Understanding Embeddings

What is Vector?

Vectors are multi-valued numeric representations of information, for example [10, 3, 1] in which each numeric element represents a particular attribute of the information. Each dimension of the vector represents some aspect or attribute of the data.

Imagine a pet's dataset where each pet is represented by three attributes:

- **Size** (on a scale from 1 to 10, where 1 is very small and 10 is very large).
- **Friendliness** (on a scale from 1 to 10, where 1 is not friendly and 10 is very friendly).
- **Energy Level** (on a scale from 1 to 10, where 1 is very low energy and 10 is very high energy).

Examples:

Dog: [6, 9, 8]

Cat: [4, 7, 5]

Hamster: [1, 6, 7]

Rabbit: [3, 8, 6]

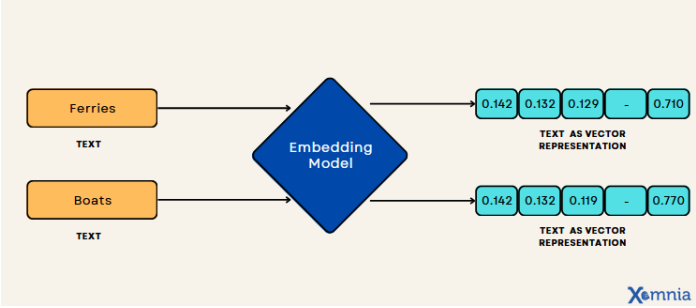
Parrot: [2, 7, 9]

Observations: Pets with similar attributes will have vectors that are close together.

- **cat** and **rabbit** are relatively close because their size, friendliness, and energy levels are similar.
- **dog** and **hamster** are far apart in vector space, since the dog is larger and has a different energy level.
- If you wanted to find a pet that's **friendly and has high energy**, **dog** and **parrot** would stand.
- If someone wanted a **low-energy** pet, **cat** or **rabbit** might be better choices.

What are [Vector] Embeddings

- Vector embeddings are numerical interpretations that **retain** the contextual significance of data, facilitating the alignment of similar entities within a vector space for similarity searches.



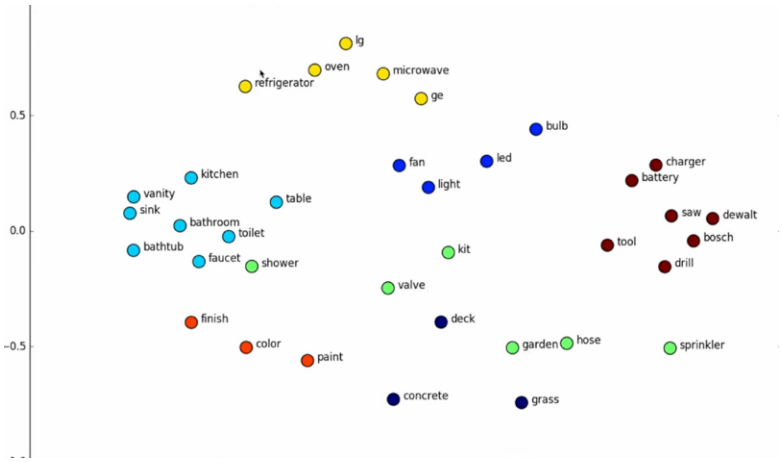
- Embeddings are the technique used to represent data in a meaningful way including semantic information.

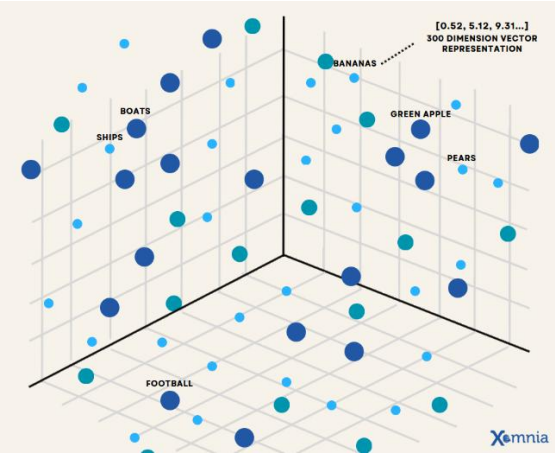
- These embeddings are learned and abstracted from the data, and their dimensions **don't directly correspond to any specific attributes** like size, friendliness, or energy.

Embeddings are commonly used for:

- **Search** (where results are ranked by relevance to a query string)
- **Clustering** (where text strings are grouped by similarity)
- **Recommendations** (where items with related text strings are recommended)
- **Anomaly detection** (where outliers with little relatedness are identified)
- **Diversity measurement** (where similarity distributions are analyzed)
- **Classification** (where text strings are classified by their most similar label)

Imagine an *n*-dimensional space with thousands of attributes about any **word's grammar, meaning, and use in sentences** mapped to a series of numbers.





Prompt	A neutron star is the collapsed core of a massive supergiant star	A star shines for most of its active life due to thermonuclear fusion.	The presence of a black hole can be inferred through its interaction with other matter

Embedding	[0.78, -0.12, 0.55, 0.65, -0.43 . . .]	[0.65, -0.15, 0.58, 0.55, -0.60]	[0.25, 0.30, -0.10, 0.15, 0.40]

- The key feature of embeddings is that similar items will have vectors that are **close together** in this vector space, even if the original data is very different.

Embedding Models

OpenAI offers two powerful third-generation embedding model (denoted by -3 in the model ID).

MODEL	PAGES PER DOLLAR	PERFORMANCE ON MTEB EVAL	MAX INPUT
text-embedding-3-small	62,500	62.3%	8191
text-embedding-3-large	9,615	64.6%	8191
text-embedding-ada-002	12,500	61.0%	8191

Deprecated Models:

- text-similarity-babbage-001
- text-similarity-curie-001
- text-search-davinci-doc-001

Example: Single Input Text

demo.py

```
from util import GetOpenAIClient


# Get OpenAI Client from Util.
client = GetOpenAIClient()
response = client.embeddings.create(
  input="Your text string goes here",
  model="text-embedding-3-small"
)
print(response.data[0].embedding)
```

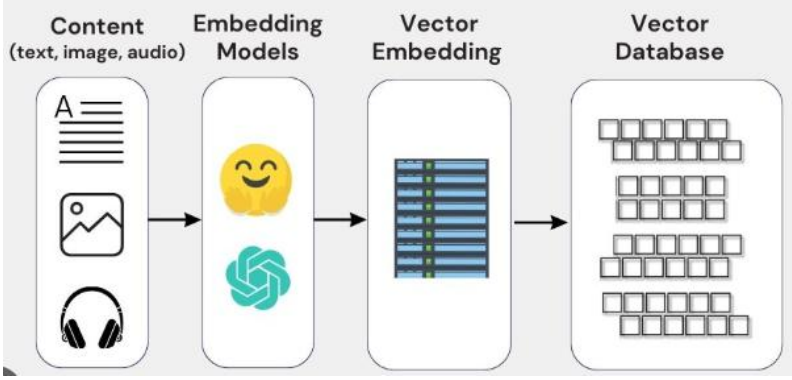
Embedding Response


```
{
"object": "list",
"data": [
 {
  "object": "embedding",
  "index": 0,
  "embedding": [
   -0.006929283495992422,
   -0.005536422007530928,
   … (omitted for spacing)
   -4.547132266452536e-05,
   -0.024047505110502243
  ],
 }
],
"model": "text-embedding-3-small",
"usage": {
 "prompt_tokens": 5,
 "total_tokens": 5
}
}
```


OpenAI embeddings rely on **cosine similarity** to compute similarity between documents and a query.
If two documents are far apart by Euclidean distance because of size, they could still have a smaller angle between them and therefore higher cosine similarity.

About Vector Database

- A **vector database** is a type of database systems designed to efficiently **store, index, and query** data in the form of vectors.
- **Indexing** employs advanced data structures such as **FAISS (Facebook AI Similarity Search)**, **HNSW (Hierarchical Navigable Small World graphs)**, or **LSH (Locality Sensitive Hashing)** for efficient querying in high-dimensional spaces.
- **Scalability:** Designed to handle millions or billions of vectors while maintaining fast query times.

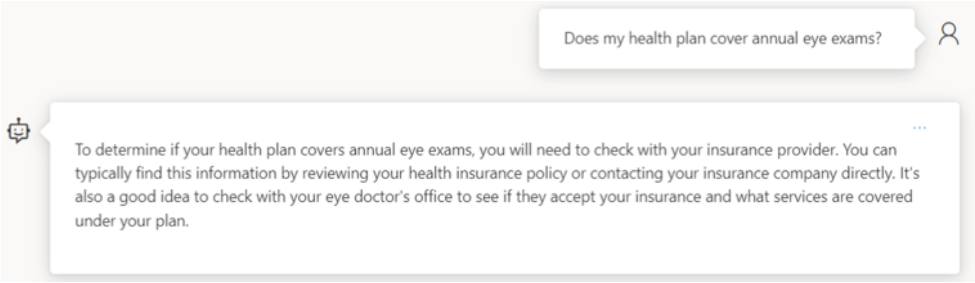


Popular Vector Databases:

1. **PostgreSQL** (with pgvector)
2. **MongoDB**
3. **Pinecone:** Specialized for machine learning use cases with fast and scalable similarity search.
4. **Weaviate:** Offers semantic search and supports various ML models.
5. **Chroma:** Focused on AI-first applications with seamless ML integration.
6. **Vespa:** Handles both structured and unstructured data queries.

Retrieval-Augmented Generation (RAG)

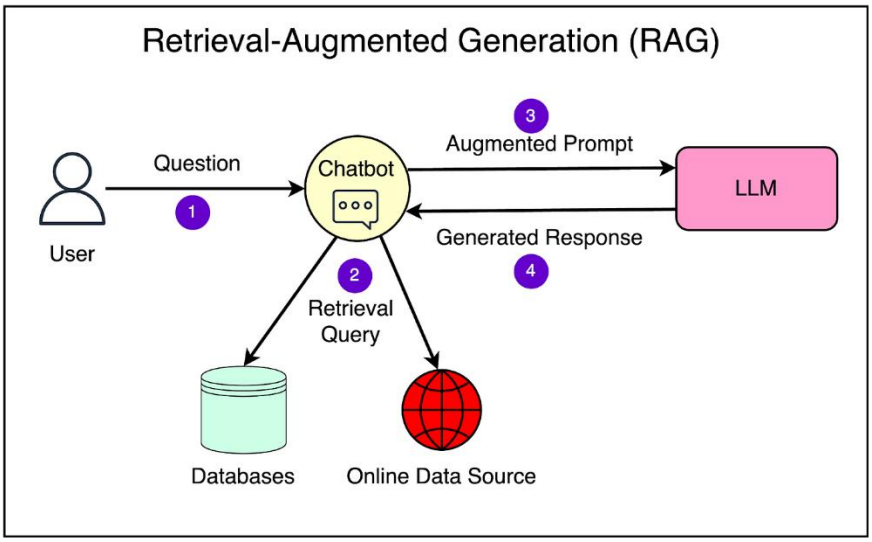
- In the landscape of conversational AI, Large Language Models (LLMs) are akin to encyclopedic repositories of general knowledge. They have an **extensive breadth** of information but often **lack depth** in specific, localized contexts, such as the intricacies of a company's internal database or the specialized findings of a research paper.



- RAG uses **your data** to generate answers to the user question.
- It allows your LLM to have domain-specific external information sources like your databases, documents, etc in real time. This way the LLM can get the most up-to-date and relevant information to answer the queries specific to your business.

Prompt : **"What is the price of Microsoft Stock today?" or "What is the temperature in London today"**
@ https://chatgpt.com/

@ https://www.perplexity.ai/



Here's a high-level overview of how a RAG system works:

1. The user poses a question to the RAG system.
2. The retrieval component searches the knowledge corpus using the question as a query and retrieves the most relevant passages or documents.
3. The retrieved content is passed to the LLM as additional context.
4. The language model processes the input and generates an answer by combining the information from the retrieved passages and its base knowledge.
5. The generated answer is returned to the user.

RAG has shown promising results in improving the accuracy and relevance of generated responses, especially in scenarios where the answer requires synthesizing information from multiple sources. It leverages the strengths of both information retrieval and language generation to provide better answers.

Advantages of RAG

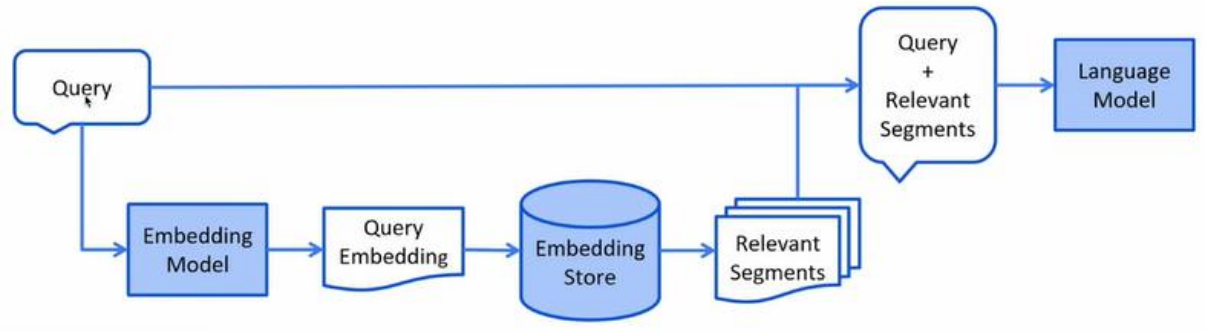
1. **Expanded Knowledge**: Enables models to answer questions outside their training data by accessing an external corpus.
2. **Efficiency**: Reduces the size of the generative model by offloading knowledge storage to the retriever. It has faster response time.
3. **Dynamic Updates**: The knowledge base can be updated independently of the model, making the system adaptable.

4. **Explainability**: Provides insight into why a response was generated by exposing the retrieved documents.

Example Use Cases:

- Handling complex customer queries that require product manuals or FAQs. Retriever fetches sections from the product manual about resetting procedures.
- Summarizing medical guidelines or providing information about rare diseases. Retrievers pull information from medical journals or trusted health databases.
- Summarizing case laws or regulations for lawyers. Retriever finds case summaries and rulings from a legal database.

RAG Demo in JavaScript



Step1: Create an account @ https://supabase.com/

Visit Home → Project Settings → Database → Connect → copy the connection string.

Eg of connection string:

postgresql://postgres:Sandeep@75@db.nnalijciffjswxgyywrn.supabase.co:5432/postgres.env.local

Step2: add to Project:

.env.local

```
OPEN_API_KEY=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
DATABASE_URL=postgresql://postgres:Sandeep@75@db.nnalijciffjswxgyywrn.supabase.co:5432/postgres
```

Step3: Add index.mjs (join all sections below)

```
import { config } from "dotenv";
```

```
import postgres from 'postgres'
import OpenAI from "openai";
import readline from 'readline';

config({ path: ".env.local" });
const openai = new OpenAI()
const connectionString = "postgresql://postgres.nnalijciffjswxgyywrn:Sandeep@75@aws-0-ap-southeast-1.pooler.supabase.com:6543/postgres"
const sql = postgres(connectionString)
```

```
async function InitializeDatabase() {
  // Connection string to the PostgreSQL database
  const connectionString = "postgresql://postgres.nnalijciffjswxgyywrn:Sandeep@75@aws-0-ap-southeast-1.pooler.supabase.com:6543/postgres"
  const sql = postgres(connectionString)

  // Create the vector extension if it doesn't exist
  await sql`
    CREATE EXTENSION IF NOT EXISTS vector;
    `;
  // Create the documents table if it doesn't exist
  await sql`
  CREATE TABLE IF NOT EXISTS "documents" (
  "id" serial PRIMARY KEY NOT NULL,
  "name" text NOT NULL,
  "content" text NOT NULL,
  "embedding" vector(256),
  "created_at" timestamp DEFAULT now() NOT NULL,
  "updated_at" timestamp DEFAULT now() NOT NULL
  );
`;
  // Create an index on the embedding column using hnsw and cosine similarity
  await sql`
  CREATE INDEX IF NOT EXISTS "embedding_index" ON "documents" USING hnsw ("embedding" vector_cosine_ops);
```

```
`;
}
```

```
export async function generateEmbeddings(texts) {
  const response = await openai.embeddings.create({
    model: "text-embedding-3-small",
    dimensions: 256,
    input: texts
  });
  return response.data.map((item) => item.embedding);
}
```

```
async function insertAndUpdateEmbeddings() {
  // Mock documents array with fun facts
  const knowledge_base = [
    { content: "A group of flamingos is called a 'flamboyance'.", name: "Fun Fact 1" },
    { content: "Octopuses have three hearts.", name: "Fun Fact 2" },
    { content: "Butterflies taste with their feet.", name: "Fun Fact 3" },
    { content: "A snail can sleep for three years.", name: "Fun Fact 4" },
    { content: "Elephants are the only animals that can't jump.", name: "Fun Fact 5" },
    { content: "A rhinoceros' horn is made of hair.", name: "Fun Fact 6" },
    { content: "Slugs have four noses.", name: "Fun Fact 7" },
    { content: "A cow gives nearly 200,000 glasses of milk in a lifetime.", name: "Fun Fact 8" },
    { content: "Bats are the only mammals that can fly.", name: "Fun Fact 9" },
    { content: "Koalas sleep up to 21 hours a day.", name: "Fun Fact 10" }
  ];
  // Generate embeddings for the content field using OpenAI API
  const response = await generateEmbeddings(knowledge_base.map((doc) => doc.content));
  // Insert documents into the database
  for (let i = 0; i < knowledge_base.length; i++) {
    const { name, content } = knowledge_base[i];

    const embeddingArray = `[${response[i].join(',')}]`;
    // Insert the record
    await sql`
```

```
        INSERT INTO documents (name, content, embedding)
        VALUES (${name}, ${content}, ${embeddingArray}::vector)
        `;
      console.log('Records inserted successfully');
  }
}
```

```
async function retrieveDocuments(query, limit = 3) {
  const embeddings = await generateEmbeddings([query]);
  const embedding = embeddings[0];
  const embeddingArray = `[${embedding.join(',')}]`;
  const documents = await sql`
    SELECT *, 1 - cosine_distance(embedding, ${embeddingArray}::vector) AS similarity
    FROM documents
    ORDER BY similarity DESC
    LIMIT ${limit}
  `;
  return documents;
}
```

Main code

```
// Check if there are any records in the documents table
const sqlResponse = await sql`
        SELECT count(*) as count FROM documents
  `;

if (sqlResponse.count != 0) {
  const rl = readline.createInterface({
    input: process.stdin,
    output: process.stdout
  });

  const answer = await new Promise((resolve) => {
    rl.question('Do you want to delete existing records? (yes/no): ', (input) => {
      rl.close();
```

```
        resolve(input.trim().toLowerCase());
      });
  });

  if (answer === 'yes') {
    console.log('Deleting existing records...');
    await sql`
      delete FROM documents
        `;
      await insertAndUpdateEmbeddings();
  } else {
    console.log('Keeping existing records...');
  }
}
```

Submitting RAG Query

```
const user_query = "I want to learn about animal sleep patterns"
const retrieved_doc = await retrieveDocuments(user_query);
const retrieved_str = JSON.stringify(retrieved_doc.map(doc => doc.content));

if (retrieved_doc != null) {
  const completion = await openai.chat.completions.create({
    model: "gpt-4o-mini",
    messages: [
      { role: "system", content: "You are a helpful assistant specialized about Animals." },
      { role: "user", content: "Question: " + user_query },
      { role: "assistant", content: "Relevant Document: " + retrieved_str }
    ],
  });
  console.log(completion.choices[0].message);
}
```