Sunil Kumar Dash — Updated On November 17th, 2023

Deep Learning Guide Intermediate Large Language Models NLP

Introduction

One of the most popular applications of large language models (LLMs) is to answer questions about custom datasets. LLMs like ChatGPT and Bard are excellent communicators. They can answer almost anything that they have been trained on. This is also one of the biggest bottlenecks for LLMs. They can only answer the questions they have seen during model training. The language models have a cap on the knowledge about the world. For example, Chatgpt has been trained on data available until 2021. Also, there is no way GPT can learn about your private files. So, how can we make the model aware of the knowledge it does not possess yet? The answer is a Retrieval Augmented Generation Pipeline. In this article, we will learn about the RAG (Retrieval Augmented Generation) pipeline and build one using the LLama Index.

Learning Objectives

- Explore what Retrieval Augmented Generation (RAG) is and when we should use it.
- Understand different components of RAG in brief.
- Learn about the Llama Index and how to use it to build a simple RAG pipeline for PDFs.
- Understand what embeddings and vector databases are and how to use Llama Index's inbuilt modules to build knowledge bases from PDFs.
- Discover the real-world use cases of RAG-based applications.

This article was published as a part of the Data Science Blogathon.

Table of contents

What is RAG?

LLMs are the most efficient and powerful NLP models to this date. We have seen the potential of LLMs in translation, essay writing, and general question-answering. But when it comes to domain-specific question-answering, they suffer from hallucinations. Besides, in a domain-specific QA app, only a few documents contain relevant context per query. So, we need a unified system that streamlines document extraction to answer generation and all the processes between them. This process is called **Retrieval Augmented Generation.**

Learn More: Retrieval-Augmented Generation (RAG) in AI

So, let's understand why RAG is most effective for building real-world domain-specific QA apps.

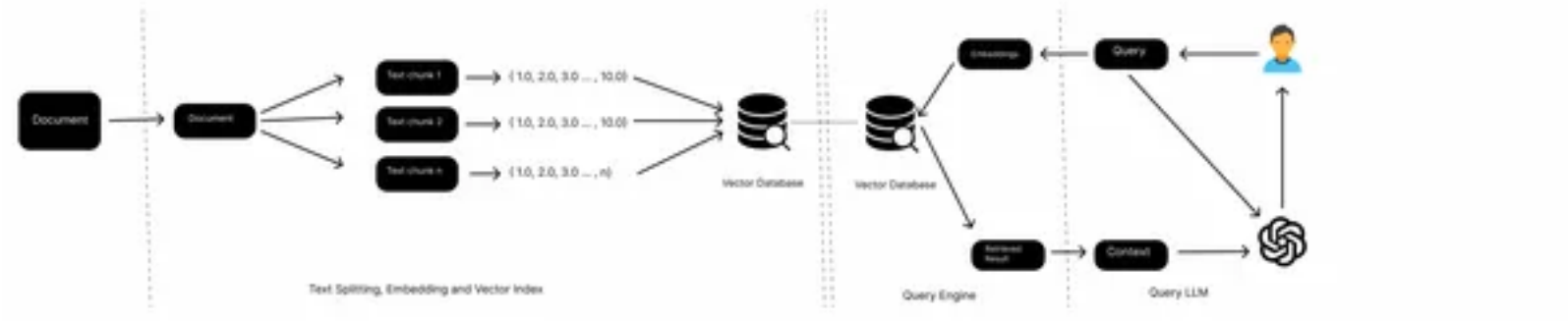
Why Should One Use RAG?

There are three ways an LLM can learn new data.

- **Training**: A large mesh of neural networks is trained over trillions of tokens with billions of parameters to create Large Language Models. The parameters of a deep learning model are the coefficients or weights that hold all the information regarding the particular model. To train a model like GPT-4 costs hundreds of millions of dollars. This way is beyond anyone's capacity. We cannot re-train such a humongous model on new data. This is not feasible.
- **Fine-tuning**: Another option is to fine-tune a model on existing data. Fine-tuning involves using a pre-trained model as a starting point during training. We use the knowledge of the pre-trained model to train a new model on different data sets. Albeit it is very potent, it is expensive in terms of time and money. Unless there is a specific requirement, fine-tuning does not make sense.
- **Prompting:** Prompting is the method where we fit new information within the context window of an LLM and make it answer the queries from the information given in the prompt. It may not be as effective as knowledge learned during training or fine-tuning, but it is sufficient for many real-life use cases, such as document Q&A.

Prompting for answers from text documents is effective, but these documents are often much larger than the context windows of Large Language Models (LLMs), posing a challenge. Retrieval Augmented Generation (RAG) pipelines address this by processing, storing, and retrieving relevant document sections, allowing LLMs to answer queries efficiently. So, let's discuss the crucial components of an RAG pipeline.

- **Text Splitter:** Splits documents to accommodate context windows of LLMs.
- **Embedding Model:** The deep learning model used to get embeddings of documents.
- **Vector Stores:** The databases where document embeddings are stored and queried along with their metadata.
- **LLM:** The Large Language Model responsible for generating answers from queries.
- **Utility Functions:** This involves additional utility functions such as Webretriver and document parsers that aid in retrieving and pre-processing files.



The above picture is of a typical RAG process. We have documents (PDFs, Web Pages, Docs), a tool to split large text into smaller chunks, embedding models to get vector representation of text chunks, Vector stores as knowledge bases, and an LLM to get answers from text chunks.

What is the Llama Index?

The Llama Index (GPTIndex) is a framework written in Python to build LLM applications. It is a simple, flexible data framework connecting custom data sources to large language models. It provides appropriate tools to support data ingestion from various sources, vector databases for data indexing, and query interfaces for querying large documents. In short, Llama Index is a one-stop shop for building retrieval augmented generation applications. It allows easy integration with other applications like Langchain, Flask, Docker, etc. Check out the official GitHub repository for more: https://github.com/run-llama/llama_index.

Also Read: How to use LlamaIndex?

Now that we know about RAG and Llama Index. So, let's build our RAG pipeline to process PDF documents and discuss individual concepts as we proceed.

Set-up Dev Environment

The first rule of building any Python project is to create a Virtual environment. Once you are done, install the following libraries.

```
llama-index
openai
tiktoken
```

Now, import the following functions.

```
from llama_index.text_splitter import TokenTextSplitter
from llama_index.node_parser import SimpleNodeParser
from llama_index import VectorStoreIndex, SimpleDirectoryReader
from llama_index import set_global_service_context
```

Now, set the Open AI API key.

```
import os
os.environ['OPENAI_API_KEY'] = "YOUR API KEY"
```

Load Documents

As we know, LLMs do not possess updated knowledge of the world nor knowledge about your internal documents. To help LLMs, we need to feed them with relevant information from knowledge sources. These knowledge sources can be structured data such as CSV, Spreadsheets, or SQL tables, unstructured data such as texts, Word Docs, Google Docs, PDFs, or PPTs, and semi-structured data such as Notion, Slack, Salesforce, etc.

In this article, we will use PDFs. Llama Index includes a class SimpleDirectoryReader, which can read saved documents from a specified directory. It automatically selects a parser based on file extension.

```
documents = SimpleDirectoryReader(input_dir='data').load_data()
```

You can have your custom implementation of a PDF reader using packages like PyMuPDF or PyPDF2.

Creating Text Chunks

Often, the data extracted from knowledge sources are lengthy, exceeding the context window of LLMs. If we send texts longer than the context window, the Chatgpt API will shrink the data, leaving out crucial information. One way to solve this is text chunking. In text chunking, longer texts are divided into smaller chunks based on separators.

Text chunking has other benefits besides making it possible to fit texts into a large language model's context window.

- Smaller text chunks result in better embedding accuracy, subsequently improving retrieval accuracy.
- Precise context: Narrowing down information will help in getting better information.

The Llama index has built-in tools for chunking texts. So, this is how we can do it.

```
text_splitter = TokenTextSplitter(
  separator=" ",
  chunk_size=1024,
  chunk_overlap=20,
  backup_separators=["\n"],
  tokenizer=tiktoken.encoding_for_model("gpt-3.5-turbo").encode

)

node_parser = SimpleNodeParser.from_defaults(
  text_splitter = TokenTextSplitter )
)
```

SimpleNodeParser creates nodes out of text chunks, and the text chunks are created using Llama Index's TokenTextSplitter. We can use a SentenceSplitter as well.

```
    chunk_overlap=20,
    paragraph_separator="\n\n\n",
    secondary_chunking_regex="[^,.;。]+[,.;。]?",
    tokenizer=tiktoken.encoding_for_model("gpt-3.5-turbo").encode

)
```

Building Knowledge Bases

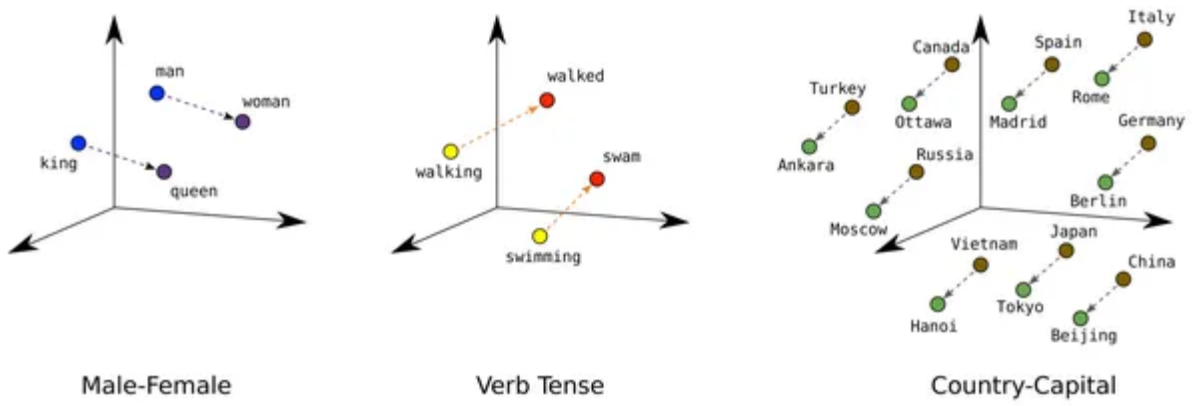
The texts extracted from the knowledge sources need to be stored somewhere. But in RAG-based applications, we need the embeddings of the data. These embeddings are floating point numbers representing data in a high-dimensional vector space. To store and operate on them, we need vector databases. Vector Databases are purpose-built data stores for storing and querying vectors.

In this section, we will understand embeddings and vector databases and implement them using the Llama Index for our RAG pipeline.

Embeddings

We can understand embeddings from a simple supermarket example. In a supermarket, you will always find apples and oranges in the same corner. To find a soap, you will have to move farther from the fruits section towards the daily apparel section, but you will easily find perfumes in the same section within a few step's distance.

This is how embeddings work. Two semantically related texts will be in proximity in the vector space, while dissimilar texts are far away. Embeddings have an extraordinary ability to map analogies between different data points. Here is a simple illustration of the same.



So, why do we need embeddings?

Embeddings generated from capable deep-learning models can efficiently capture the semantic meaning of text chunks. When a user sends a text query, we convert it to embeddings using the same model, compare the distances of the text embeddings stored in the vector database, and retrieve the closest "n" text chunks. These chunks are the most semantically similar chunks to the queried text.

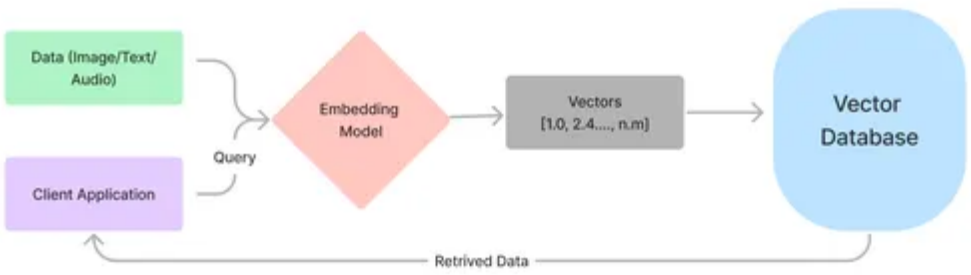
For embedding models, we need not do anything special. Llama Index has a custom implementation of popular embedding models, such as OpeanAI's Ada, Cohere, Sentence transformers, etc.

To customize the embedding model, we need to use ServiceContext and PromptHelper.

```
prompt_helper = PromptHelper(
  context_window=4096,
  num_output=256,
  chunk_overlap_ratio=0.1,
  chunk_size_limit=None
)
service_context = ServiceContext.from_defaults(
  llm=llm,
  embed_model=embed_model,
  node_parser=node_parser,
  prompt_helper=prompt_helper
)
```

Vector Database

Vector Databases are purpose-built for storing and organizing embeddings and associated metadata to provide maximum querying efficiency. They provide semantic retrieval of data, which helps augment LLMs with new information. This is how it works.



Now, you must be asking, Why can't we use the traditional databases? Technically, we can use a database like SQLite or MySQL to store vectors and linearly compare the embeddings of the query text with all others. But the problem is, you might have guessed, the linear search with O(n) time complexity. While a GPU-augmented machine can handle a few thousand data points perfectly fine, it will fail miserably when processing hundreds of millions of embeddings in any real-world application.

So, how do we solve this? The answer is indexing embeddings using different ANN algorithms such as HNSW. The HNSW is a graph-based algorithm that can efficiently handle billions of embeddings. The average query complexity of HNSW is O(log n).

Apart from HNSW, there are a few other indexing techniques, such as Product quantization, Scalar quantization, and Inverted file indexing. However, HNSW is used as the default indexing algorithm for most of the vector databases.

Weaviate, Qdrant, Milvus, etc.

```
index = VectorStoreIndex.from_documents(
    documents,
    service_context = service_context
    )
```

An index is created using the documents from our directory and the defaults from the service context we defined earlier.

Query Index

The final step is to query from the index and get a response from the LLM. Llama Index provides a query engine for querying and a chat engine for a chat-like conversation. The difference between the two is the chat engine preserves the history of the conversation, and the query engine does not.

```
query_engine = index.as_query_engine(service_context=service_context)
response = query_engine.query("What is HNSW?")
print(response)
```

```
HNSW stands for Hierarchical Navigable Small World. It is a new approach for approximate K-nearest neighbor search based on navigable small world graphs with controllable hierarchy. HNSW
incrementally builds a multi-layer structure consisting of hierarchical sets of proximity graphs for nested subsets of the stored elements. It is a fully graph-based solution that does not
require additional search structures typically used in other proximity graph techniques. HNSW has been shown to outperform previous state-of-the-art approaches in terms of search performance
and scalability.
```

GitHub repository for images and Code: [sunilkumardash9/llama_rag]

Full code:

```
from llama_index.node_parser import SimpleNodeParser
import tiktoken
llm = OpenAI(model='gpt-3.5-turbo', temperature=0, max_tokens=256)
embed_model = OpenAIEmbedding()
text_splitter = TokenTextSplitter(
  separator=" ",
  chunk_size=1024,
  chunk_overlap=20,
  backup_separators=["\n"],
  tokenizer=tiktoken.encoding_for_model("gpt-3.5-turbo").encode
)
node_parser = SimpleNodeParser.from_defaults(
  text_splitter=text_splitter
)
prompt_helper = PromptHelper(
  context_window=4096,
  num_output=256,
  chunk_overlap_ratio=0.1,
  chunk_size_limit=None
)

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

documents = SimpleDirectoryReader(input_dir='data').load_data()
index = VectorStoreIndex.from_documents(
    documents,
    service_context = service_context
    )
index.storage_context.persist()

query_engine = index.as_query_engine(service_context=service_context)
response = query_engine.query("What is HNSW?")
print(response)
```

Real-Life Use Cases

A RAG-based application can be helpful in many real-life use cases.

1. **Academic Research**: Researchers often deal with numerous research papers and articles in PDF format. A RAG pipeline could help them extract relevant information, create bibliographies, and organize their references efficiently.
2. **Law Firms**: Law firms often deal with numerous legal documents. A RAG-enabled Q&A chatbot can streamline the document retrieval process. This will save a lot of time from wasting.
3. **Educational Institutions**: Teachers and educators can extract content from educational resources to create customized learning materials or to prepare course content. Students can remove applicable information from large PDFs within a fraction of the time.
4. **Administration**: Government and Private administrative departments often deal with large amounts of documents, applications, and reports. Employing a RAG chatbot can streamline mundane document retrieval processes.

Conclusion

As we have seen, RAG streamlines retrieval generation. It helps querying the proper context out of heaps of documents within a fraction of the time. Prompting with the correct context of the query is what prompts the LLMs to generate answers. The additional contexts essentially ground the LLM to keep the answers to the context only. This prohibits the LLM from hallucinating while keeping its superior phrasing and writing ability.

Key Takeaways

- The ideal way to make LLMs learn about personal documents and reduce hallucinations is to augment the LLMs with retrieved documents from knowledge bases.
- RAG stands for Retrieval Augmented Generation. RAG is used to augment LLM with information from knowledge bases from custom documents.
- Embeddings are numerical representations of text data in high-dimensional vector space. Embeddings capture the semantic meaning of texts.
- Users employ Vector Databases as knowledge bases, utilizing various indexing algorithms to organize high-dimensional vectors, enabling fast and robust querying ability.
- Llama Index provides in-built tools and methods to build production-grade RAG-based applications.

Frequently Asked Questions

Q1. What is the Lama Index?

Ans. Lama Index is a Python framework that provides the essential tools to augment your LLM applications with external data.

Q2. What are Knowledge bases?

Ans. A knowledge base is a database that stores information, including embeddings and their metadata, from different sources.

Q3. What is the Llama Index used for?

Ans. Llama Index is an open-source framework for building LLM-based applications. It provides data ingestion tools, indexing tools, and a query interface to build production-grade RAG applications.

Q4. What is a RAG pipeline?

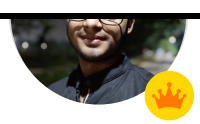
Ans. A RAG pipeline retrieves documents from external data stores, processes them to store them in a knowledge base, and provides tools to query them.

Q5. What is the difference between the Langchain and Llama Index?

Ans. Llama Index explicitly designs search and retrieval applications, while Langchain offers flexibility for creating custom AI agents.

The media shown in this article is not owned by Analytics Vidhya and is used at the Author's discretion.

blogathon language models Llama Index LLM RAG pipeline



Sunil Kumar Dash

Our Top Authors

view more

Download

Analytics Vidhya App for the Latest blog/Article

Next Post

Top 10 Online Data Science Courses in USA

Top Resources

10 Best AI Image Generator Tools to Use in 2023

Nitika Sharma - AUG 17, 2023