Retrieval Augmented Generation (RAG)

CS450: Modern Software Engineering

Vector Embeddings & Semantic Search

Today's Topics

1. The Problem: LLM Knowledge Limits
2. Vector Embeddings Explained
3. Semantic Similarity & Search
4. Building RAG Systems
5. When to Use RAG

The LLM Knowledge Problem

Challenge: LLMs have fixed knowledge cutoff dates

```
# LLM without RAG
Q: "When was WWU founded?"
A: "I'm not sure about the specific date..."
```

Solution: Give LLMs access to external knowledge

```
# LLM with RAG
Q: "When was WWU founded?"
→ Retrieve: "WWU was founded in 1892..."
A: "Walla Walla University was founded in 1892."
```

What Are Vector Embeddings?

Embeddings = Dense numerical representations of text

- Text → Vector of numbers
- Capture semantic meaning
- Similar meanings → Similar vectors

```
"cat" → [0.2, -0.5, 0.8, ..., 0.3]  # 768 dimensions
"dog" → [0.3, -0.4, 0.7, ..., 0.2]  # Similar!
"car" → [-0.8, 0.9, -0.1, ..., 0.7] # Different!
```

How Embeddings Capture Meaning

Traditional approach: Exact word matching

- "cat" matches "cat" only
- Misses "feline", "kitten", etc.

Embedding approach: Semantic similarity

- "cat" ≈ "feline" ≈ "kitten"
- Captures relationships and context

Embedding Example

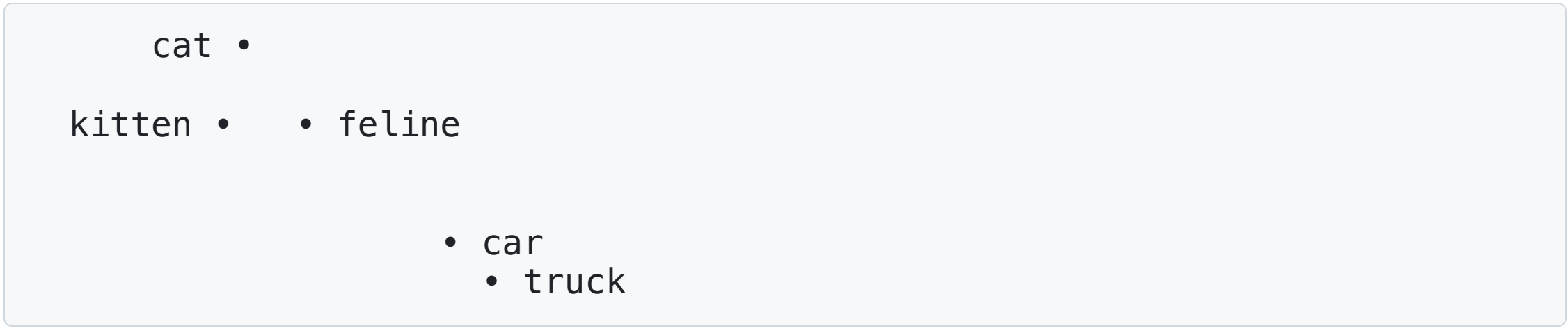
```
import ollama

client = ollama.Client(host='http://localhost:11434')
response = client.embeddings(
    model='nomic-embed-text',
    prompt="The cat sat on the mat"
)

embedding = response['embedding']
print(len(embedding))  # 768 dimensions
```

Result: Array of 768 floating-point numbers

Visualizing Embeddings

Real embeddings are high-dimensional (768+)

Let's visualize in 2D for intuition:



Similar concepts cluster together!

Cosine Similarity

How to measure similarity between vectors?

Cosine Similarity = angle between vectors

```
Score range: -1 to 1
  1.0  = Identical
  0.0  = Unrelated
 -1.0  = Opposite
```

Formula: $\cos(\theta) = (A \cdot B) / (||A|| \times ||B||)$

Cosine Similarity Example

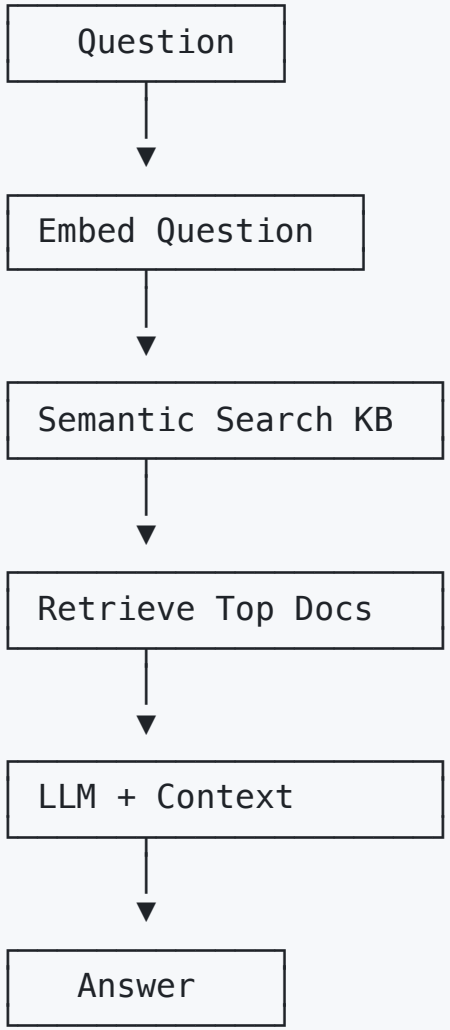
```
from sklearn.metrics.pairwise import cosine_similarity

emb1 = get_embedding("The cat sat on the mat")
emb2 = get_embedding("A feline rested on a rug")
emb3 = get_embedding("Python programming language")

sim_1_2 = cosine_similarity([emb1], [emb2])[0][0]
# Result: 0.85 (very similar!)

sim_1_3 = cosine_similarity([emb1], [emb3])[0][0]
# Result: 0.12 (not similar)
```

Semantic Search Process

1. **Index documents:** Convert all docs to embeddings
2. **Query conversion:** Convert question to embedding
3. **Similarity search:** Find closest document embeddings
4. **Return results:** Ranked by similarity score

```
Query: "Python usage" → [0.3, 0.7, ...]
                          ↓ compare
Documents: [doc1_emb, doc2_emb, doc3_emb, ...]
                          ↓ rank
Results: [doc3 (0.92), doc1 (0.78), ...]
```

RAG Architecture



RAG Prompt Structure

```
prompt = f"""Answer the question based on the context below.
If the answer is not in the context, say "I don't know."

Context:
{retrieved_documents}

Question: {user_question}

Answer:"""
```

Key: Provide retrieved context before the question

RAG vs Direct LLM

Direct LLM:

- Limited to training data
- May hallucinate facts
- Can't access private data

RAG:

- Uses current documents
- Grounds responses in sources
- Works with private knowledge
- More accurate for specific domains

When to Use RAG

Good for:

- Frequently updated information
- Domain-specific knowledge
- Private/proprietary data
- Need for source attribution

Not ideal for:

- General knowledge tasks
- Creative writing
- Real-time computation
- When latency is critical

RAG Limitations

1. **Quality depends on retrieval**
    - Bad retrieval → Bad answers
2. **Context window limits**
    - Can only include top-k documents
3. **Latency**
    - Embedding + search + generation
4. **Relevance threshold**
    - Must decide when to say "I don't know"

Lab time!