A PRACTICAL APPROACH
TO

Retrieval Augmented Generation

Mehdi Allahyari | Angelina Yang

Table of contents

List of Figures

Preface

In the ever-evolving landscape of artificial intelligence, Retrieval Augmented Generation (RAG) systems have emerged as a powerful and versatile tool. These systems, which blend the strengths of information retrieval and natural language generation, have the potential to revolutionize how we access, understand, and interact with vast amounts of data. As we embark on this journey through the pages of "**A Practical Approach to Retrieval Augmented Generation Systems**", we delve into the core principles, strategies, and techniques that underpin the development and implementation of these systems.

This book is a testament to the culmination of knowledge, insights, and experiences from both experts and enthusiasts in the field of RAG. We will explore the intricacies of combining retrieval and generation techniques, understand the nuances of handling diverse data sources, and witness the transformative impact of query routing, memory, and observability tools.

At the heart of our exploration, we'll leverage a popular real-world use case of "*chat with your PDF document*" to learn about RAG. Whether you are a seasoned AI practitioner or a curious learner, the practical guidance, and hands-on examples within these chapters aim to empower you to harness the full potential of RAG systems.

Together, we embark on a journey to master the art of Retrieval-Augmented Generation, discover innovative ways to solve complex problems, and ultimately transform the way we interact with AI systems. Join us in this exploration of practical RAG applications, and let's explore a transformative adventure into the heart of AI's future.

1 Introduction

In this chapter, we will lay the foundation for building a chat-to-PDF app using Large Language Models (LLMs) with a focus on the Retrieval-Augmented Generation approach. We'll explore the fundamental concepts and technologies that underpin this project.

1.1 The Role of LLMs in NLP

Large Language Models (LLMs) play a crucial role in Natural Language Processing (NLP). These models have revolutionized the field of NLP by their ability to understand and generate human-like text. With advances in deep learning and neural networks, LLMs have become valuable assets in various NLP tasks, including language translation, text summarization, and chatbot development.

One of the key strengths of LLMs lies in their capacity to learn from vast amounts of text data. By training on massive datasets, LLMs can capture complex linguistic patterns and generate coherent and contextually appropriate responses. This enables them to produce high-quality outputs that are indistinguishable from human-generated text.

LLMs are trained using a two-step process: pre-training and fine-tuning. During pre-training, models are exposed to a large corpus of text data and learn to predict the next word in a sentence. This helps them develop a strong understanding of language structure and semantics. In the fine-tuning phase, the models are further trained on task-specific data to adapt their knowledge to specific domains or tasks.

The versatility and effectiveness of LLMs make them a powerful tool in advancing the field of NLP. They have not only improved the performance of existing NLP systems but have also opened up new possibilities for developing innovative applications. With continued research and development, LLMs are expected to further push the boundaries of what is possible in natural language understanding and generation.

Large Language Models (LLMs) represent a breakthrough in NLP, allowing machines to understand and generate human-like text at an unprecedented level of accuracy and fluency. Some of the key roles of LLMs in NLP include:

1. **Natural Language Understanding (NLU):** LLMs can comprehend the nuances of human language, making them adept at tasks such as sentiment analysis, entity recognition, and language translation.
2. **Text Generation:** LLMs excel at generating coherent and contextually relevant text. This capability is invaluable for content generation, chatbots, and automated writing.
3. **Question Answering:** LLMs are particularly powerful in question answering tasks. They can read a given text and provide accurate answers to questions posed in natural language.
4. **Summarization:** LLMs can summarize lengthy documents or articles, distilling the most important information into a concise form.
5. **Conversational AI:** They serve as the backbone of conversational AI systems, enabling chatbots and virtual assistants to engage in meaningful and context-aware conversations.
6. **Information Retrieval:** LLMs can be used to retrieve relevant information from vast corpora of text, which is crucial for applications like search engines and document retrieval.
7. **Customization:** LLMs can be fine-tuned for specific tasks or domains, making them adaptable to a wide range of applications.

1.2 The Importance of Question Answering over PDFs

Question answering over PDF documents addresses a critical need in information retrieval and document processing. Here, we'll explore why it is important and how LLMs can play a pivotal role:

The Importance of Question Answering over PDFs:

1. **Document Accessibility:** PDF is a widely used format for storing and sharing documents. However, extracting information from PDFs, especially in response to specific questions, can be challenging for users. Question answering over PDFs enhances document accessibility.
2. **Efficient Information Retrieval:** For researchers, students, and professionals, finding answers within lengthy PDF documents can be time-consuming. Question-answering systems streamline this process, enabling users to quickly locate the information they need.
3. **Enhanced User Experience:** In various domains, including legal, medical, and educational, users often need precise answers from PDF documents. Implementing question answering improves the user experience by providing direct and accurate responses.
4. **Automation and Productivity:** By automating the process of extracting answers from PDFs, organizations can save time and resources. This automation can be particularly beneficial in scenarios where large volumes of documents need to be processed.
5. **Scalability:** As the volume of digital documents continues to grow, scalable solutions for question answering over PDFs become increasingly important. LLMs can handle large datasets and diverse document types.

In various industries, there is a growing demand for efficient information retrieval from extensive collections of PDF documents. Take, for example, a legal firm or department collaborating with the Federal Trade Commission (FTC) to process updated information about legal cases and proceedings. Their task

often involves processing a substantial volume of documents, sifting through them, and extracting relevant case information—a labor-intensive process.

> *Bachground: Every year the FTC brings hundreds of cases against individuals and companies for violating consumer protection and competition laws that the agency enforces. These cases can involve fraud, scams, identity theft, false advertising, privacy violations, anti-competitive behavior and more.*

The advent of the Retrieval-Augmented Generation (RAG) approach marks a new era in question and answering that promises to revolutionize workflows within these industries.

1.3 The Retrieval-Augmented Generation Approach

The Retrieval-Augmented Generation approach is a cutting-edge technique that combines the strengths of information retrieval and text generation. Let's explore this approach in detail:

The Retrieval-Augmented Generation Approach:

The Retrieval-Augmented Generation approach combines two fundamental components, retrieval and generation, to create a powerful system for question answering and content generation. Here's an overview of this approach:

1. **Retrieval Component:** This part of the system is responsible for searching and retrieving relevant information from a database of documents. It uses techniques such as indexing, ranking, and query expansion to find the most pertinent documents.
2. **Generation Component:** Once the relevant documents are retrieved, the generation component takes over. It uses LLMs to process the retrieved information and generate coherent and contextually accurate responses to user queries.

3. **Benefits:** The key advantage of this approach is its ability to provide answers based on existing knowledge (retrieval) while also generating contextually rich responses (generation). It combines the strengths of both worlds to deliver high-quality answers.
4. **Use Cases:** Retrieval-Augmented Generation is particularly useful for question answering over large document collections, where traditional search engines may fall short in providing concise and informative answers.
5. **Fine-Tuning:** Successful implementation of this approach often involves fine-tuning LLMs on domain-specific data to improve the quality of generated responses.

By understanding the role of LLMs in NLP, the importance of question answering over PDFs, and the principles behind the Retrieval-Augmented Generation approach, you have now laid the groundwork for building your chat-to-PDF app using these advanced technologies. In the following chapters, we will delve deeper into the technical aspects and practical implementation of this innovative solution.

1.4 A Brief History of LLMs

Lately, ChatGPT, as well as DALL-E-2 and Codex, have been getting a lot of attention. This has sparked curiosity in many who want to know more about what's behind their impressive performance. ChatGPT and other Generative AI (GAI) technologies fall into a category called Artificial Intelligence Generated Content (AIGC). This means they're all about using AI models to create content like images, music, and written language. The whole idea behind AIGC is to make creating conetent faster and easier.

> *AIGC is achieved by extracting and understanding intent information from instructions provided by human, and generating the content according to its knowledge and the intent information. In recent years, large-scale models have become increasingly important in AIGC as*

> *they provide better intent extraction and thus, improved generation results.*

With more data and bigger models, these AI systems can make things that look and sound quite realistic and high-quality. The following shows an example of text prompting that generates images according to the instructions, leveraging the OpenAI DALL-E-2 model.

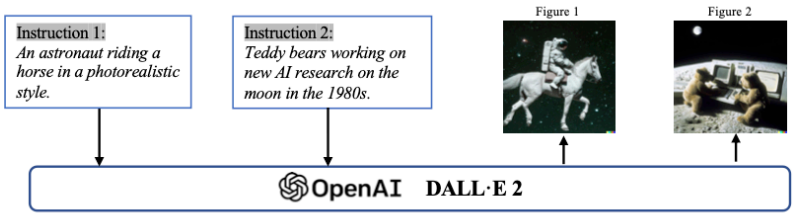


Figure 1.1: Examples of AIGC in image generation. Image source

In the realm of Generative AI (GAI), models can typically be divided into two categories: unimodal models and multimodal models. Unimodal models operate by taking instructions from the same type of data as the content they generate, while multimodal models are capable of receiving instructions from one type of data and generating content in another type. The following figure illustrates these two categories of models.

These models have found applications across diverse industries, such as art and design, marketing, and education. It's evident that in the foreseeable future, AIGC will remain a prominent and continually evolving research area with artificial intelligence.

1.4.1 Foundation Models

Speaking of LLMs and GenAI, we cannot overlook the significant role played by Transformer models.

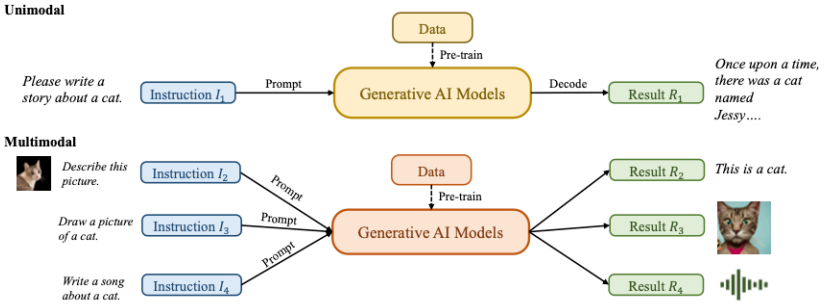


Figure 1.2: Overview of AIGC model types. Image source

> *Transformer is the backbone architecture for many state-of-the-art models, such as GPT, DALL-E, Codex, and so on.*

Transformer started out to address the limitations of traditional models like RNNs when dealing with variable-length sequences and context. The heart of the Transformer is its self-attention mechanism, allowing the model to focus on different parts of an input sequence. It comprises an encoder and a decoder. The encoder processes the input sequence to create hidden representations, while the decoder generates an output sequence. Each encoder and decoder layer includes multi-head attention and feed-forward neural networks. Multi-head attention, a key component, assigns weights to tokens based on relevance, enhancing the model's performance in various NLP tasks. The Transformer's inherent parallelizability minimizes inductive biases, making it ideal for large-scale pre-training and adaptability to different downstream tasks.

Transformer architecture has dominated natural language processing, with two main types of pre-trained language models based on training tasks: masked language modeling (e.g., BERT) and autoregressive language modeling (e.g., GPT-3). Masked language models predict masked tokens within a sentence, while autoregressive models focus on predicting the next token given previous ones, making them more suitable for generative tasks. RoBERTa and XL-Net are clas-

sic examples of masked language models and have further improved upon the BERT architecture with additional training data and techniques.

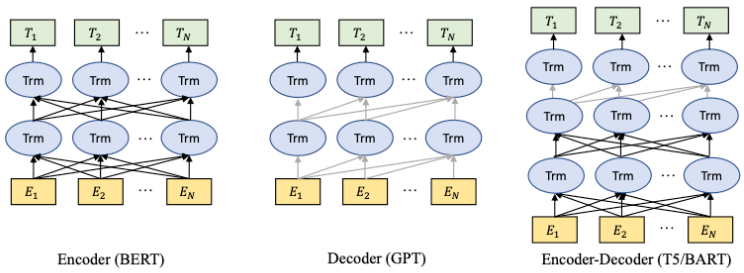


Figure 1.3: Categories of pre-trained LLMs. Image source

In this graph, you can see two types of information flow indicated by lines: the black line represents bidirectional information flow, while the gray line represents left-to-right information flow. There are three main model categories:

1. Encoder models like BERT, which are trained with context-aware objectives.
2. Decoder models like GPT, which are trained with autoregressive objectives.
3. Encoder-decoder models like T5 and BART, which merge both approaches. These models use context-aware structures as encoders and left-to-right structures as decoders.

1.4.2 Reinforcement Learning from Human Feedback

To improve AI-generated content (AIGC) alignment with user intent, i.e., considerations in *usefulness* and *truthfulness*, reinforcement learning from human feedback (RLHF) has been applied in models like Sparrow, InstructGPT, and ChatGPT.

The RLHF pipeline involves three steps: *pre-training*, *reward learning*, and *fine-tuning with reinforcement learning*. In reward learning, human feedback on diverse responses is used to create reward scalars. Fine-tuning is done through reinforcement learning with Proximal Policy Optimization (PPO), aiming to maximize the learned reward.

However, the field lacks benchmarks and resources for RL, which is seen as a challenge. But this is changing day-by day. For example, an open-source library called RL4LMs was introduced to address this gap. Claude, a dialogue agent, uses *Constitutional AI*, where the reward model is learned via RL from AI feedback. The focus is on reducing harmful outputs, with guidance from a set of principles provided by humans. See more about the topic of *Constitutional AI* in one of our blog post here.

1.4.3 GAN

Generative Adversarial Networks (GANs) are widely used for image generation. GANs consist of a generator and a discriminator. The generator creates new data, while the discriminator decides if the input is real or not.

The design of the generator and discriminator influences GAN training and performance. Various GAN variants have been developed, including LAPGAN, DCGAN, Progressive GAN, SAGAN, BigGAN, StyleGAN, and methods addressing mode collapse like D2GAN and GMAN.

The following graph illustrates some of the categories of vision generative models.

Although GAN models are not the focus of our book, they are essential in powering multi-modality applications such as the diffusion models.

1.4.4 Applications

Chatbots are probably one of the most popular applications for LLMs.

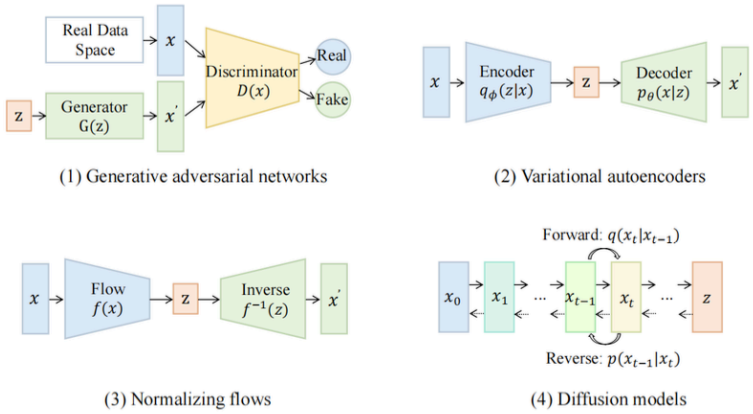


Figure 1.4: Categories of vision generative models. Image source

Chatbots are computer programs that mimic human conversation through text-based interfaces. They use language models to understand and respond to user input. Chatbots have various use cases, like customer support and answering common questions. Our "*chat with your PDF documents*" is a up-and-coming use case!

Other notable examples include Xiaoice, developed by Microsoft, which expresses empathy, and Google's Meena, an advanced chatbot. Microsoft's Bing now incorporates ChatGPT, opening up new possibilities for chatbot development.

This graph illustrates the relationships among current research areas, applications, and related companies. Research areas are denoted by dark blue circles, applications by light blue circles, and companies by green circles.

In addition, we have previously written about chatbots and now they are part of history, but still worth reviewing:

- Blogpost: What Does A Chatbot Look Like Under the Hood?

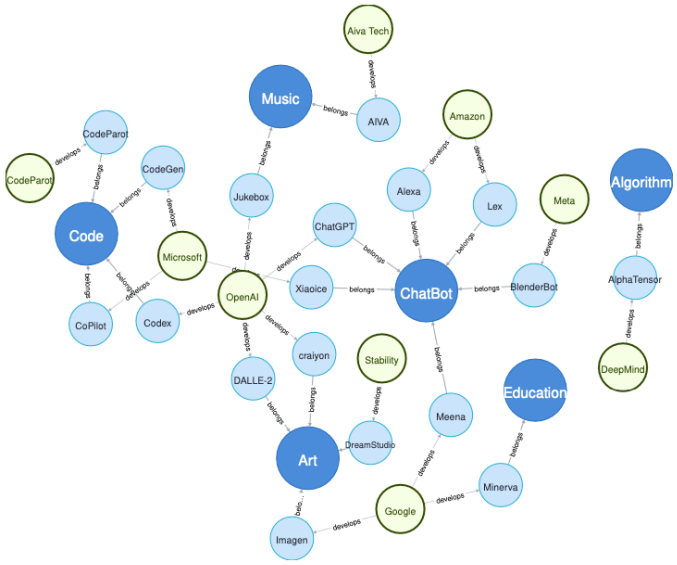


Figure 1.5: Knowlege Graph for Applications. Image source

- Blogpost: What Is Behind the Scene of A Chatbot NLU?
- Blogpost: What More Can You Do with Chatbots?

Of course, chatbots are not the only application. There are vast possibilities in arts and design, music generation, education technology, coding and beyond - your imagination doesn't need to stop here.

1.4.5 Prompt Learning

Prompt learning is a new concept in language models. Instead of predicting y given x, it aims to find a template x' that predicts $P(y|x')$.

> *Normally, prompt learning will freeze the language model and directly perform few-shot or zero- shot learning on it. This enables the language models to be pre-trained on large amount of raw text data and be adapted to new domains without tuning it again. Hence, prompt learning could help save much time and efforts.*

Traditionally, prompt learning involves prompting the model with a task, and it can be done in two stages: prompt engineering and answer engineering.

Prompt engineering: This involves creating prompts, which can be either discrete (manually designed) or continuous (added to input embeddings) to convey task-specific information.

Answer engineering: After reformulating the task, the generated answer must be mapped to the correct answer space.

Besides single-prompt, *multi-prompt methods* combine multiple prompts for better predictions, and *prompt augmentation* basically beefs up the prompt to generate better results.

Moreover, in-context learning, a subset of prompt learning, has gained popularity. It enhances model performance by incorporating a pre-trained language model and supplying input-label pairs and task-specific instructions to improve alignment with the task.

Overall, in the dynamic landscape of language models, tooling and applications, the graph below illustrates to the evolution of language model engineering. With increasing flexibility along the x-axis and rising complexity along the y-axis, this graph offers a bird's-eye view of the choices and challenges faced by developers, researchers and companies.

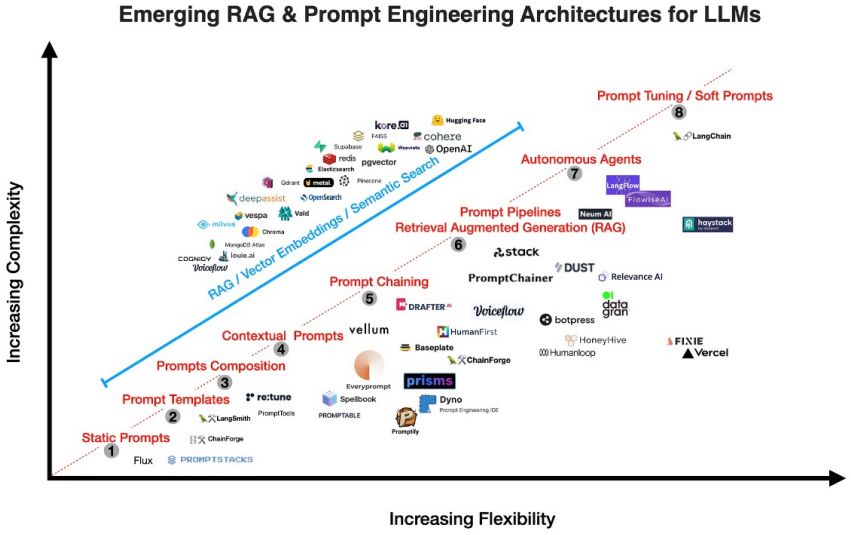


Figure 1.6: Emerging RAG & Prompt Engineering Architecture for LLMs. Image source

In the top-right corner, you can see the complex, yet powerful tools like OpenAI, Cohere, and Anthropic (to-be-added), which have pushed the boundaries of what language models can achieve. Along the diagonal, the evolution of prompt engineering is displayed, from static prompts to templates, prompt chaining, RAG pipelines, autonomous agents, and prompt tuning. On the more flexible side, options like Haystack and LangChain have excelled, presenting broader horizons for those seeking to harness the versatility of language models.

This graph serves as a snapshot of the ever-evolving landscape of toolings in the realm of language model and prompt engineering today, providing a roadmap

for those navigating the exciting possibilities and complexities of this field. It is likely going to be changing every day, reflecting the continuous innovation and dynamism in the space.

In the next Chapter we'll turn our focus to more details of Retrieval Augmented Generation (RAG) pipelines. We will break down their key components, architecture, and the key steps involved in building an efficient retrieval system.

2 Retrieval Augmented Generation (RAG)

Generative AI, a subset of artificial intelligence, has revolutionized the field of text generation. It has paved the way for machines to generate human-like text, offering a myriad of benefits in various applications. From content creation and chatbots to language translation and natural language understanding, generative AI has proven to be a powerful tool in the world of natural language processing. However, it is essential to recognize that despite its remarkable capabilities, generative AI systems have limitations, one of which is their reliance on the data they have been trained on to generate responses.

2.1 The Limitation of Generative AI

Generative AI models, such as GPT3 (Generative Pre-trained Transformer), have been trained on vast datasets containing text from the internet. While this training process equips them with a broad understanding of language and context, it also introduces limitations. These models can only generate text that aligns with the patterns and information present in their training data. As a result, their responses may not always be accurate or contextually relevant, especially when dealing with niche topics or recent developments that may not be adequately represented in their training data.

2.1.1 Example: Transforming Customer Support with RAG

Imagine you're the owner of a thriving e-commerce platform, selling a wide range of products from electronics to fashion. You've recently integrated a chatbot to assist your customers with inquiries, but you're starting to see its limitations. Let's explore how Retrieval-Augmented Generation (RAG) can help overcome these limitations and enhance the customer support experience.

Limitations of a Traditional Large Language Model (LLM)

Your existing chatbot is built around a traditional Large Language Model (LLM). While it's knowledgeable about general product information, your customers are increasingly seeking more specific and real-time assistance. Here are some challenges you've encountered:

Product Availability: Customers often inquire about the availability of specific items, especially during sales or promotions. The LLM can provide information based on its training data, but it doesn't have access to real-time inventory data.

Shipping and Delivery: Customers frequently ask about shipping times, tracking information, and potential delays. The LLM can provide standard shipping policies, but it can't offer real-time updates on the status of an individual order.

Product Reviews: Shoppers want to know about recent product reviews and ratings to make informed decisions. The LLM lacks access to the latest customer reviews and sentiment analysis.

Promotions and Discounts: Customers seek information about ongoing promotions, discounts, and special offers. The LLM can only provide details based on the data it was trained on, missing out on time-sensitive deals.

2.1.2 How RAG Transforms Customer Support

Now, let's introduce RAG into your e-commerce customer support system:

Retrieval of Real-Time Data: With RAG, your chatbot can connect to your e-commerce platform's databases and data warehouses in real-time. It can retrieve the latest information about product availability, stock levels, and shipping status.

Incorporating User Reviews: RAG can scrape and analyze customer reviews and ratings from your website, social media, and other sources. It can then generate responses that include recent reviews, helping customers make informed choices.

Dynamic Promotions: RAG can access your promotion database and provide up-to-the-minute details about ongoing discounts, flash sales, and limited-time offers. It can even suggest personalized promotions based on a user's browsing history.

Order Tracking: RAG can query your logistics system to provide customers with real-time tracking information for their orders. It can also proactively notify customers of any delays or issues.

2.2 Introducing Retrieval Augmented Generation (RAG)

To address the limitation of generative AI, researchers and engineers have developed innovative approaches, one of which is the Retrieval Augmented Generation (RAG) approach. RAG initially caught the interest of generative AI developers following the release of a seminal paper titled "*Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks*" (Lewis et al. 2020) at Facebook AI Research. RAG combines the strengths of generative AI with retrieval techniques to enhance the quality and relevance of generated text. Unlike traditional

generative models that rely solely on their internal knowledge, RAG incorporates an additional step where it retrieves information from external sources, such as databases, documents, or the web, before generating a response. This integration of retrieval mechanisms empowers RAG to access up-to-date information and context, making it particularly valuable for applications where accurate and current information is critical.

In this chapter, we will delve deeper into the Retrieval Augmented Generation (RAG) approach, exploring its architecture, advantages, and real-world applications. By doing so, we will gain a better understanding of how RAG represents a significant step forward in improving the capabilities of generative AI and overcoming the limitations posed by reliance on static training data. Understanding the key concepts and components of this approach is essential for building an effective chat-to-PDF app.

2.2.1 Key Concepts and Components

To grasp the essence of Retrieval-Augmented Generation, let's explore its key concepts and components:

1. **Retrieval Component:** The retrieval component is responsible for searching and selecting relevant information from a database or corpus of documents. This component utilizes techniques like document indexing, query expansion, and ranking to identify the most suitable documents based on the user's query.
2. **Generation Component:** Once the relevant documents are retrieved, the generation component takes over. It leverages Large Language Models (LLMs) such as GPT-3 to process the retrieved information and generate coherent and contextually accurate responses. This component is responsible for converting retrieved facts into human-readable answers.
3. **Interaction Loop:** Retrieval-Augmented Generation often involves an interaction loop between the retrieval and generation components. The

initial retrieval may not always return the perfect answer, so the generation component can refine and enhance the response iteratively by referring back to the retrieval results.

4. **Fine-Tuning:** Successful implementation of this approach often requires fine-tuning LLMs on domain-specific data. Fine-tuning adapts the model to understand and generate content relevant to the specific knowledge domain, improving the quality of responses.
5. **Latent Space Representations:** Retrieval models often convert documents and queries into latent space representations, making it easier to compare and rank documents based on their relevance to a query. These representations are crucial for efficient retrieval.
6. **Attention Mechanisms:** Both the retrieval and generation components typically employ attention mechanisms. Attention mechanisms help the model focus on the most relevant parts of the input documents and queries, improving the accuracy of responses.

2.2.2 How It Improves Question Answering

The Retrieval-Augmented Generation approach offers several advantages for question answering:

1. **Access to a Wide Knowledge Base:** By integrating retrieval, the system can access a vast knowledge base, including large document collections. This enables the model to provide answers that may not be present in its pre-training data, making it highly informative.
2. **Contextual Understanding:** The generation component uses the context provided by the retrieval results to generate answers that are not only factually accurate but also contextually relevant. This contextual understanding leads to more coherent and precise responses.
3. **Iterative Refinement:** The interaction loop between retrieval and generation allows the system to iteratively refine its responses. If the initial response is incomplete or incorrect, the generation component can make

further inquiries or clarifications based on the retrieval results, leading to improved answers.

4. **Adaptability to Diverse Queries:** Retrieval-Augmented Generation can handle a wide range of user queries, including complex and multi-faceted questions. It excels in scenarios where simple keyword-based search engines may fall short.
5. **Fine-Tuning for Specific Domains:** By fine-tuning the model on domain-specific data, you can tailor it to excel in particular knowledge domains. This makes it a valuable tool for specialized question answering tasks, such as legal or medical consultations.

In summary, Retrieval-Augmented Generation is a dynamic approach that combines the strengths of retrieval and generation to provide accurate, contextually relevant, and informative answers to user queries. Understanding its key components and advantages is essential as we move forward in building our chat-to-PDF app, which will leverage this approach to enhance question answering over PDF documents.

2.3 RAG Architecture

At its core, RAG is a framework that synergizes two vital components:

Retrieval Model: This component specializes in searching and retrieving relevant information from extensive datasets, such as documents, articles, or databases. It identifies passages or documents that contain information related to a user's query.

Generation Model: On the other hand, the generation model excels in crafting coherent and contextually rich responses to user queries. It's often based on large language models (LLMs) like GPT-3, which can generate human-like text. Figure 2.1 shows the RAG architecture.

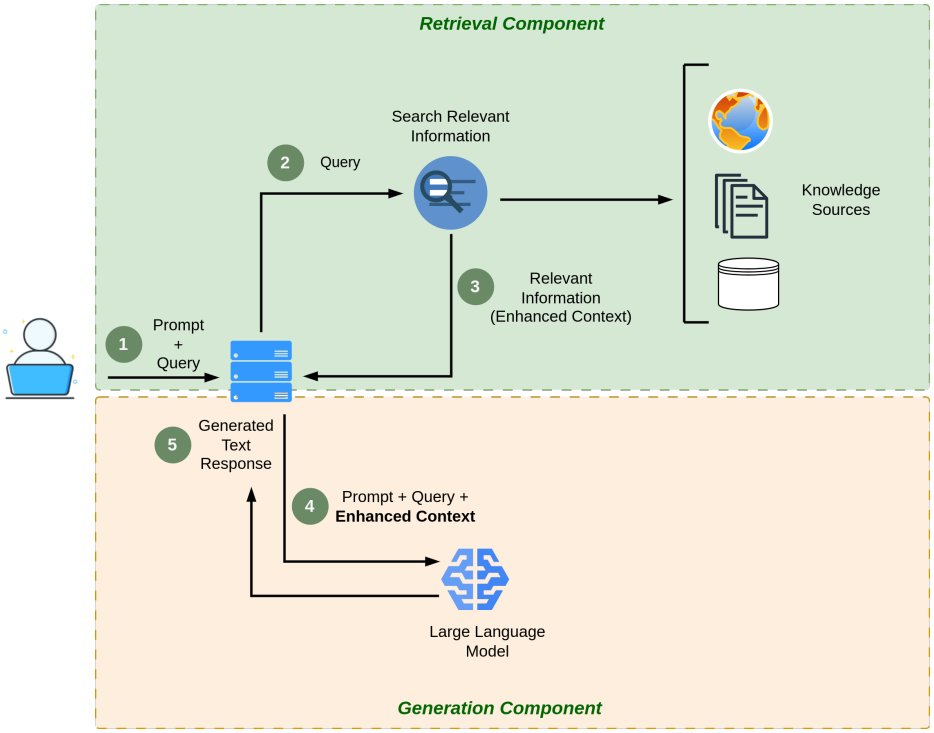


Figure 2.1: RAG architechture

2.4 Building the Retrieval System

In this section, we will focus on building the retrieval system, a critical component of the chat-to-PDF app that enables the extraction of relevant information from PDF documents. This section is essential for implementing the Retrieval-Augmented Generation approach effectively.

2.4.1 Choosing a Retrieval Model

Choosing the right retrieval model is a crucial decision when building your chat-to-PDF app. Retrieval models determine how efficiently and accurately the system can find and rank relevant documents in response to user queries. Here are some considerations when selecting a retrieval model:

- **TF-IDF (Term Frequency-Inverse Document Frequency):** TF-IDF is a classical retrieval model that calculates the importance of terms in a document relative to a corpus. It's simple to implement and effective for certain tasks.
- **BM25:** BM25 is an improved version of TF-IDF that accounts for document length and term saturation. It's often more effective in modern retrieval tasks.
- **Vector Space Models:** These models represent documents and queries as vectors in a high-dimensional space. Cosine similarity or other distance metrics are used to rank documents. Implementations like Latent Semantic Analysis (LSA) and Word Embeddings (e.g., Word2Vec) can be used.
- **Neural Ranking Models:** Modern neural models, such as BERT-based models, are increasingly popular for retrieval tasks due to their ability to capture complex semantic relationships. They can be fine-tuned for specific tasks and domains.
- **Hybrid Models:** Combining multiple retrieval models, such as a combination of TF-IDF and neural models, can offer the benefits of both approaches.

- **Domain and Data Size:** Consider the specific requirements of your chat-to-PDF app. Some retrieval models may be more suitable for small or specialized document collections, while others excel in handling large, diverse corpora.
- **Scalability:** Ensure that the chosen retrieval model can scale to meet the needs of your application, especially if you anticipate handling a substantial volume of PDF documents.

2.5 Embeddings and Vector Databases for Retrieval in RAG

In addition to selecting an appropriate retrieval model, leveraging embeddings and vector databases can significantly enhance the performance and efficiency of the retrieval component within your chat-to-PDF app. Vector embeddings are a fundamental concept in modern information retrieval and natural language processing. They transform textual data into numerical vectors, enabling computers to understand and manipulate text data in a mathematical, geometric space. These embeddings capture semantic and contextual relationships between words, documents, or other textual entities, making them highly valuable in various applications, including the retrieval component of Retrieval Augmented Generation (RAG).

2.5.1 Vector Embeddings: An Overview

Vector embeddings represent words, phrases, sentences, or even entire documents as points in a high-dimensional vector space. The key idea is to map each textual element into a vector in such a way that semantically similar elements are located close to each other in this space, while dissimilar elements are further apart. This geometric representation facilitates similarity calculations, clustering, and other operations.

Examples of Vector Embeddings:

1. **Word Embeddings (Word2Vec, GloVe):** Word embeddings represent individual words as vectors. For example, "king" and "queen" may be represented as vectors that are close together in the vector space because they share similar semantic properties.
2. **Document Embeddings (Doc2Vec, BERT):** Document embeddings map entire documents (such as PDFs) into vectors. Two documents discussing similar topics will have embeddings that are close in the vector space.

There are abundent of tutorials and resources that can help you learn more about vector embeddings. Here are some resources that can help you get started:

- Vector Embeddings Explained
- Google Vector embeddings
- What are vector embeddings

2.5.2 Vector Databases and Their Role in Enhancing Retrieval

Vector databases, also known as similarity search engines or vector index databases, play a crucial role in the retrieval component of RAG by efficiently storing and retrieving these vector embeddings. They are specialized databases designed for retrieving vectors based on similarity, making them well-suited for scenarios where similarity between data points needs to be calculated quickly and accurately.

How Vector Databases Enhance Retrieval in RAG:

1. **Fast Retrieval:** Vector databases employ indexing structures optimized for similarity search. They use algorithms like approximate nearest neighbor (ANN) search to quickly locate the most similar vectors, even in large datasets containing numerous documents.

2. **Scalability:** Vector databases can efficiently scale as the corpus of documents grows. This ensures that retrieval performance remains consistent, regardless of the dataset's size.
3. **Advanced Similarity Scoring:** These databases offer a range of similarity metrics, such as cosine similarity or Jaccard index, allowing you to fine-tune the relevance ranking of retrieved documents based on your specific requirements.
4. **Integration with Retrieval Models:** Vector databases can be seamlessly integrated into your retrieval system. They complement retrieval models like TF-IDF, BM25, or neural ranking models by providing an efficient means of candidate document selection based on vector similarity.

All of these factors has resulted in numerous new vector databases. Selecting and depending on one of these databases can have long-lasting consequences and dependencies within your system. Ideally, we opt for a vector database that exhibits strong scalability, all while maintaining cost-efficiency and minimizing latency. Some of these vector databases are: Qdrant, Weaviate, Pinecone, pgvector, Milvus, and Chroma.

Product	Free tier	queries per second (approximate recall of 0.9) From ANN Benchmarks on glove-100-angular	Cost at Scale per Month (Information and Not Comparison)	Is in Current Database	Self-Host	Managed in Cloud	SOC-2	HIPAA	GDPR	Open Source	BM25	Aggregations	Notes	@	Metadata Filtering	Aggregations	Notes
Weaviate	Yes	517	(Vector size 1,536, 10M vectors, 1,000,000 queries/month) - Standard: **$770/month** (High	No	Yes	Yes	Can be (Depending on hosting)	Can be (Depending on hosting)		Yes	Yes	Yes			Yes	No	
Pinecone	Yes	From Pinecone website (queries per second for 1M vectors of size 768; top_10): - s1 pod: 10 - p1 pod: 30 - p2 pod: 150	For 10M vectors of size 1,536 on AWS and one replica: - s1 pod (4 pods needed): - standard: **$320/month**; enterprise: **$480/month** - p1 pod (20 pods needed): - standard: **$1,599/month;** enterprise: **$2,398/month** - p2 pod (19 pods needed): - standard: **$2,278/month**; enterprise: **$3,417/month**	No	No	Yes	Yes	No		No	No	No			Yes	Yes	
pgvector on PostgreSQL	Depending on hosting	84 (Supabase article claims around 470 with improvements)	(Each vector takes 4 * dimensions + 8 bytes of storage) 10M vectors of size 1,536 occupy around 62 GB. Assuming using Azure PostgreSQL Flexible Server: - Burstable: - 1 vCore: **$19.77/month** - 2 vCore: **$107/month** - General purpose: - 2 vCore: **$137/month** - 4 vCore: **$267/month** - Memory optimized: - 2 vCore: **$190/month** - 4 vCore: **$372/month**	Yes	Yes	Yes (various vendors)	Can be (Depending on hosting)	Can be (Depending on hosting)		Yes	Yes	Yes	- Improvements to pgvector can improve pgvector's requests per second		Yes	No	https://github.com/milvus-io/milvus
Milvus	Yes	1,751	- Capacity-oprimized on Zilliz cloud: **$450/month** - Cost optimized: **$300/month** - Performance optimized: **$1,375/month**	No	Yes	Yes	Yes	? (Depending on Hosting?)		Yes	No	No			Yes	No	
MongoDB	Yes	? (There are available data on traditional MongoDB, but could not find about the vector part)	- MongoDB Atlas; Dedicated with 80 GB storage on AWS costs around **$750/month**	Yes	Yes	Yes	Yes	Yes		Yes	Yes	Yes	- The vector search is new and not much tested yet				
Qdrant	Self-hosted is free	300 by ANN (Around 350 by NirantK)	- Storage optimized: **$280/month** - Not storage optimized: **$820/month**	No	Yes	Yes	? (Depending on Hosting?)	? (Depending on Hosting?)		Yes	No	No					
ChromaDB	In memory of server	?	Hosted product is under development	No	Yes	Not Yet	? (Depending on Hosting?)	? (Depending on Hosting?)		Yes	No	No					
Elasticsearch	Self-hosted is free			Yes	Yes	Yes	Yes	Yes		No	Yes	Yes	kNN Documentation				
Hyperspace	Yes	?	?	No	No	Yes	Yes	?		No	Yes	Yes	website				
Retake	Yes	300 (Lucene kNN)	Hosted product is under development	Yes	Yes	Not Yet	Can be (Depending on hosting)	Can be (Depending on hosting)		Yes	Yes	Yes	Github Repo				
DataStax Astra/Apache Cassandra	Yes			Yes	Yes	Yes	Yes	Yes	Yes	Yes							

Figure 2.2: Vector databases comparison. Image source

2.6 RAG Data Ingestion Pipeline

Before your Chat-to-PDF app can effectively retrieve information from a vector database, it's imperative to preprocess the PDF documents and create a structured and searchable index for the preprocessed data. This searchable index serves as the cornerstone of your application, akin to a meticulously organized library catalog. It empowers your system to swiftly and accurately locate relevant information within PDF documents, enhancing the efficiency and precision of the retrieval process.

Figure 2.3 illustrates the RAG data ingestion pipeline. in Chapter 3, we will fully discuss how to prepare, index, and store the documents in a vector database.

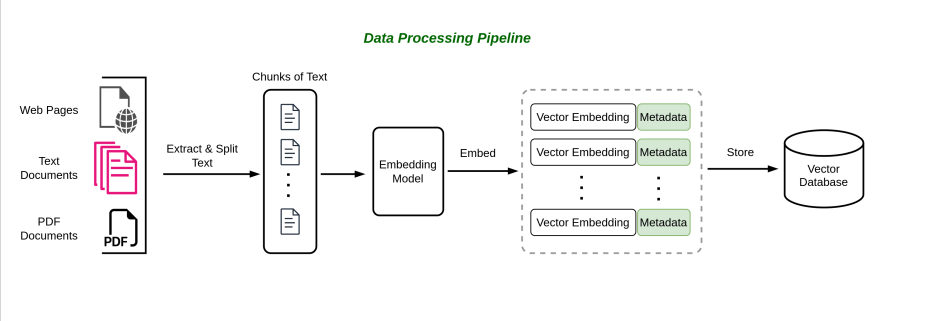


Figure 2.3: RAG data ingestion pipeline

2.7 Challenges of Retrieval-Augmented Generation

The adoption of Retrieval-Augmented Generation (RAG) represents a significant advancement in natural language processing and information retrieval. However, like any complex AI system, RAG presents a set of challenges that must be addressed to fully harness its potential. In this section, we explore some of the key challenges associated with RAG.

2.7.1 Data Quality and Relevance

RAG heavily relies on the availability of high-quality and relevant data for both retrieval and generation tasks. Challenges in this area include:

- **Noisy Data:** Incomplete, outdated, or inaccurate data sources can lead to retrieval of irrelevant information, impacting the quality of generated responses.
- **Bias and Fairness:** Biases present in training data may lead to biased retrieval and generation, perpetuating stereotypes or misinformation.

2.7.2 Integration Complexity

Integrating retrieval and generation components seamlessly is non-trivial, as it involves bridging different architectures and models. Challenges include:

- **Model Compatibility:** Ensuring that the retrieval and generation models work harmoniously, especially when combining traditional methods (e.g., TF-IDF) with neural models (e.g., GPT-3).
- **Latency and Efficiency:** Balancing the need for real-time responsiveness with the computational resources required for retrieval and generation.

2.7.3 Scalability

Scaling RAG systems to handle large volumes of data and user requests can be challenging:

- **Indexing Efficiency:** As the document corpus grows, maintaining an efficient and up-to-date index becomes crucial for retrieval speed.
- **Model Scaling:** Deploying large-scale neural models for both retrieval and generation may require substantial computational resources.

2.7.4 Evaluation Metrics

Evaluating the performance of RAG systems presents difficulties:

- **Lack of Gold Standards:** In some cases, there may be no clear gold standard for evaluating the relevance and quality of retrieved documents.
- **Diverse User Needs:** Users have diverse information needs, making it challenging to develop universal evaluation metrics.

2.7.5 Domain Adaptation

Adapting RAG systems to specific domains or industries can be complex:

- **Domain-Specific Knowledge:** Incorporating domain-specific knowledge and jargon into retrieval and generation.
- **Training Data Availability:** The availability of domain-specific training data for fine-tuning models.

Addressing these challenges is essential to unlock the full potential of RAG in various applications, from question answering to content generation. As research and development in this field continue, finding innovative solutions to these challenges will be critical for building robust and reliable RAG systems that deliver accurate, relevant, and trustworthy information to users.

As we conclude our exploration of the foundations and the retrieval component of Retrieval-Augmented Generation (RAG) systems in this Chapter, we now turn our attention to the practical implementation of RAG pipelines in Chapter 3. In this next chapter, we'll delve into the nitty-gritty details of how these systems come to life, starting with the preprocessing of PDF documents and the data ingestion pipeline. We'll also discuss the generation components that make RAG systems work. Further, Chapter 3 explores the impact of text splitting methods on RAG quality and the crucial role of metadata in enhancing the overall RAG experience.

3 RAG Pipeline Implementation

3.1 Preprocessing PDF documents

Before we can harness the power of Large Language Models (LLMs) and particularly RAG method for question answering over PDF documents, it's essential to prepare our data. PDFs, while a common format for documents, pose unique challenges for text extraction and analysis. In this section, we'll explore the critical steps involved in preprocessing PDF documents to make them suitable for our Chat-to-PDF app. These steps are not only essential for PDFs but are also applicable to other types of files. However, our primary focus is on PDF documents due to their prevalence in various industries and applications.

3.1.1 PDF Text Extraction

PDFs may contain a mix of text, images, tables, and other elements. To enable text-based analysis and question answering, we need to extract the textual content from PDFs. Here's how you can accomplish this:

- **Text Extraction Tools:** Explore available tools and libraries like PyPDF2, pdf2txt, or PDFMiner to extract text from PDF files programmatically.
- **Handling Scanned Documents:** If your PDFs contain scanned images instead of selectable text, you may need Optical Character Recognition (OCR) software to convert images into machine-readable text.
- **Quality Control:** Check the quality of extracted text and perform any necessary cleanup, such as removing extraneous characters or fixing formatting issues.

3.1.2 Handling Multiple Pages

PDF documents can span multiple pages, and maintaining context across pages is crucial for question answering. Here's how you can address this challenge:

- **Page Segmentation:** Segment the document into logical units, such as paragraphs or sections, to ensure that context is preserved.
- **Metadata Extraction:** Extract metadata such as document titles, authors, page numbers, and creation dates. This metadata can aid in improving searchability and answering user queries.

3.1.3 Text Cleanup and Normalization

PDFs may introduce artifacts or inconsistencies that can affect the quality of the extracted text. To ensure the accuracy of question answering, perform text cleanup and normalization:

- **Whitespace and Punctuation:** Remove or replace excessive whitespace and special characters to enhance text readability.
- **Formatting Removal:** Eliminate unnecessary formatting, such as font styles, sizes, and colors, which may not be relevant for question answering.
- **Spellchecking:** Check and correct spelling errors that might occur during the extraction process.

3.1.4 Language Detection

If your PDF documents include text in multiple languages, it is a good idea to implement language detection algorithms to identify the language used in each section. This information can be useful when selecting appropriate LLM models for question answering.

3.2 Data Ingestion Pipeline Implementation

As it has been depicted in Figure 2.3 the first step of the data ingestion pipeline is *extracting and spliting text from the pdf documents*. There are several packages for this goal including:

- PyPDF2
- pdfminer.six
- unstructured

> **Note**
>
> If you have scanned pdfs, you can utilize libraries such as **unstructured**, **pdf2image** and **pytesseract**.

Additionally, there are data loaders hub like llamahub that contains tens of data loaders for reading and connecting a wide variety data sources to a Large Language Model (LLM).

Finally, there are packages like llamaindex, and langchain. These are frameworks that faciliates developing applications powered by LLMs. Therefore, they have implemented many of these data loaders including extracting and spliting text from pdf files.

Step 1: Install necessary libraries

```
pip install llama-index
pip install langchain
```

Step 2: Load the pdf file and extract the text from it

Code below will iterate over the pages of the pdf file, extract the text and add it to the `documents` list object, see Figure 3.1.

```python
from PyPDF2 import PdfReader

PDF_PATH = 'pdf_files/ml_interview.pdf'

# Load the pdf file
reader = PdfReader(PDF_PATH)

# Get all the pages of the pdf
pages = reader.pages
documents = []

for page in pages:
    documents.append(page.extract_text())

print(documents[0])
# 41 Essential Machine \nLearning Interview \nQuestions\nwww.springboard.com\n18 mins read'
```

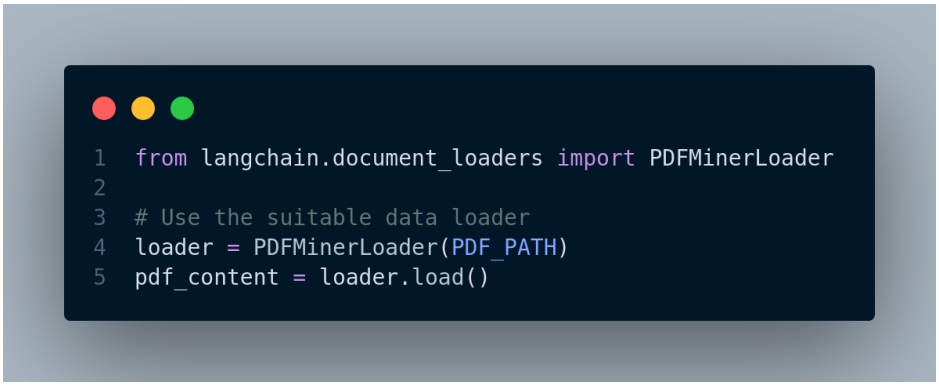
Figure 3.1: Load pdf files

Now every page has become a separate document that we can later *embed (vectorize)* and *store* in the vector database. However, some pages could be very lengthy and other ones could be very short as page length varies. This could signaficantly impact the quality of the document search and retrieval.

Additionally, LLMs have a limited context window (token limit), i.e. they can handle certain number of tokens (a token roughly equals to a word). Therefore, we instead first concatenate all the pages into a long text document and then split that document into smaller reletively equal size chunks. We then embed each chunk of text and insert into the vector database.

Nevertheless, since we are going to use `llamaindex` and `langchain` frameworks for the RAG pipeline, Let's utilize the features and functions these frameworks offer. They have data loaders and splitters that we can use to read and split pdf files. You can see the code in Figure 3.2.

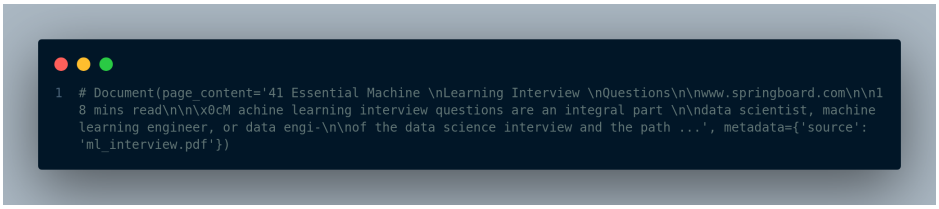
`pdf_content[0]` contains the entire content of pdf, and has s special structure. It is a `Document` object with some properties including `page_content` and `metadata`. `page_content` is the textual content and `metadata` contains some meta-



```python
from langchain.document_loaders import PDFMinerLoader

# Use the suitable data loader
loader = PDFMinerLoader(PDF_PATH)
pdf_content = loader.load()
```

Figure 3.2: Langchain data loader

data about the pdf. Here's the partial output the `Document` object of our pdf in Figure 3.3.



```
# Document(page_content='41 Essential Machine \nLearning Interview \nQuestions\n\nwww.springboard.com\n\n1
8 mins read\n\n\x0cM achine learning interview questions are an integral part \n\ndata scientist, machine
learning engineer, or data engi-\n\nof the data science interview and the path ...', metadata={'source':
'ml_interview.pdf'})
```

Figure 3.3: Langchain data loader output

A `Document` object is a generic class for storing a piece of unstructured text and its associated metadata. See here for more information.

Step 3: Split the text into smaller chunks

There are several different text splitters. For more information see langchain API, or llamaIndex documentation. Two common ones are:

i. `CharacterTextSplitter`: Split text based on a certain number characters.

ii. `TokenTextSplitter`: Split text to tokens using model tokenizer.

The following code in Figure 3.4 chunks the pdf content into sizes no greater than 1000, with a bit of overlap to allow for some continued context.

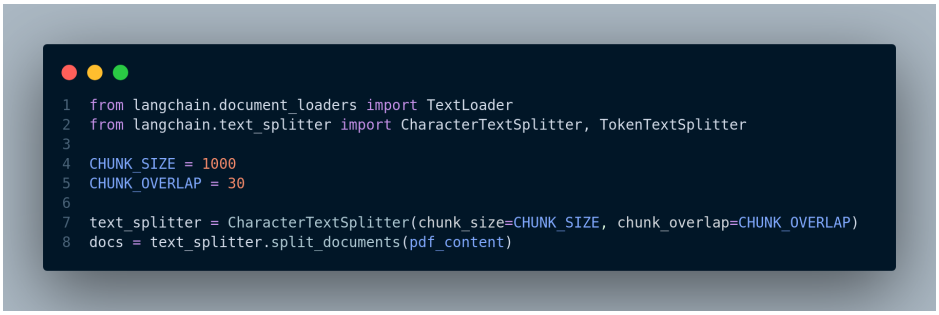


Figure 3.4: Langchain text split method

Here's the number of chunks created from splitting the pdf file.

```
# Print the number of chunks (list of Document objects)
print(len(docs))
# 30
```

Step 4: Embed and store the documents in the vector database

In this step we need to convert chunks of text into embedding vectors. There are plenty of embedding models we can use including OpenAI models, Huggingface models, and Cohere models. You can even define your own custom embedding model. Selecting an embedding model depnds on several factors:

- **Cost:** Providers such as OpenAI or Cohere charge for embeddings, albeit it's cheap, when you scale to thusands of pdf files, it will become prohibitive.
- **Latency and speed:** Hosting an embedding model on your server reduce the latency, whereas using vendors' API increases the latency.

- **Convenience:** Using your own embedding model needs more compute resource and maintainance whereas using vendors APIs like OpenAI gives you a hassle-free experience.

Similar to having several choices for embedding models, there are so many options for choosing a vector database, which is out the scope of this book.

Figure 3.5 shows some of the most popular vector database vendors and some of the features of their hosting. This blog fully examines these vector databases from different perspective.

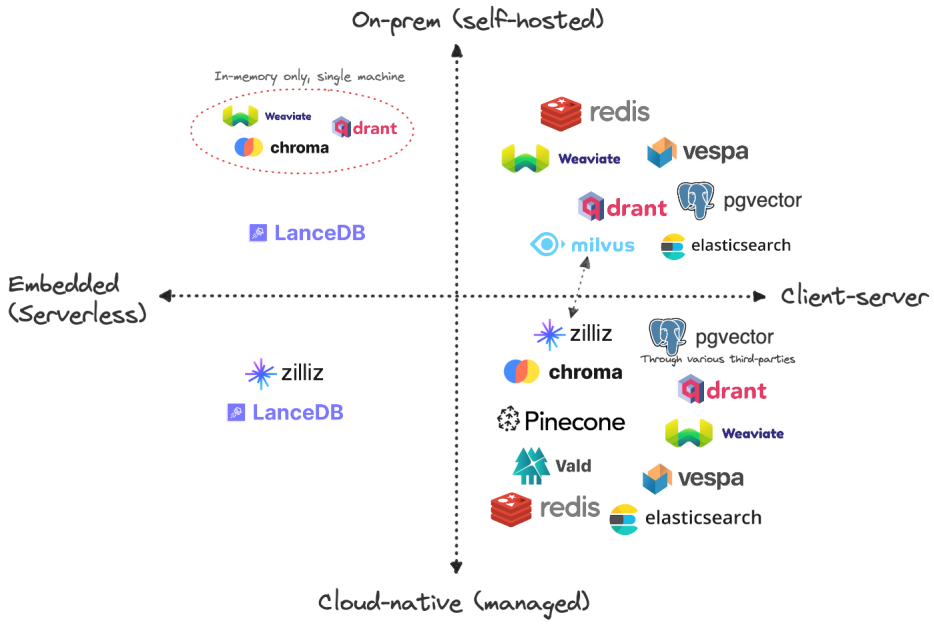


Figure 3.5: Various vector databases. Image source

We are going to use OpenAI models, particularly `text-embedding-ada-002` for embedding. Furthermore, we choose Qdrant as our vector database. It's open source, fast, very flexible, and offers a free clould-based tier.

We first install the `openai` and `qdrant` package.

```
pip install openai
pip install qdrant-client
```

We also require an API key that we can get it from here.

If we set `OPENAI_API_KEY` environment variable to our API key, we can easily call the functions that needs it without getting any error. Otherwise we can pass the API key parameter to functions requiring it. Figure 3.6 shows how to do it.

```python
# Set the environment variable
OPENAI_API_KEY = 'YOUR API KEY'
os.environ['OPENAI_API_KEY'] = OPENAI_API_KEY

from langchain.embeddings import OpenAIEmbeddings
from langchain.vectorstores import Qdrant

# initialize OpenAIEmbedding
embeddings = OpenAIEmbeddings(model='text-embedding-ada-002')

# If not set OPENAI_API_KEY environment variable use this
# embeddings = OpenAIEmbeddings(openai_api_key=OPENAI_API_KEY, model='text-embedding-ada-002')

# Create a qdrant client, embed documents and insert them into the qdrant vector database
qdrant = Qdrant.from_documents(
    docs,
    embeddings,
    path="/tmp/local_qdrant",  # Location to store the vector database Index
    collection_name="my_documents",
    force_recreate=True,  # Force to override the collection if exists
)
```

Figure 3.6: Qdrant vector database setup via Langchain

Please note that there are several different ways to achieve the same goal (embedding and storing in the vector database). You can use `Qdrant` client library directly instead of using the langchain wrapper for it. Also, you can

first create embeddings separately and then store them in the Qdrant vector database. Here, we embedded the documents and stored them all by calling `Qdrant.from_documents()`.

In addition, you can use Qdrant cloud vector database to store the embeddings and use their REST API to interact with it, unlike this example where the index is stored locally in the `/tmp/local_qdrant` directory. This approach is suitable for testing and POC (Proof-Of-Concept), not for production environment.

We can try and see how we can search and retrieve relevant documents from the vector database. For instance, let's see what the answer to the question *"what is knearest neighbor?"*. See the output in Figure 3.7.

```
query = "what is knearest neighbor"

# Retrieve the top-2 relevant documents from the vector database
found_docs = qdrant.similarity_search(query, k=2)

# Print the first returned document
print(found_docs[0].page_content)
# OUTPUT
# More reading: How is the k-nearest neighbor algorithm different from k-means clustering?
(Quora) K-Nearest Neighbors is a supervised classification algorithm, while the
# mechanisms may seem similar at first, what this really means is that in order for K-Near
est Neighbors to work, you need labeled data you want to classify an unlabeled point into
# (thus the nearest neighbor part). K-means clustering requires only a set of unlabeled po
ints and a threshold: the algorithm will take unlabeled points and gradually
# learn how to cluster them into groups by computing the mean of the distance between diff
erent points.The critical difference here is that KNN needs labeled points and is
# thus supervised learning, while k-means doesn't — and is thus unsupervised learning.Q4-
Explain how a ROC curve works.More reading: Receiver operating characteristic (Wikipedia)
# k-means clustering is an unsupervised
```

Figure 3.7: Question answering example with output

Awesome! The retrieved answer seems quite relevant.

The entire code is displayed in Figure 3.8.

```python
from langchain.document_loaders import TextLoader, PDFMinerLoader
from langchain.text_splitter import CharacterTextSplitter, TokenTextSplitter
from langchain.embeddings import OpenAIEmbeddings
from langchain.vectorstores import Qdrant

PDF_PATH = 'ml_interview.pdf'
CHUNK_SIZE = 1000
CHUNK_OVERLAP = 30

loader = PDFMinerLoader(PDF_PATH)
pdf_content = loader.load()

text_splitter = CharacterTextSplitter(chunk_size=CHUNK_SIZE, chunk_overlap=CHUNK_OVERLAP)
docs = text_splitter.split_documents(pdf_content)

# initialize OpenAIEmbedding
embeddings = OpenAIEmbeddings(model='text-embedding-ada-002')

# If not set OPENAI_API_KEY environment variable use this
# embeddings = OpenAIEmbeddings(openai_api_key=OPENAI_API_KEY, model='text-embedding-ada-0
02')

# Create a qdrant client
qdrant = Qdrant.from_documents(
    docs,
    embeddings,
    path="/tmp/local_qdrant",  # Location to store the vector database Index
    collection_name="my_documents",
    force_recreate=True,  # Force to override the collection if exists
)

query = "what is knearest neighbor?"
found_docs = qdrant.similarity_search(query)

# Print the first returned document
print(found_docs[0].page_content)
```

Figure 3.8: The entire code for retrieval component

3.3 Generation Component Implementation

Figure 3.9 illustrates a simplified version of the RAG pipeline we saw in Chapter 2. So far our **Retrieval** component of the RAG is implemented. In the next section we will implement the **Generation** component.

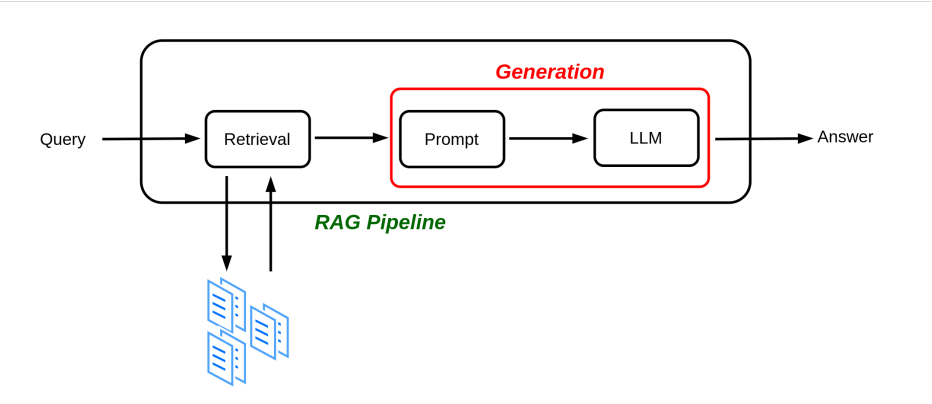


Figure 3.9: RAG pipeline

The steps for generating a response for a user's question are:

- **Step 1:** Embed the user's query using the same model used for embedding documents
- **Step 2:** Pass the query embedding to vector database, search and retrieve the top-k documents (i.e. context) from the vector database
- **Step 3:** Create a "prompt" and include the user's query and context in it
- **Step 4:** Call the LLM and pass the the prompt
- **Step 5:** Get the generated response from LLM and display it to the user

Again, we can follow each step one by one, or utilize the features langchain or llamaIndex provide. We are going to use langchain in this case.

Langchain includes several kinds of built-in question-answering chains. A *chain* in LangChain refers to a sequence of calls to components, which can include

other chains or external tools. In order to create a question answering chain, we use:

1. **load_qa_chain:** `load_qa_chain()` is a function in Langchain that loads a pre-configured question answering chain. It takes in a language model like OpenAI, a chain type (e.g. "stuff" for extracting answers from text), and optionally a prompt template and memory object. The function returns a `QuestionAnsweringChain` instance that is ready to take in documents and questions to generate answers.
2. **load_qa_with_sources_chain:** This is very similar to `load_qa_chain` except it contains sources/metadata along with the returned response.
3. **RetrievalQA:** `RetrievalQA` is a class in Langchain that creates a question answering chain using retrieval. It combines a retriever, prompt template, and LLM together into an end-to-end QA pipeline. The prompt template formats the question and retrieved documents into a prompt for the LLM. This chain retrieves relevant documents from a vector database for a given query, and then generates an answer using those documents.
4. **RetrievalQAWithSourcesChain:** It is a variant of RetrievalQA that returns relevant source documents used to generate the answer. This chain returns an `AnswerWithSources` object containing the answer string and a list of source IDs.

Here's the code demonstraing the implementation, Figure 3.10:

Figure 3.11 shows how to use `load_qa_with_sources_chain`:

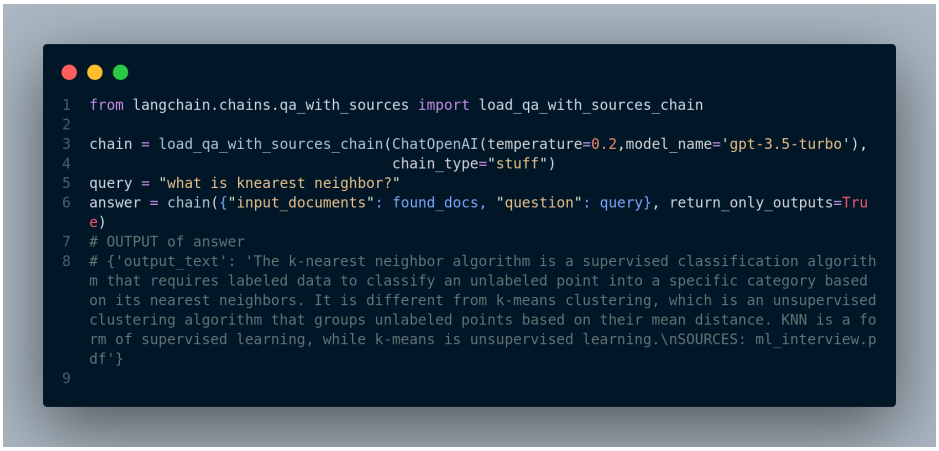
Similarly, if we use `RetrievalQA`, we will have Figure 3.12:

And here's the code when we use `RetrievalQAWithSourcesChain`, Figure 3.13:

As you can see, it's fairly straight forward to implement RAG (or say a prototype RAG application) using frameworks like langchain or llamaIndex. However, when it comes to deploying RAG to production and scaling the system, it becomes notoriously challenging. There are a lot of nuances that will affect

```python
from langchain.chains.question_answering import load_qa_chain
from langchain.llms import OpenAI

# Create the QA chain
chain = load_qa_chain(ChatOpenAI(temperature=0), chain_type="stuff")

query = "what is knearest neighbor?"
found_docs = qdrant.similarity_search(query)

# Pass the retrieved documents from vector database, and the query
answer = chain.run(input_documents=found_docs, question=query)
print(answer)
#'K-nearest neighbor (KNN) is a supervised classification algorithm. It is used to classify an unlabeled data point by finding the K nearest labeled data points in the training set and assigning the majority label among those neighbors to the unlabeled point. In other words, KNN determines the class of a data point based on the classes of its nearest neighbors.'

```

Figure 3.10: Response generation using Langchain chain

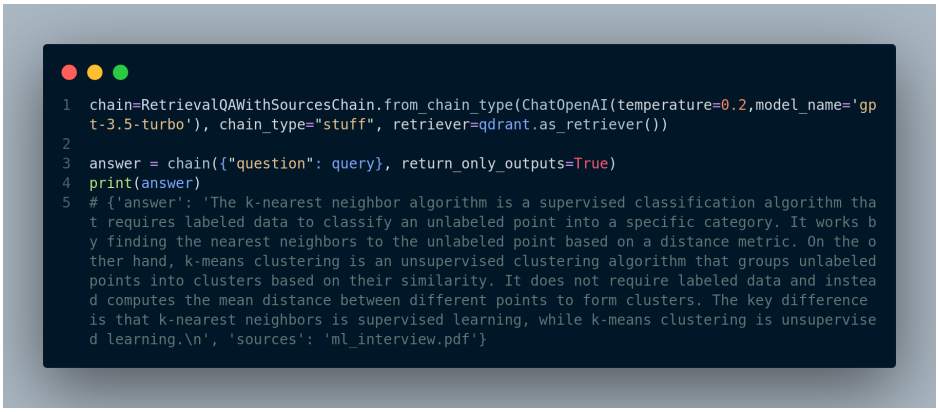


```python
from langchain.chains.qa_with_sources import load_qa_with_sources_chain

chain = load_qa_with_sources_chain(ChatOpenAI(temperature=0.2,model_name='gpt-3.5-turbo'),
                                   chain_type="stuff")
query = "what is knearest neighbor?"
answer = chain({"input_documents": found_docs, "question": query}, return_only_outputs=True)
# OUTPUT of answer
# {'output_text': 'The k-nearest neighbor algorithm is a supervised classification algorithm that requires labeled data to classify an unlabeled point into a specific category based on its nearest neighbors. It is different from k-means clustering, which is an unsupervised clustering algorithm that groups unlabeled points based on their mean distance. KNN is a form of supervised learning, while k-means is unsupervised learning.\nSOURCES: ml_interview.pdf'}

```

Figure 3.11: Using `load_qa_with_sources_chain` chain for response generation

```python
from langchain.chains import RetrievalQA
from langchain.chains import RetrievalQAWithSourcesChain

qa=RetrievalQA.from_chain_type(llm=ChatOpenAI(temperature=0.2,model_name='gpt-3.5-turbo'),
chain_type="stuff", retriever=qdrant.as_retriever())

answer = qa.run(query)
# 'K-Nearest Neighbors (KNN) is a supervised classification algorithm. It is used to classi
fy an unlabeled data point by finding the k nearest labeled data points in the training set
and assigning the majority label among those neighbors to the unlabeled point. In KNN, the
value of k determines the number of neighbors to consider.'

```

Figure 3.12: The usage of `RetrievalQA` chain



```python
chain=RetrievalQAWithSourcesChain.from_chain_type(ChatOpenAI(temperature=0.2,model_name='gp
t-3.5-turbo'), chain_type="stuff", retriever=qdrant.as_retriever())

answer = chain({"question": query}, return_only_outputs=True)
print(answer)
# {'answer': 'The k-nearest neighbor algorithm is a supervised classification algorithm tha
t requires labeled data to classify an unlabeled point into a specific category. It works b
y finding the nearest neighbors to the unlabeled point based on a distance metric. On the o
ther hand, k-means clustering is an unsupervised clustering algorithm that groups unlabeled
points into clusters based on their similarity. It does not require labeled data and instea
d computes the mean distance between different points to form clusters. The key difference
is that k-nearest neighbors is supervised learning, while k-means clustering is unsupervise
d learning.\n', 'sources': 'ml_interview.pdf'}
```

Figure 3.13: Code snippet for using Langchain `RetrievalQAWithSourcesChain` for response generation

the quality of the RAG, and we need to take them into consideration. We will discuss some of the main challenges and how to address them in the next few sections.

3.4 Impact of Text Splitting on Retrieval Augmented Generation (RAG) Quality

In the context of building a Chat-to-PDF app using Large Language Models (LLMs), one critical aspect that significantly influences the quality of your Retrieval Augmented Generation (RAG) system is how you split text from PDF documents. Text splitting can be done at two levels: *splitting by character* and *splitting by token*. The choice you make between these methods can have a profound impact on the effectiveness of your RAG system. Let's delve into the implications of each approach.

3.4.1 Splitting by Character

Advantages:

Fine-Grained Context: Splitting text by character retains the finest granularity of context within a document. Each character becomes a unit of input, allowing the model to capture minute details.

Challenges:

- **Long Sequences:** PDF documents often contain long paragraphs or sentences. Splitting by character can result in extremely long input sequences, which may surpass the model's maximum token limit, making it challenging to process and generate responses.

- **Token Limitations:** Most LLMs, such as GPT-3, have token limits, often around 4,000 tokens. If a document exceeds this limit, you'll need to truncate or omit sections, potentially losing valuable context.
- **Increased Inference Time:** Longer sequences require more inference time, which can lead to slower response times and increased computational costs.

3.4.2 Splitting by Token

Advantages:

- **Token Efficiency:** Splitting text by token ensures that each input sequence remains within the model's token limit, allowing for efficient processing.
- **Balanced Context:** Each token represents a meaningful unit, striking a balance between granularity and manageability.
- **Scalability:** Splitting by token accommodates documents of varying lengths, making the system more scalable and adaptable.

Challenges:

- **Contextual Information:** Token-based splitting may not capture extremely fine-grained context, potentially missing nuances present in character-level splitting.

3.4.3 Finding the Right Balance

The choice between character-level and token-level splitting is not always straightforward and may depend on several factors:

- **Document Types:** Consider the types of PDF documents in your collection. Technical manuals with precise details may benefit from character-level splitting, while general documents could work well with token-level splitting.
- **Model Limitations:** Take into account the token limits of your chosen LLM. If the model's limit is a significant constraint, token-level splitting becomes a necessity.
- **User Experience:** Assess the trade-off between detailed context and response time. Character-level splitting might provide richer context but at the cost of slower responses.

3.4.4 Hybrid Approaches

In practice, you can also explore hybrid approaches to text splitting. For instance, you might use token-level splitting for most of the document and switch to character-level splitting when a specific question requires fine-grained context.

The impact of text splitting on RAG quality cannot be overstated. It's a critical design consideration that requires a balance between capturing detailed context and ensuring system efficiency. Carefully assess the nature of your PDF documents, the capabilities of your chosen LLM, and user expectations to determine the most suitable text splitting strategy for your Chat-to-PDF app. Regular testing and user feedback can help refine this choice and optimize the overall quality of your RAG system.

3.5 Impact of Metadata in the Vector Database on Retrieval Augmented Generation (RAG)

The inclusion of metadata about the data stored in the vector database is another factor that can significantly enhance the quality and effectiveness of your Retrieval Augmented Generation (RAG) system. Metadata provides valuable

contextual information about the PDF documents, making it easier for the RAG model to retrieve relevant documents and generate accurate responses. Here, we explore the ways in which metadata can enhance your RAG system.

3.5.1 Contextual Clues

Metadata acts as contextual clues that help the RAG model better understand the content and context of each PDF document. Typical metadata includes information such as:

- **Document Title:** The title often provides a high-level summary of the document's content.
- **Author:** Knowing the author can offer insights into the document's perspective and expertise.
- **Keywords and Tags:** Keywords and tags can highlight the main topics or themes of the document.
- **Publication Date:** The date of publication provides a temporal context, which is crucial for understanding the relevance of the document.
- **Document Type:** Differentiating between research papers, user manuals, and other types of documents can aid in tailoring responses appropriately.

3.5.2 Improved Document Retrieval

With metadata available in the vector database, the retrieval component of your RAG system can become more precise and efficient. Here's how metadata impacts document retrieval:

- **Relevance Ranking:** Metadata, such as document titles, keywords, and tags, can be used to rank the documents based on relevance to a user query. Documents with metadata matching the query can be given higher priority during retrieval. For example, if a user asks a question related

to "machine learning," documents with "machine learning" in their keywords or tags might be given priority during retrieval.
- **Filtering:** Metadata can be used to filter out irrelevant documents early in the retrieval process, reducing the computational load and improving response times. For instance, if a user asks about "biology," documents with metadata indicating they are engineering manuals can be excluded from consideration.
- **Enhanced Query Understanding:** Metadata provides additional context for the user's query, allowing the RAG model to better understand the user's intent and retrieve documents that align with that intent. For example, if the metadata includes the publication date, the RAG model can consider the temporal context when retrieving documents.

3.5.3 Contextual Response Generation

Metadata can also play a crucial role in the generation component of your RAG system. Here's how metadata impacts response generation:

- **Context Integration:** Metadata can be incorporated into the response generation process to provide more contextually relevant answers. For example, including the publication date when answering a historical question.
- **Customization:** Metadata can enable response customization. For instance, the tone and style of responses can be adjusted based on the author's information.
- **Enhanced Summarization:** Metadata can aid in the summarization of retrieved documents, allowing the RAG model to provide concise and informative responses. For instance, if the metadata includes the document type as "research paper," the RAG system can generate a summary that highlights the key findings or contributions of the paper.

3.5.4 User Experience and Trust

Including metadata in the RAG system not only enhances its technical capabilities but also improves the overall user experience. Users are more likely to trust and find value in a system that provides contextually relevant responses. Metadata can help build this trust by demonstrating that the system understands and respects the nuances of the user's queries.

Overall, incorporating metadata about data in the vector database of your Chat-to-PDF app's RAG system can significantly elevate its performance and user experience. Metadata acts as a bridge between the user's queries and the content of PDF documents, facilitating more accurate retrieval and generation of responses.

As we conclude our exploration of the nuts and bolts of RAG pipelines in this Chapter, it's time to move on to more complex topics. In Chapter 4, we'll take a deep dive and try to address some of the retrieval and generation challenges that come with implementing advanced RAG systems.

We'll discuss the optimal chunk size for efficient retrieval, consider the balance between context and efficiency, and introduce additional resources for evaluating RAG performance. Furthermore, we'll explore retrieval chunks versus synthesis chunks and ways to embed references to text chunks for better understanding.

We'll also investigate how to rethink retrieval methods for heterogeneous document corpora, delve into hybrid document retrieval, and examine the role of query rewriting in enhancing RAG capabilities.

4 From Simple to Advanced RAG

4.1 Introduction

As we traverse the path from development to production, the world of Retrieval Augmented Generation (RAG) applications unveils its potential to transform the way we interact with vast collections of information. In the preceding chapters, we've laid the groundwork for building RAG systems that can answer questions, provide insights, and deliver valuable content. However, the journey is far from over.

Please note that LlamaIndex framework has been used for several of the code implementations in this chapter. This framework also contains a lot of advanced tutorials about RAG that inspired the content of this chapter.

The transition from a well-crafted RAG system in the development environment to a real-world production application is a monumental step, one that demands careful consideration of a myriad of factors and deals with limitations of existing approaches. These considerations ensure your RAG application operates seamlessly in a real-world production environment. There are so many challenges and consideration for building *production-ready* RAG. Nevertheless, we will discuss some of the primary ones.

RAG pipeline consists of two components: i) Retrieval and, ii) response generation (synthesis). Each component has its own challenges.

Retrieval Challenges

- **Low precision:** When retrieving the top-k chunks, there's a risk of including irrelevant content, potentially leading to issues like hallucination and generating inaccurate responses.
- **Low recall:** In certain cases, even when all the relevant chunks are retrieved, the text chunks might lack the necessary global context beyond the retrieved chunks to generate a coherent response.
- **Obsolete information:** Ensuring the data remains up-to-date is critical to avoid reliance on obsolete information. Regular updates are essential to maintain data relevance and accuracy.

Generation Challenges

- **Hallucination:** The model generates responses that include information not present in the context, potentially leading to inaccuracies or fictional details.
- **Irrelevance:** The model produces answers that do not directly address the user's question, resulting in responses that lack relevance or coherence.
- **Bias:** The model generates answers that contai n harmful or offensive content, potentially reflecting biases and undermining user trust and safety.

What Can be done

In order to effectively tackle the challenges in Retrieval Augmented Generation (RAG) systems, several key strategies can be employed:

1. **Data Augmentation:** Can we enrich our dataset by storing supplementary information beyond raw text chunks, such as metadata or structured data, to provide richer context for retrieval and generation processes?

2. **Optimized Embeddings:** Can we refine and enhance our embedding representations to capture context more effectively, improve the relevance of retrieved information, and enable more coherent response generation?

3. **Advanced Retrieval Techniques:** Can we go beyond basic top-k embedding lookup and implement advanced retrieval methods, such as semantic search, or hybrid search (keyword search + semantic search) to enhance the precision and recall of information retrieval?

4. **Multi-Purpose Use of LLMs:** Can we leverage Large Language Models (LLMs) for tasks beyond text generation, such as question answering, summarization, or knowledge graph construction, to augment the capabilities of RAG systems and provide more comprehensive responses to user queries?

Let's dive in a bit more deeply into aforementioned challenges and propose how to alleiviate each one.

4.2 Optimul Chunk Size for Efficient Retrieval

The chunk size in a RAG system is the size of the text passages that are extracted from the source text and used to generate the retrieval index. The chunk size has a significant impact on the system's efficiency and performance in several ways:

4.2.1 Balance Between Context and Efficiency

The chunk size should strike a balance between providing sufficient context for generating coherent responses (i.e. Relevance and Granularity) and ensuring efficient retrieval and processing (i.e Performance).

A smaller chunk size results in more granular chunks, which can improve the relevance of the retrieved chunks. However, it is important to note that the most relevant information may not be contained in the top retrieved chunks, especially if the *similarity_top_k* setting is low (e.g. k=2). A smaller chunk size also results in more chunks, which can increase the system's memory and processing requirements.

A larger chunk size can improve the system's performance by reducing the number of chunks that need to be processed. However, it is important to note that a larger chunk size can also reduce the relevance of the retrieved chunks.

The optimal chunk size for a RAG system depends on a number of factors, including the size and complexity of the source text, the desired retrieval performance, and the available system resources. However, it is important to experiment with different chunk sizes to find the one that works best for your specific system. But, how do we know what works and what doesn't?

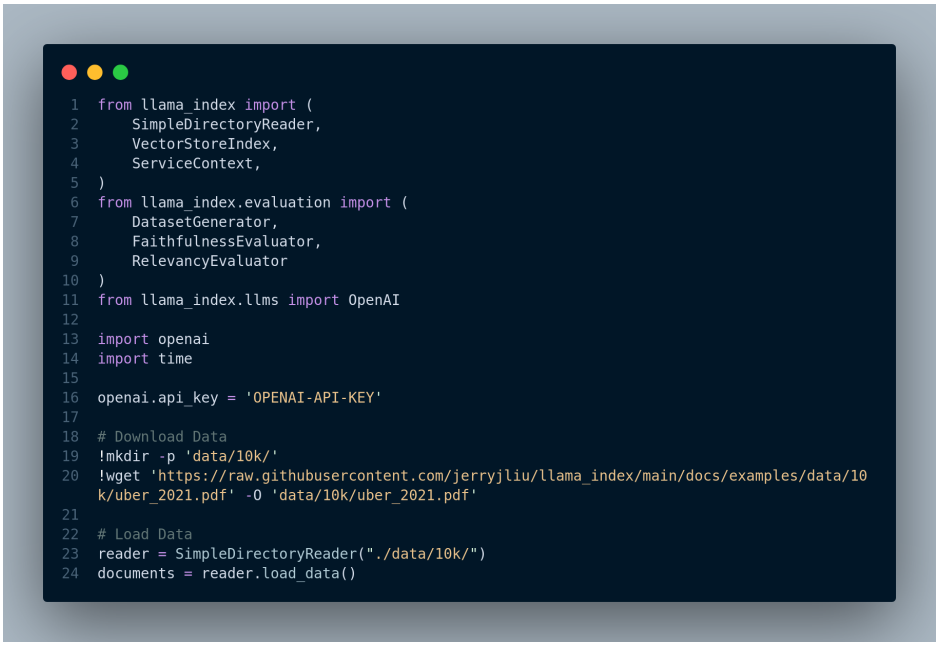
Question: *How can we measure the performance of RAG?*

Answer: We have to define evaluation metrics and then use evaluation tools to measure how the RAG performs considering the metrics. There are several tools to evaluate the RAG including LlamaIndex **Response Evaluation** module to test, evaluate and choose the right *chunk size*. It contains a few components, particularly:

1. **Faithfulness Evaluator:** This tool assesses whether the response includes fabricated information and determines if the response aligns with any source nodes from the query engine.
2. **Relevancy Evaluator:** This tool gauges whether the response effectively addresses the query and evaluates if the combined information from the response and source nodes corresponds to the query.

The code below shows how to use *Evaluation module* and determine the optimal *chunk size* for retrieval. To read the full article, see this link.

Figure 4.1 shows the data preparation for evaluation module.



```python
from llama_index import (
    SimpleDirectoryReader,
    VectorStoreIndex,
    ServiceContext,
)
from llama_index.evaluation import (
    DatasetGenerator,
    FaithfulnessEvaluator,
    RelevancyEvaluator
)
from llama_index.llms import OpenAI

import openai
import time

openai.api_key = 'OPENAI-API-KEY'

# Download Data
!mkdir -p 'data/10k/'
!wget 'https://raw.githubusercontent.com/jerryjliu/llama_index/main/docs/examples/data/10k/uber_2021.pdf' -O 'data/10k/uber_2021.pdf'

# Load Data
reader = SimpleDirectoryReader("./data/10k/")
documents = reader.load_data()
```

Figure 4.1: data preparation for reponse evaluation

Figure 4.2 displays the criteria to set for evaluation.

```
# To evaluate for each chunk size, we will first generate a set of 40 questions from first
20 pages.
eval_documents = documents[:20]
data_generator = DatasetGenerator.from_documents()
eval_questions = data_generator.generate_questions_from_nodes(num = 20)

# We will use GPT-4 for evaluating the responses
gpt4 = OpenAI(temperature=0, model="gpt-4")

# Define service context for GPT-4 for evaluation
service_context_gpt4 = ServiceContext.from_defaults(llm=gpt4)

# Define Faithfulness and Relevancy Evaluators which are based on GPT-4
faithfulness_gpt4 = FaithfulnessEvaluator(service_context=service_context_gpt4)
relevancy_gpt4 = RelevancyEvaluator(service_context=service_context_gpt4)
```

Figure 4.2: Define criteria for evaluation

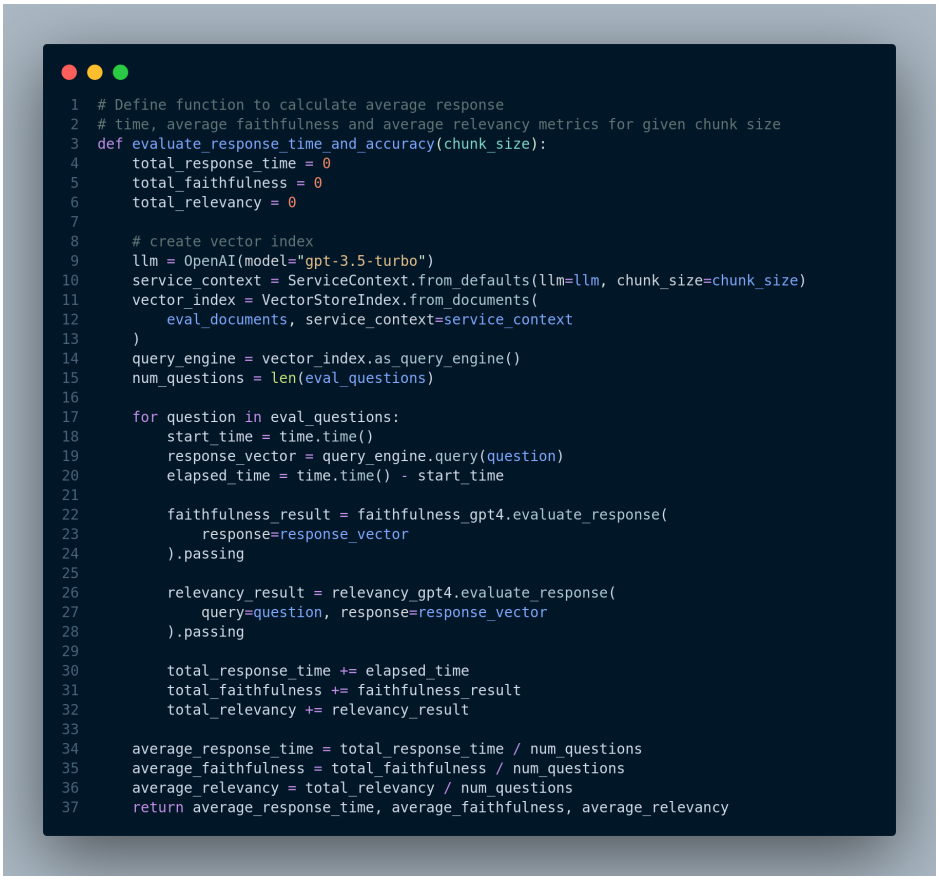
Figure 4.3 shows the evaluation function definition.

And Figure 4.4 demonstrates testing the evaluation function with different chunk sizes.

They test across different chunk sizes and conclude (in this experiment) that `chunk_size = 1024` results in peaking of *Average Faithfulness* and *Average Relevancy*.

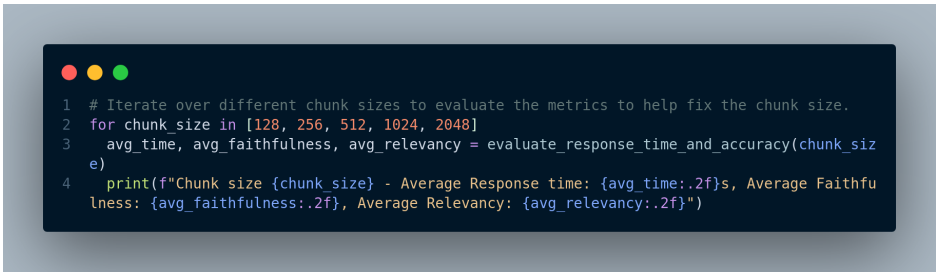
Here are summary of the tips for choosing the optimal chunk size for a RAG system:

- Consider the size and complexity of the source text. Larger and more complex texts will require larger chunk sizes to ensure that all of the relevant information is captured.
- Consider the desired retrieval performance. If you need to retrieve the most relevant chunks possible, you may want to use a smaller chunk size.



```python
# Define function to calculate average response
# time, average faithfulness and average relevancy metrics for given chunk size
def evaluate_response_time_and_accuracy(chunk_size):
    total_response_time = 0
    total_faithfulness = 0
    total_relevancy = 0

    # create vector index
    llm = OpenAI(model="gpt-3.5-turbo")
    service_context = ServiceContext.from_defaults(llm=llm, chunk_size=chunk_size)
    vector_index = VectorStoreIndex.from_documents(
        eval_documents, service_context=service_context
    )
    query_engine = vector_index.as_query_engine()
    num_questions = len(eval_questions)

    for question in eval_questions:
        start_time = time.time()
        response_vector = query_engine.query(question)
        elapsed_time = time.time() - start_time

        faithfulness_result = faithfulness_gpt4.evaluate_response(
            response=response_vector
        ).passing

        relevancy_result = relevancy_gpt4.evaluate_response(
            query=question, response=response_vector
        ).passing

        total_response_time += elapsed_time
        total_faithfulness += faithfulness_result
        total_relevancy += relevancy_result

    average_response_time = total_response_time / num_questions
    average_faithfulness = total_faithfulness / num_questions
    average_relevancy = total_relevancy / num_questions
    return average_response_time, average_faithfulness, average_relevancy
```

Figure 4.3: Define a function to perform evaluation



```
# Iterate over different chunk sizes to evaluate the metrics to help fix the chunk size.
for chunk_size in [128, 256, 512, 1024, 2048]
  avg_time, avg_faithfulness, avg_relevancy = evaluate_response_time_and_accuracy(chunk_size)
  print(f"Chunk size {chunk_size} - Average Response time: {avg_time:.2f}s, Average Faithfulness: {avg_faithfulness:.2f}, Average Relevancy: {avg_relevancy:.2f}")
```

Figure 4.4: Run the evaluation function with different parameters

However, if you need to retrieve chunks quickly, you may want to use a larger chunk size.

- Consider the available system resources. If you are limited on system resources, you may want to use a smaller chunk size. However, if you have ample system resources, you may want to use a larger chunk size. You can evaluate the optimal chunk size for your RAG system by using a variety of metrics, such as:
- **Relevance:** The percentage of retrieved chunks that are relevant to the query.
- **Faithfulness:** The percentage of retrieved chunks that are faithful to the source text.
- **Response time:** The time it takes to retrieve chunks for a query. Once you have evaluated the performance of your RAG system for different chunk sizes, you can choose the chunk size that strikes the best balance between relevance, faithfulness, and response time.

4.2.2 Additional Resources for RAG Evaluation

Evaluation of RAG applications remains an unsolved problem today and is an active research area. Here are just a few references for practical evaluation of

RAG: 1. This blog from Databricks has some best practices to evaluate RAG applications. Figure 4.5 illustrates what their experiment setup looks like.

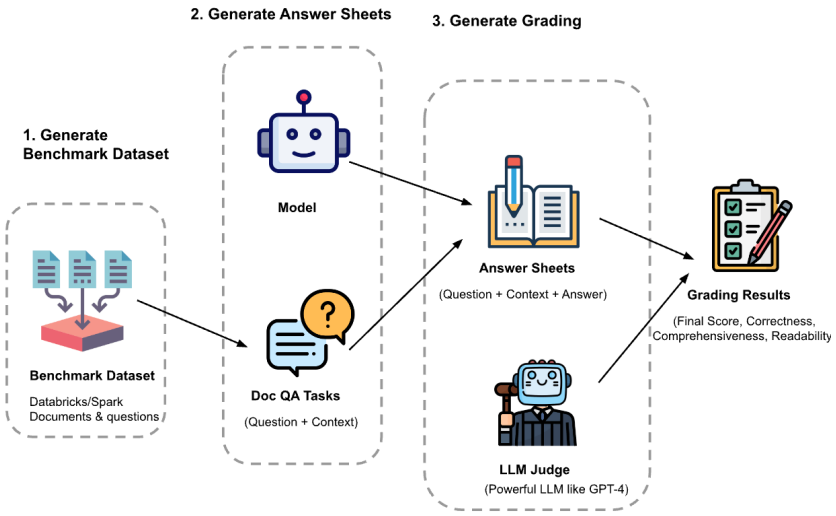


Figure 4.5: Databricks evaluation experiment setup. Image source

2. Zheng et al. (2023) propose a strong LLMs as judges to evaluate these models on more open-ended questions.
3. RAG Evaluation is another interesting blog that discuss how to use Langchain for evaluating RAG applications.

4.3 Retrieval Chunks vs. Synthesis Chunks

Another fundamental technique for enhancing retrieval in Retrieval Augmented Generation (RAG) systems is the *decoupling of chunks* used for retrieval from those used for synthesis (i.e. response generation). The main idea is optimal

chunk representation for retrieval may not necessarily align with the requirements for effective synthesis. While a raw text chunk could contain essential details for the LLM to generate a comprehensive response, it might also contain filler words or information that could introduce biases into the embedding representation. Furthermore, it might lack the necessary global context, making it challenging to retrieve the chunk when a relevant query is received. To give an example, think about having a question answering system on emails. Emails often contain so much *fluff* (a big portion of the email is "looking forward", "great to hear from you", etc.) and so little information. Thus, retaining semantics in this context for better and more accurate question answering is very important.

There are a few ways to implement this technique including:

1. Embed references to text chunks
2. Expand sentence-level context window

4.3.1 Embed References to Text Chunks

The main idea here is that, we create an index in the vector database for storing document summaries. When a query comes in, we first fetch relevant document(s) at the high-level (i.e. summaries) before retrieving smaller text chunks directly, because it might retrieve irrelavant chunks. Then we can retrieve smaller chunks from the fetched document(s). In other words, we store the data in a hierarchical fashion: summaries of documents and chunks for each document. We can consider this approach as *Dense Hierarchical Retrieval*, in which a document-level retriever (i.e. summary index) first identifies the relevant documents, and then a passage-level retriever finds the relevant passages/chunks. Y. Liu et al. (2021) and Zhao et al. (2022) gives you a deep understanding of this approach. Figure 4.6 shows how this technique works.

We can choose different strategies based on the type of documents we are dealing with. For example, if we have a list of web pages, we can consider each page as a *document* that we summarize, and also we split each document into a set of

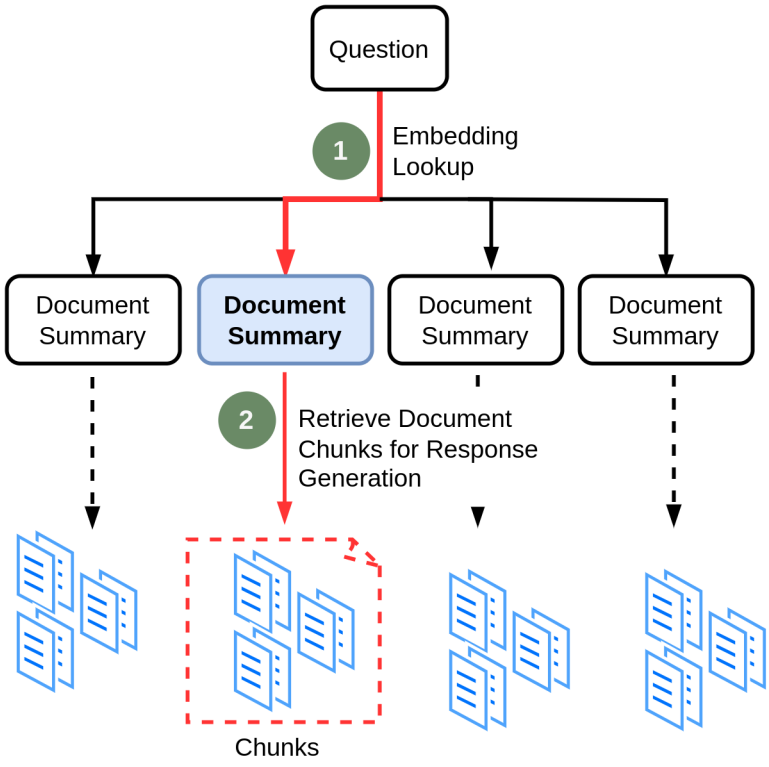


Figure 4.6: Document summary index

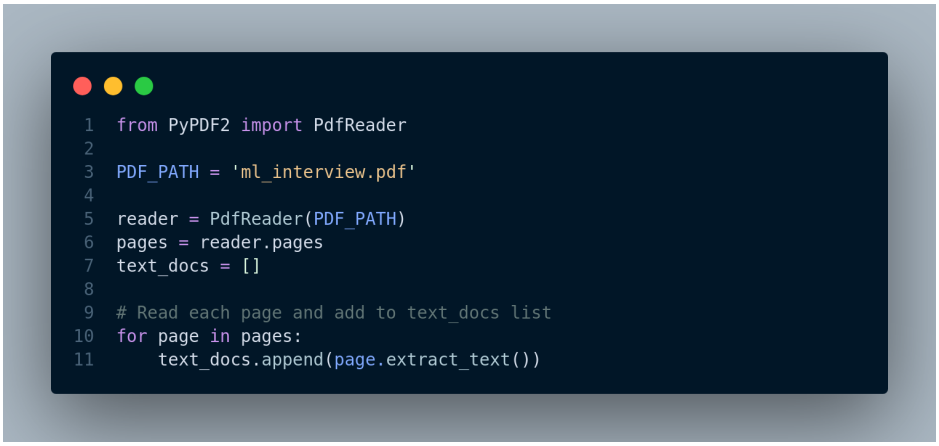
smaller chunks as the second level of our data store strategy. When user asks a question, we first find the relevant page using the summary embeddings, and then we can retrieve the relevant chunks from that particular page.

If we have a pdf document, we can consider each page of the pdf as a separate document, and then split each page into smaller chunks. If we have a list of pdf files, we can choose the entire content of each pdf to be a document and split the it into smaller chunks.

Let's code it up for our pdf file.

Step 1: Read the PDF file

We read the pdf file, Figure 4.7, and create a list of pages as later we will view each page as a separate document.



```python
from PyPDF2 import PdfReader

PDF_PATH = 'ml_interview.pdf'

reader = PdfReader(PDF_PATH)
pages = reader.pages
text_docs = []

# Read each page and add to text_docs list
for page in pages:
    text_docs.append(page.extract_text())
```

Figure 4.7: Read a list of documents from each page of the pdf

Step 2: Create Document Summary Index

In order to create an *index*, first we have to convert a list of texts into a list of *Document* that is compatible with LlamaIndex.

Definition: A Document is a generic container around any data source, for instance, a PDF, an API output, or retrieved data from a database. It stores text along with some other properties such as *metadata* and *relationships (Links to other Documents/Nodes)*

Figure 4.8 shows the code.

We can use the summary index to get the summary of each page/document using the document `id`, for instanc, Figure 4.9 shows the output of a summary of a document.

Step 3: Retrieve and Generate Response using Document Summary Index

In this step, when a query comes, we run a retrieval from the document summary index to find the relevant pages. Retrieved document has links to its corresponding chunks, that are used to generate the final response to the query.

There are multiple ways to that in LlamaIndex:

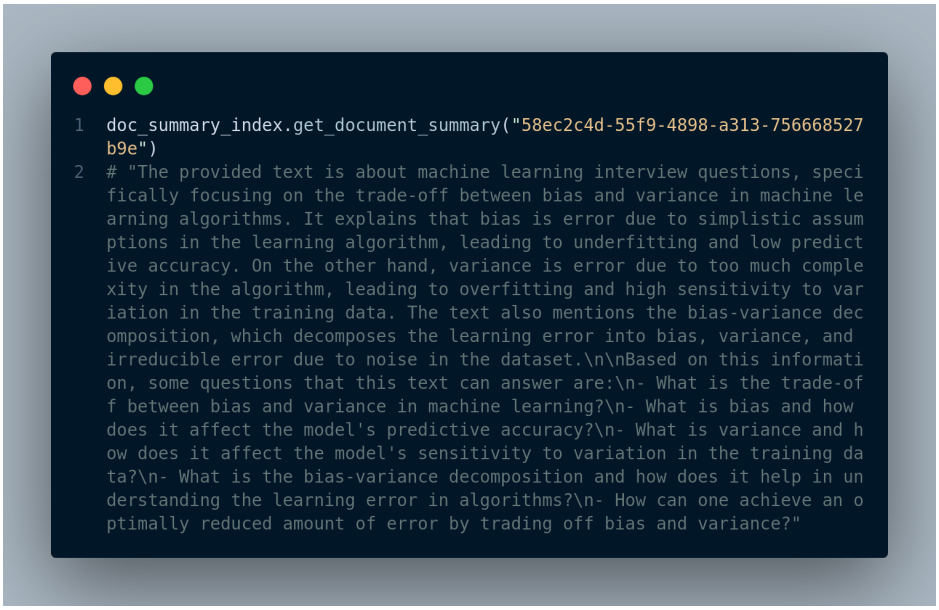
- High-level query execution
- LLM based retrieval
- Embedding based retrieval

The high-level approach is depicted in Figure 4.10

LLM based retrieval: This approach is low-level so we can view and change the parameters. Figure 4.11 below displays the code snippet:

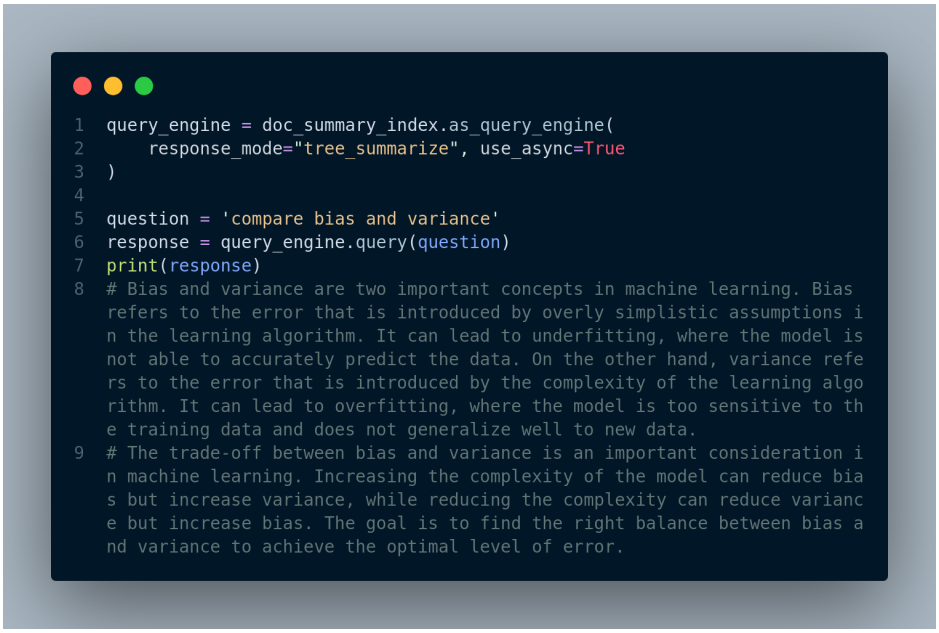
```python
from llama_index import Document
from llama_index import (
    ServiceContext,
    get_response_synthesizer,
)
from llama_index.indices.document_summary import DocumentSummaryIndex
from llama_index.llms import OpenAI

documents = [Document(text=t) for t in text_docs]
print(documents[0])
# Document(id_='04d1ba81-0000-432f-9605-34b34f2176b6', embedding=None, metadata={}, excluded_embed_metadata_keys=[], excluded_llm_metadata_keys=[], relationships={}, hash='90e6c77318f9863b478fd82b3a56bb68bfb1eedbac5c153d0eb307aa19b052b6', text='Q2- What is the difference between supervised....  metadata_seperator='\n')


# Set the model and chunk size in terms of tokens
chatgpt = OpenAI(temperature=0, model="gpt-3.5-turbo")
service_context = ServiceContext.from_defaults(llm=chatgpt, chunk_size=50)

# Define response generator
response_synthesizer = get_response_synthesizer(
    response_mode="tree_summarize", use_async=True
)

# Build the summary index
doc_summary_index = DocumentSummaryIndex.from_documents(
    documents,
    service_context=service_context,
    response_synthesizer=response_synthesizer,
    show_progress=False,
)

```

Figure 4.8: Build a document summary index



```
doc_summary_index.get_document_summary("58ec2c4d-55f9-4898-a313-756668527b9e")
# "The provided text is about machine learning interview questions, specifically focusing on the trade-off between bias and variance in machine learning algorithms. It explains that bias is error due to simplistic assumptions in the learning algorithm, leading to underfitting and low predictive accuracy. On the other hand, variance is error due to too much complexity in the algorithm, leading to overfitting and high sensitivity to variation in the training data. The text also mentions the bias-variance decomposition, which decomposes the learning error into bias, variance, and irreducible error due to noise in the dataset.\n\nBased on this information, some questions that this text can answer are:\n- What is the trade-off between bias and variance in machine learning?\n- What is bias and how does it affect the model's predictive accuracy?\n- What is variance and how does it affect the model's sensitivity to variation in the training data?\n- What is the bias-variance decomposition and how does it help in understanding the learning error in algorithms?\n- How can one achieve an optimally reduced amount of error by trading off bias and variance?"
```

Figure 4.9: Example of a document summary



```
query_engine = doc_summary_index.as_query_engine(
    response_mode="tree_summarize", use_async=True
)

question = 'compare bias and variance'
response = query_engine.query(question)
print(response)
# Bias and variance are two important concepts in machine learning. Bias
refers to the error that is introduced by overly simplistic assumptions i
n the learning algorithm. It can lead to underfitting, where the model is
not able to accurately predict the data. On the other hand, variance refe
rs to the error that is introduced by the complexity of the learning algo
rithm. It can lead to overfitting, where the model is too sensitive to th
e training data and does not generalize well to new data.
# The trade-off between bias and variance is an important consideration i
n machine learning. Increasing the complexity of the model can reduce bia
s but increase variance, while reducing the complexity can reduce varianc
e but increase bias. The goal is to find the right balance between bias a
nd variance to achieve the optimal level of error.
```

Figure 4.10: High-level query execution approach (default approach)

```python
from llama_index.indices.document_summary import DocumentSummaryIndexRetriever
from llama_index.query_engine import RetrieverQueryEngine # use retriever as part of a query engine

retriever = DocumentSummaryIndexRetriever(
    doc_summary_index,
    # choice_select_prompt=None,
    # choice_batch_size=10,
    # choice_top_k=1,
    # format_node_batch_fn=None,
    # parse_choice_select_answer_fn=None,
    # service_context=None
)

# You can see the retrieved nodes and chunks using command below
# retrieved_nodes = retriever.retrieve(question)

# configure response synthesizer
response_synthesizer = get_response_synthesizer(response_mode="tree_summarize")

# assemble query engine
query_engine = RetrieverQueryEngine(
    retriever=retriever,
    response_synthesizer=response_synthesizer,
)

# query
response = query_engine.query(question)
print(response)
```

Figure 4.11: LLM based retrieval approach

Embedding based retrieval: In this technique, we first define `DocumentSummaryIndexEmbeddingRetriever` retriever, and configure the response generator to use this retriever. We then, integrate these two components into a `RetrieverQueryEngine` and run that for the query. Figure 4.12 shows the code snippet for this approach.

```python
from llama_index.indices.document_summary import DocumentSummaryIndexEmbeddingRetriever
from llama_index.query_engine import RetrieverQueryEngine


retriever = DocumentSummaryIndexEmbeddingRetriever(
    doc_summary_index,
    similarity_top_k=1,
)

# configure response synthesizer
response_synthesizer = get_response_synthesizer(response_mode="tree_summarize")

# assemble query engine
query_engine = RetrieverQueryEngine(
    retriever=retriever,
    response_synthesizer=response_synthesizer,
)

# query
response = query_engine.query(question)
print(response)
```

Figure 4.12: Embedding based retrieval

4.3.2 Expand sentence-level context window

In this approach, we have split the text into sentence level chunks to be able to perform fine-grained retrieval. However, before passing the fetched sentences to LLM response generator, we include the sentences surrounding the retrieved sentence, to enlarge the context window for better accuaracy. Please be mindful of *lost in the middle* problem when splitting large textual content at a very fine-grained level, such as sentence-level.

Figure 4.13 illustrates this technique.

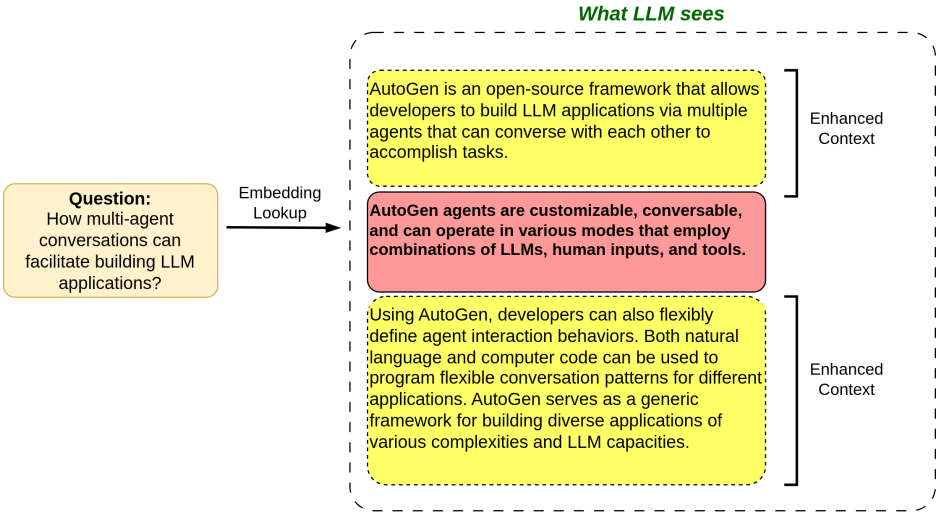


Figure 4.13: Expanding the sentence level context, so LLM has a bigger context to use to generate the response

Implementation

Again, we rely on LlamaIndex to implement this technique. We use `SentenceWindowNodeParser` to split document into sentences and save each sentence in a node. Node contains a *window* property where we can adjust. During the retrieval step, each fetched sentence will be replaced with surrounding sentences depending on the window size via `MetadataReplacementNodePostProcessor` function.

Figure 4.14 shows the basic setup such as importing necessary modules, reading the pdf file and initializing the LLM and embedding models.

```python
from llama_index import ServiceContext, VectorStoreIndex
from llama_index import SimpleDirectoryReader
from llama_index.llms import OpenAI
from llama_index.embeddings import HuggingFaceEmbedding
from llama_index.node_parser import SentenceWindowNodeParser, SimpleNodeParser
from llama_index.indices.postprocessor import MetadataReplacementPostProcessor


# Load the pdf and create a list of Document objects
documents = SimpleDirectoryReader(
    input_files=[PDF_PATH]
).load_data()


# create the sentence window node parser with default settings
node_parser = SentenceWindowNodeParser.from_defaults(
    window_size=3,  # 5 is default value
    window_metadata_key="window",
    original_text_metadata_key="original_text",
)


simple_node_parser = SimpleNodeParser.from_defaults()

# COnfigure the LLM model
llm = OpenAI(model="gpt-3.5-turbo", temperature=0)

# Select an embedding model. HuggingFace is used since being faster and free
embed_model = HuggingFaceEmbedding(
    model_name="sentence-transformers/all-mpnet-base-v2", max_length=512
)
```

Figure 4.14: Basic setup for sentence window implementation

Next, we have to define nodes that are going to be stored in the `VectorIndex` as well as sentence index. Then, we create a query engine and run the query. Figure 4.15 shows the steps.

```
# Create ServiceContext object
ctx = ServiceContext.from_defaults(
    llm=llm,
    embed_model=embed_model,
)


# Define nodes to be stored in the VectorIndex
nodes = node_parser.get_nodes_from_documents(documents)

# Build the sentence index
sentence_index = VectorStoreIndex(nodes, service_context=ctx)

query_engine = sentence_index.as_query_engine(
    similarity_top_k=2,
    # the target key defaults to `window` to match the node_parser's default
    node_postprocessors=[
        MetadataReplacementPostProcessor(target_metadata_key="window")
    ],
)
window_response = query_engine.query(
    "how to avoid overfitting?"
)
print(window_response)
```

Figure 4.15: Build the sentence index, and run the query

Figure 4.16 displays the response output.

We can see the original sentence that is retrieved for each node (we show the first node below) and also the actual window of sentences in the Figure 4.17.

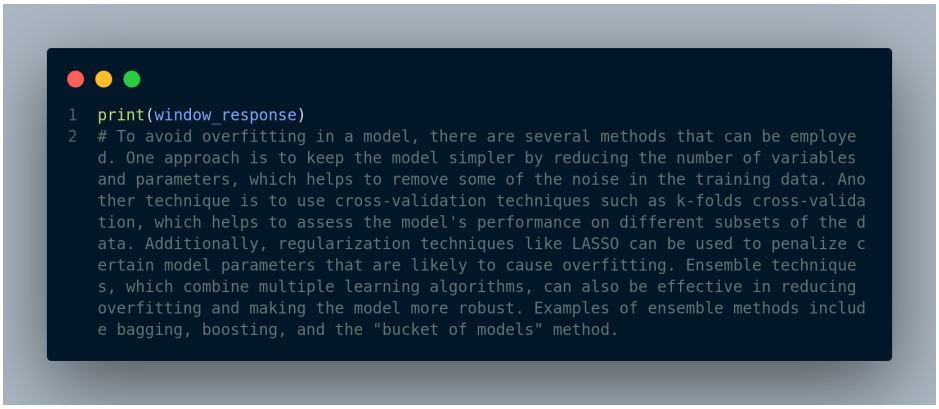


Figure 4.16: Output of the window response

4.3.3 Lost in the Middle Problem

Retrieval process in a RAG based application is all about retrieving the right and most relevant documents for a given user's query. The way we find these documents is that retrieval method assigns a relevance score to each document based on their similarity to the query. Then, sorts them descendingly and returns them. Nevertheless, this approach might not work well when we are returning many documents such as `top-k >= 10`. The reason is when we pass a very long context to LLM, it tend to ignore or overlook the documents in the middle. Consequently, putting the least relevant document to the bottom of the fetch documents is not the best strategy. A better way is to place the least relevant documents in the middle.

N. F. Liu et al. (2023) in Lost in the Middle: How Language Models Use Long Contexts demonstrated interesting findings about LLMs behavior. They realized that performance of LLMs is typically at its peak when relevant information is located at the beginning or end of the input context. However, it notably deteriorates when models are required to access relevant information buried within the middle of lengthy contexts. Figure 4.18 demonstrates the results.

```
window = window_response.source_nodes[0].node.metadata["window"]
sentence = window_response.source_nodes[0].node.metadata["original_text"]

print(f"Window: {window}")
print("------------------")
print(f"Original Sentence: {sentence}")
# Window: Q22- How do you ensure you're not overfitting with a
# model?  More reading: How can I avoid overfitting?  (Quora)
# This is a simple restatement of a fundamental problem in machine
# learning: the possibility of overfitting training data and carrying the
# noise of that data through to the test set, thereby providing inaccu-
# rate generalizations.
#  There are three main methods to avoid overfitting:
# 1- Keep the model simpler: reduce variance by taking into account
# fewer variables and parameters, thereby removing some of the noise
# in the training data.
#  2- Use cross-validation techniques such as k-folds cross-validation.
#  3- Use regularization techniques such as LASSO that penalize certain
# model parameters if they're likely to cause overfitting.
#
# ------------------
# Original Sentence: There are three main methods to avoid overfitting:
# 1- Keep the model simpler: reduce variance by taking into account
# fewer variables and parameters, thereby removing some of the noise
# in the training data.

```

Figure 4.17: Original sentence that was retrieved for each node, as well as the actual window of sentences

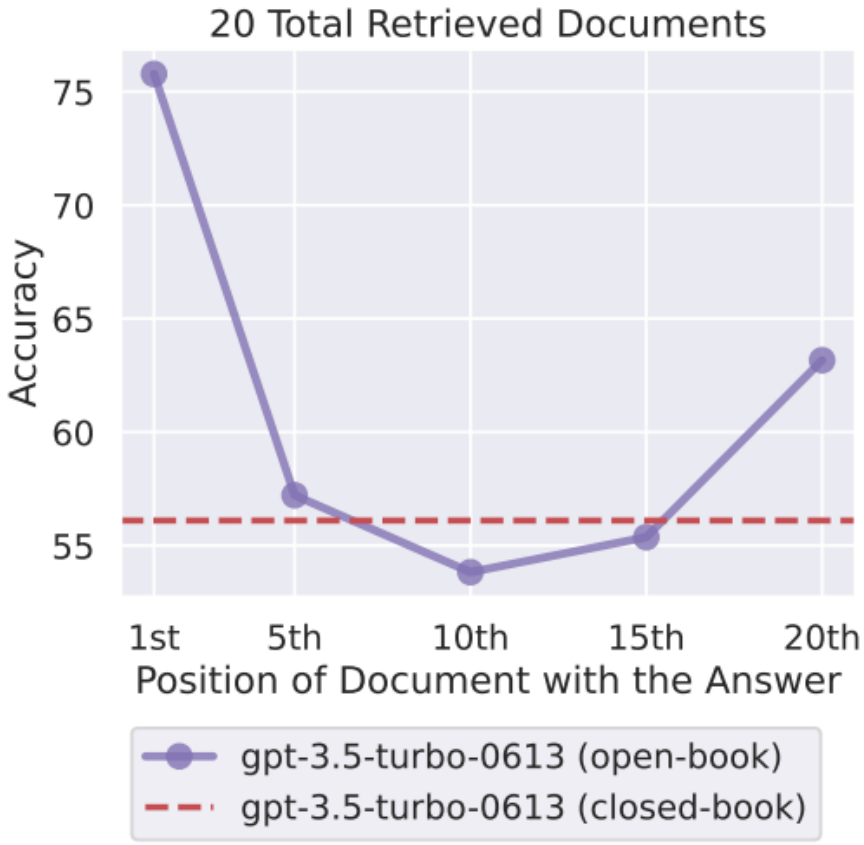


Figure 4.18: Accuracy of the RAG based on the postions of the retrieved documents. Image source

They also show that LLMs with longer context windows still face this problem and increasing the context window doesn't solve this issue. The following demonstrates this experiment.

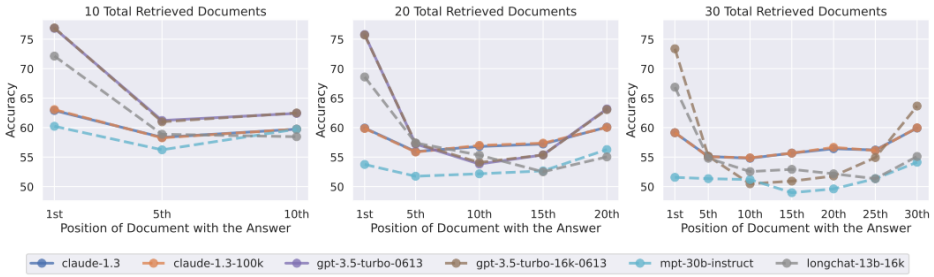


Figure 4.19: Comparing LLM models with various context size and the impact of changing the position of relevant documents

How can we alleviate this problem? The answer is to reorder the retrieved documents in such a way that most similar documents to the query are placed at the top, the less similar documents at the bottom, and the least similar documents in the middle.

For implementation, we need a function to get the retrieved documents from the `retriever` and reorder them, i.e. place most relevant documents at the beginning and end. Figure 4.20 shows our code.

We can instead utilize Langchain solution: `LongContextReorder`. It essentially implements a similar approach to Figure 4.20 function. You can read the documentation for more details.

Figure 4.21 shows how to use Langchain solution to deal with this problem.

We can also use Haystack to deal with this problem. Haystack is the open source framework for building custom NLP apps with large language models (LLMs) in an end-to-end fashion. It offers a few components that are building blocks for performing various tasks like document retrieval, and summarization. We can connect these components and create an end-to-end *pipeline*. The two very

```python
def reorder_documents(documents):
    # Calculate the index for placing the middle documents
    middle_index = len(documents) // 2

    # Split the sorted documents into two halves
    top_half = documents[:middle_index]
    bottom_half = documents[middle_index:]

    # Reverse the order of the bottom half to place the least similar documents in the middle
    reordered_documents = top_half + bottom_half[::-1]

    return reordered_documents
```

Figure 4.20: Pseudocode of a function to solve lost in the middle problem

```python
# Create a retriever, below is just an example
retriever = Chroma.from_texts(texts, embedding=embeddings).as_retriever(
    search_kwargs={"k": 10}
)
query = "What can you tell me about the Celtics?"

# Get relevant documents ordered by relevance score
docs = retriever.get_relevant_documents(query)

# Reorder the documents:
# Less relevant document will be at the middle of the list and more
# relevant elements at beginning / end.
reordering = LongContextReorder()
reordered_docs = reordering.transform_documents(docs)
```

Figure 4.21: Langchain approach for solving lost in the middle problem

useful components that we can utilize are *DiversityRanker* and *LostInTheMiddleRanker*.

> "**DiversityRanker** is designed to maximize the variety of given documents. It does so by selecting the most semantically similar document to the query, then selecting the least similar one, and continuing this process with the remaining documents until a diverse set is formed. It operates on the principle that a diverse set of documents can increase the LLM's ability to generate answers with more breadth and depth."

> "**LostInTheMiddleRanker** sorts the documents based on the "Lost in the Middle" order. The ranker positions the most relevant documents at the beginning and at the end of the resulting list while placing the least relevant documents in the middle."

Please check their documentation for more details.

4.3.4 Embedding Optimization

Optimizing embeddings can have a significant impact on the results of your use cases. There are various APIs and providers of embedding models, each catering to different objectives:

- Some are best suited for coding tasks.
- Others are designed specifically for the English language.
- And there are also embedding models that excel in handling multilingual datasets (e.g., Multilingual BERT/mBERT).

However, determining which embedding model is the best fit for your dataset requires an effective evaluation method.

So which embedding models should we use?

One approach is to rely on existing academic benchmarks. However, it's important to note that these benchmarks may not fully capture the real-world usage of retrieval systems in AI use cases. They are often synthetic benchmarks specifically designed for information retrieval problems.

For example, there is a benchmark called MTEB (Massive Text Embedding Benchmark). MTEB's leaderboard showcases embedding models across 8 tasks, including multilingual tasks, and currently features 132 models. You can compare the performance, speed, or both for these models (refer to the graph below: Figure 4.22).

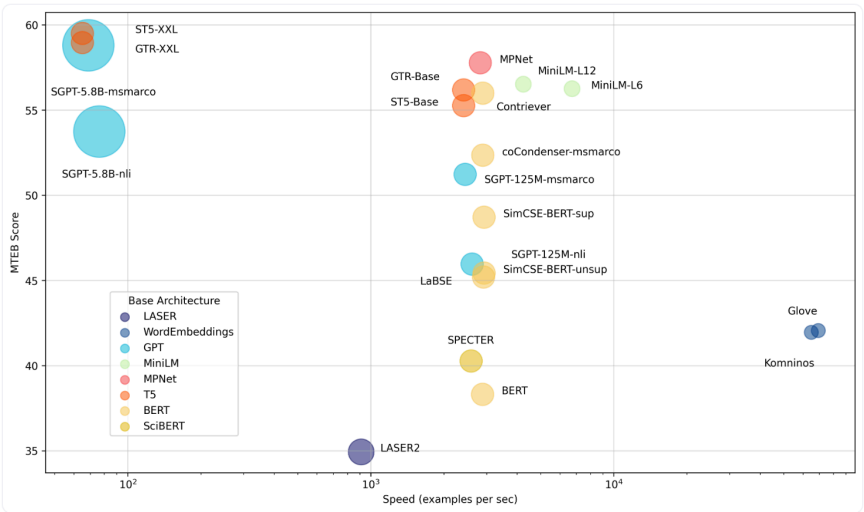


Figure 4.22: Models by average English MTEB score (y) vs speed (x) vs embedding size (circle size). Image source

For better results, you can still utilize open-source tools by applying them to your specific data and use case. Additionally, you can enhance relevance by incorporating human feedback through a simple relevance feedback endpoint.

Constructing your own datasets is also important as you have a deep under-

standing of your production data, relevant metrics, and what truly matters to you. This allows you to tailor the training and evaluation process to your specific needs.

In terms of evaluating the performance of embedding models, there are excellent evaluation tools available in the market. These tools can help you assess the effectiveness of different models and make informed decisions.

It is worth noting that recent research and experiments have shown that embedding models with the same training objective and similar data tend to learn very similar representations, up to an affine linear transform. This means that it is possible to project one model's embedding space into another model's embedding space using a simple linear transform.

By understanding and leveraging linear identifiability, you can explore ways to transfer knowledge between embedding models and potentially improve their performance in specific tasks."

This is called linear identifiability, and it was discussed in the 2020 paper by Roeder and Kingma (2021) On Linear Identifiability of Learned Representations from Google Brain. The paper states,

> *"We demonstrate that as one increases the representational capacity of the model and dataset size, learned representations indeed tend towards solutions that are equal up to only a linear transformation."*

See an example below:

Therefore, the selection of a particular embedding model may not be that important if you are able to discover and implement a suitable transformation from your own dataset.

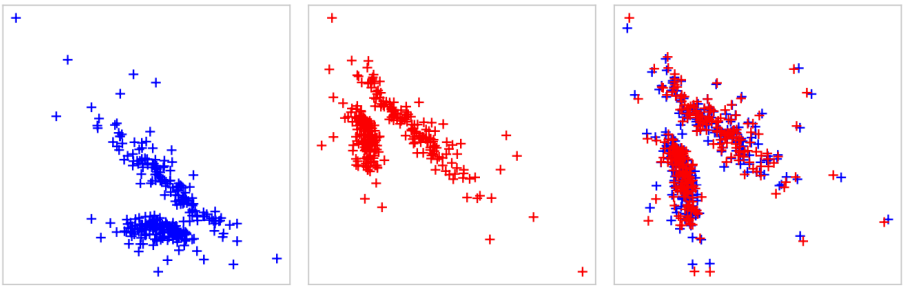


Figure 1: Two distinct 2-D representation functions parameterized by deep neural networks $f_{\theta_1}(\mathcal{B})$, $f_{\theta_2}(\mathcal{B})$ on a subset $\mathcal{B}$ of validation dataset $\mathcal{D}$. We train on LM1B [Chelba et al., 2013] using a word embedding model from Mnih and Teh [2012] (see Appendix A.1 for training details and code URL). The rightmost pane shows $\boldsymbol{A} f_{\theta_1}(\mathcal{B})$ and $f_{\theta_2}(\mathcal{B})$, where $\boldsymbol{A}$ is a linear transformation learned after training. This model exhibits *linear identifiability* (see Section 3): different representation functions, learned on the same data distribution, live within linear transformations of each other.

Figure 4.23: Linear Identifiability. Image source

4.4 Rethinking Retrieval Methods for Heterogeneous Document Corpora

Retrieval-augmented generation (RAG) applications, especially when dealing with a substantial volume of documents (e.g. having many pdf files), often face challenges related to performance, relevance, and latency.

Example: Assume a user asks a question and the answer to user's question only involves two pdf files, we would rather first get those two relevant pdf documents and then find the actual answer from their chunks, instead of searching through thusands of text chunks. But how to do that?

There are multiple ways to achieve that goal:

- Have multi-level embeddings, i.e. embed document summaries, where each document summary is related to its text chunks. This approach is implemented in Section 4.3.1.

- Add metadata about each document and store that along with the document in the vector database.

4.4.1 How metadata can help

Including metadata in the indexing process can be a powerful strategy to address these challenges and significantly enhance the overall system's efficiency. By narrowing down the search scope using metadata filters, the system can reduce the number of documents to be considered, resulting in faster retrieval times.

Figure 4.24 shows how metadata filtering can help the retrieval process. *When user asks a question, they can explicitly give metadata for instance by specifying filters such as dropdown list, etc., or we can use LLM to determine the metadata filters from the query and search the vector database using the filters. Vector database utilizes the filters to narrow down search to the documents that match with the filters, and then finds the most similar chunks from documents and returns the top-k chunks.*

Please note that although we can add metadata to the text chunks after they are stored in the vector database, we should do that in the preprocessing step while we are splitting and embedding the documents, because if the vector database index becomes very large (i.e. we already have a great deal of embeddings in vector database), updating it will be significantly time consuming.

Use Langchain for Metadata Filtering

Figure 4.25 shows how to define new metadata for text chunks, and how to use filters to perform retrieval.

In this example, we load two pdf files, one file is about machine learning interview questions, and the other file is a research paper. We would like to add a topic or category of each file as metadata, so later we can restrict our search to

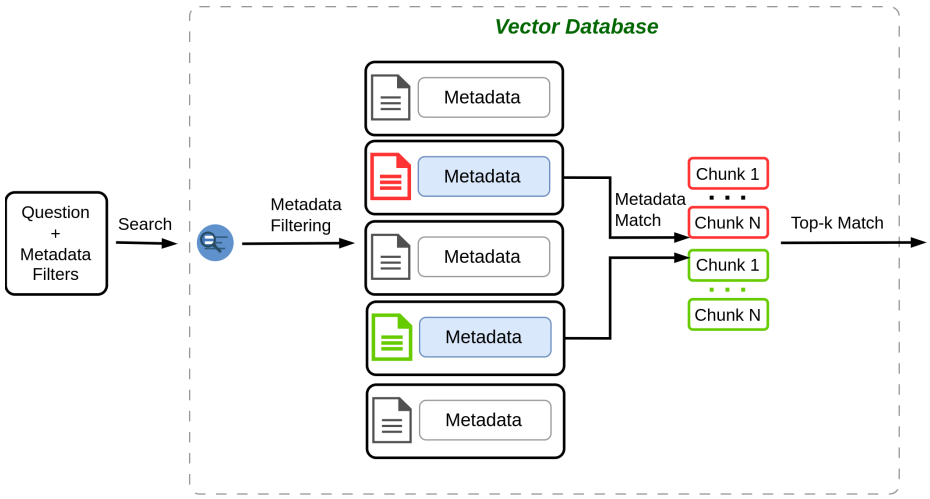


Figure 4.24: How metadata filtering can improve retrieval process

the category. Therefore, we update the initial metadata field by adding a `category` property to each text chunk and then store them in the vector database.

If we print out the `metadata` of documents, we can see the `category` property has been added, shown in Figure 4.26.

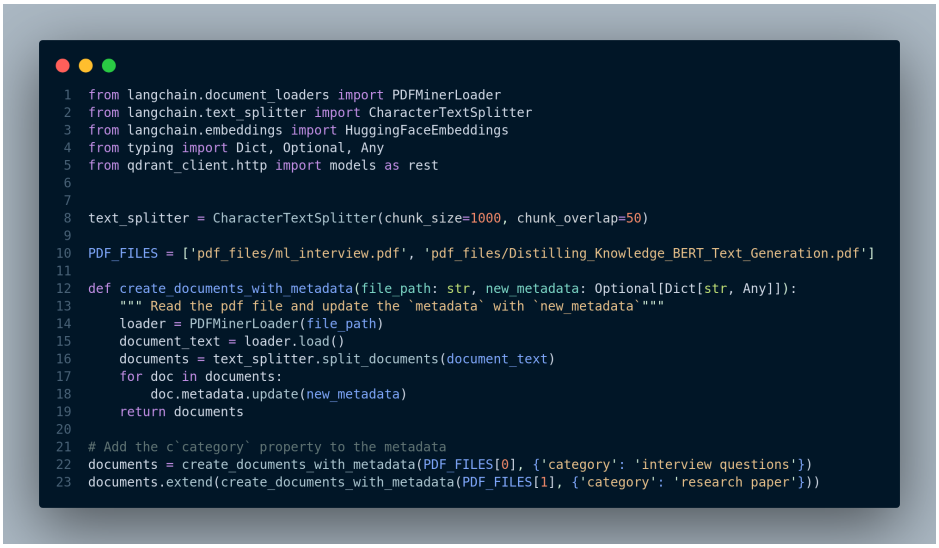
Now, we define the index and use it to perform search and retrieval, which is shown in Figure 4.27.

Use LlamaIndex for Metadata Filtering

Figure 4.28 shows how to use LlamaIndex for metadata filtering.

How to let LLM infer the metadata from user question

In both of the previous techniques, we have to explicitly define the metadata/filters while doing the retrieval. However, the questions is: *Can we ask*



```python
from langchain.document_loaders import PDFMinerLoader
from langchain.text_splitter import CharacterTextSplitter
from langchain.embeddings import HuggingFaceEmbeddings
from typing import Dict, Optional, Any
from qdrant_client.http import models as rest


text_splitter = CharacterTextSplitter(chunk_size=1000, chunk_overlap=50)

PDF_FILES = ['pdf_files/ml_interview.pdf', 'pdf_files/Distilling_Knowledge_BERT_Text_Generation.pdf']

def create_documents_with_metadata(file_path: str, new_metadata: Optional[Dict[str, Any]]):
    """ Read the pdf file and update the `metadata` with `new_metadata`"""
    loader = PDFMinerLoader(file_path)
    document_text = loader.load()
    documents = text_splitter.split_documents(document_text)
    for doc in documents:
        doc.metadata.update(new_metadata)
    return documents

# Add the c`category` property to the metadata
documents = create_documents_with_metadata(PDF_FILES[0], {'category': 'interview questions'})
documents.extend(create_documents_with_metadata(PDF_FILES[1], {'category': 'research paper'}))
```

Figure 4.25: Read the files and update the metadata property

```python
for d in documents:
    print(d.metadata)

# {'source': 'pdf_files/ml_interview.pdf', 'category': 'interview questions'}
# ...
# {'source': 'pdf_files/Distilling_Knowledge_BERT_Text_Generation.pdf', 'category': 'research paper'}
```

Figure 4.26: Output example of metadata for text chunks

```
embedding_model = HuggingFaceEmbeddings(model_name = 'sentence-transformers/all-mpnet-base-v2', m
odel_kwargs = {'device': 'cpu'})

qdrant = Qdrant.from_documents(
    documents,
    embedding_model,
    location=":memory:",  # Local mode with in-memory storage only
    collection_name="qa_index",
    force_recreate=True,
)

# Define a filter. Filter below only searches through the
# text chunks belonging to `interview questions` pdf file
filter={'category': 'interview questions'}

query = "What is k-nearest neighbor"

# return the top-2 relevant chunks after metadata filtering is done
found_docs = qdrant.similarity_search(query, k=2, filter=filter)
```

Figure 4.27: Insert text chunks into the vector database and perform retrieval

LLM to infer the metadata from user query? The short answer is: *Yes.*

Therefore, the general approach is we need to define a particular prompt for LLM, so it can use that to extract entities or metadata from the user query, map them to the existing metadata stored with text chunks in the vector database, and then perform the retrieval.

That said, LlamaIndex provides us an implemented version of this approach. The following code is from the LlamaIndex documentation here. For this technique, we define the metadata (in this example `category` and `country`) along with each text chunk. Figure 4.29 shows the code snippet for this step.

Figure 4.30 shows the retrieval process including how to define vector index, vector store, and `VectorIndexAutoRetriever` object.

```python
from llama_index import ServiceContext, VectorStoreIndex
from llama_index.storage.storage_context import StorageContext
from llama_index.vector_stores.qdrant import QdrantVectorStore
from llama_index.vector_stores.types import ExactMatchFilter, MetadataFilters
from llama_index import SimpleDirectoryReader
from qdrant_client import QdrantClient

# Load the pdf and create a list of Document objects
documents = SimpleDirectoryReader(
    input_dir='pdf_files'
).load_data()

for i,d in enumerate(documents):
    if d.metadata['file_name'] == 'ml_interview.pdf':
        d.metadata['category'] = 'interview questions'
    else:
        d.metadata['category'] = 'research paper'


# Define qdrant vector database client
client = QdrantClient(
    # you can use :memory: mode for fast and light-weight experiments,
    location=":memory:"
    # otherwise set Qdrant instance address with:
    # uri="http://<host>:<port>"
    # set API KEY for Qdrant Cloud
    # api_key="<qdrant-api-key>",
)

service_context = ServiceContext.from_defaults()

# Create vector database with an index
vector_store = QdrantVectorStore(client=client, collection_name="qa_collection")
storage_context = StorageContext.from_defaults(vector_store=vector_store)

# Insert docuemtns into vector database
index = VectorStoreIndex.from_documents(
    documents, storage_context=storage_context, service_context=service_context
)

# Define filter(s)
# Filter below only searches through the text chunks that `category` = 'research paper`
filters = MetadataFilters(filters=[
    ExactMatchFilter(
        key="category",
        value="research paper"
    )
])

# Retrieve the top-3 results
retriever = index.as_retriever(similarity_top_k=3, filters=filters)
results = retriever.retrieve("how bert distilation works?")
```

Figure 4.28: Metadata filtering in LlamaIndex for document retrieval

```python
from llama_index import VectorStoreIndex, StorageContext
from llama_index.vector_stores import ChromaVectorStore

from llama_index.schema import TextNode

# Define the text nodes with metadata
nodes = [
    TextNode(
        text=(
            "Michael Jordan is a retired professional basketball player,"
            " widely regarded as one of the greatest basketball players of all"
            " time."
        ),
        metadata={
            "category": "Sports",
            "country": "United States",
        },
    ),
    TextNode(
        text=(
            "Angelina Jolie is an American actress, filmmaker, and"
            " humanitarian. She has received numerous awards for her acting"
            " and is known for her philanthropic work."
        ),
        metadata={
            "category": "Entertainment",
            "country": "United States",
        },
    )
]
```

Figure 4.29: Define text node and metadata for auto retrieval

```python
from llama_index.indices.vector_store.retrievers import (
    VectorIndexAutoRetriever,
)
from llama_index.vector_stores.types import MetadataInfo, VectorStoreInfo

vector_store = ChromaVectorStore(chroma_collection=chroma_collection)
storage_context = StorageContext.from_defaults(vector_store=vector_store)

index = VectorStoreIndex(nodes, storage_context=storage_context)

# Create VectorStoreInfo, which contains a structured description of the
# vector store collection and the metadata filters it supports.
vector_store_info = VectorStoreInfo(
    content_info="brief biography of celebrities",
    metadata_info=[
        MetadataInfo(
            name="category",
            type="str",
            description=(
                "Category of the celebrity, one of [Sports, Entertainment,"
                " Business, Music]"
            ),
        ),
        MetadataInfo(
            name="country",
            type="str",
            description=(
                "Country of the celebrity, one of [United States, Barbados,"
                " Portugal]"
            ),
        ),
    ],
)

# Define VectorIndexAutoRetriever module.
retriever = VectorIndexAutoRetriever(
    index, vector_store_info=vector_store_info
)

# Perform retrieval
retriever.retrieve("Tell me about two celebrities from United States")
```

Figure 4.30: Define VectorIndexAutoRetriever retriever and VectorStoreInfo, which contains a structured description of the vector store collection and the metadata filters it supports.

4.5 Hybrid Document Retrieval

Hybrid document retrieval is an approach that combines traditional keyword-based search like BM25 with semantic (dense) search using embeddings, such as BERT or word2vec. Integrating this technique to Retrieval-Augmented Generation (RAG) applications can significantly enhance the effectiveness of document retrieval. It addresses scenarios where a basic keyword-based approach can outperform semantic search and demonstrates how combining these methods improves retrieval in RAG applications.

In addition, often times a complete migration to a semantic-based search using RAG is challenging for most companies. They might already have a keyword-based search system and have been using it for quite a long time. Performing an overhaul to the company's information architecture, and migrating to a vector database is just infeasible.

Scenarios Favoring Keyword-Based Search:

1. **Highly Specific Queries:** In cases where a user's query is highly specific and focuses on precise terms or phrases, a keyword-based approach can outperform semantic search. Keyword matching excels at finding exact matches within documents.

2. **Niche Domains:** In specialized domains with industry-specific jargon or acronyms, keyword search can be more effective as it directly matches terminology without requiring extensive semantic understanding.

3. **Short Documents:** When dealing with very short documents, such as tweets or headlines, keyword-based search can be more efficient. Semantic models often require longer text to derive meaningful embeddings.

4. **Low Resource Usage:** Keyword search typically requires fewer computational resources compared to semantic search. This can be advantageous when resource efficiency is a critical concern.

Combining Keyword and Semantic Approaches for Improved Retrieval:

To harness the strengths of both keyword-based and semantic (dense) retrievers effectively, a pragmatic approach is to integrate two retrievers into the pipeline and merge their outputs. This two-pronged strategy capitalizes on the unique advantages of each retriever, resulting in more comprehensive and accurate retrieval in Retrieval-Augmented Generation (RAG) applications.

The process begins by employing both **keyword-based** as well as **dense** retrievers within the RAG pipeline. However, the challenge lies in merging the results obtained from these two retrievers, each of which returns ranked lists of results with relevance scores assigned to each document. Figure 4.31 illustrates the hybrid retrieval pipeline.

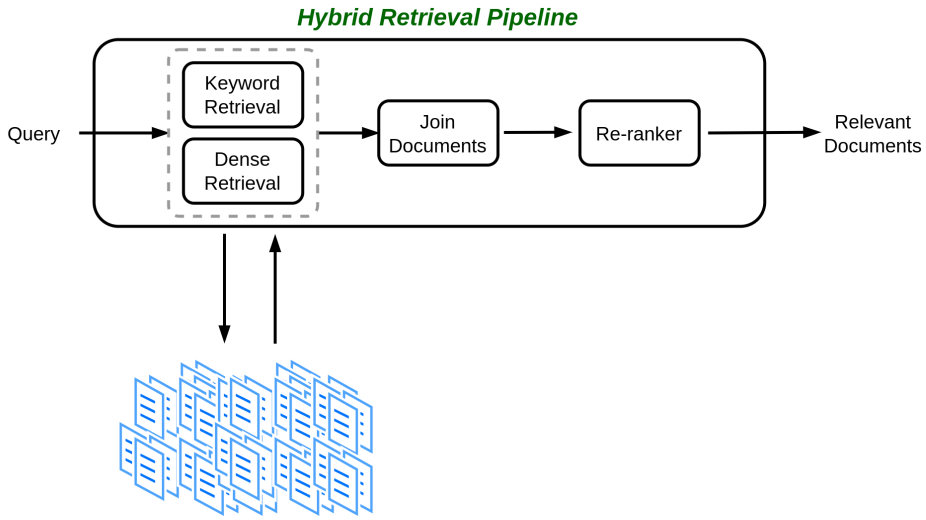


Figure 4.31: Hybrid retrieval pipeline

Merging the results from keyword-based and semantic retrievers can be approached in several ways, depending on the nature of the RAG application:

1. **Concatenation:** This method involves simply appending all documents from both retrievers (excluding duplicates) to create the final list of results. Concatenation is suitable when you intend to use all retrieved documents and the order of the results is not crucial. This approach can be valuable in extractive question-answering pipelines, where you aim to extract information from multiple sources and are less concerned about ranking.

2. **Reciprocal Rank Fusion (RRF):** RRF operates with a formula that reranks documents from both retrievers, giving priority to those that appear in both results lists. Its purpose is to elevate the most relevant documents to the top of the list, thereby enhancing the overall relevance of the results. RRF is particularly useful when the order of the results is important or when you intend to pass on only a subset of results to the subsequent processing stages.

3. **Merging by Scoring:** In this approach, documents are ranked based on the scores assigned by the individual retrievers. This method is suitable when you aim to prioritize results from one retriever over the other. If the relevance scores assigned by the two retrievers are comparable and you wish to emphasize results from a particular retriever, this method can be employed. For instance, if you're using dense retrievers from different sources that return documents from various document stores, this method allows you to choose one retriever's output over another.

Advantages of Hybrid Retrieval:

- **Enhanced Relevance:** Hybrid retrieval leverages the strengths of both keyword and semantic approaches, increasing the chances of returning highly relevant documents.

- **Coverage:** It addresses scenarios where purely keyword or purely semantic approaches might fail, providing a broader scope of relevant documents.

- **Resource Efficiency:** By initially narrowing the search with keyword-based filtering, the system conserves computational resources, making the retrieval process more efficient.
- **Adaptability:** Hybrid retrieval allows for adaptability to different user queries and document types, striking a balance between precision and recall.

There are frameworks which support hybrid retrieval out of the box such as ElasticSearch, Haystack, Weaviate, and Cohere Rerank. Let's find out how to implement this approach using Haystack. The following code is from Haystack documentation, you can see all the implementation details here. The documents that are used for this example are abstracts of papers from PubMed. You can find and download the dataset here.

```
pip install datasets>=2.6.1
pip install farm-haystack[inference]
```

Step 1: We load the dataset and initialize the document store (i.e. vector database). Figure 4.32 shows this step.

Step 2: Define the retrievers, insert the documents and embeddings into the document store and choose the *join document* strategy. You can see this step in Figure 4.33.

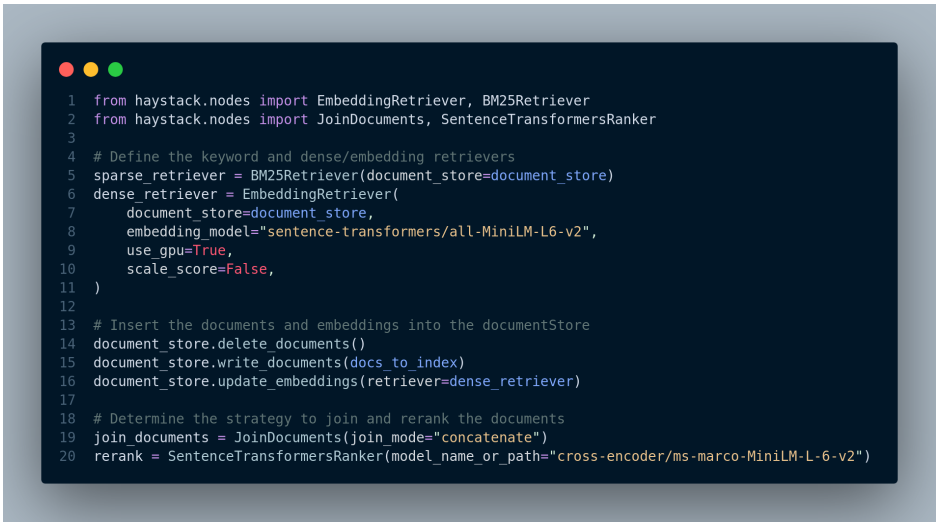
Step 3: Create an end-to-end pipeline in Haystack and perform the hybrid retrieval for a query. This step is depicted in Figure 4.34.

4.6 Query Rewriting for Retrieval-Augmented Large Language Models

Query rewriting is a sophisticated technique that plays a pivotal role in enhancing the performance of Retrieval-Augmented Large Language Models (RAG). The fundamental idea behind query rewriting is to optimize and fine-tune the

```python
from datasets import load_dataset
from haystack.document_stores import InMemoryDocumentStore
from haystack.schema import Document
from haystack.nodes import PreProcessor

# Load the dataset
dataset = load_dataset("ywchoi/pubmed_abstract_3", split="test")

# Initialize the document store/vector database
document_store = InMemoryDocumentStore(use_bm25=True, embedding_dim=384)

# Create a list of text documents with some metadata
documents = []
for doc in dataset:
    documents.append(
        Document(
            content=doc["title"] + " " + doc["text"],
            meta={"title": doc["title"], "abstract": doc["text"], "pmid": doc["pmid"]},
        )
    )

# Clean and split the text
preprocessor = PreProcessor(
    clean_empty_lines=True,
    clean_whitespace=True,
    clean_header_footer=True,
    split_by="word",
    split_length=512,
    split_overlap=32,
    split_respect_sentence_boundary=True,
)

docs_to_index = preprocessor.process(documents)
```

Figure 4.32: Load documents and initialize document store



```python
from haystack.nodes import EmbeddingRetriever, BM25Retriever
from haystack.nodes import JoinDocuments, SentenceTransformersRanker

# Define the keyword and dense/embedding retrievers
sparse_retriever = BM25Retriever(document_store=document_store)
dense_retriever = EmbeddingRetriever(
    document_store=document_store,
    embedding_model="sentence-transformers/all-MiniLM-L6-v2",
    use_gpu=True,
    scale_score=False,
)

# Insert the documents and embeddings into the documentStore
document_store.delete_documents()
document_store.write_documents(docs_to_index)
document_store.update_embeddings(retriever=dense_retriever)

# Determine the strategy to join and rerank the documents
join_documents = JoinDocuments(join_mode="concatenate")
rerank = SentenceTransformersRanker(model_name_or_path="cross-encoder/ms-marco-MiniLM-L-6-v2")
```

Figure 4.33: Define keyword and embedding based retrievers

queries presented to the retrieval component of RAG systems, ultimately leading to more accurate and contextually relevant results.

The core concept is to transform the initial user query into an improved form that effectively captures the user's intent and aligns with the document retrieval phase. This often involves various steps and considerations, including:

1. **Expanding User Queries**: Query rewriting can involve expanding the initial user query by adding synonyms, related terms, or concepts that might improve the retrieval of relevant documents. This expansion can be based on linguistic analysis or external knowledge sources.

2. **Rephrasing for Clarity**: Queries are often rewritten to improve their clarity and conciseness. Ambiguous or complex phrasings can be simplified to make the user's intent more explicit.

3. **Contextual Adaptation**: The rewriting process may take into account the specific context of the documents available for retrieval. It can adapt

```python
from haystack.pipelines import Pipeline

# Define a pipeline and connect all the components end to end
pipeline = Pipeline()
pipeline.add_node(component=sparse_retriever, name="SparseRetriever", inputs=["Query"])
pipeline.add_node(component=dense_retriever, name="DenseRetriever", inputs=["Query"])
pipeline.add_node(component=join_documents, name="JoinDocuments", inputs=["SparseRetriever", "DenseRetriever"])
pipeline.add_node(component=rerank, name="ReRanker", inputs=["JoinDocuments"])

# Run the retriever for a given query
prediction = pipeline.run(
    query="treatment for HIV",
    params={
        "SparseRetriever": {"top_k": 10},
        "DenseRetriever": {"top_k": 10},
        "JoinDocuments": {"top_k_join": 15},  # comment for debug
        # "JoinDocuments": {"top_k_join": 15, "debug":True}, #uncomment for debug
        "ReRanker": {"top_k": 5},
    },
)

def pretty_print_results(prediction):
    for doc in prediction["documents"]:
        print(doc.meta["title"], "\t", doc.score)
        print(doc.meta["abstract"])
        print("\n", "\n")

# Print out the retrieved documents
pretty_print_results(prediction)
```

Figure 4.34: Create end-to-end pipeline and run the retrievers

the query to the characteristics of the document corpus, which is particularly valuable in domain-specific applications.

Query rewriting is closely tied to the document retrieval phase in RAG systems. It contributes to better retrieval rankings, which, in turn, leads to more informative and contextually relevant answers generated by the language model. The goal is to ensure that the retrieved documents align closely with the user's intent and cover a wide spectrum of relevant information.

4.6.1 Leveraging Large Language Models (LLMs) for Query Rewriting in RAGs

Question: *With the advent of Large Language Models (LLMs) that have revolutionized natural language understanding and generation tasks, can we use them for query rewriting?* The answer is: *Yes.* They can help in two primary ways: query expansion and generating better prompts. Figure 4.36 shows how query rewriting works. We can use LLM to either expand (enhance) a query or generate multiple (sub)queries for better retreival process.

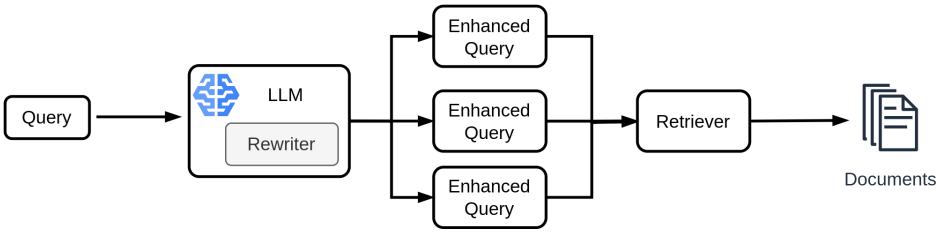


Figure 4.35: Query re-writing using LLMs. LLM can expand the query or create multiple sub-queries.

4.6.1.1 Query Expansion with LLMs

LLMs possess an extensive understanding of language and vast knowledge repositories, which makes them ideal for query expansion. Here's how LLMs

can aid in query expansion:

Synonym Generation: LLMs can identify synonyms and related terms for words in the user's query. By expanding the query with synonyms, it increases the chances of retrieving documents that may use different terminology but are contextually relevant.

Example: *User Query:* "Renewable energy sources". *LLM Query Expansion:* "Renewable energy sources" -> "Green energy sources," "Sustainable energy sources," "Eco-friendly energy sources".

By suggesting synonyms for "renewable," the LLM broadens the query's scope to retrieve documents that may use alternative terminology.

Conceptual Expansion: LLMs can identify concepts and entities related to the query. They can suggest adding relevant concepts, entities, or phrases that can lead to more comprehensive results. For instance, if a user queries about "climate change," the LLM might suggest adding "global warming" to ensure a broader document retrieval.

Example: *User Query:* "Mars exploration". *LLM Query Expansion:* "Mars exploration" -> "Mars mission," "Red planet research," "Space exploration of Mars".

Multilingual Query Expansion: For multilingual queries, LLMs can assist in translating and expanding the query into multiple languages, broadening the search scope and retrieving documents in various languages.

4.6.1.2 Generating Better Prompts with LLMs

LLMs can assist in generating more effective prompts for retrieval, especially when a prompt-based retrieval mechanism is employed. Here's how LLMs contribute to prompt generation.

Query Refinement: LLMs can refine a user query by making it more concise, unambiguous, and contextually relevant. The refined query can then serve as a prompt for the retrieval component, ensuring a more focused search.

Example: *User Query:* "How does photosynthesis work?" *LLM-Generated Prompt:* "Explain the process of photosynthesis."

Multi-step Prompts: In complex queries, LLMs can generate multi-step prompts that guide the retrieval component through a series of sub-queries. This can help break down intricate requests into more manageable retrieval tasks.

Example: *User Query:* "Market trends for electric cars in 2021". *LLM-Generated Multi-Step Prompts:*

- "Retrieve market trends for electric cars."
- "Filter results to focus on trends in 2021."

The LLM generates sequential prompts to guide the retrieval component in finding documents related to market trends for electric cars in 2021.

Context-Aware Prompts: LLMs can consider the context of the available document corpus and generate prompts that align with the characteristics of the documents. They can adapt prompts for specific domains or industries, ensuring that the retrieval phase retrieves contextually relevant documents.

Example: *User Query:* "Legal documents for the healthcare industry". *LLM-Generated Domain-Specific Prompt:* "Retrieve legal documents relevant to the healthcare industry."

Understanding the context, the LLM generates a prompt tailored to the healthcare industry, ensuring documents retrieved are pertinent to that domain.

Query rerwriting for RAGs is an active research area, and new approaches are suggested regularly. One recent research is Ma et al. (2023), where they propose a new framework for query generation. See here to learn more about their method.

4.7 Query Routing in RAG

Query routing in RAG, often facilitated by a router, is the process of automatically selecting and invoking the most suitable retrieval technique or tool for a given user query. It enables a dynamic, adaptive approach to choose how to retrieve information based on the specific requirements of each query.

Rather than relying on a fixed retrieval method, query routing empowers the system to intelligently assess the user's query and select the appropriate retrieval mechanism. This approach is particularly powerful in scenarios where a diverse range of retrieval techniques or tools can be employed to answer different types of queries. The following shows the generate architecture of this approach.

How Query Routing Works:

1. **User Query Input**:

   A user enters a query into the RAG system. This query could encompass various types of information needs, such as fact-based lookup, summarization, translation, question answering, etc.

2. **Query Analysis**:

   The router or the query routing component first performs an analysis of the user's query. This analysis involves understanding the nature of the query, its intent, and the type of information required.

3. **Detection of Retrieval Technique Requirement**:

   Based on the analysis, the router detects the retrieval technique or tool that best matches the query's requirements. This detection is often driven by heuristics, pre-defined rules, machine learning models, or a combination of these methods.

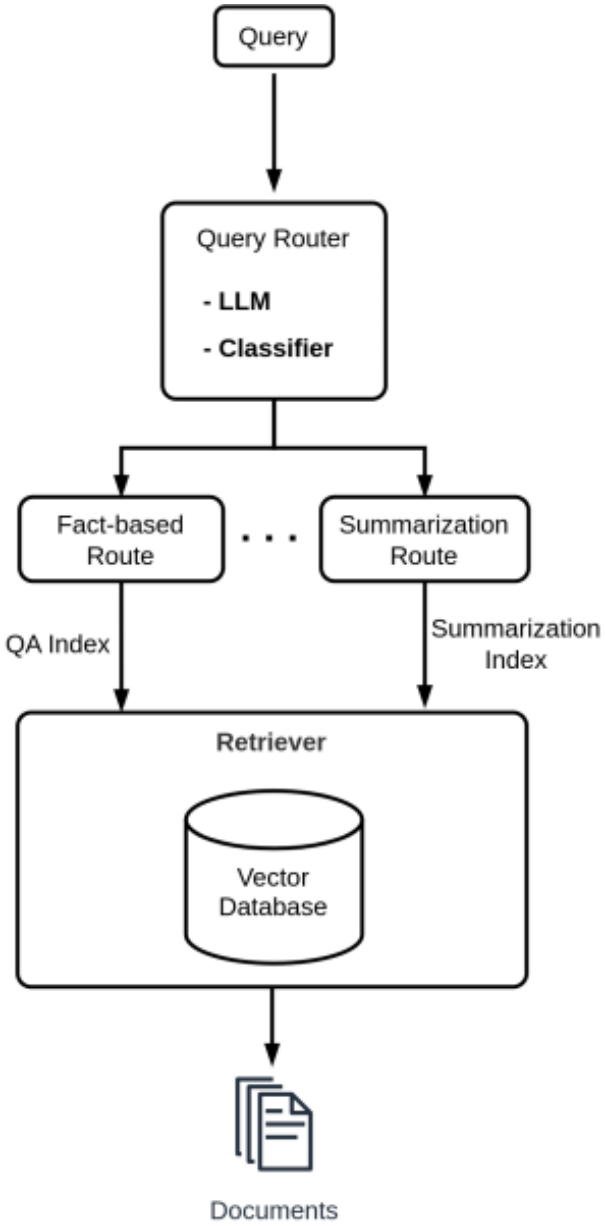


Figure 4.36: Query router architecture

4. **Selection of Retrieval Technique**:

   The router selects the most appropriate retrieval technique from a predefined set, which can include methods like fact-based lookup in a vector store, summarization, document retrieval, question answering, or translation, among others.

5. **Invoke the Retrieval Component**:

   The router then calls the relevant retrieval component or "tool" that specializes in the chosen retrieval technique. This component may be a vector store interface, a summarization tool, a translation service, or any other retrieval method.

6. **Information Retrieval**:

   The selected retrieval component performs the necessary information retrieval or processing based on the chosen technique. For example, if fact-based lookup is required, it retrieves facts from a vector store. If summarization is needed, it generates a concise summary. If translation is the goal, it translates the content.

7. **Generation of Output**:

   The retrieved or processed information is then used by the generation component in the RAG system to compose a response, which is presented to the user.

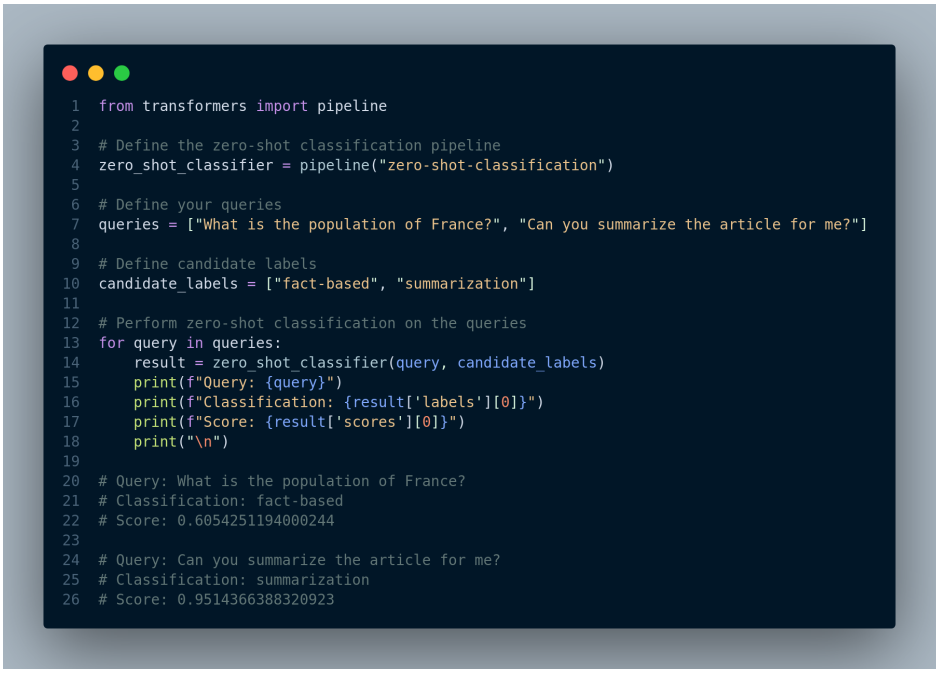
There are multiple ways to implement query routing in Retrieval-Augmented Generation (RAG) systems:

1. **Intent Classification Model:**

   - **Use a Classifier**: Employ a pre-trained classifier model that can categorize user queries based on their intent. This classifier can have predefined categories such as fact-based lookup, summarization, translation, question answering, etc. The selected category then dictates the retrieval method to be used.

- **Machine Learning**: Train a custom intent classification model on labeled query data to predict the query's intent. This model can be fine-tuned on specific intent detection tasks relevant to the RAG system.

Figure 4.37 shows how to use a zero-shot classifer to categorize user queries.



```python
from transformers import pipeline

# Define the zero-shot classification pipeline
zero_shot_classifier = pipeline("zero-shot-classification")

# Define your queries
queries = ["What is the population of France?", "Can you summarize the article for me?"]

# Define candidate labels
candidate_labels = ["fact-based", "summarization"]

# Perform zero-shot classification on the queries
for query in queries:
    result = zero_shot_classifier(query, candidate_labels)
    print(f"Query: {query}")
    print(f"Classification: {result['labels'][0]}")
    print(f"Score: {result['scores'][0]}")
    print("\n")

# Query: What is the population of France?
# Classification: fact-based
# Score: 0.6054251194000244

# Query: Can you summarize the article for me?
# Classification: summarization
# Score: 0.9514366388320923
```

Figure 4.37: Using a zero-shot classifier to categorize and route user queries

2. **Prompt-Based Routing**:

   - **Leverage LLMs**: Utilize Large Language Models (LLMs), such as GPT-4 or similar models, to perform query classification. A prompt can be designed to guide the LLM in classifying the query based on its intent.

- **Template Prompts**: Create a set of template prompts that are specifically designed to categorize queries. These templates can include leading questions or cues to elicit the intent of the query.

The code in Figure 4.38 shows how to use a LLM `text-davinci-002` for query routing.

```python
import openai

# Set your OpenAI API key
api_key = OPENAI_API_KEY

# Define your queries
queries = ["What is the capital of France?", "Summarize the key points of the Paris Agreement."]

# Create a prompt for zero-shot classification
prompt = """ Classify this query as either fact-based or summarization:\n\n 0: fact-based\n 1: summarization\n\n Query: """

# Perform zero-shot classification for each query
for query in queries:
    classification_input = f"{prompt} {query}"
    # print(classification_input)
    response = openai.Completion.create(
        engine="text-davinci-002",
        prompt=classification_input,
        max_tokens=10,
        api_key=api_key,
    )

    # Determine the category based on the model's response
    category = response.choices[0].text.strip()

    print(f"Category:{category}")

# Category:0: fact-based
# Category:1: summarization
```

Figure 4.38: Using a LLM to categorize and route user queries

3. **Rule-Based Routing**:

   - **Rule-Based System**: Develop a rule-based system that consists of predefined rules or conditions to categorize queries. For example, if a query starts with "Translate," it is routed to a translation retrieval method.

- **Regular Expressions**: Use regular expressions to match query patterns and automatically route queries based on predefined patterns, keywords, or structures.

The choice of implementation depends on the complexity of the RAG system, the available resources, and the specific requirements of the application. Implementing a combination of these methods can provide a robust and adaptive query routing mechanism that enhances the overall performance of RAG systems.

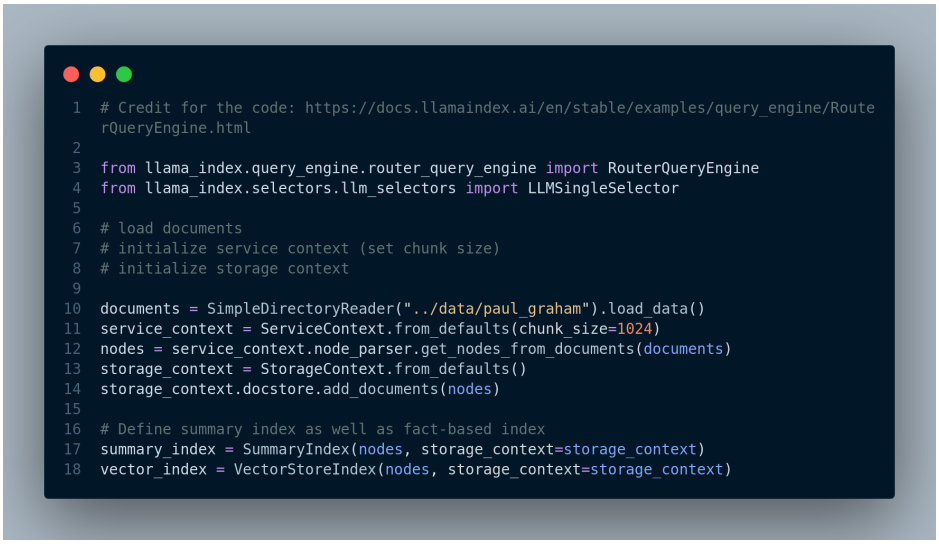
Let's take a look at an example where we use LlamaIndex framework for routing queries between a *summarization route* and *fact-based route*. LlamaIndex has a concept called `RouterQueryEngine` which accepts a set of query engines `QueryEngineTool` objects as input. A `QueryEngineTool` is essentially an index used for retrieving documents from vector database. First step is to load the documents and define the summary index and fact-based index, which is displayed in Figure 4.39.

Then, we create `QueryEngine` objects for each index and add them to the `RouterQueryEngine` object. Figure 4.40 shows this step.

By dynamically routing queries to the most suitable retrieval techniques, query routing enhances RAG by ensuring that the system adapts to the specific requirements of each user query, providing more accurate and context-aware responses. It overcomes the challenge of needing to know in advance which retrieval technique to apply and optimizes the retrieval process for different types of information needs.

4.8 Leveraging User History to Enhance RAG Performance

Retrieval-Augmented Generation (RAG) systems have gained prominence in natural language processing applications due to their ability to combine the



```python
# Credit for the code: https://docs.llamaindex.ai/en/stable/examples/query_engine/RouterQueryEngine.html

from llama_index.query_engine.router_query_engine import RouterQueryEngine
from llama_index.selectors.llm_selectors import LLMSingleSelector

# load documents
# initialize service context (set chunk size)
# initialize storage context

documents = SimpleDirectoryReader("../data/paul_graham").load_data()
service_context = ServiceContext.from_defaults(chunk_size=1024)
nodes = service_context.node_parser.get_nodes_from_documents(documents)
storage_context = StorageContext.from_defaults()
storage_context.docstore.add_documents(nodes)

# Define summary index as well as fact-based index
summary_index = SummaryIndex(nodes, storage_context=storage_context)
vector_index = VectorStoreIndex(nodes, storage_context=storage_context)
```

Figure 4.39: Query routing example in LlamaIndex. First we load documents and create different indecies.

```python
# get list_query_engine (summary engine) and vector_query_engine (fact-based engine)
list_query_engine = summary_index.as_query_engine(
    response_mode="tree_summarize",
    use_async=True,
)
vector_query_engine = vector_index.as_query_engine()

# define tool over summarization/vector query engines
list_tool = QueryEngineTool.from_defaults(
    query_engine=list_query_engine,
    description='Useful for summarization questions related to Paul Graham eassy on What I Worked On.',
)

vector_tool = QueryEngineTool.from_defaults(
    query_engine=vector_query_engine,
    description='Useful for retrieving specific context from Paul Graham essay on What I Worked On.',
)

# define query engine
query_engine = RouterQueryEngine(
    selector=LLMSingleSelector.from_defaults(),
    query_engine_tools=[
        list_tool,
        vector_tool,
    ]
)

# ask summarization question
response = query_engine.query('What is the summary of the document?')
print(response)

# ask fact-based lookup question
response = query_engine.query('What did Paul Graham do after RISD?')
print(response)
```

Figure 4.40: Define `QueryEngine` and `RouterQueryEngine` objects, and run the engine for user queries.

strengths of information retrieval and language generation. However, RAG applications often involve significant computational and financial costs due to the necessity of embeddings, retrievals and even response generation. When dealing with recurrent user queries or similar questions, these costs can be optimized by leveraging the memory of previously asked questions.

4.8.1 Challenge

In many real-world applications, users tend to ask similar or nearly identical questions repeatedly. For instance, in a customer support chatbot, users may have common queries related to product information or troubleshooting procedures. While the RAG framework excels at generating relevant responses based on retrieval, it may not be efficient to re-embed and retrieve information for the same questions each time they are asked. This is where the concept of leveraging user history comes into play.

4.8.2 How User History Enhances RAG Performance

Leveraging user history to enhance RAG performance involves building a memory or cache of previously asked questions and their corresponding responses. When a user query is received, instead of immediately triggering an embedding and retrieval process, the system can check the user history to see if it has encountered a similar or identical query before. This approach offers several advantages:

1. **Faster Response Times (i.e. reduced latency):** By comparing the new query to the user history, the system can identify duplicate or closely related questions. In such cases, it can bypass the embedding and retrieval steps, significantly reducing response times.

2. **Cost Reduction:** Skipping the resource-intensive retrieval process for recurrent queries leads to a notable reduction in computational and financial costs, as the need for additional API calls or data processing diminishes.

3. **Consistency and Accuracy:** Responses to repeated questions can remain consistent and accurate over time. The use of cached responses from the user history ensures that users receive reliable information without relying on new retrievals. Additionally, by keeping track of user history, RAG can learn to better understand the user's intent and context. This can lead to more accurate answers to questions, even if the questions are not perfectly phrased.

4.8.3 How Memory/User History Works

A simple RAG memory can be implemented as follows: We need to maintain a log of user queries and their corresponding answers. Therefore, we can define the memory as a key-value store, where the keys are questions and the values are answers. When a user asks a question, RAG first checks its memory to see if it has already answered a similar question. If it has, then it simply retrieves the answer from its memory and returns it to the user. If it has not already answered a similar question, then it performs the embedding and retrieval process as usual.

We need to have a mechanism that constantly updates the RAG's memory with new questions and answers. As a user asks more questions, RAG's memory grows and it becomes better able to answer future questions accurately and efficiently.

Implementation: The snippet code in Figure 4.41 shows a very basic implementation of memory for RAG.

While a simple in-memory dictionary, as shown in the previous example, can be effective for maintaining recent user history, it might not be sufficient for handling a large-scale conversation history or long-term memory. To address

```python
class RagMemory:
    def __init__(self):
        # Initialize an empty memory dictionary to store user history.
        self.memory = {}

    def add_to_memory(self, question, answer):
        # Add a question-answer pair to memory.
        self.memory[question] = answer

    def get_from_memory(self, question):
        # Retrieve an answer from memory if the question exists.
        return self.memory.get(question, None)

def answer_question(rag_memory, question):
    # Check if the question is already in memory.
    cached_answer = rag_memory.get_from_memory(question)

    if cached_answer:
        # If the question is in memory, retrieve the answer.
        print("RAG with Memory:", cached_answer)
    else:
        # If the question is not in memory, you can perform retrieval and generation here.
        # For simplicity, we'll just simulate a new answer.
        new_answer = "I don't have a cached response for that. Generating a new response."

        # Add the new question-answer pair to memory for future use.
        rag_memory.add_to_memory(user_input, new_answer)
        print("RAG with Memory:", new_answer)

        return new_answer

# Initialize the RAG system with memory.
rag_memory = RagMemory()
question = "What is the capital of United States?"

answer = answer_question(rag_memory, question)
print(answer)
```

Figure 4.41: A basic key-value implementation of memory for RAG.

these challenges, RAG systems can benefit from using a vector database for memory.

The implementation of a vector database for memory in a RAG system involves several key steps:

1. **Data Ingestion:** Store historical question-answer pairs along with their corresponding embeddings in the vector database. This process typically involves a batch or incremental data ingestion pipeline.
2. **Retrieval:** When a user poses a question, retrieve the most relevant historical answers by calculating the similarity between the question's embedding and the embeddings in the vector database.
3. **Updating Memory:** Periodically update the vector database with new question-answer pairs and their embeddings to keep the memory up to date.
4. **Caching:** Implement a caching mechanism to enhance retrieval speed by temporarily storing frequently accessed data in memory.
5. **Query Optimization:** Use efficient indexing and search algorithms to optimize the retrieval process, reducing query response times.

Incorporating a vector database for memory in a RAG system enhances its ability to provide contextually relevant and coherent responses by efficiently managing and retrieving historical information. This approach is particularly valuable in scenarios where extensive conversation histories or long-term memory are essential for the application's success.

As we wrap up our exploration of advanced RAG systems in this Chapter, we are on the cusp of a new frontier. In Chapter 5 - "Observability Tools for RAG," we will discuss various observability tools tailored for RAG systems. We will explore their integration with LlamaIndex, including Weights & Biases, Phoenix, and HoneyHive. These tools will not only help us monitor and evaluate the performance of our RAG systems but also provide valuable insights for continuous improvement.

5 Observability Tools for RAG

An observability tool is a software or platform designed to help monitor, analyze, and gain insights into the performance, behavior, and health of a complex system, such as a machine learning model, a RAG system, or a software application. These tools provide visibility into various aspects of a system, allowing operators, administrators, and developers to understand how the system operates and to detect and troubleshoot issues.

Key components and features of an observability tool typically include:

1. **Data Collection:** Observability tools collect data from various sources within the system. This data can include metrics (e.g., CPU usage, memory usage), logs, traces, events, and more. The broader the range of data collected, the more comprehensive the observability.

2. **Storage:** Data collected by the tool is stored for analysis and historical reference. The storage can be in the form of time-series databases, log repositories, or other storage solutions designed to handle large volumes of data.

3. **Analysis:** Observability tools provide analytical capabilities to process and make sense of the collected data. This includes querying data, aggregating metrics, and identifying patterns or anomalies.

4. **Visualization:** The tools offer visualization capabilities to create graphs, charts, dashboards, and reports that make it easier for users to interpret data. Visualizations help spot trends, issues, and performance bottlenecks.

5. **Alerting:** Many observability tools allow users to define alerting rules. When certain conditions are met, the tool sends notifications, enabling operators to respond to issues promptly.

6. **Tracing:** For distributed systems, tracing is important. Observability tools often provide tracing features to track requests as they move through various services or components of a system. This helps pinpoint performance bottlenecks and issues.

7. **User Interface:** A user-friendly interface is essential for interacting with the data and insights generated by the observability tool. It should allow users to explore data, set up alerts, and visualize information.

8. **Integration:** Observability tools should integrate with various components of the system, such as applications, databases, containers, and cloud services, to capture relevant data.

9. **Scalability:** The tool should be able to scale with the system it monitors. It needs to handle growing data volumes and provide insights without compromising performance.

10. **Customization:** Users should be able to customize the tool to meet the specific needs of their system. This includes defining custom dashboards, alerts, and data collection methods.

There are several observability tools for RAG based systems. Frameworks like LlamaIndex also provides an easy way to integrate some of these tools with RAG application. This enable us to:

- View LLM prompt inputs and outputs
- Make sure that all the components such as embedding models, LLMs and vector databases are working as expected
- View indexing and querying traces

In order to integrate observability tools with LlamaIndex, we simply need to do the following, Figure 5.1:

Figure 5.1: General pattern for integrating observability tools into LlamaIndex

5.1 Weights & Biases Integration with LlamaIndex

Weights & Biases is a machine learning platform that empowers developers to enhance their models efficiently. It provides versatile tools for experiment tracking, dataset versioning, model evaluation, result visualization, regression identification, and seamless collaboration with peers.

The code depicted in Figure 5.2 shows how to integrate W&B with LlamaIndex. For complete example, please see here.

We can even see the logs as shown in Figure 5.3.

If we go the W&B website and login, we can see all the details, Figure 5.4 displays our project including charts, artifacts, logs, and traces.

5.2 Phoenix Integration with LlamaIndex

Phoenix is an observability tool designed for LLM applications, offering insight into their inner workings. It provides a visual representation of query engine calls and highlights problematic execution spans based on factors like latency and token count, aiding in performance evaluation and optimization.

Figure 5.5 shows the general usage pattern to use Phoenix.

```python
import llama_index
from llama_index import set_global_handler
from llama_index.callbacks import WandbCallbackHandler

set_global_handler("wandb", run_args={"project": "llamaindex"})
wandb_callback = llama_index.global_handler

# Create your index
docs = SimpleDirectoryReader("./data/paul_graham/").load_data()
index = GPTVectorStoreIndex.from_documents(
    docs, service_context=service_context
)

# Persist Index as W&B Artifacts
wandb_callback.persist_index(index, index_name="simple_vector_store")
# OUTPUT: wandb: Adding directory to artifact


# load storage context
storage_context = llama_index.global_handler.load_storage_context(
    artifact_url="ayut/llamaindex/composable_graph:v0"
)

query_engine = index.as_query_engine()
response = query_engine.query("What did the author do growing up?")
print(response, sep="\n")
# OUTPUT
# The author worked on writing and programming outside of school before college. They wrot
e short stories and tried writing programs on an IBM 1401 computer using an early version
of Fortran. They later got a microcomputer and started programming on it, writing simple g
ames and a word processor. They also mentioned their interest in philosophy and AI.
```

Figure 5.2: W&B integration with LlamaIndex

```
wandb: Logging into wandb.ai. (Learn how to deploy a W&B server locally: https://wandb.me/wandb-server)
wandb: You can find your API key in your browser here: https://wandb.ai/authorize
wandb: Paste an API key from your profile and hit enter, or press ctrl+c to quit:
  ........
wandb: Appending key for api.wandb.ai to your netrc file: /home/mehdi/.netrc
wandb: Streaming LlamaIndex events to W&B at https://wandb.ai/mehdi-allahyari/llamaindex/runs/ll24ao1q
wandb: `WandbCallbackHandler` is currently in beta.
wandb: Please report any issues to https://github.com/wandb/wandb/issues with the tag `llamaindex`.
wandb: Adding directory to artifact (/home/mehdi/environments/expr_env/wandb/run-20231026_143712-ll24ao1q/files/storage)... Done
wandb:   3 of 3 files downloaded.
```

Figure 5.3: W&B logs at different steps

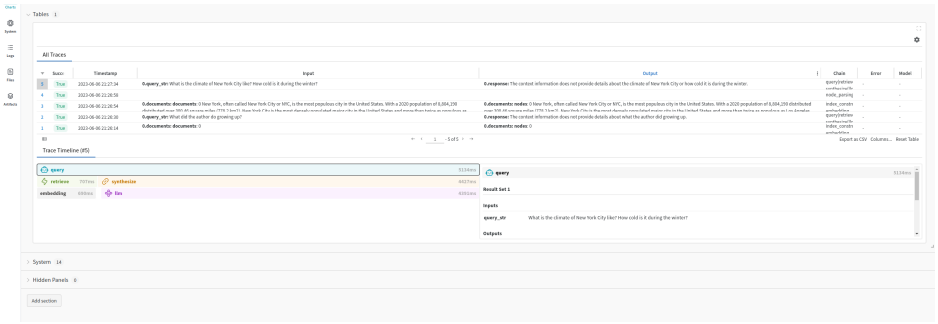


Figure 5.4: W&B dashboard

```python
# Phoenix can display in real time the traces automatically
# collected from your LlamaIndex application.
import phoenix as px
# Look for a URL in the output to open the App in a browser.
px.launch_app()
# The App is initially empty, but as you proceed with the steps below,
# traces will appear automatically as your LlamaIndex application runs.

import llama_index
llama_index.set_global_handler("arize_phoenix")

# Run all of your LlamaIndex applications as usual and traces
# will be collected and displayed in Phoenix.
```

Figure 5.5: Phoenix integration with LlamaIndex RAG applications

When we run queries, we can see the traces in real time in the Phoenix UI. Figure 5.6 illustrates the Phoenix UI for a RAG application.

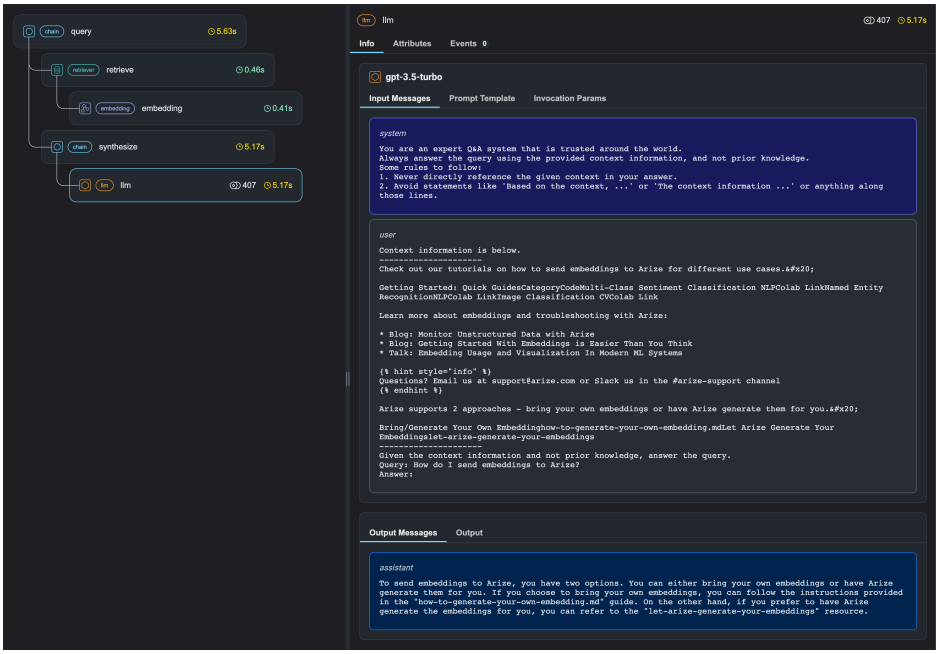


Figure 5.6: Phoenix UI that shows traces of queries in real time.

A complete example of tracing a LlamaIndex RAG application using Phoenix is available at this link.

5.3 HoneyHive Integration with LlamaIndex

HoneyHive is a framework that can be used to test and evaluate, monitor and debug LLM applications. It can be seamlessly integrated as displayed in Figure 5.7 into LlamaIndex applications.

The HoneyHive dashaboard looks like Figure 5.8 below:

```python
from llama_index import set_global_handler
set_global_handler(
    "honeyhive",
    project="My HoneyHive Project",
    name="My LLM Pipeline Name",
    api_key="MY HONEYHIVE API KEY",
)
```

Figure 5.7: HoneyHive integration with LlamaIndex

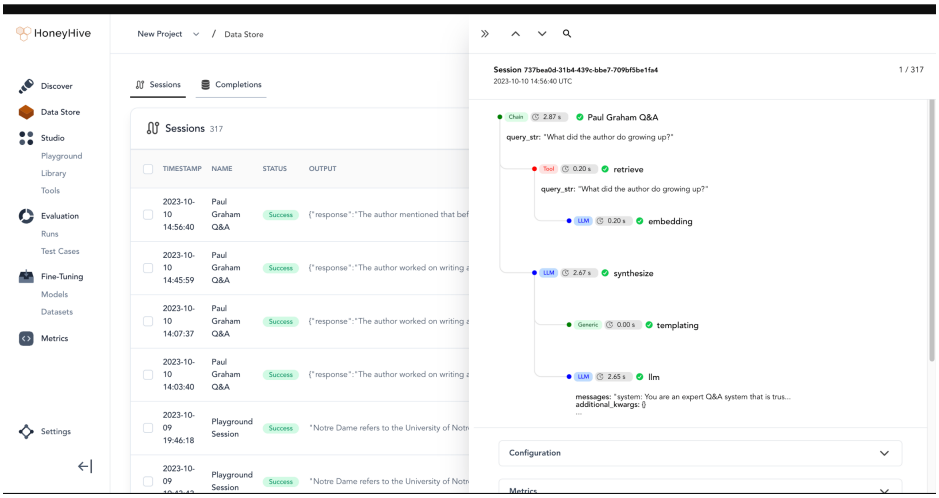


Figure 5.8: HoneyHive dashboard

For a complete guide, see this tutorial.

Other observability tools we can use include Truera, databricks, and Elastic observability among many other tools that are available.

6 Ending Note

In concluding our journey through the pages of "**A Practical Approach to Retrieval Augmented Generation Systems**," we hope you have gained valuable insights into the world of Retrieval-Augmented Generation. As the landscape of AI continues to evolve, RAG systems present an exciting intersection of retrieval and generation technologies, with immense potential across a multitude of industries and applications. With each chapter, we've delved deeper into the core principles, strategies, and techniques that underpin the development and implementation of RAG systems.

Remember that the field of AI is dynamic and ever-changing. While this book has aimed to provide a comprehensive understanding of RAG, new developments and possibilities are always on the horizon. We encourage you to continue exploring, experimenting, and innovating in the realm of Retrieval-Augmented Generation. Your journey doesn't end here; it's just the beginning.

Thank you for joining us on this transformative adventure into the heart of AI's future.

6.1 Acknowledgements

We would like to express our gratitude to the teams behind LlamaIndex, LangChain, and Haystack for their invaluable contributions to the field of Retrieval-Augmented Generation (RAG). Their comprehensive documentations and tutorials have been instrumental in our journey, allowing us to learn from their expertise and leverage the fascinating tools they have built.

References

Lewis, Patrick, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, et al. 2020. "Retrieval-Augmented Generation for Knowledge-Intensive Nlp Tasks." *Advances in Neural Information Processing Systems* 33: 9459–74.

Liu, Nelson F, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2023. "Lost in the Middle: How Language Models Use Long Contexts." *arXiv Preprint arXiv:2307.03172*.

Liu, Ye, Kazuma Hashimoto, Yingbo Zhou, Semih Yavuz, Caiming Xiong, and Philip S Yu. 2021. "Dense Hierarchical Retrieval for Open-Domain Question Answering." *arXiv Preprint arXiv:2110.15439*.

Ma, Xinbei, Yeyun Gong, Pengcheng He, Hai Zhao, and Nan Duan. 2023. "Query Rewriting for Retrieval-Augmented Large Language Models." *arXiv Preprint arXiv:2305.14283*.

Roeder, Luke Metz, Geoffrey, and Durk Kingma. 2021. "On Linear Identifiability of Learned Representations." *arXiv Preprint arXiv:2007.00810*.

Zhao, Wayne Xin, Jing Liu, Ruiyang Ren, and Ji-Rong Wen. 2022. "Dense Text Retrieval Based on Pretrained Language Models: A Survey." *arXiv Preprint arXiv:2211.14876*.

Zheng, Lianmin, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, et al. 2023. "Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena." *arXiv Preprint arXiv:2306.05685*.