Affiliates, before, on or after the Effective Date. This includes the fact that discussions and negotiations are taking place concerning the Purpose and the status of those discussions and negotiations.*"
- Question: What is the definition of Confidential Information?

Biomedical Dataset:

- Chunk: "*The P2 64 and P3 regions encode the non-structural proteins 2B and 2C and 3A, 3B (1-3) (VPg), 3C pro and 4 structural protein-coding regions is replaced by reporter genes, allow the study of genome 68 replication without the requirement for high containment.*"
- Question: What are the non-structural proteins encoded by the P2 64 and P3 regions?

Finance Dataset:

- Chunk: "*the deferred fuel cost revisions variance resulted from a revised unbilled sales pricing estimate made in december 2002 and a further revision made in the first quarter of 2003 to more closely align the fuel component of that pricing with expected recoverable fuel costs . the asset retirement obligation variance was due to the implementation of sfas 143 , "accounting for asset retirement obligations" adopted in january 2003 . see "critical accounting estimates" for more details on sfas 143 . the increase was offset by decommissioning expense and had no effect on net income . the volume variance was due to a decrease in electricity usage in the service territory . billed usage decreased 1868 gwh in the industrial sector including the loss of a large industrial customer to cogeneration.*"
- Question: What was the impact of the asset retirement obligation variance on net income?

Providing those inputs to the Vector Store, we can upload the Dataset we created to the Active Loop Dataset Repository. After this step, the Deep Memory feature will do 2 things automatically:

1. Generate the embeddings the embedding model we defined
2. Finetune the indices using the Deep Memory feature

And we are all set to try our improved RAG applications!
But how to try it out now?

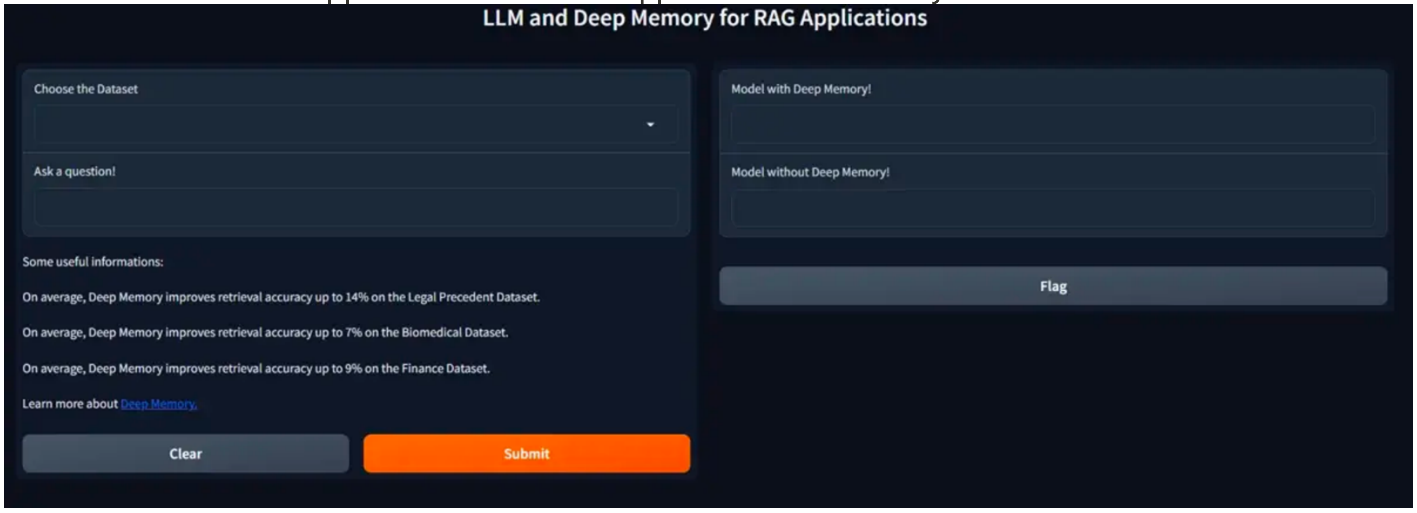
Deep Memory Search

After creating the Deep Memory Dataset, we can search for the right piece of text for our question using the following code:

```
def get_answer(vector_store_db, user_question, deep_memory):

    # deep memory inside the vectore store ==> deep_memory=True

    answer = vector_store_db.search(
        embedding_data=user_question,
        embedding_function=embeddings_function.embed_query,
        deep_memory=deep_memory,
        return_view=False,)
    return answer
```

Developing a Deep Memory Search with Gradio

We created a Gradio application to test our application more easily.



Some useful informations:

On average, Deep Memory improves retrieval accuracy up to 14% on the Legal Precedent Dataset.

On average, Deep Memory improves retrieval accuracy up to 7% on the Biomedical Dataset.

On average, Deep Memory improves retrieval accuracy up to 9% on the Finance Dataset.

Learn more about Deep Memory.

The interface allows us to select the dataset we want to test, write a question, and instantly generate the answer. We can also compare the response returned by the Deep Memory model with that returned by the model without.

Classic RAG vs Deep Memory

To test out the improvements of the Deep Memory step, we prepared and shared 3 datasets we mentioned earlier: **Legal, Medical, Finance**. In the output windows, you can see the benefits of this amazing tool when compared to more classical approaches.
If you want to try these models, we suggest you to try one of these questions:

- Legal Dataset:
  - What are the provisions of this Agreement regarding the disclosure of Confidential Information to third parties?
- Biomedical Dataset:
  - What are the advantages of using the new package to visualize data?
- Finance Dataset:
  - What were the primary factors that contributed to the improvement in net cash provided by operating activities during 2015?

The following example shows how the model with deep memory and dataset finance is more efficient in the response generated:

Deep Memory model:

```
The provisions of this Agreement state that disclosure of Confidential Information to
third party consultants and professional advisors is allowed, as long as those third
parties agree to be bound by this Agreement. Additionally, both parties are required
to keep any confidential information they may have access to confidential, unless
required by law or necessary to perform their obligations under this Agreement. This
includes not only the information itself, but also the terms of the Agreement and the
fact that the parties are considering a business arrangement.
```

Non Deep Memory model:

```
The provisions of this Agreement state that disclosure of Confidential Information to
third party consultants and professional advisors is allowed, as long as those third
parties agree to be bound by this Agreement. Additionally, the Confidential
Information includes the terms of this agreement, the fact that the information is
being made available, and the possibility of a business arrangement between the
parties.
```

Evaluation Metrics

After testing out our datasets, we can see that the Deep Memory contribution is visible in retrieving more suitable information to the query provided as a question by the user. The following metrics show how the **Deep Memory feature** can improve performance:

Legal Dataset:

```
---- Evaluating without Deep Memory ----

Recall@1: 12.0%

Recall@3: 37.0%

Recall@5: 47.0%

Recall@10: 57.0%

Recall@50: 87.0%

Recall@100: 94.0%


---- Evaluating with Deep Memory ----

Recall@1: 19.0%

Recall@3: 56.0%

Recall@5: 66.0%

Recall@10: 79.0%

Recall@50: 88.0%

Recall@100: 95.0%
```

Biomedical Dataset:

```
---- Evaluating without Deep Memory ----

Recall@1: 59.0%
```

```
Recall@3: 75.0%

Recall@5: 78.0%

Recall@10: 81.0%

Recall@50: 91.0%

Recall@100: 94.0%


---- Evaluating with Deep Memory ----

Recall@1: 69.0%

Recall@3: 81.0%

Recall@5: 83.0%

Recall@10: 86.0%

Recall@50: 97.0%

Recall@100: 98.0%
```

Financial Dataset:

```
---- Evaluating without Deep Memory ----

Recall@1:         18.0%

Recall@3:         51.0%

Recall@5:         65.0%

Recall@10:        71.0%

Recall@50:        98.0%

Recall@100:       99.0%

---- Evaluating with Deep Memory ----

Recall@1:         26.0%

Recall@3:         66.0%

Recall@5:         75.0%

Recall@10:        81.0%

Recall@50:        99.0%

Recall@100:       99.0%
```

In conclusion, it is crucial to recognize that in NLP, success is not only determined by the richness of the data, but also depends on the effectiveness of the retrieval strategy. Although collecting large and diverse datasets is undeniably valuable, how the information is retrieved and presented plays a key role in optimizing model performance. As we have seen in this brief guide, there are tools such as Deep Memory that allow us to be more accurate and efficient and thus generate more relevant answers.

Biomedical Use Case: Advanced Retrieval Augmented Generation (RAG) for Pill Searching

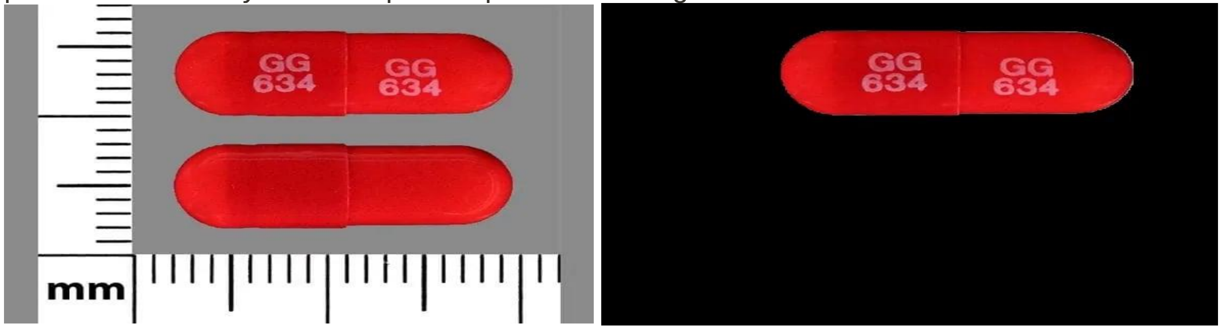
This project exploits the most advanced artificial intelligence techniques, specifically those relating to NLP and computer vision, and is made available to healthcare, allowing the user to take a photo of a pill and find information about it.
The goal is to upload a photo of a pill and recognize it. To obtain noteworthy results it was decided to divide the problem into different phases, which will be explained in the following paragraphs.

Segmentation

Initially the image is segmented so that the background does not generate false positives or false negatives, for this phase an algorithm called **FastSAM** was used. This algorithm is able to perform well on both GPU and CPU and has some characteristics to consider:

- **Real-time Solution:** FastSAM capitalizes on the computational prowess of Convolutional Neural Networks (CNNs) to offer a real-time solution for the 'segment anything' task. This feature is particularly beneficial for industrial applications where swift and efficient results are paramount.
- **Practical Applications:** FastSAM introduces a novel and practical approach for a wide array of vision tasks, delivering results at a speed that is tens to hundreds of times faster than existing methods, revolutionizing the field.
- **Based on YOLOv8-seg:** At its core, FastSAM utilizes YOLOv8-seg, a sophisticated object detector with an integrated instance segmentation branch, enabling it to efficiently generate segmentation masks for all instances in an image.
- **Efficiency and Performance:** FastSAM stands out by dramatically lowering computational and resource usage while maintaining high-quality performance. It rivals the performance of SAM but requires substantially

Here is an example of how segmentation is performed, which will allow the algorithm of the next phase to focus only on the important part of the image:



After performing the segmentation, we examine which image in our dataset is similar to the one we just segmented. This practice is performed via a neural network called **ResNet-18** and allows you to capture important image information and use it in the similarity search phase.
It is important to underline that even the images of the dataset that are being compared were all initially segmented to avoid the problem described above.
If you want to know more about how this technique works you can read the article [ResNet-18 from Deep Residual Learning for Image Recognition].

Visual Similarity

ResNet-18 is a compelling choice for computing visual similarity between images, such as in our application for identifying similarities between pill images. Its effectiveness lies in its architecture and its capability for feature extraction. Here's a breakdown of why ResNet-18 is well-suited for this task:

- **Deep Residual Learning:** ResNet-18, a variant of the Residual Network (ResNet) family, incorporates deep residual learning. In ResNet-18, there are 18 layers, including convolutional layers, batch normalization, ReLU activations, and fully connected layers.
- **Feature Extraction:** One of the primary strengths of ResNet-18 is its feature extraction capability. In the context of pill images, ResNet-18 can learn and identify intricate patterns, shapes, and colors that are unique to different pills. During the forward pass, as the image goes through successive layers, the network learns hierarchically more complex and abstract features. Initial layers might detect edges or basic shapes, while deeper layers can identify more specific features relevant to different types of pills.
- **Efficiency and Speed:** Despite being deep, ResNet-18 is relatively lightweight compared to other deeper models (like ResNet-50 or ResNet-101). This makes it a good choice for applications where computational resources or inference time might be a concern, without significantly compromising on the accuracy of feature extraction.
- **Performance in Feature Embedding:** For tasks like visual similarity, it's essential to have a robust feature embedding, which is a compressed representation of the input image. ResNet-18, due to its deep structure, can create rich, discriminative embeddings. When we input two pill images, ResNet-18 processes them to produce feature vectors.

  The similarity between these vectors can then be computed using metrics like cosine similarity or Euclidean distance. The closer these vectors are in the feature space, the more similar the images are.
  This **similarity** is **performed** directly **in** Activeloop's **Deep Lake Vector Stores**, simply take the input image and pass it to Activeloop and it will return the `n` most similar images that were found. If you want to delve deeper into this topic you can find a guide on `Activeloop` at the following link.

  Going into a little more detail in this project we can see how visual similarity search is not the only one that has been used. Once the `n` most similar images have been returned they are split into two groups:
- the `3` most similar images
- the remaining `n - 3` images

  To talk about the second type of similarity we take an intermediate step and show what the user interface looks like.

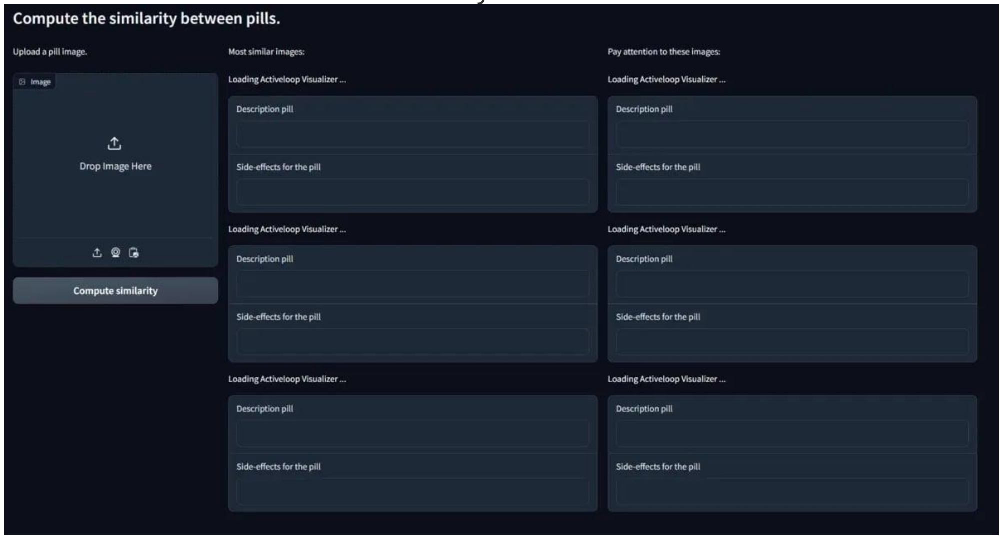
Text Extraction and Identification

In order to extract the text engraved in the pill, purely computer vision approaches were initially used and subsequently GPT-4 vision was chosen. This SAAS (software as a service) allows us to recover the text present in the pill which will then be compared with those present in the database. If a perfect match occurs, this pill will be identified as the input one, otherwise the closest image will be chosen.

Gradio Interface

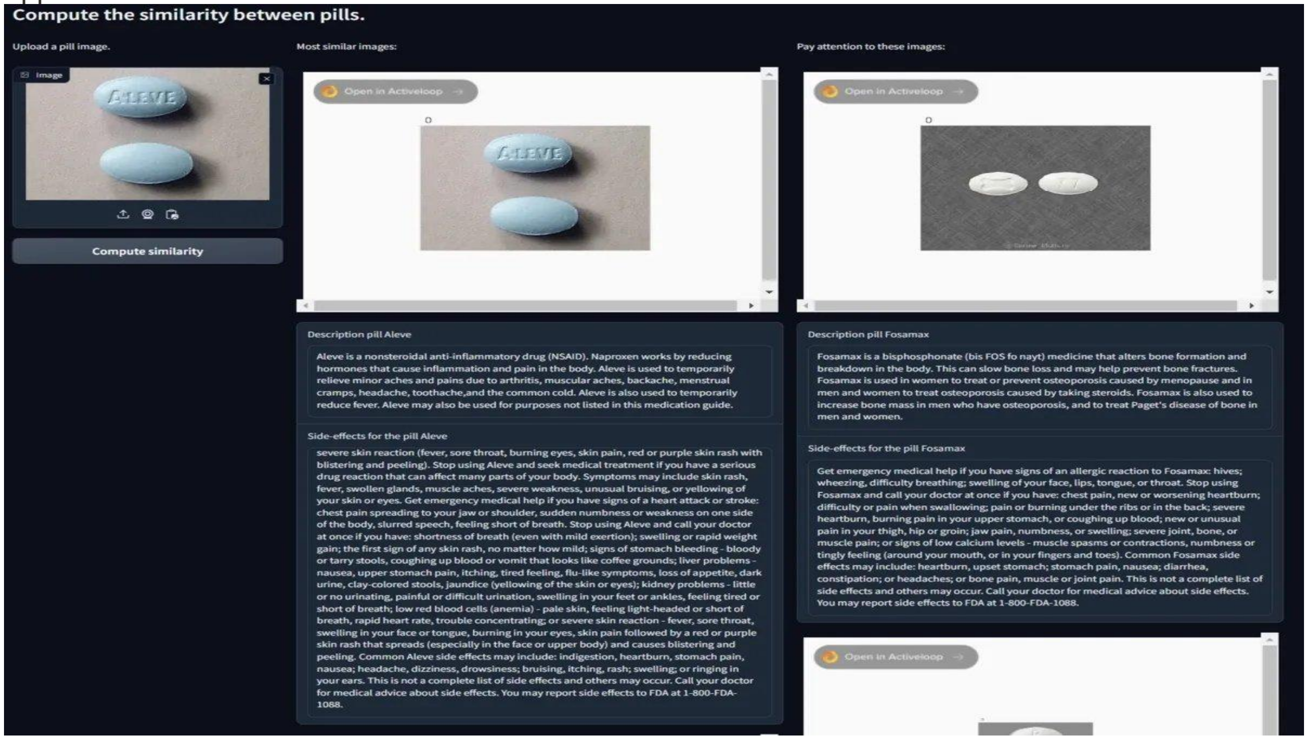
Gradio is an open-source Python library that provides an easy way to create customizable web interfaces for machine learning models. In this pill project, we have utilized Gradio to build a user-friendly interface that allows users to upload pill images and interact with the pipeline.
The results are then displayed back to the user through the Gradio interface and are divided into two different columns:
- in the first there are the `3 images` most similar to the input one
- in the second there are `3 similar images` to which we must pay attention because they have a description of the pill as different as possible from the one inserted.

It is necessary to specify that since the input image has no text but is just an image, the description taken is that of the image whose unique identification code is equal to one of those present in the dataset or, in case there is no exact match, that of the image which is absolutely most similar to the input one.
The interface that the user will initially load is as follows:


Compute the similarity between pills.
Upload a pill image.
Image
Drop Image Here
Compute similarity
Most similar images:
Pay attention to these images:
Loading Activeloop Visualizer ...
Description pill
Side-effects for the pill

Activeloop Visualizer

To show the images returned after the search, the Activeloop rendering engine called Visualizer was used. This functionality allows us to view the data present in the Deep Lake by loading it in HTML format. It was then possible to embed the Activeloop visualization engine into our RAG applications.



Description pill Aleve

Aleve is a nonsteroidal anti-inflammatory drug (NSAID). Naproxen works by reducing hormones that cause inflammation and pain in the body. Aleve is used to temporarily relieve minor aches and pains due to arthritis, muscular aches, backache, menstrual cramps, headache, toothache,and the common cold. Aleve is also used to temporarily reduce fever. Aleve may also be used for purposes not listed in this medication guide.

Side-effects for the pill Aleve

severe skin reaction (fever, sore throat, burning eyes, skin pain, red or purple skin rash with blistering and peeling). Stop using Aleve and seek medical treatment if you have a serious drug reaction that can affect many parts of your body. Symptoms may include skin rash, fever, swollen glands, muscle aches, severe weakness, unusual bruising, or yellowing of your skin or eyes. Get emergency medical help if you have signs of a heart attack or stroke: chest pain spreading to your jaw or shoulder, sudden numbness or weakness on one side of the body, slurred speech, feeling short of breath. Stop using Aleve and call your doctor at once if you have: shortness of breath (even with mild exertion); swelling or rapid weight gain; the first sign of any skin rash, no matter how mild; signs of stomach bleeding - bloody or tarry stools, coughing up blood or vomit that looks like coffee grounds; liver problems - nausea, upper stomach pain, itching, tired feeling, flu-like symptoms, loss of appetite, dark urine, clay-colored stools, jaundice (yellowing of the skin or eyes); kidney problems - little or no urinating, painful or difficult urination, swelling in your feet or ankles, feeling tired or short of breath; low red blood cells (anemia) - pale skin, feeling light-headed or short of breath, rapid heart rate, trouble concentrating; or severe skin reaction - fever, sore throat, swelling in your face or tongue, burning in your eyes, skin pain followed by a red or purple skin rash that spreads (especially in the face or upper body) and causes blistering and peeling. Common Aleve side effects may include: indigestion, heartburn, stomach pain, nausea; headache, dizziness, drowsiness; bruising, itching, rash; swelling; or ringing in your ears. This is not a complete list of side effects and others may occur. Call your doctor for medical advice about side effects. You may report side effects to FDA at 1-800-FDA-1088.

Description pill Fosamax

Fosamax is a bisphosphonate (bis FOS fo nayt) medicine that alters bone formation and breakdown in the body. This can slow bone loss and may help prevent bone fractures. Fosamax is used in women to treat or prevent osteoporosis caused by menopause and in men and women to treat osteoporosis caused by taking steroids. Fosamax is also used to increase bone mass in men who have osteoporosis, and to treat Paget's disease of bone in men and women.

Side-effects for the pill Fosamax

Get emergency medical help if you have signs of an allergic reaction to Fosamax: hives; wheezing, difficulty breathing; swelling of your face, lips, tongue, or throat. Stop using Fosamax and call your doctor at once if you have: chest pain, new or worsening heartburn; difficulty or pain when swallowing; pain or burning under the ribs or in the back; severe heartburn, burning pain in your upper stomach, or coughing up blood; new or unusual pain in your thigh, hip or groin; jaw pain, numbness, or swelling; severe joint, bone, or muscle pain; or signs of low calcium levels - muscle spasms or contractions, numbness or tingly feeling (around your mouth, or in your fingers and toes). Common Fosamax side effects may include: heartburn, upset stomach; stomach pain, nausea; diarrhea, constipation; or headaches; or bone pain, muscle or joint pain. This is not a complete list of side effects and others may occur. Call your doctor for medical advice about side effects. You may report side effects to FDA at 1-800-FDA-1088.

In this case for each cell we chose to return only the image we needed but if desired it is possible to view all the images in a single window and move between them using the cursor.
If you want to delve deeper into this functionality and integrate it into your application via Python or Javascript code you can find the guide here.
Now we can move on to the last phase, the similarity search via description of the pill.

Advanced Retrieval Strategies

A technique that is becoming increasingly popular in this period is the Retrieval-Augmented Generation (RAG) which enhances large language models (LLMs) by integrating external authoritative knowledge sources beyond their initial training datasets for response generation. LLMs, trained on extensive data and utilizing billions of parameters, excel in tasks such as question answering, language translation, and sentence completion.
RAG builds upon these strengths, tailoring LLMs to particular domains or aligning them with an organization's internal knowledge, without necessitating model retraining. This method provides a cost-efficient solution to refine LLM outputs, ensuring their continued relevance, precision, and utility across diverse applications.

There are five key stages within RAG, which in turn will be a part of any larger application you build. These are:

- **Loading:** This involves transferring data from its original location, such as text files, PDFs, websites, databases, or APIs, into your processing pipeline. LlamaHub offers a wide range of connectors for this purpose.
- **Indexing:** This process entails developing a data structure that supports data querying. In the context of LLMs, it typically involves creating vector embeddings, which are numerical representations of your data's meaning, along with various metadata strategies to facilitate the accurate retrieval of contextually relevant data.
- **Storing:** After indexing, it's common practice to store the index and other metadata. This step prevents the need for re-indexing in the future.
- **Querying:** Depending on the chosen indexing method, there are multiple ways to deploy LLMs and LlamaIndex structures for querying, including options like sub-queries, multi-step queries, and hybrid approaches.
- **Evaluation:** This crucial phase in the pipeline assesses its effectiveness against other methods or following modifications. Evaluation provides objective metrics of the accuracy, reliability, and speed of your system's responses to queries.

These processes can be easily and clearly represented by the following diagram in the LLamaIndex guide:
Since we have a description for each pill we used these descriptions as if they were documents in order to then be able to obtain the most similar ones (and therefore also the least similar ones) once the description of the input pill was passed to the model.
After initially loading our data onto Activeloop's Deep Lake we can show how the data present in this space is used to do an Advanced Retrieval.

```
import DeepLakeVectorStore

def create_upload_vectore_store(

                                    chunked_text: list,

                                    vector_store_path: Union[str, os.PathLike],
                        filename: str,
                        metadata: Optional[list[dict]] = None ):

    vector_store = DeepLakeVectorStore(
        dataset_path=vector_store_path,
        runtime={"tensor_db": True},
        overwrite=True,
        tensor_params=[
            {"name": "text", "htype": "text"},
            {"name": "embedding", "htype": "embedding"},
            {"name": "filename", "htype": "text"},
            {"name": "metadata", "htype": "json"}         ])
    vector_store = vector_store.vectorstore
    vector_store.add(
        text=chunked_text,
        embedding_function=embedding_function_text,
        filename=filename,
        embedding_data=chunked_text,
        rate_limiter={
            "enabled": True,
            "bytes_per_minute": 1500000,
            "batch_byte_size": 10000      },
        metadata=metadata if metadata else None    )
```

For all subsequent steps we used LlamaIndex which is a data framework for LLM-based applications used to capture, structure and access private or domain-specific data.

Indexing Phase

Once we've ingested the data, LlamaIndex will help us index the data into a structure that's easy to retrieve. This involves generating vector embeddings which are stored in a specialized database called a vector store, in our case we stored them into the Deep Lake Vector Store. Indexes can also store a variety of metadata about your data.
Under the hood, Indexes store data in Node objects (which represent chunks of the original documents), and expose a Retriever interface that supports additional configuration and automation.
This part is made up of two main blocks:

- **Embedding**: we used OpenAI's `text-embedding-ada-002` as the embedding model
- **Retriever**: we tried different approaches which will be described below

Retriever Phase

Retrievers are responsible for fetching the most relevant context given a user query (or chat message). It can be built on top of indexes, but can also be defined independently. It is used as a key building block in query engines (and Chat Engines) for retrieving relevant context.

Vector Store Index

A VectorStoreIndex is by far the most frequent type of Index you'll encounter. The Vector Store Index takes your Documents and splits them up into Nodes. It then creates vector embeddings of the text of every node, ready to be queried by an LLM. The vector store index stores each Node and a corresponding embedding in a Vector Store.
Source Image
Querying a vector store index involves fetching the top-k most similar Nodes, and passing those into our Response Synthesis module.
Source Image

BM25

BM25 is a popular ranking function used by search engines to estimate the relevance of documents to a given search query. It's based on probabilistic models and improves upon earlier models like TF-IDF (Term Frequency-Inverse Document Frequency). BM25 considers factors like term frequency and document length to provide a more nuanced approach to relevance scoring. It handles the issue of term saturation (where the importance of a term doesn't always increase linearly with frequency) and length normalization (adjusting scores based on document length to prevent bias toward longer documents). BM25's effectiveness in various search tasks has made it a standard in information retrieval.
To use this retriever we need to take documents from Activeloop's Deep Lake and transform them into nodes, these nodes will then be returned in an orderly manner once a question is asked. This process, as already mentioned previously, exploits the similarity between the description of the pill (which will be passed as a query) and the description of the `n - 3` most similar pills.
In the code below we used the name of the images to make a specific query that returned only the values of those images and by extrapolating their description we created the nodes and indexes.

```python
def get_index_and_nodes_after_visual_similarity(filenames: list):
    vector_store = load_vector_store(vector_store_path=VECTOR_STORE_PATH_DESCRIPTION)


    conditions = " or ".join(f"filename == '{name}'" for name in filenames)
    tql_query = f"select * where {conditions}"


    filtered_elements = vector_store.vectorstore.search(query=tql_query)
    chunks = []
    for el in filtered_elements["text"]:
        chunks.append(el)


    string_iterable_reader = download_loader("StringIterableReader")
    loader = string_iterable_reader()
    documents = loader.load_data(texts=chunks)
    node_parser = SimpleNodeParser.from_defaults(separator="\n")
    nodes = node_parser.get_nodes_from_documents(documents)

    # To ensure same id's per run, we manually set them.
    for idx, node in enumerate(nodes):
        node.id_ = f"node_{idx}"

    llm = OpenAI(model="gpt-4")

    service_context = ServiceContext.from_defaults(llm=llm)
    index = VectorStoreIndex(nodes=nodes)
    return index, nodes, service_context, filtered_elementsCopy
```

Since we have the `n` most similar images (obtained in the previous step through the visual similarity), we can extract the description of these `n` images and use them to generate the nodes.

Why do we need to care about different retrieval methods and how are they different from each other?

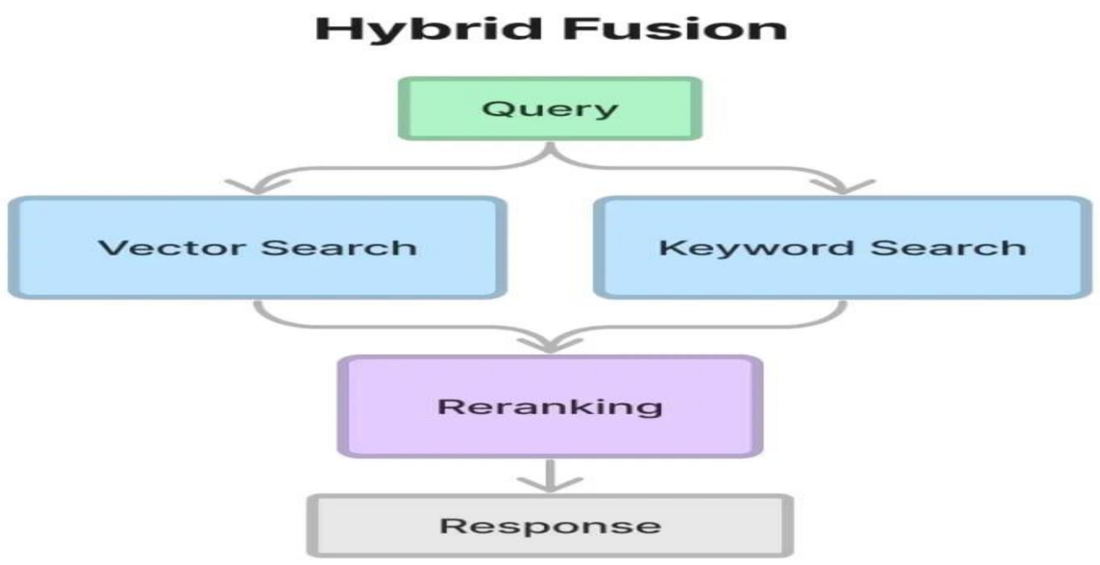
RAG (Retrieval-Augmented Generation) systems retrieve relevant information from a given knowledge base, thereby allowing it to generate factual, contextually relevant, and domain-specific information. However, RAG faces a lot of challenges when it comes to effectively retrieving relevant information and generating high-quality responses.

Traditional search engines work by parsing documents into chunks and indexing these chunks. The algorithm then searches this index for relevant results based on a user's query. Retrieval Augmented Generation is a new paradigm in machine learning that uses large language models (LLMs) to improve search and discovery. The LLMs, like the GPT-4, generate relevant content based on context.

The advanced technique utilized in this project is the **Hybrid Search**. It is a technique that combines multiple search algorithms to improve the accuracy and relevance of search results. It uses the best features of **both keyword-based search algorithms with vector search techniques**. By leveraging the strengths of different algorithms, it provides a more effective search experience for users.

Hybrid Fusion Retriever

In advanced technique, we merge the **vector store based retriever and the BM25 based retriever**. This will enable us to capture **both semantic relations and keywords** in our input queries. Since both of these retrievers calculate a score, we can use the **reciprocal rerank** algorithm to re-sort our nodes without using additional models or excessive computation. We can see the scheme in the image below taken from the LlamaIndex guide:



BM25 Retriever + Re-Ranking technique (classic approach with BM25)

In this first case we use a classic retriever based only on the **BM25 retriever**, the nodes generated by the query will then be re-ranked and filtered. This allows us to keep the intermediate top-k values large and filter out unnecessary nodes.

```
_, nodes, service_context = get_index_and_nodes_from_activeloop(
    vector_store_path=VECTOR_STORE_PATH_BASELINE)
```

These nodes will then be used by the bm25 retriever. It is useful to note that the index variable is not currently used because bm25 manages this entire part internally.

```
bm25_retriever = BM25Retriever.from_defaults(nodes=nodes, similarity_top_k=10)
```

Now that we have the nodes we need to obtain the similarity by passing the description of the input image as a query.

```
nodes_bm25_response = bm25_retriever.retrieve(description)
```

ClassicRetrieverBM25 is an object that we used to manage the creation of the BM25-based retriever in a more orderly way.

```
class ClassicRetrieverBM25(BaseRetriever):
    def __init__(self, bm25_retriever):
        self.bm25_retriever = bm25_retriever
        super().__init__()

    def _retrieve(self, query, **kwargs):
        bm25_nodes = self.bm25_retriever.retrieve(query, **kwargs)
        all_nodes = []
        node_ids = set()
        for n in bm25_nodes:
            if n.node.node_id not in node_ids:
                all_nodes.append(n)
                node_ids.add(n.node.node_id)
        return all_nodesCopy
```

The final part of this process is generated by the re-renker which takes care of ordering the nodes according to their score.

```
reranker = SentenceTransformerRerank(top_n=4, model="BAAI/bge-reranker-base")
# nodes retrieved by the bm25 retriever with the reranker
reranked_nodes_bm25 = reranker.postprocess_nodes(
    nodes_bm25_response,
    query_bundle=QueryBundle(QUERY),)
```

Advanced - Hybrid Retriever + Re-Ranking technique with BM25 & vector retriever
Here we extend the base retriever class and create a custom retriever that always uses the **vector retriever** and **BM25 retreiver**. In this test, the previous approach was used but the vector store was added as a retriever which uses the index variable returned by the `get_index_and_nodes_after_visual_similarity` function to manage the nodes.
At the beginning, we create the standard retriever:

```
index, nodes, _ = get_index_and_nodes_from_activeloop(
    vector_store_path=VECTOR_STORE_PATH_COMPLETE_SEQUENTIALLY)
self.vector_retriever = index.as_retriever(similarity_top_k=2)
self.bm25_retriever = BM25Retriever.from_defaults(
    nodes=nodes, similarity_top_k=10)
```

The nodes will then be calculated both via the vector store and bm25 and once they are all put together they will be re-ranked and filtered. This process allows us to get the best of both models and filter out the `k best nodes`.

```
reranked_nodes_bm25 = self.reranker.postprocess_nodes(
                                    self.nodes_bm25_response,
                                    query_bundle=QueryBundle(QUERY),)
print("Reranked Nodes BM25\n\n")
for el in reranked_nodes_bm25:
    print(f"{el.score}\n")
```

```
reranked_nodes_vector = self.reranker.postprocess_nodes(
self.nodes_vector_response,
query_bundle=QueryBundle(QUERY)   )
print("Reranked Nodes Vector\n\n")
for el in reranked_nodes_vector:
    print(f"{el.score}\n")
    unique_nodes = keep_best_k_unique_nodes(
        reranked_nodes_bm25, reranked_nodes_vector )
    print("Unique Nodes\n\n")
for el in unique_nodes:
    print(f"{el.id} : {el.score}\n")
```

Advanced - Hybrid Retriever + Re-Ranking technique with BM25 and the vector retriever and QueryFusionRetriever

In the last case we can see how through the **QueryFusionRetriever** object the entire process described previously can be represented with a single function.
We fuse our index with a BM25 based retriever, this will enable us to capture both semantic relations and keywords in our input queries.
Since both of these retrievers calculate a score, we can use the reciprocal rerank algorithm to re-sort our nodes without using an additional models or excessive computation.

```
from llama_index.retrievers import BM25Retriever
vector_retriever = index.as_retriever(similarity_top_k=2)
bm25_retriever = BM25Retriever.from_defaults(    docstore=index.docstore,
similarity_top_k=2)
```

Here we can create our fusion retriever, which will return the top-2 most similar nodes from the 4 returned nodes from the retrievers:

```
from llama_index.retrievers import QueryFusionRetriever
retriever = QueryFusionRetriever(
    [vector_retriever, bm25_retriever],
    similarity_top_k=2,
    num_queries=4,  # set this to 1 to disable query generation
    mode="reciprocal_rerank",
    use_async=True,     verbose=True )
```

Finally, we can perform the query search:

```
retriever.retrieve(description)
```

Metrics and Conclusions

Advanced retrievers, like the ones we've been discussing, represent a significant leap in our ability to process and analyze vast amounts of data. These systems, armed with sophisticated algorithms like BM25, vector store, and the latest developments in embedding models from OpenAI, are not just tools; they are gateways to a new era of information accessibility and knowledge discovery.

The power of these retrievers lies in their understanding of context, their ability to sift through massive data sets to find relevant information, and their efficiency in providing accurate results. They have become indispensable in sectors where precision and speed are essential. In the healthcare sector, for example, their application to identify and cross-reference medical information can represent a game changer, improving both the quality of care and patient safety.

All tested models work very well for our use case & in favor of this thesis are following metrics:

BM25 Retriever + Re-Ranking technique (classic approach with BM25)

```
retrievers  hit_rate    mrr

top-2 eval  0.964643    0.944501
```

Advanced - Hybrid Retriever + Re-Ranking technique with BM25 and the vector retriever

```
retrievers  hit_rate    mrr

top-2 eval  0.975101    0.954078
```

Advanced - Hybrid Retriever + Re-Ranking technique with BM25 and the vector retriever and QueryFusionRetriever

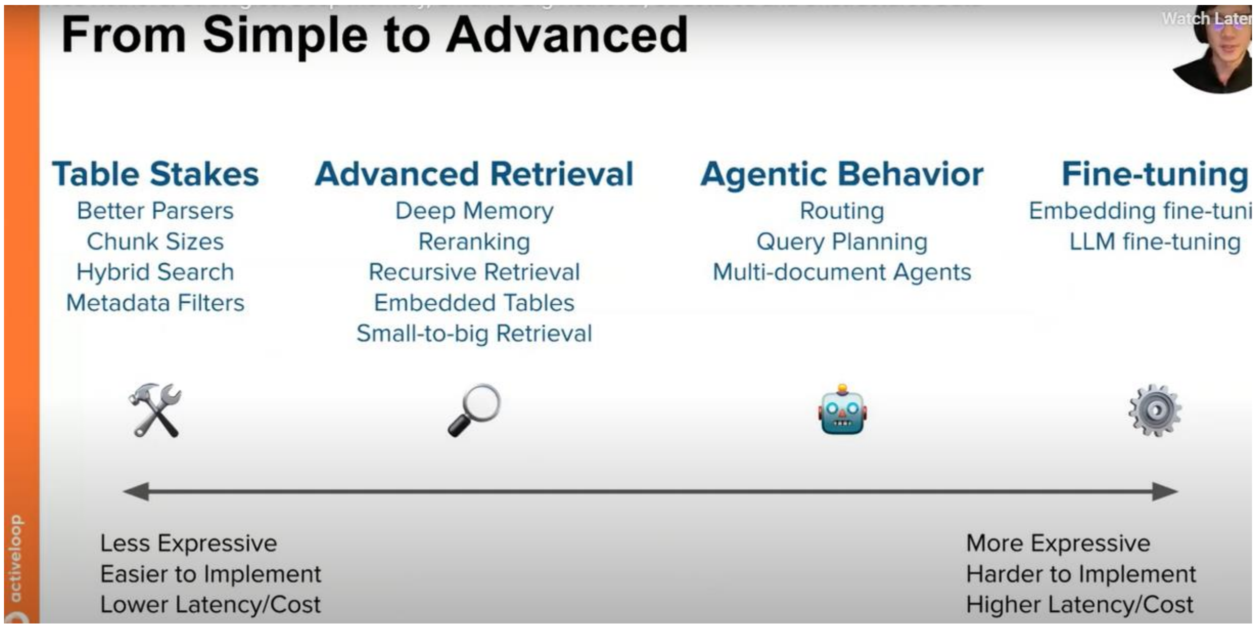
```
retrievers  hit_rate    mrr

top-2 eval  0.977138    0.954235
```

Where the **hit_rate** and **MRR (Mean Reciprocal Rank)** are two metrics commonly used to evaluate the performance of information retrieval systems, search algorithms, and recommendation systems.

Hit Rate: The hit rate is a measure of accuracy, specifically the proportion of times a system successfully retrieves relevant documents or items. It's often used in the context of recommendation systems to evaluate if any of the recommended items are relevant. A hit is usually defined by whether the relevant item appears in the top-N recommendations or retrieval results. For instance, if the system provides 10 recommendations, and at least one of them is relevant, it's considered a hit. The hit rate is the number of hits divided by the total number of recommendations or queries made.

Mean Reciprocal Rank (MRR): MRR is a statistic that measures the average of the reciprocal ranks of results for a sample of queries. The reciprocal rank of a query response is the multiplicative inverse of the rank of the first correct answer. For example, if the first relevant document for a query is located at the third position in a list of ranked items, the reciprocal rank is 1/3. MRR is calculated by taking the average of the reciprocal ranks across all queries. It gives a higher score to systems where the relevant item appears earlier in the recommendation or search results list, therefore considering the ranking of results, unlike hit rate which is binary.

Both metrics are critical for assessing how effectively a system presents the relevant information to users, with the hit rate focusing on presence in the results and MRR on the rank or position of the first relevant result.

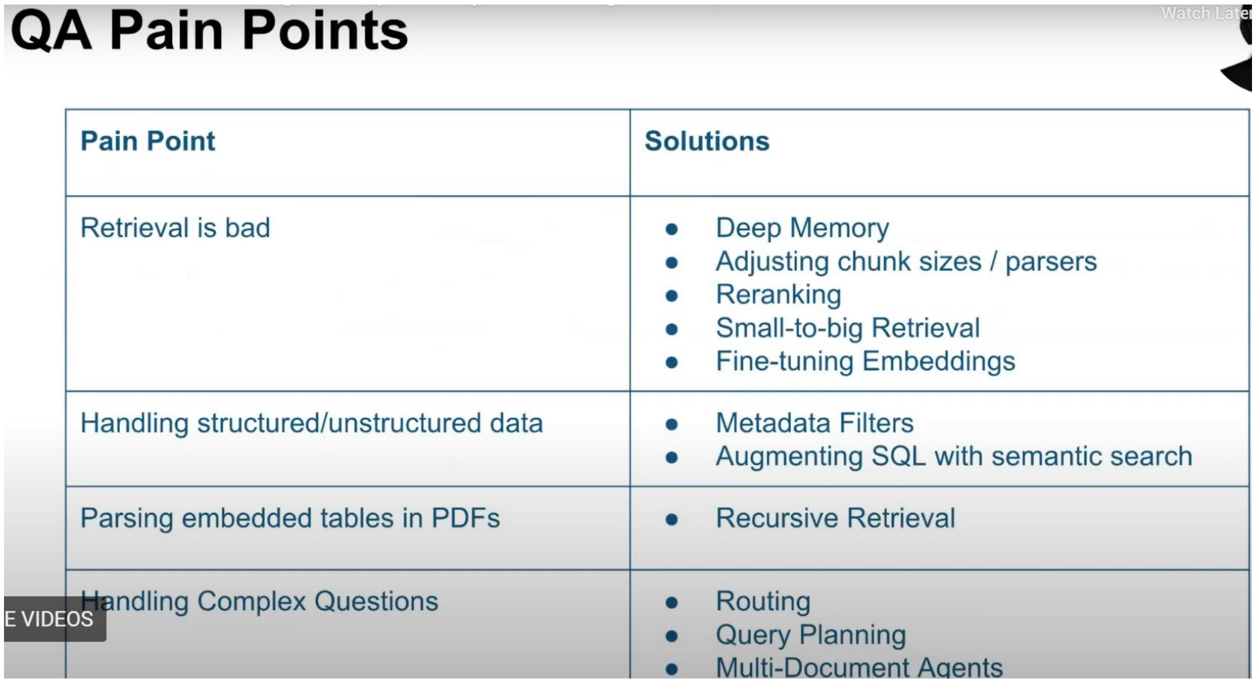


From Simple to Advanced

Table Stakes	Advanced Retrieval	Agentic Behavior	Fine-tuning
Better Parsers	Deep Memory	Routing	Embedding fine-tuni
Chunk Sizes	Reranking	Query Planning	LLM fine-tuning
Hybrid Search	Recursive Retrieval	Multi-document Agents	
Metadata Filters	Embedded Tables		
	Small-to-big Retrieval		

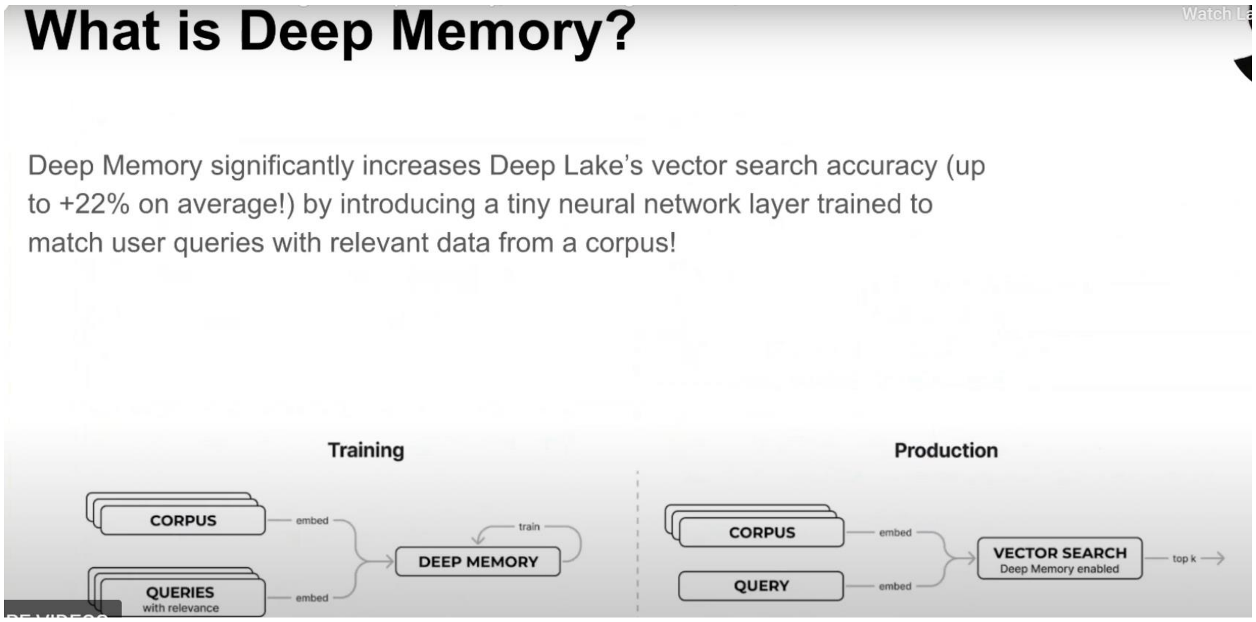
Less Expressive
Easier to Implement
Lower Latency/Cost

More Expressive
Harder to Implement
Higher Latency/Cost



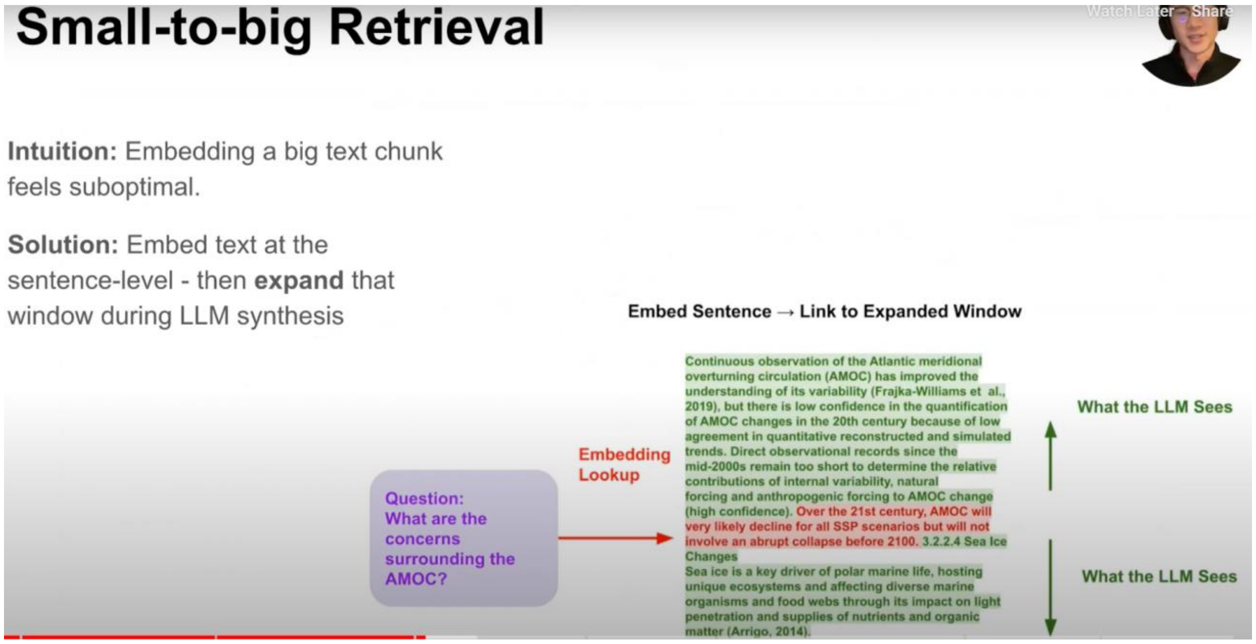
QA Pain Points

Pain Point	Solutions
Retrieval is bad	• Deep Memory • Adjusting chunk sizes / parsers • Reranking • Small-to-big Retrieval • Fine-tuning Embeddings
Handling structured/unstructured data	• Metadata Filters • Augmenting SQL with semantic search
Parsing embedded tables in PDFs	• Recursive Retrieval
Handling Complex Questions	• Routing • Query Planning • Multi-Document Agents



What is Deep Memory?

Deep Memory significantly increases Deep Lake's vector search accuracy (up to +22% on average!) by introducing a tiny neural network layer trained to match user queries with relevant data from a corpus!



Small-to-big Retrieval

Intuition: Embedding a big text chunk feels suboptimal.

Solution: Embed text at the sentence-level - then **expand** that window during LLM synthesis

Embed Sentence → Link to Expanded Window

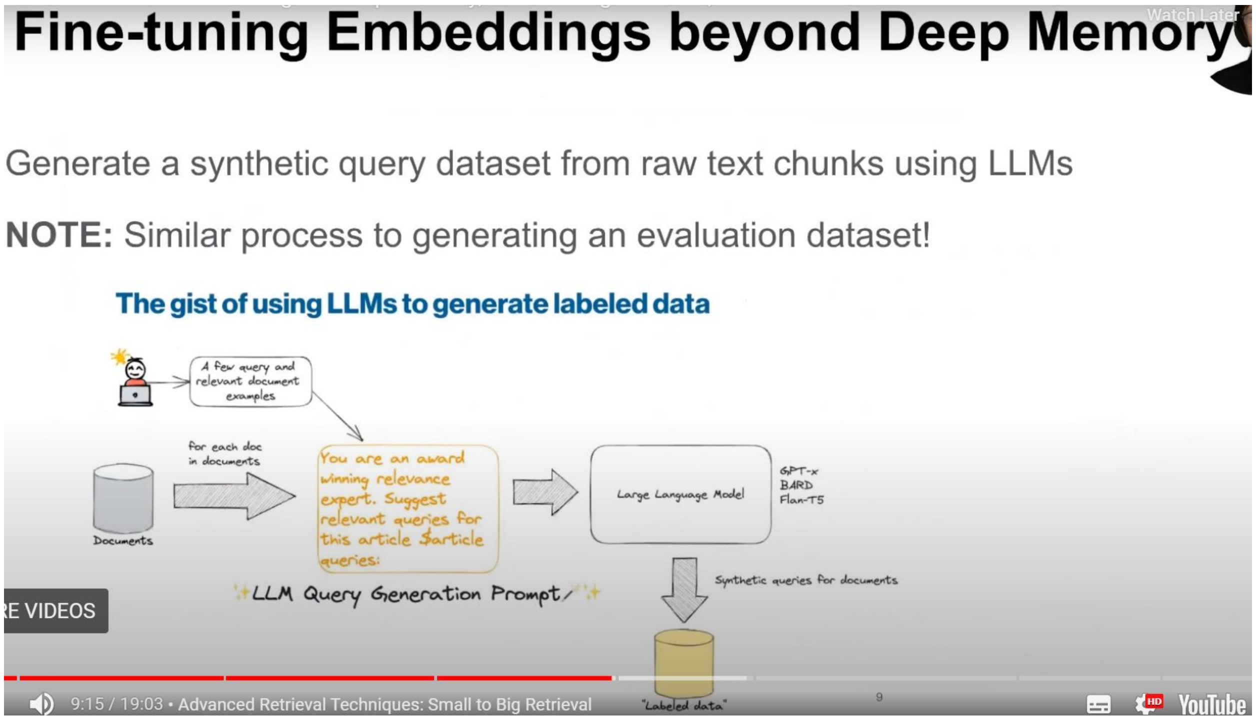
Question: What are the concerns surrounding the AMOC?

Embedding Lookup

Continuous observation of the Atlantic meridional overturning circulation (AMOC) has improved the understanding of its variability (Frajka-Williams et al., 2019), but there is low confidence in the quantification of AMOC changes in the 20th century because of low agreement in quantitative reconstructed and simulated trends. Direct observational records since the mid-2000s remain too short to determine the relative contributions of internal variability, natural forcing and anthropogenic forcing to AMOC change (high confidence). Over the 21st century, AMOC will very likely decline for all SSP scenarios but will not involve an abrupt collapse before 2100. 3.2.2.4 Sea Ice Changes
Sea ice is a key driver of polar marine life, hosting unique ecosystems and affecting diverse marine organisms and food webs through its impact on light penetration and supplies of nutrients and organic matter (Arrigo, 2014).

What the LLM Sees

What the LLM Sees



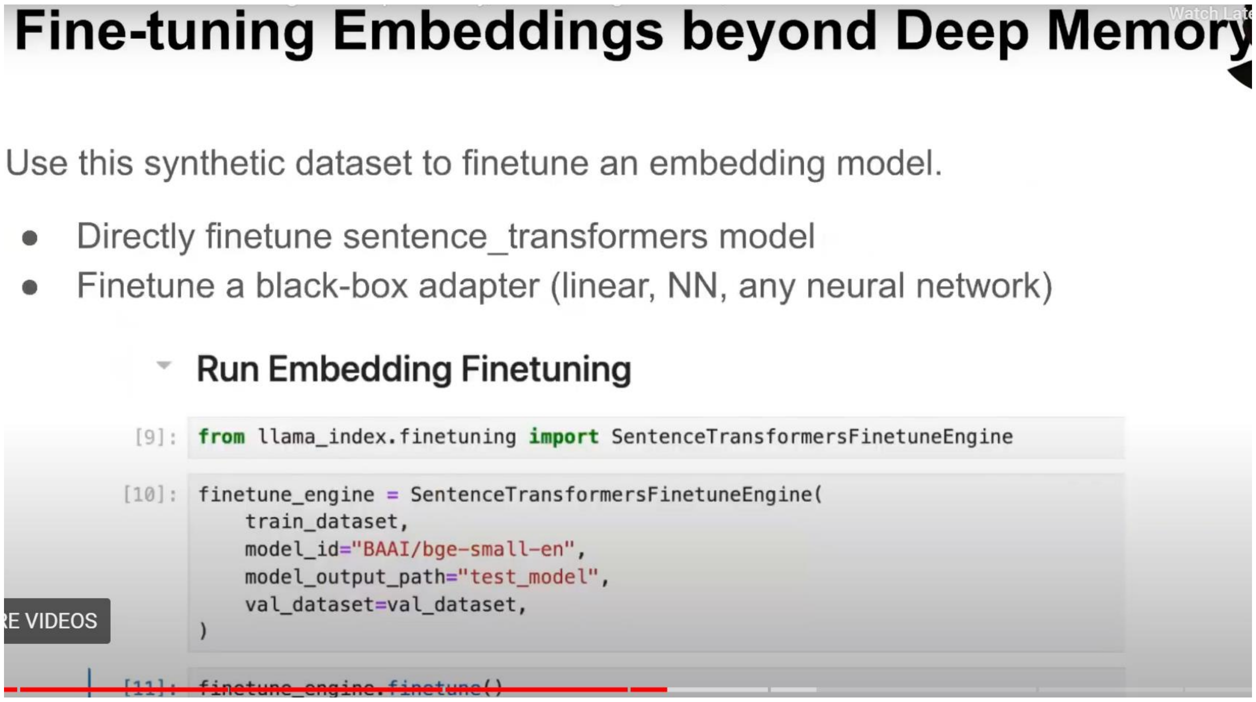
Fine-tuning Embeddings beyond Deep Memory

Generate a synthetic query dataset from raw text chunks using LLMs

NOTE: Similar process to generating an evaluation dataset!

The gist of using LLMs to generate labeled data

A few query and relevant document examples
For each doc in documents
Documents
You are an award winning relevance expert. Suggest relevant queries for this article $article queries:
LLM Query Generation Prompt
Large Language Model
GPT-x
BARD
Flan-T5
Synthetic queries for documents
"Labeled data"
Watch Later
E VIDEOS
9:15 / 19:03 • Advanced Retrieval Techniques: Small to Big Retrieval
YouTube



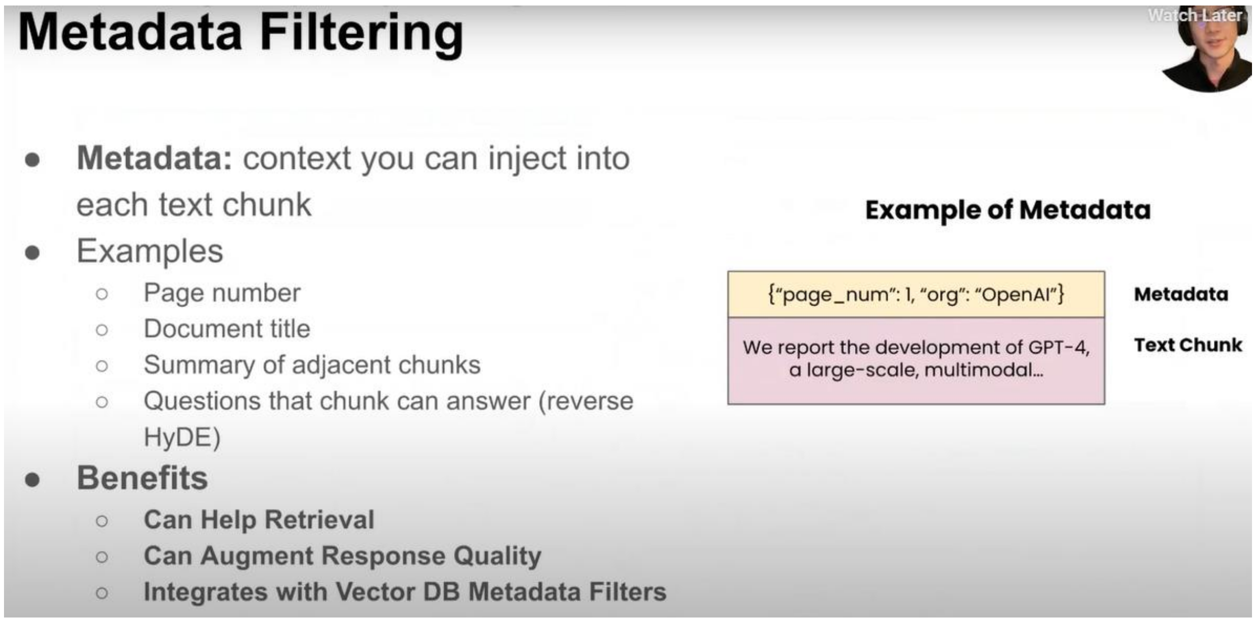
Fine-tuning Embeddings beyond Deep Memory

Use this synthetic dataset to finetune an embedding model.

- Directly finetune sentence_transformers model
- Finetune a black-box adapter (linear, NN, any neural network)

Run Embedding Finetuning

```
[9]: from llama_index.finetuning import SentenceTransformersFinetuneEngine

[10]: finetune_engine = SentenceTransformersFinetuneEngine(
          train_dataset,
          model_id="BAAI/bge-small-en",
          model_output_path="test_model",
          val_dataset=val_dataset,
      )

[11]: finetune_engine.finetune()
```

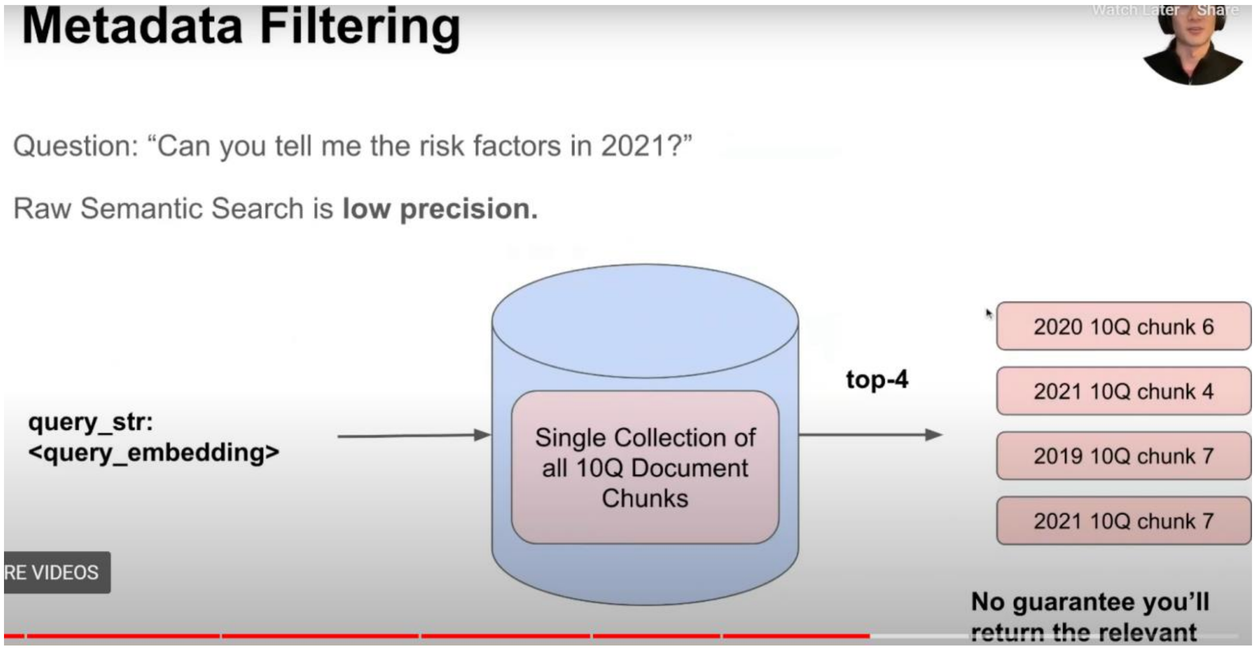


Metadata Filtering

- **Metadata:** context you can inject into each text chunk
- Examples
  - Page number
  - Document title
  - Summary of adjacent chunks
  - Questions that chunk can answer (reverse HyDE)
- **Benefits**
  - **Can Help Retrieval**
  - **Can Augment Response Quality**
  - **Integrates with Vector DB Metadata Filters**

Example of Metadata

{"page_num": 1, "org": "OpenAI"}	**Metadata**
We report the development of GPT-4, a large-scale, multimodal...	**Text Chunk**

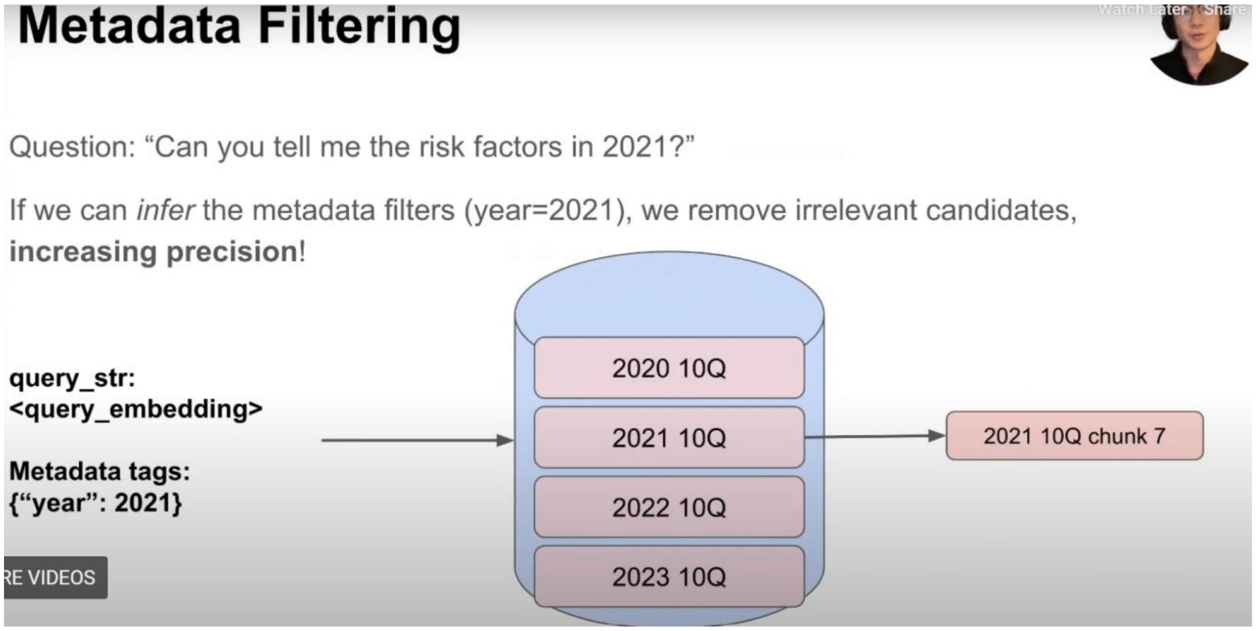


Metadata Filtering

Question: "Can you tell me the risk factors in 2021?"

Raw Semantic Search is **low precision.**

No guarantee you'll return the relevant



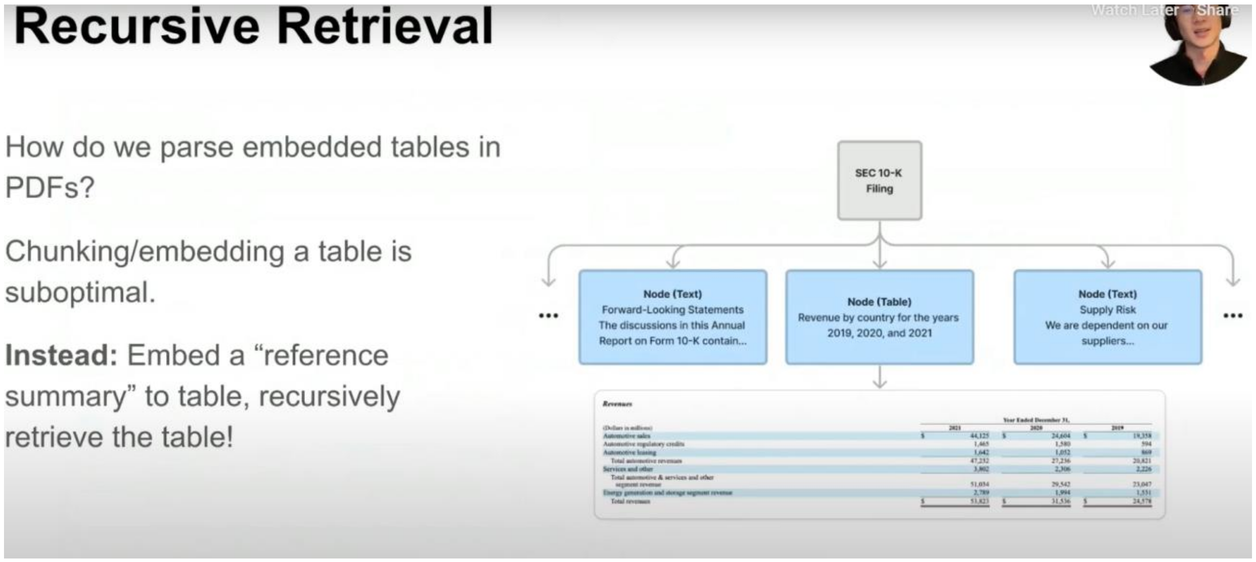
Metadata Filtering

Question: “Can you tell me the risk factors in 2021?”

If we can *infer* the metadata filters (year=2021), we remove irrelevant candidates, **increasing precision**!

query_str:
<query_embedding>

Metadata tags:
{“year”: 2021}



Recursive Retrieval

How do we parse embedded tables in PDFs?

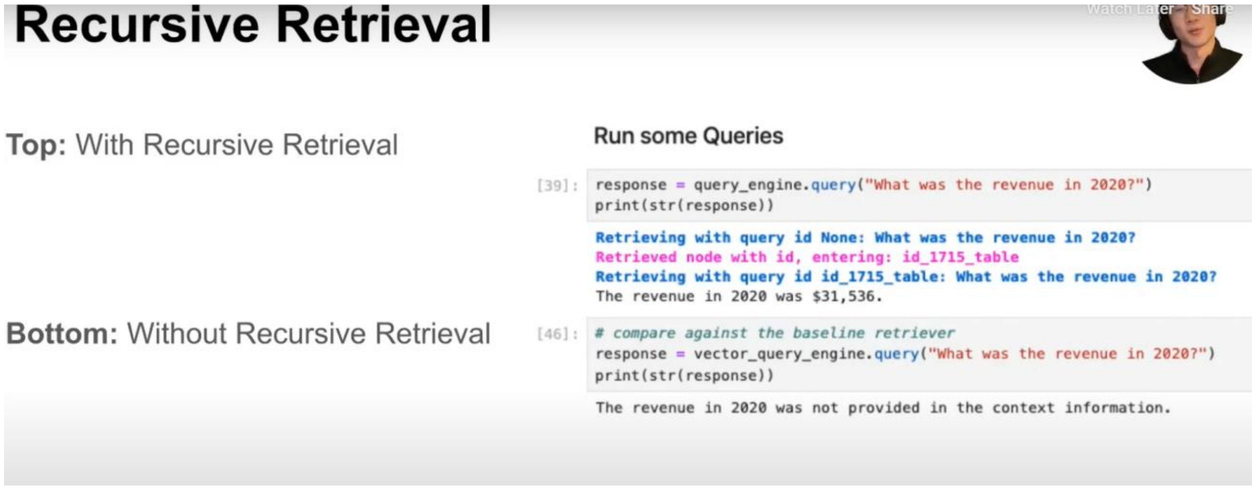
Chunking/embedding a table is suboptimal.

Instead: Embed a “reference summary” to table, recursively retrieve the table!

SEC 10-K Filing
Node (Text) Forward-Looking Statements The discussions in this Annual Report on Form 10-K contain...
Node (Table) Revenue by country for the years 2019, 2020, and 2021
Node (Text) Supply Risk We are dependent on our suppliers...
...
...

Revenues

	Year Ended December 31,		
(Dollars in millions)	2021	2020	2019
Automotive sales	$ 44,125	$ 24,604	$ 19,358
Automotive regulatory credits	1,465	1,580	594
Automotive leasing	1,642	1,052	869
Total automotive revenues	47,232	27,236	20,821
Services and other	3,802	2,306	2,226
Total automotive & services and other segment revenue	51,034	29,542	23,047
Energy generation and storage segment revenue	2,789	1,994	1,531
Total revenues	$ 53,823	$ 31,536	$ 24,578



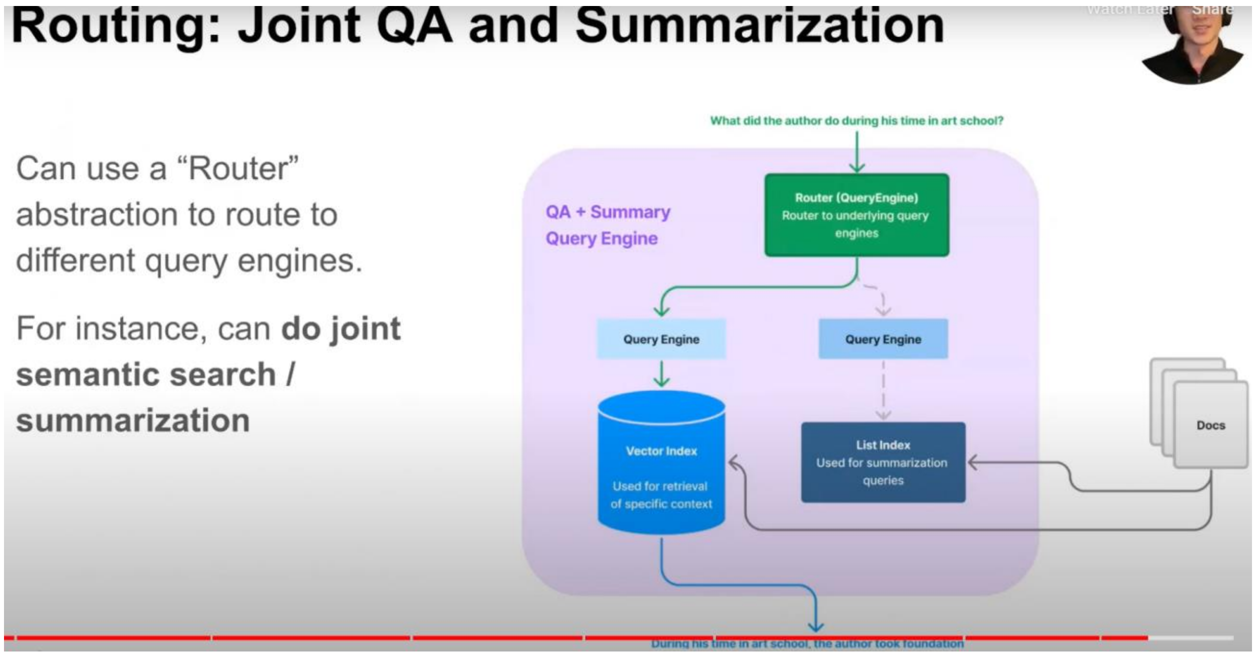
Recursive Retrieval

Top: With Recursive Retrieval

Bottom: Without Recursive Retrieval

Run some Queries

```
[39]: response = query_engine.query("What was the revenue in 2020?")
      print(str(response))

Retrieving with query id None: What was the revenue in 2020?
Retrieved node with id, entering: id_1715_table
Retrieving with query id id_1715_table: What was the revenue in 2020?
The revenue in 2020 was $31,536.

[46]: # compare against the baseline retriever
      response = vector_query_engine.query("What was the revenue in 2020?")
      print(str(response))

The revenue in 2020 was not provided in the context information.
```



Routing: Joint QA and Summarization

Can use a “Router” abstraction to route to different query engines.

For instance, can **do joint semantic search / summarization**

What did the author do during his time in art school?
QA + Summary Query Engine
Router (QueryEngine) Router to underlying query engines
Query Engine
Query Engine
Vector Index Used for retrieval of specific context
List Index Used for summarization queries
Docs
During his time in art school, the author took foundation

https://youtu.be/TdjvzSpjBuI