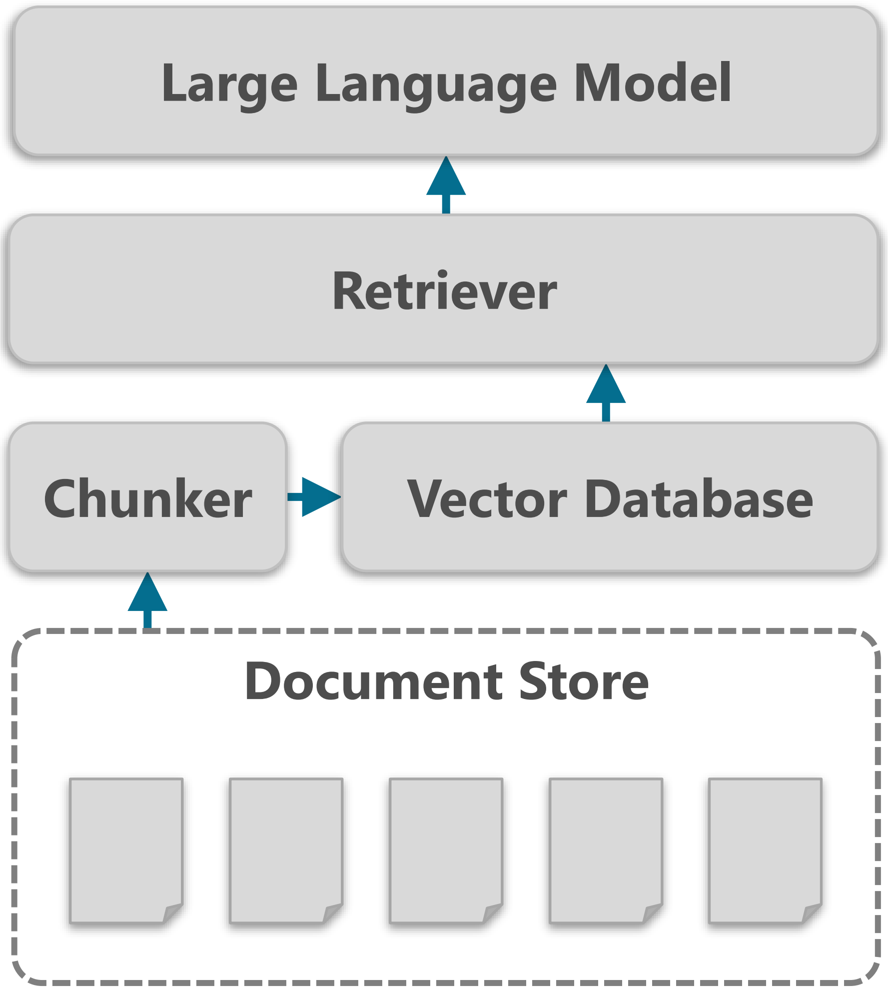
Retrieval-Augmented Generation

Jeff Prosise

@jprosise

Retrieval-Augmented Generation (RAG)

- Enables LLMs to answer questions from custom knowledge bases consisting of documents
  - Text files, PDFs, DOCX files, etc.
- Puts up guardrails that make an LLM less likely to hallucinate
  - Say "I don't know"
- The #1 use case for LLMs in industry today



Text Embeddings

- Vectors of floating-point numbers that quantify text

`Cars that get great gas mileage`

-0.43	0.02	0.85	0.03	-0.40	0.07	-0.13	0.25	0.43	...

- Compute similarity of two text samples by measuring distance between their embedding vectors using cosine similarity, dot products, or other measures
- Useful for semantic-search systems, recommender systems, deduplication systems, and other similarity-based systems

OpenAI Text Embedding Models

- Multilingual models that generate normalized embeddings
- Permit embeddings to be shortened for compatibility with vector stores that limit embedding lengths

text-embedding-3-small

Produces vectors of up to **1,536** floating-point numbers. Max input length is **8K**.

text-embedding-3-large

Produces vectors of up to **3,072** floating-point numbers. Max input length is **8K**. Scores **slightly higher** than text-embedding-3-small on key benchmarks.

.

Generating an Embedding Vector

```python
from openai import OpenAI

client = OpenAI(api_key='OPENAI_API_KEY')

response = client.embeddings.create(
    model='text-embedding-3-small',
    input='Cars that get great gas mileage'
)

embedding = response.data[0].embedding
```

Generating a Short Embedding Vector

```
from openai import OpenAI

client = OpenAI(api_key='OPENAI_API_KEY')

response = client.embeddings.create(
    model='text-embedding-3-small',
    input='Cars that get great gas mileage',
    dimensions=256 # Limit embedding to 256 values
)

embedding = response.data[0].embedding
```

Comparing Embedding Vectors

```
x = client.embeddings.create(
    model='text-embedding-3-small',
    input='Cars that get great gas mileage'
).data[0].embedding

y = client.embeddings.create(
    model='text-embedding-3-small',
    input='Cars that are fuel-efficient'
).data[0].embedding

similarity = np.dot(np.array(x), np.array(y))
# 0.8362645039208086
```

Google Embedding Models

- Models that generate high-quality normalized embeddings
- Support **task_type** parameter that permits embeddings to be optimized for various use cases (retrieval, semantic similarity, etc.)

text-embedding-004

English-only model that produces vectors of up to **768** floating-point numbers. Max input length is **2K** tokens.

text-multilingual-embedding-002

Multilingual model that produces vectors of up to **768** floating-point numbers. Max input length is **2K** tokens. **Only available through Vertex AI**.

Generating an Embedding Vector

```
import google.generativeai as genai

genai.configure(api_key='GOOGLE_API_KEY')

response = genai.embed_content(
    model='models/text-embedding-004',
    content='Cars that get great gas mileage'
)

embedding = response['embedding']
```

Generating a Short Embedding Vector

```
import google.generativeai as genai

genai.configure(api_key='GOOGLE_API_KEY')

response = genai.embed_content(
    model='models/text-embedding-004',
    content='Cars that get great gas mileage',
    output_dimensionality=256 # Limit embedding to 256 values
)

embedding = response['embedding']
```

Comparing Embedding Vectors

```
x = genai.embed_content(
    model='models/text-embedding-004',
    content='Cars that get great gas mileage'
)['embedding']

y = genai.embed_content(
    model='models/text-embedding-004',
    content='Cars that are fuel-efficient'
)['embedding']

similarity = np.dot(np.array(x), np.array(y))
# 0.8746150746638693
```

Measuring Semantic Similarity

```
x = genai.embed_content(
    model='models/text-embedding-004',
    content='Cars that get great gas mileage',
    task_type='SEMANTIC_SIMILARITY'
)['embedding']

y = genai.embed_content(
    model='models/text-embedding-004',
    content='Cars that are fuel-efficient',
    task_type='SEMANTIC_SIMILARITY'
)['embedding']

similarity = np.dot(np.array(x), np.array(y))
# 0.9059963501637823
```

Hugging Face Text Embedding Models

- Hugging Face hosts more than 800 text embedding models that vary by context length, languages supported, and other factors

all-MiniLM-L12-v2

English-only model that supports input lengths up to **256 tokens**. Generates **384-dimensional** embedding vectors.

all-miniLM-L6-v2

English-only model that supports input lengths up to **256 tokens**. Generates **384-dimensional** embedding vectors. **5X faster** than **all-MiniLM-L12-v2.**

paraphrase-multilingual-MiniLM-L12-v2

Multilingual model trained on **more than 50 languages** that supports input lengths up to **512 tokens**. Generates **384-dimensional** embedding vectors.

jina-embeddings-v2-small-en

English-only model based on BERT that supports input lengths up to **8K tokens**. Generates **384-dimensional** embedding vectors.

Generating an Embedding with all-MiniLM-L6-v2

```python
from sentence_transformers import SentenceTransformer

model = SentenceTransformer('sentence-transformers/all-MiniLM-L6-v2')
embedding = model.encode(['Cars that get great gas mileage'])[0]
```

Comparing all-MiniLM-L6-v2 Embeddings

```
model = SentenceTransformer('sentence-transformers/all-MiniLM-L6-v2')
x = model.encode(['Cars that get great gas mileage'])[0]
y = model.encode(['Cars that are fuel-efficient'])[0]

similarity = np.dot(np.array(x), np.array(y))
# 0.7341136
```

Demo

Text Embeddings

Vector Databases

- Databases that store text and optional metadata
  - Items are keyed with embedding vectors generated from item text
  - Database is queried with embedding vectors generated from query text
  - Queries return the top *n* matches based on embedding similarity
- Contemporary vector databases such as **Pinecone**, **ChromaDB**, and **Qdrant** retrieve text samples based on similarity to input text and scale to millions of vectors
  - Many are free and open-source
- The basis for modern RAG systems

Creating and Populating a ChromaDB Collection

```
import chromadb

# Create the collection
client = chromadb.PersistentClient('chroma') # Path to where database is stored
collection = client.create_collection(name='Great_Speeches')

# Add an item
collection.add(
    documents=['Four score and seven years ago...'], # Text of item
    ids=['Paragraph-001'] # Unique ID
)
```

Querying a Collection

```
# Get a reference to the collection
client = chromadb.PersistentClient('chroma')
collection = client.get_collection(name='Great_Speeches')

# Query without metadata filtering
results = collection.query(
    query_texts=['How old is the nation?'], # Query text
    n_results=1
)
```

Using OpenAI Embeddings

```
from chromadb.utils.embedding_functions import OpenAIEmbeddingFunction

embedding_function = OpenAIEmbeddingFunction(
    api_key='OPENAI_API_KEY',
    model_name='text-embedding-3-small'
)
```

Using OpenAI Embeddings, Cont.

```
client = chromadb.PersistentClient('chroma')

# Create collection with custom embedding function
collection = client.create_collection(
    name='Great_Speeches',
    embedding_function=embedding_function
)

# Retrieve reference to collection with custom embedding function
collection = client.get_collection(
    name='Great_Speeches',
    embedding_function=embedding_function
)
```

Using Google Embeddings

```python
from chromadb.utils.embedding_functions import GoogleGenerativeAiEmbeddingFunction

embedding_function = GoogleGenerativeAiEmbeddingFunction(
    api_key='GOOGLE_API_KEY',
    model_name='models/text-embedding-004'
)
```

Demo

ChromaDB

Answering Questions with an LLM

```
content = f'''
    Answer the following question, and if you don't know the answer, say "I don't know:"
    Question: Who was the first president of the United States?
    Answer:
    '''

messages = [{ 'role': 'user', 'content': content }]

response = client.chat.completions.create(
    model='gpt-4o',
    messages=messages
)
```

Answering Contextual Questions

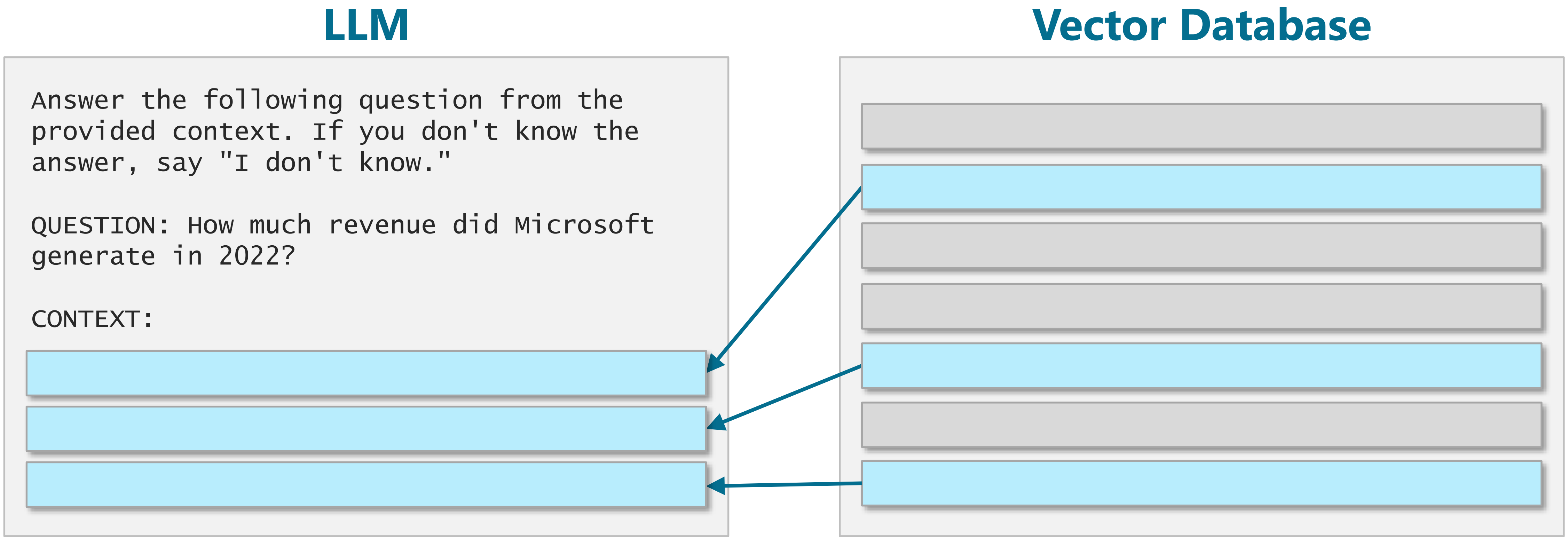
```
content = f'''
    Answer the following question using the provided context, and if the answer isn't
    contained in the context, say "I don't know:"
    Context: {context} # Insert text to be searched for an answer
    Question: Who was the first president of the United States?
    Answer:
    '''

messages = [{ 'role': 'user', 'content': content }]

response = client.chat.completions.create(
    model='gpt-4o',
    messages=messages
)
```

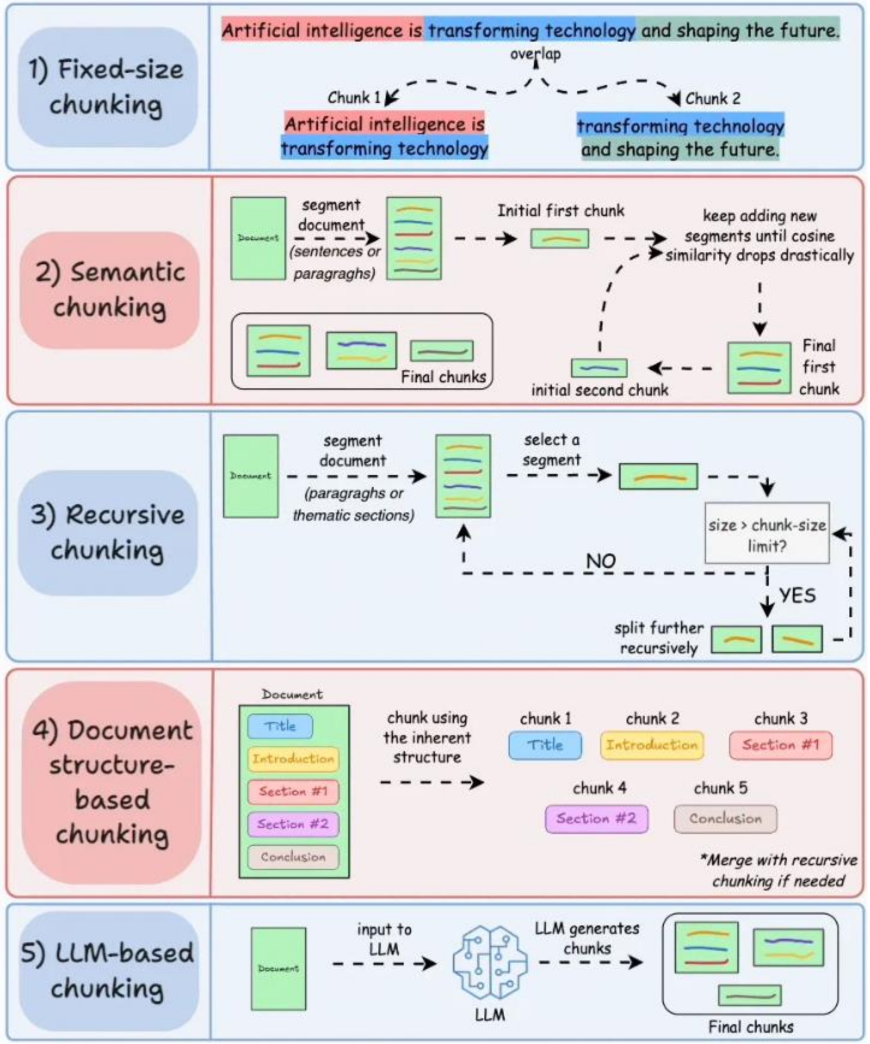
Retrieval-Augmented Generation (RAG)

- Use LLM to answer questions using documents as context
- "Chunk" documents and use embeddings to identify relevant chunks



Chunking Strategies

- Several strategies exist for "chunking" documents, ranging from simple and free (fixed-size) to costly (LLM-based)
- Chunking can be done manually or with help from libraries such as **LllamaIndex**
- **LlamaIndex** also has parsers for extracting text from numerous file types, including PDF, DOCX, PPTX, HTML, EPUB, and MD (markdown) files



https://blog.dailydoseofds.com/p/5-chunking-strategies-for-rag

Fixed-Size Chunking with LlamaIndex

```python
from llama_index.readers.file.docs import DocxReader
from llama_index.core.node_parser import SentenceSplitter

reader = DocxReader()
document = reader.load_data('PATH_TO_DOCX_FILE')

splitter = SentenceSplitter(chunk_size=1024, chunk_overlap=20)
nodes = splitter.get_nodes_from_documents(document)

for node in nodes:
    print(node.text)
```

Semantic Chunking with LlamaIndex

```
from llama_index.readers.file.docs import PDFReader
from llama_index.core.node_parser import SemanticSplitterNodeParser
from llama_index.embeddings.openai import OpenAIEmbedding

reader = PDFReader()
document = reader.load_data('PATH_TO_PDF_FILE')

splitter = SemanticSplitterNodeParser(
    buffer_size=1,
    breakpoint_percentile_threshold=95,
    embed_model=OpenAIEmbedding(api_key='OPENAI_API_KEY')
)

nodes = splitter.get_nodes_from_documents(document)
```

Demo

Naïve RAG

Reranking

- Identifying relevant chunks using embedding vectors isn't perfect

Cars that get great gas mileage

Cars that don't get great gas mileage

0.9070775256753464

Similarity score computed by text-embedding-3-small

- Rerankers rank chunks in order of relevance using semantic understanding and are used to implement two-stage retrieval
  - Query vector database for *m* chunks based on embedding similarity
  - Rerank chunks by descending order of relevance and take the top *n* chunks, where *n* is less than *m*

Reranking Methods

Method	Example(s)	Comments
Cross encoder	jina-reranker, BGE	Fast, free, and effective
Multi-vector model	ColBERT	Faster than cross encoders due to late interaction, but not quite as effective
Large language model	GPT-4o, Gemini, Llama	Highly effective, but higher cost and latency
Reranking API	Cohere, Jina	Highly effective, but higher cost and latency

Cross Encoding

- Computes similarity of two text samples using a heightened understanding of semantic meaning
- Built by fine-tuning pretrained language models such as BERT or a variation of BERT

Cars that get great gas mileage

Cars that don't get great gas mileage

0.49530423

Similarity score computed by jina-reranker-v1-turbo-en

Using jina-reranker-v1-turbo-en

```
from sentence_transformers import CrossEncoder

model = CrossEncoder('jinaai/jina-reranker-v1-turbo-en', trust_remote_code=True)

ranked_chunks = model.rank(
    'Cars that get great gas mileage',
    chunks, # Contexts retrieved from vector database
    return_documents=True,
    top_k=5
)
```

Using jina-reranker-v2-base-multilingual

```
from sentence_transformers import CrossEncoder

model = CrossEncoder('jinaai/jina-reranker-v2-base-multilingual', trust_remote_code=True)

ranked_chunks = model.rank(
    'Cars that get great gas mileage',
    chunks, # Contexts retrieved from vector database
    return_documents=True,
    top_k=5
)
```

Demo

Reranking

Metadata Extraction

- In some cases, incorporating metadata into vector database queries makes RAG more accurate
  - Example: Knowledge base includes annual reports from Microsoft, Google, and Meta for the years 2020-present
  - Questions such as "What was Microsoft's revenue in 2022?" must target only chunks from Microsoft's 2022 annual report
- Solution: Use an LLM to extract metadata values from the user input and use those values to filter vector-database queries

Inserting with Metadata

```
collection.add(
    documents=[text],
    metadatas=[{ 'company': 'Microsoft', 'year': '2022' }],
    ids=['00001']
)
```

Filtering Queries with Metadata

```
# Filtering with a single value
results = collection.query(
    query_texts=["Who is Microsoft's CEO?"],
    where={ 'company': 'Microsoft' }
    n_results=5
)

# Filtering with multiple values
results = collection.query(
    query_texts=["What was Microsoft's revenue in 2022?"],
    where={ '$and': [{ 'company': 'Microsoft' }, { 'year': '2022' }] }
    n_results=5
)
```

Demo

Metadata Extraction