<DevSum>

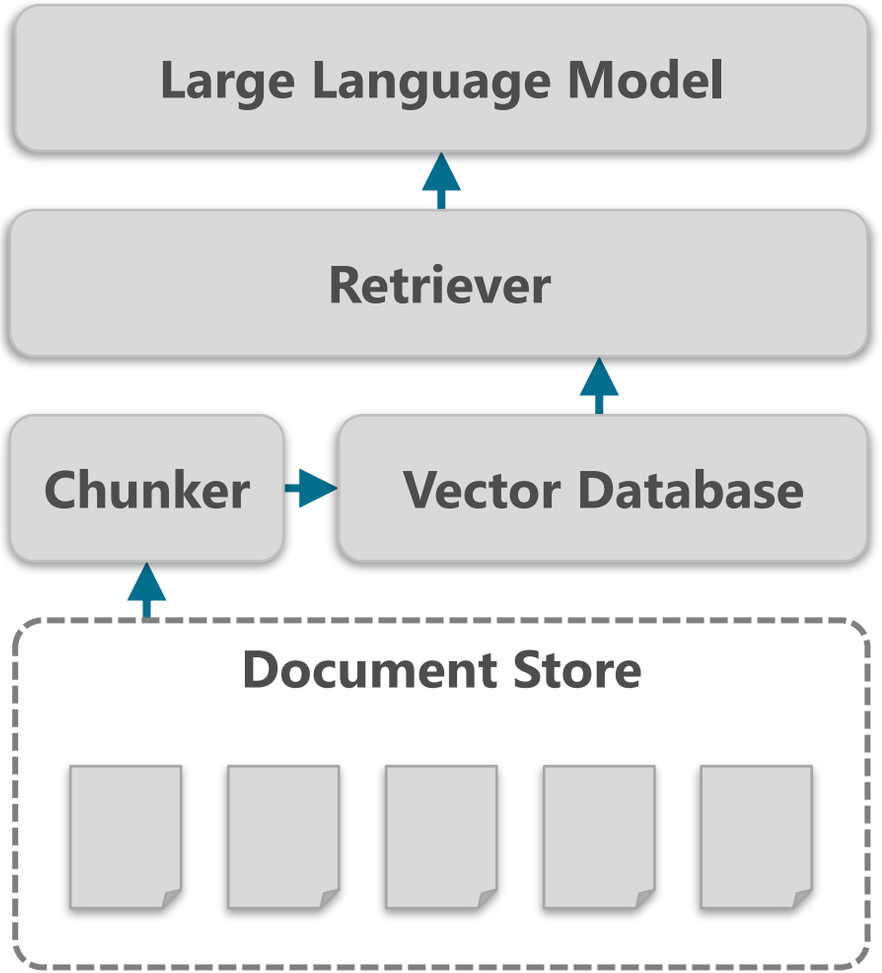
Making LLMs Smarter

Jeff Prosise

https://wintellect.blob.core.windows.net/public/devsum.zip

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

Generating an 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'
)

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
```

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
)
```

Answering Questions with an LLM

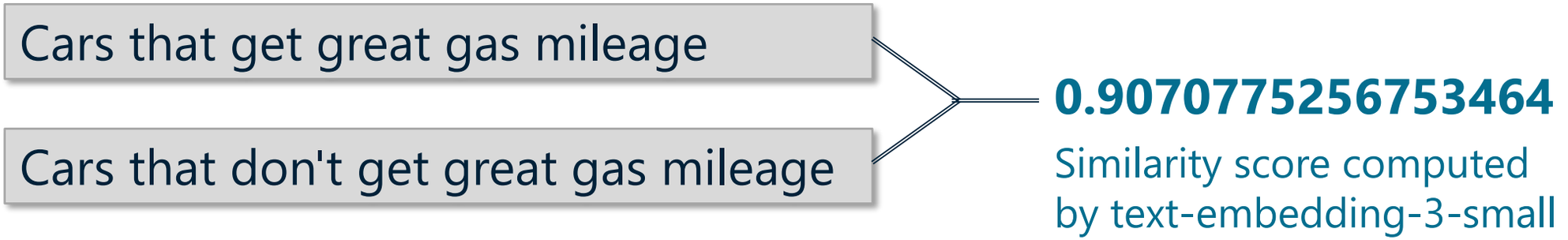
```
content = f'''
    Answer the following question, and if you don't know the answer, say "I don't know:"
    Question: Who was the first president of the United States?
    Answer:
    '''


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
)
```

Reranking

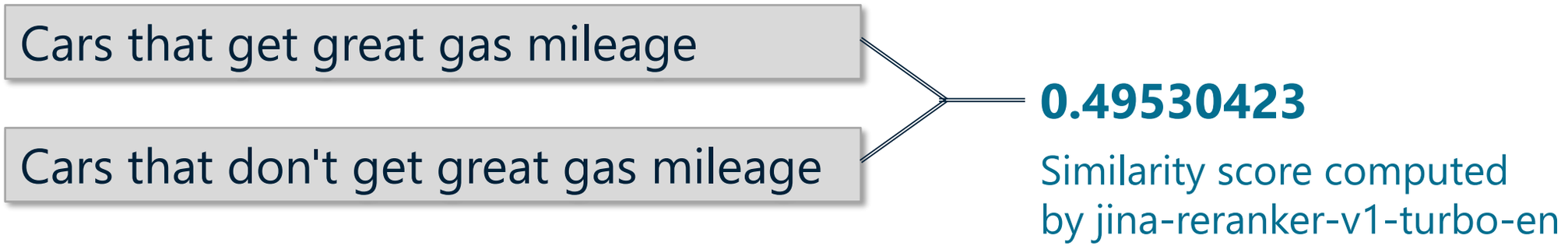
- Identifying relevant chunks using embedding vectors isn't perfect



- 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*

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



Using jina-reranker-v1-turbo-en

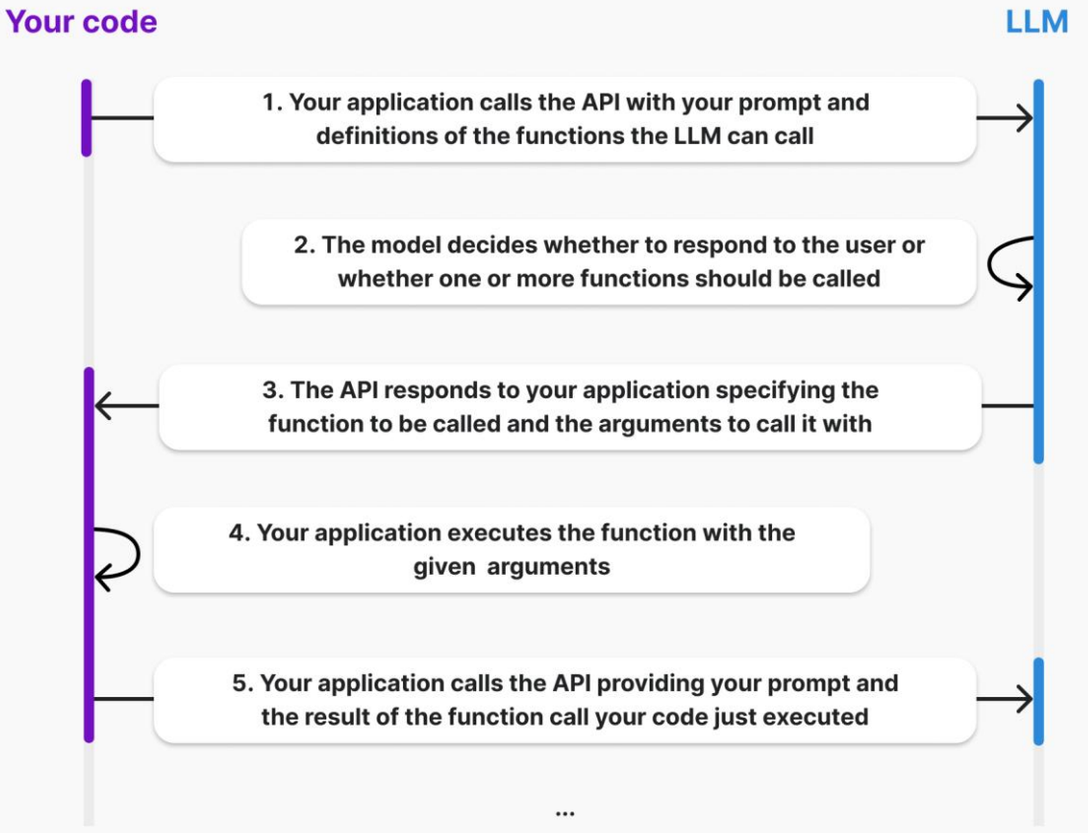
```
from sentence_transformers import CrossEncoder

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
)
```

Demo

Retrieval-Augmented Generation

Function Calling

- Extend an LLM's powers by making user-defined functions available to it
  - Call external APIs
  - Query a database
  - Perform math
- LLM parses input, tells you which function(s) to call, and provides values for function parameters



Source: https://platform.openai.com/docs/guides/function-calling

Describing a Function to an LLM

```
weather_tool = {
    'type': 'function',
    'function': {
        'name': 'get_current_weather',
        'description': 'Retrieves the current weather at the specified location',
        'parameters': {
            'type': 'object',
            'properties': {
                'location': {
                    'type': 'string', # number, string, boolean, array, or object
                    'description': 'The location whose weather is to be retrieved.'
                }
            },
            'required': ['location']
        }
    }
}
```

Making a Function Available to an LLM

```
client = OpenAI(api_key='OPENAI_API_KEY')
messages = [{ 'role': 'user', 'content': 'Is it raining in London?' }]

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

Processing Function Calls

```
# If one or more tool calls are required, execute them
if response.choices[0].message.tool_calls:
    for tool_call in response.choices[0].message.tool_calls:
        function_name = tool_call.function.name

        if function_name == 'get_current_weather':
            # Get the function argument(s) and call the function
            location = json.loads(tool_call.function.arguments)['location']
            output = get_current_weather(location) # Convert to JSON if not already JSON

            # Append the function output to the messages list
            messages.append({
                'role': 'function', 'name': function_name, 'content': output
            })
```

Processing Function Calls, Cont.

```
    # After all function calls are complete, pass the output to the LLM
    response = client.chat.completions.create(
        model='gpt-4o',
        messages=messages # Includes original message and function output
    )

# Show the response
print(response.choices[0].message.content)
```

Demo

Function Calling

Generating Code

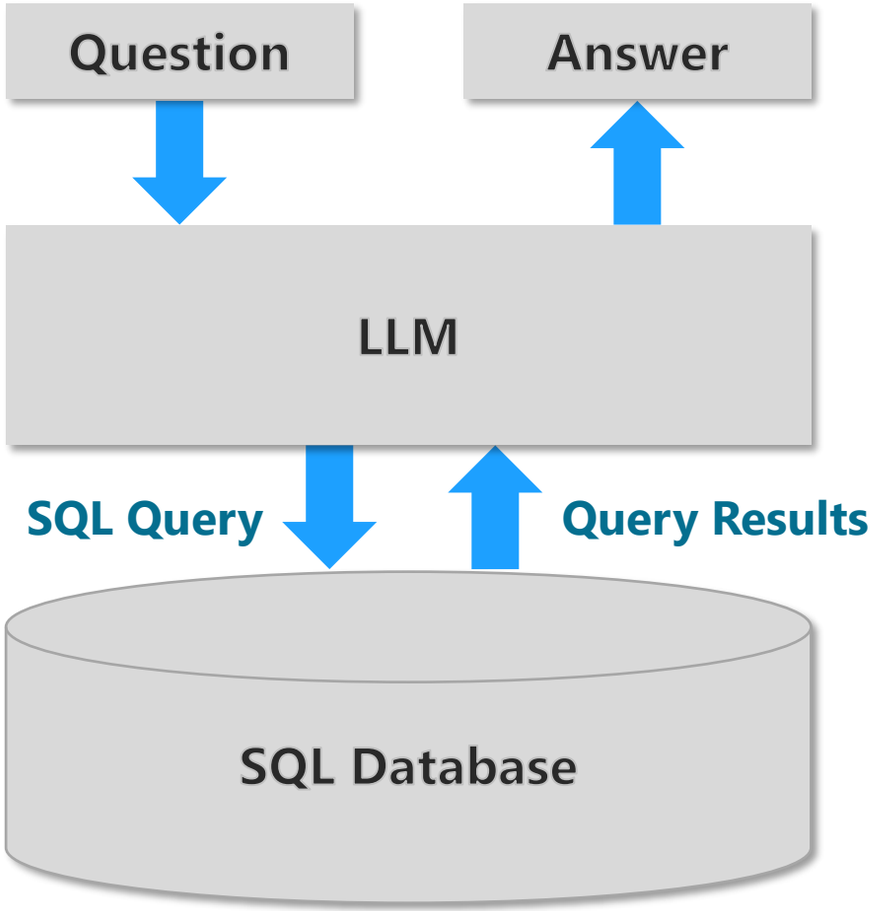
```
prompt = '''
    Create a Python function that accepts an array of numbers as
    input, bubble sorts the numbers, and returns a sorted array
    '''


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

response = client.chat.completions.create(
    model='gpt-4o',
    messages=messages,
    temperature=0 # Use a low temperature setting for code generation
)
```

Generating SQL

```
prompt = f'''
    Generate a well-formed SQL query from the following input:
    INPUT: {input}

    Assume the database has the following schema:
    SCHEMA: {database_schema}
    '''

response = client.chat.completions.create(
    model='gpt-4o',
    messages=[{ 'role': 'user', 'content': prompt }],
    temperature=0
)
```

SQL-Augmented Generation

- Use an LLM to provide a natural-language interface to databases
  - Convert natural-language questions into SQL queries
  - Execute the queries against the database
  - Use an LLM to phrase query results in terms a human can understand
- Use CREATE TABLE statements in prompts to communicate schema



Self-Correcting Queries

- Even with examples, an LLM will still occasionally generate bad SQL
- Solution: If a query fails, pass it back to the LLM with the original input and the error message and ask the LLM to correct the query

```
prompt = '''
    The SQL query below was generated from the input below. The query failed and
    returned the error message below. Modify the query to rectify the error.

    INPUT: {input}
    SCHEMA: {database_schema)
    QUERY: {sql}
    ERROR: {error_message}
    '''
```

Demo

LLMs Over Databases

Fine-Tuning

- Add to a model's knowledge base by training with additional samples at a reduced learning rate
  - Expand a model's knowledge to include current events
  - Make a model aware of domain-specific terms and information
  - Train a model to mimic someone's style or tone of voice

Pros	Cons
▪ Shorter prompts yield less cost and less latency ▪ Model can inherently do things it couldn't do before	▪ Time and expense required to assemble training data and fine-tune the model ▪ To benefit from improvements to the base model, fine-tuning must be repeated

JSONL

```
{ "messages": [
    { "role": "system", "content": "You are a helpful HR person" }, # Optional
    { "role": "user", "content": "Who is our current CEO?" },
    { "role": "assistant", "content": "Satya Nadella" }
]}
{ "messages": [
    { "role": "system", "content": "You are a helpful HR person" },
    { "role": "user", "content": "How much PTO do I receive each year?" },
    { "role": "assistant", "content": "Microsoft employees receive unlimited PTO" }
]}
      .
      .
      .
```

Uploading a Training File

```
client = OpenAI(api_key='OPENAI_API_KEY')

train_file = client.files.create(
    file=open('PATH_TO_JSONL_FILE', 'rb'),
    purpose='fine-tune'
)
```

Fine-Tuning a Model

```
model = client.fine_tuning.jobs.create(
    training_file=train_file.id,
     model='gpt-4o-mini-2024-07-18',
     hyperparameters={
        'n_epochs': 5,
        'batch_size': 5,
        'learning_rate_multiplier': 0.2
    }
)
```

Enumerating Fine-Tuned Models

```
from datetime import datetime

client = OpenAI(api_key='OPENAI_API_KEY')
jobs = client.fine_tuning.jobs.list()

for job in jobs:
    if job.status == 'succeeded':
        completion_date = datetime.utcfromtimestamp(job.finished_at) \
            .strftime('%Y-%m-%d %H:%M:%S')
        print(f'{job.fine_tuned_model} ({completion_date})')

# ft:gpt-4o-mini-2024-07-18:personal::APFNGbzX (2024-11-02 20:51:00)
```

Calling a Fine-Tuned Model

```
client = OpenAI(api_key='OPENAI_API_KEY')
messages = [{ 'role': 'user', 'content': "Who is Microsoft's CEO?" }]

response = client.chat.completions.create(
    model=model, # e.g., ft:gpt-4o-mini-2024-07-18:personal::APFNGbzX
    messages=messages
)

print(response.choices[0].message.content)
```

Demo

Fine-Tuning