Weekly Research Report

Prepared by: Anbhi Thakur
Submission Date: Week Ending Saturday, 29,June,2025
Focus Areas: Vector Embeddings and Retrieval-Augmented Generation (RAG)

Task 1: In-Depth Report on Vector Embeddings and Their Workflow

1. Principles of Vector Embeddings

Vector embeddings are numerical representations of data that capture semantic meaning and relationships in a high-dimensional space. Primarily used in machine learning and NLP tasks, embeddings transform unstructured data (text, images, audio) into fixed-size vectors that models can understand.

Core Concepts:

- **Semantic Similarity:** Embeddings allow models to compare and measure similarity between inputs based on their context and meaning, not just surface-level features.
- **Dimensionality Reduction:** High-dimensional data (like full sentences or documents) is transformed into compact vectors, often of 128 to 768 dimensions.
- **Contextualization:** Modern embeddings (e.g., from transformers like BERT) provide context-aware representations. The meaning of "bank" in "river bank" and "money bank" is encoded differently.
- **Reusability:** Once embedded, the data can be used in a range of downstream tasks like classification, retrieval, clustering, or semantic search.

Popular Models:

- Word2Vec, GloVe (static)
- BERT, Sentence-BERT, OpenAI embeddings (contextualized)

---

2. Splitting Files into Manageable Chunks

Before embedding large text files, it's essential to divide them into chunks that are semantically coherent and size-appropriate for embedding models.

Key Strategies:

- **Fixed-length Chunking:** Dividing text into a fixed number of tokens (e.g., 512 tokens). This method is simple but can split context unnaturally.
- **Sentence-based Splitting:** Uses punctuation to break content into sentences or paragraphs, preserving context better.
- **Recursive Text Splitting:** A recursive strategy that attempts sentence splitting first, then falls back to smaller units (e.g., phrases or token-level) if chunks exceed limits.

- **Sliding Window Technique:** Overlapping chunks to preserve context at the boundaries, commonly used in RAG systems.

Recommendations for Implementation:

- Use libraries like LangChain or HuggingFace's Tokenizer tools.
- Define a tokenizer-aware chunking strategy (e.g., max 512 tokens per chunk for BERT).
- Maintain metadata (source, position) for each chunk to trace original context later.

---

3. Converting Chunks into Vector Embeddings

Once the file is chunked, each chunk is passed through a transformer model or embedding API to generate a dense vector representation.

Typical Workflow:

1. **Load Embedding Model:** (e.g., sentence-transformers/all-MiniLM-L6-v2)
2. **Preprocessing:** Clean text (remove special characters, normalize whitespace)
3. **Encode:** Use the model to transform each chunk into a high-dimensional vector.
4. **Store Output:** Return or store vector, often a list of floats (e.g., [0.123, -0.443, ...])

Model Considerations:

- Choose model based on task (retrieval vs. classification).
- API-based embeddings (e.g., OpenAI, Cohere) are quick to integrate but incur costs.
- Open-source alternatives like sentence-transformers offer customization and offline capability.

---

4. Storage in Vector Databases

Vector embeddings need to be stored in databases optimized for similarity search. Traditional SQL or NoSQL databases are not suitable for high-dimensional nearest-neighbor search.

Recommended Vector Databases:

Vector DB	Key Features	Ideal Use
FAISS (by Meta)	Fast, CPU/GPU, supports millions of vectors	Local experiments, prototyping
Pinecone	Fully managed, scalable, real-time search	Production-grade applications
Weaviate	Schema-based, hybrid search, RESTful APIs	RAG pipelines, semantic search
ChromaDB	Lightweight, used in LangChain	Personal and experimental apps
Milvus	Distributed, scalable, enterprise-ready	High-volume production systems

Steps to Store Vectors:

1. Generate unique ID for each chunk.
2. Store (ID, vector, metadata) tuple in the vector store.
3. Index for efficient retrieval (typically using HNSW or IVF algorithms).
4. Ensure capability for cosine similarity or dot product queries.

Practical Takeaways:

- **Chunking is as critical as embedding** — semantic preservation is key.
- **Use embedding models that align with your downstream tasks.**
- **Always include metadata (document ID, chunk number) in your storage plan.**
- **Choose the vector database based on your infrastructure and scalability needs.**

Task 2: Short Report on Retrieval-Augmented Generation (RAG)

What is RAG and How Does It Work?

Retrieval-Augmented Generation (RAG) is a hybrid technique that combines traditional information retrieval with generative AI models to improve the relevance and factual accuracy of responses.

How it works:

1. **Retrieve Phase:** Given a query, a retriever searches a document database using vector embeddings to find the most relevant content.
2. **Augment Phase:** Retrieved content is passed into a generative model (like GPT or T5), which uses it as context for crafting a more informed and grounded response.

Example Workflow:

- Input: “What is quantum computing?”
- Retriever fetches top-5 text chunks on quantum computing.
- Generator uses these chunks to generate a concise and accurate answer.

This pipeline allows the system to "know" about facts outside its training data by dynamically incorporating external knowledge.

Types of RAG

1. **RAG-Sequence:** Retrieves documents once and feeds them as a sequence to the generator.
2. **RAG-Token:** Retrieves documents at each decoding step, enabling more dynamic generation.
3. **Fusion-in-Decoder (FiD):** Embeds each document separately and merges them within the decoder.

4. **Retriever-Only + Generator (Two-Stage):** First retrieves, then generates without fusion; allows modular development.

Role of Vector Embeddings and Chunking in RAG

Both are foundational to RAG systems:

- **Vector Embeddings:** Used for representing both queries and documents in a shared semantic space. The closer the vectors, the more relevant the match.
- **Chunking:** Large documents must be split into digestible parts for retrieval. Poor chunking results in context loss or irrelevance during generation.

Best Practice: Use overlapping chunks and retain metadata so the generator knows where each piece came from.

Common Use Cases of RAG

- **Enterprise Q&A Bots:** Fetch company-specific documentation to answer internal queries.
- **Customer Support Systems:** Combine CRM knowledge with generative AI to handle user issues.
- **Healthcare Assistants:** Retrieve medical literature for condition-specific answers.
- **Academic Research Assistants:** Augment papers with up-to-date citations.
- **Legal Document Review:** Retrieve statutes and rulings dynamically for legal drafting.

Conclusion

Both vector embeddings and RAG are reshaping how machines understand and generate information. Embeddings allow for compact, meaningful representation of knowledge, while RAG empowers AI to remain current, verifiable, and domain-adaptable. When we move toward implementation, the focus should remain on effective chunking strategies, robust embedding models, and scalable, searchable vector storage.