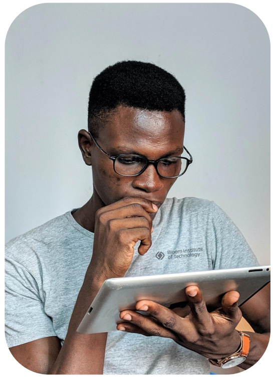
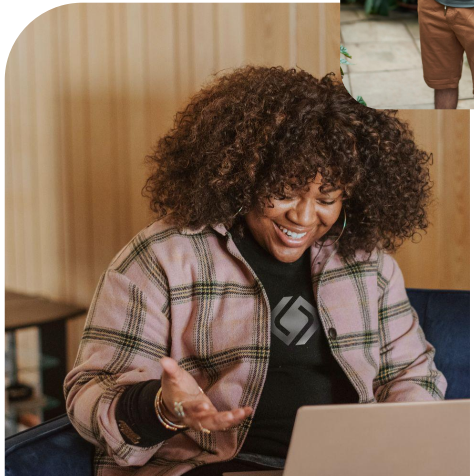
Bloom Institute
of Technology

Retrieval-Augmented Generation

Master the cutting-edge technique that integrates relevant documents into the generation process, enhancing the quality and contextuality of LLM responses.



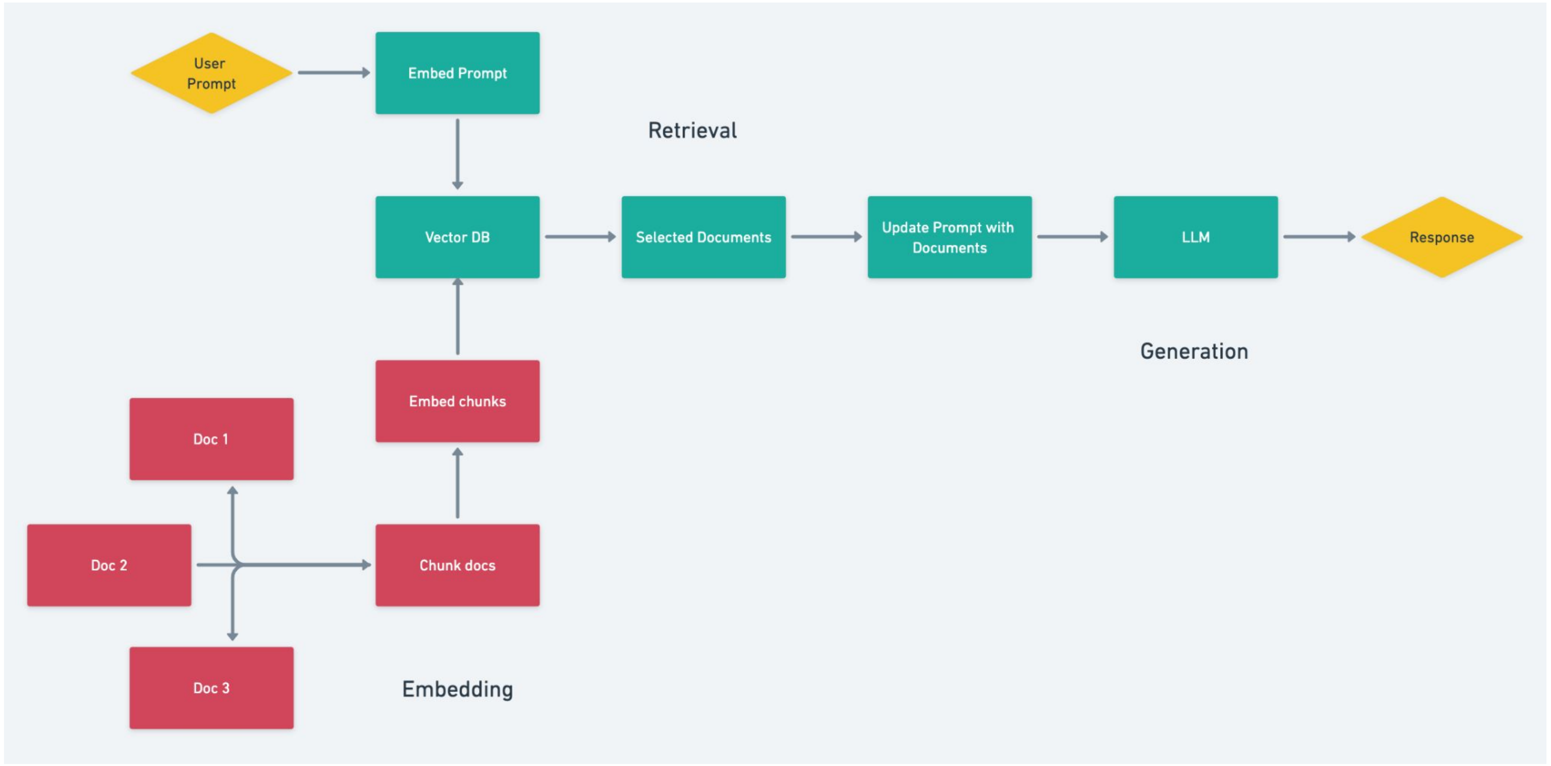


Core Competencies

The student must demonstrate…

1. A high-level understanding of RAG (10 min)
2. A deeper understanding of embedding and its purpose (5 min)
3. How to prepare documents for RAG (10 min)
4. An understanding of vector databases and what databases are available (5 min)
5. How to store documents in a vector DB (10 min)
6. (Bonus) Summary lookup RAG strategy (5 min)
7. (Bonus) RAG Fusion (10 min)

Introduction to RAG



Relevance: Automatically integrating proprietary data into chatbot memory via embeddings in a vector database enables seamless retrieval of relevant information, minimizing prompt overload.

The Role of Embeddings

Understanding Embeddings:

- Embeddings are **numerical representations capturing semantic meanings**, critical for machine learning, especially in NLP and image recognition.

Use Cases Beyond RAG:

- Embeddings enhance LLMs, aiding in text comprehension and generation.
- **Semantic chunking, powered by embeddings, breaks documents into meaningful segments** for tasks like summarization.

```
# Initialize the embeddings class
embeddings =
OpenAIEmbeddings(model="text-embedding-3-large")

# Embed a single query
query = "Hello, world!"
vector = embeddings.embed_query(query)
print(vector[:5])

# Embed multiple documents at once
documents = ["Alice works in finance", "Bob is a
database administrator", "Carl manages Bob and Alice"]
vectors = embeddings.embed_documents(documents)
print(len(vectors), len(vectors[0]))

db = FAISS.from_documents(documents, embeddings)
query = "Tell me about Alice"
docs = db.similarity_search(query)
print(docs[0].page_content)

# Perform a similarity search with scores
docs_and_scores = db.similarity_search_with_score(query)
print(docs_and_scores)
```

Preparing documents for RAG

A good RAG application requires **carefully curated documents** as they become the "memory" of the AI. Preparation involves meticulous planning and execution.

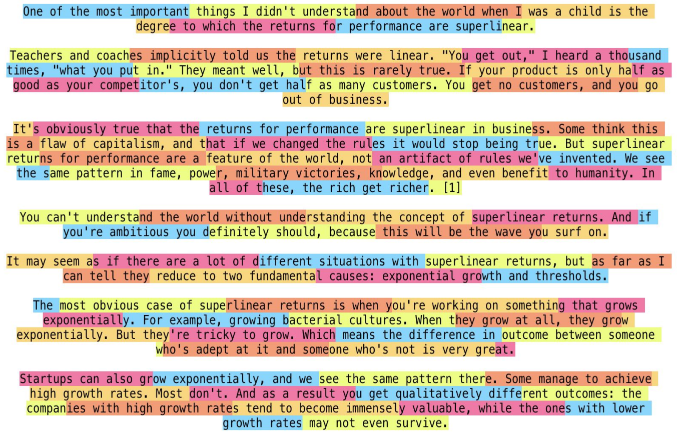
Document Splitting:

- Utilize ChunkViz to demonstrate text splitting.
- Simple character splitting has limitations, potentially dividing sentences and missing crucial meaning.
- Recursive splitters aim for cleaner breaks.
- Semantical splitting (not shown in ChunkViz) uses embeddings to identify shifts in ideas, particularly useful for documents with headers indicating significant context changes.

Editing for LLMs:

- Replace pronouns with proper nouns for clarity (e.g., "Tim went to the store. He bought some milk." becomes "Tim went to the store. Tim bought some milk.").
- Remove idioms and omit extraneous content like marketing messages that don't contribute to context.

Example: Character Splitting
Chunk size: 25 characters
Chunk overlap: 0



One of the most important things I didn't understand about the world when I was a child is the degree to which the returns for performance are superlinear.

Teachers and coaches implicitly told us the returns were linear. "You get out," I heard a thousand times, "what you put in." They meant well, but this is rarely true. If your product is only half as good as your competitor's, you don't get half as many customers. You get no customers, and you go out of business.

It's obviously true that the returns for performance are superlinear in business. Some think this is a flaw of capitalism, and that if we changed the rules it would stop being true. But superlinear returns for performance are a feature of the world, not an artifact of rules we've invented. We see the same pattern in fame, power, military victories, knowledge, and even benefit to humanity. In all of these, the rich get richer. [1]

You can't understand the world without understanding the concept of superlinear returns. And if you're ambitious you definitely should, because this will be the wave you surf on.

It may seem as if there are a lot of different situations with superlinear returns, but as far as I can tell they reduce to two fundamental causes: exponential growth and thresholds.

The most obvious case of superlinear returns is when you're working on something that grows exponentially. For example, growing bacterial cultures. When they grow at all, they grow exponentially. But they're tricky to grow. Which means the difference in outcome between someone who's adept at it and someone who's not is very great.

Startups can also grow exponentially, and we see the same pattern there. Some manage to achieve high growth rates. Most don't. And as a result you get qualitatively different outcomes: the companies with high growth rates tend to become immensely valuable, while the ones with lower growth rates may not even survive.

Vector Databases

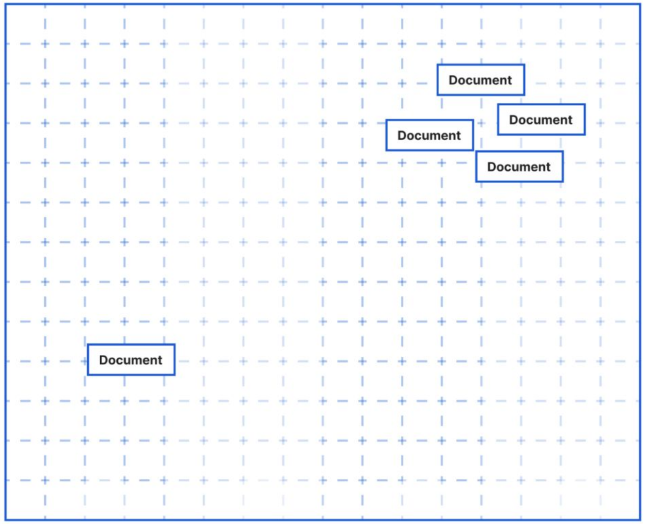
Introducing Pinecone: The Vector Database for AI. Streamline AI workflows with **lightning-fast similarity searches and nearest neighbor queries**. Explore Pinecone's comprehensive user guide on YouTube, Pinecone User Guide.

How to setup your first Pinecone Vector Store:

```
embeddings =
OpenAIEmbeddings(model="text-embedding-3-small")

document_vectorstore =
PineconeVectorStore(index_name="my-rag-index",
embedding=embeddings)

retriever = document_vectorstore.as_retriever()
context = retriever.get_relevant_documents(prompt)
```



In this simple vector database, the documents in the upper right are likely similar to each other.

Bonus: Summary Lookup

RAG extends beyond traditional document augmentation. Another approach to accessing relevant documents is through summary lookup. **Summary lookup quickly finds relevant data by comparing queries to document summaries.**

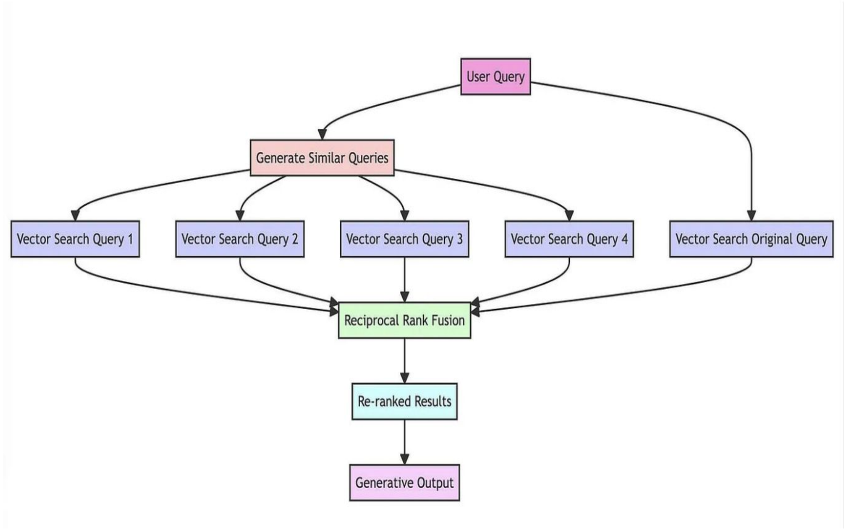
Problem	Solution
• Retrieving specific information from a large corpus, such as finding Warren's Coca-Cola investment in 2023 among all shareholder letters, risks retrieving irrelevant data from other years.	• Create a separate vector database index containing document summaries. • Compare the query to summaries to identify the most relevant documents, such as those from 2023. • Narrow down the search to the most pertinent document, possibly excluding specific details like "2023" from the prompt to refine the search.

Bonus: RAG Fusion

RAG-Fusion enhances search accuracy by **generating multiple user queries, reranking the results, and using** Reciprocal Rank Fusion with custom vector score weighting. It aims to bridge the gap between explicit user queries and their underlying intent.

Reciprocal Rank Fusion

- Leverages LLMs to rewrite a user query multiple times with different focuses, then uses each of those queries to get context.
- The system can refocus the prompt "How can I save money?" to focus on:
    - Inventory ("How can the company find alternative drugs to cut costs?"),
    - Contracts ("How can I better utilize existing contracts to save cost?"),
    - Economic Environment ("What items are experiencing a high increase due to inflation?").
- Each query (the original one could be included, too) is then sent to the retriever.



Simple RAG Fusion Using LangChain

Let's walk through a simple RAG fusion example using Pinecone, LangChain, and OpenAI.
Access the complete code here.

Overview of Code:

1. Setup and Load Data:
   a. Configure Pinecone by setting the `PINECONE_API_KEY` and `OPENAI_API_KEY` env variables.
   b. Load necessary libraries and define a set of fake documents related to any one topic.

2. Create Vector Store:
   a. Use Pinecone to create a vector store from the loaded documents.
   b. Convert the document texts into embeddings and store them in the Pinecone index.

3. Define the Query Generator:
   a. Load and configure the prompt template for query generation using `langchainhub`.
   b. Define a query generation chain using the loaded prompt, OpenAI's chat model, and an output parser to generate multiple search queries.

4. Define and Execute Full Chain:
   a. Define the original query and create a retriever.
   b. Define the Reciprocal Rank Fusion function to re-rank the search results based on multiple queries.
   c. Combine the query generation, retrieval, and rank fusion steps into a single chain.
   d. Invoke the chain with the original query to get the re-ranked search results.

Hands-on Homework.

Code a RAG application that queries the Langchain docs website (or another documentation website of your choice). Use ReadTheDocsLoader to easily pull down documentation and load it into a retriever.