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]

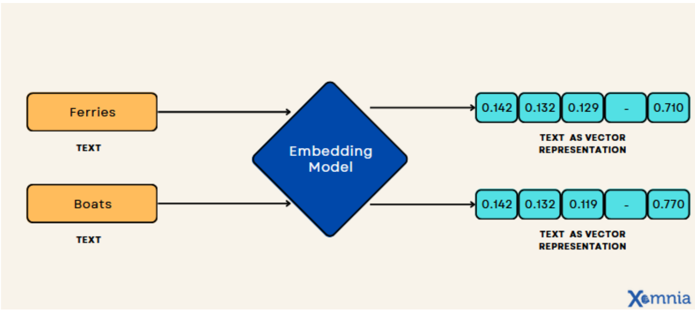
Parrot: [2, 7, 9]

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.



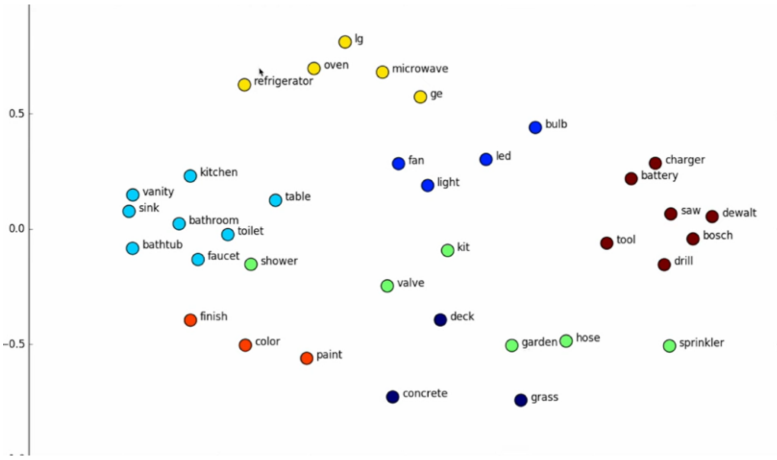
- Embeddings are the technique used to represent data in a meaningful way including **semantic information**.
- These embeddings are learned and abstracted from the data, and their dimensions **don't directly correspond to any specific attributes** like size, friendliness, or energy.

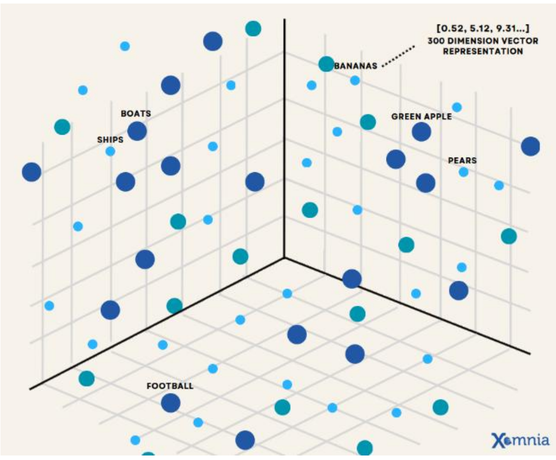
Embeddings are commonly used for:

- **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	MAX INPUT	NUMBER OF DIMENSIONS
text-embedding-3-small	62,500	8191	512
text-embedding-3-large	9,615	8191	1536
~~text-embedding-ada-002~~	~~12,500~~	~~8191~~	1536

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,
     -0.005336422007530928,
     ... (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.

Euclidean distance = 1 – cosine similary

Multiple Inputs in an Array:

Install Package

```
pip install numpy
```

demo.py

```python
import openai
import numpy as np
from util import getOpenAIClient

# Get OpenAI Client from Util.
client = getOpenAIClient()

# Sentences to be embedded
sentences = [
  "This is a Sample Code of OpenAI",
  "OpenAI Sample Code:",
  "Today is a holiday"
]

# Function to get embeddings from OpenAI
def get_embeddings(texts):
  response = client.embeddings.create(
    input=texts,
    dimensions=256,
```

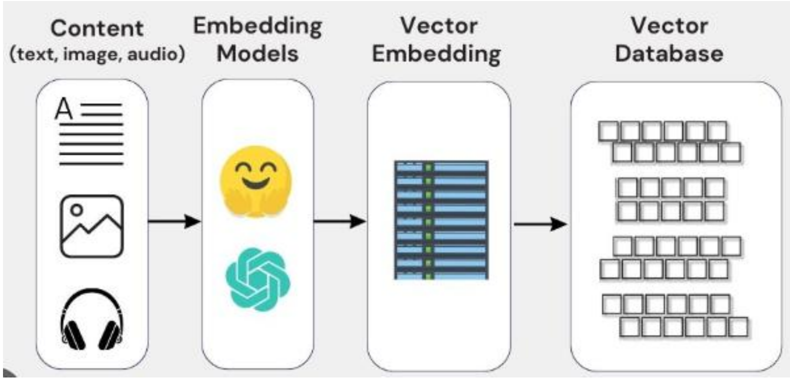
```python
        model="text-embedding-3-small"
    )
    #print(response.data)

    embeddings = []
    for data in response.data:
        embeddings.append(data.embedding)
        print(len(data.embedding))
    return embeddings

# Calculate cosine similarities
def cosine_similarity(a, b):
    return np.dot(a, b) / (np.linalg.norm(a) * np.linalg.norm(b))

# Get embeddings
embeddings = get_embeddings(sentences)
# Print cosine similarities between all texts in sentences array
print("Cosine Similarities:")
for i in range(len(sentences)):
    for j in range(i + 1, len(sentences)):
        similarity = cosine_similarity(embeddings[i], embeddings[j])
        print(f"Similarity between '{sentences[i]}' and '{sentences[j]}': {similarity}")
```

Output:

```
Cosine Similarities:
Similarity between 'This is a Sample Code of OpenAI' and 'OpenAI Sample Code': 0.9526287168427625
Similarity between 'This is a Sample Code of OpenAI' and 'Today is a holiday': 0.719602182615793
Similarity between 'OpenAI Sample Code' and 'Today is a holiday': 0.6786497460228316
```

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.

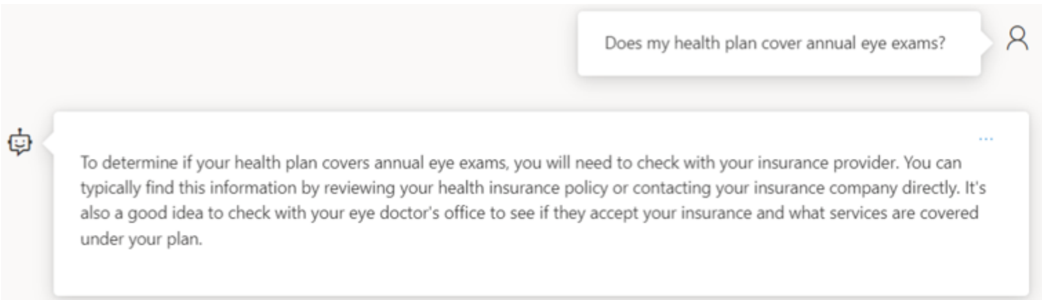


Popular Vector Databases:

1. **PostgreSQL** (with pgvector plugin)
2. **Mysql**
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.
7. **OpenSearch** with KNN Plugin

Retrieval-Augmented Generation (RAG)

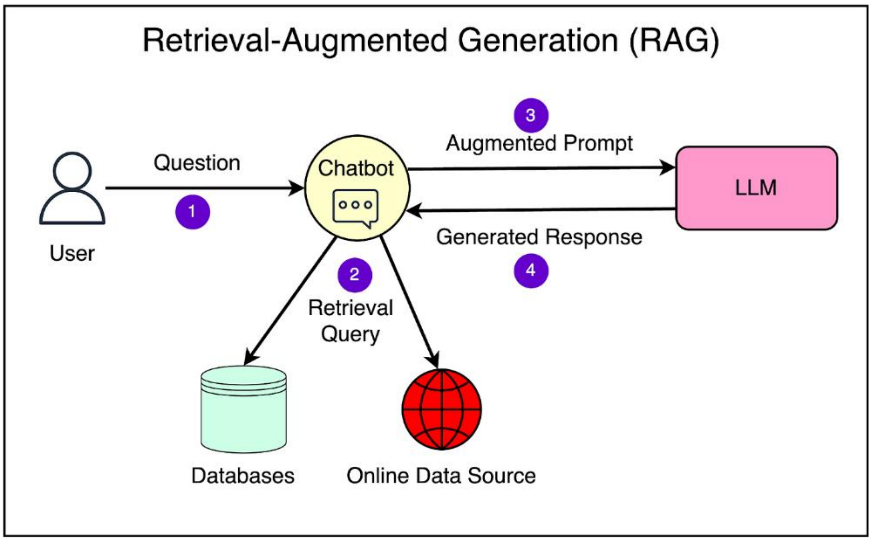
- In the landscape of conversational AI, Large Language Models (LLMs) are akin to encyclopedic repositories of general knowledge. They have an **extensive breadth** of information but often **lack depth** in specific, localized contexts, such as the intricacies of a company's internal database or the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

Prompt : **"What is the price of Microsoft Stock today?" or "What is the temperature in London today"**

@ https://chatgpt.com/



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.

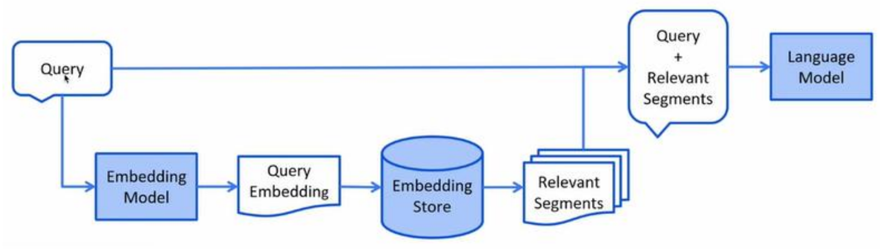
Advantages of RAG

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 Python example with OpenSearch as backend



Step-1: Create Table (Index) in OpenSearch and Insert Embeddings

Add the following to the .env file

```
OPENSEARCH_HOST=localhost
OPENSEARCH_PORT=9200
OPENSEARCH_USERNAME=admin
OPENSEARCH_PASSWORD=Opensearch#01
```

Install Package

```
pip install opensearch-py
```

initialize_db.py

```python
import os
from dotenv import load_dotenv
from opensearchpy import OpenSearch
from opensearchpy.helpers import bulk
from util import getOpenAIClient

load_dotenv()

host = os.environ.get('OPENSEARCH_HOST')
port = os.environ.get('OPENSEARCH_PORT')
username = os.environ.get('OPENSEARCH_USERNAME')
password = os.environ.get('OPENSEARCH_PASSWORD')

# OpenSearch configuration dictionary
OPENSEARCH_CONFIG = {
  "hosts": [{"host": host, "port": port}],
  "http_auth": (username, password),
  "http_compress": True,
  "use_ssl": True,
  "verify_certs": False,
```

```
    "ssl_assert_hostname": False,
    "ssl_show_warn": False
}

INDEX_NAME = "documents"

# Function to generate embeddings using OpenAI
def generate_embeddings(texts):
    # Generate embeddings for the given list of texts using OpenAI API.
    openai_client = getOpenAIClient();
    print("Got OPENAI CLIENT")
    response = openai_client.embeddings.create(input=texts, dimensions=256, model="text-embedding-3-
small")
    embeddings = [item.embedding for item in response.data]
    return embeddings

# Function to create OpenSearch index with knn_vector mapping
def create_opensearch_index(client):
    index_body = {
        "settings": {
            "index": {
                "knn": True  # Enable k-NN
            },
        },
        "mappings": {
            "properties": {
                "id": {"type": "long"},  # ID field (similar to serial)
                "name": {"type": "text"}, # Text field for the document name
                "content": {"type": "text"}, # Text field for the document content
                "embedding": {
                    "type": "knn_vector", # k-NN vector field for embeddings
                    "dimension": 256, # Dimension of the embedding vector
                    "method": { # Method for indexing the embeddings
                        "name": "hnsw", # Hierarchical Navigable Small World Graph used for indexing
                        "space_type": "cosinesimil", # Cosine similarity used for distance calculation
                        "engine": "nmslib" # NMSLIB library used for indexing
                    }
                },
                "created_at": {"type": "date"},  # Timestamp field for created_at
```

```
            "updated_at": {"type": "date"} # Timestamp field for updated_at
          }
        }
      }

    # Create the index (Table) if it does not exist
    if not client.indices.exists(INDEX_NAME):
        client.indices.create(index=INDEX_NAME, body=index_body)
        print(f"Index '{INDEX_NAME}' created.")

# Function to insert documents into OpenSearch
def insert_documents(client, knowledge_base, embeddings):
    """
    Insert documents into OpenSearch with embeddings.
    """
    actions = []
    for i, doc in enumerate(knowledge_base):
        action = {
            "_index": INDEX_NAME,
            "_id": i,
            "_source": {
                "name": doc["name"],
                "content": doc["content"],
                "embedding": embeddings[i]
            }
        }
        actions.append(action)

    success, _ = bulk(client, actions)
    print(f"Successfully inserted {success} documents into OpenSearch.")

# Main function to generate embeddings and insert documents
def main():
    # Mock documents array with fun facts
    knowledge_base = [
        {"content": "A group of flamingos is called a 'flamboyance'.", "name": "Fun Fact 1"},
        {"content": "Octopuses have five hearts.", "name": "Fun Fact 2"},
        {"content": "Butterflies taste with their feet.", "name": "Fun Fact 3"},
        {"content": "A snail can sleep for Five 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"}
    ]

    # Extract contents from the documents
    contents = []
    for doc in knowledge_base:
        contents.append(doc["content"])

    # Generate embeddings for the content
    embeddings = generate_embeddings(contents)

    # Connect to OpenSearch
    client = OpenSearch(**OPENSEARCH_CONFIG)

    # Create the OpenSearch index
    create_opensearch_index(client)

    # Insert documents with embeddings
    insert_documents(client, knowledge_base, embeddings)

# Entry point of the script
if __name__ == "__main__":
    main()
```

Step-2: Create Table (Index) in OpenSearch and Insert Embeddings

rag.py

```
import os
import numpy as np
from dotenv import load_dotenv
from opensearchpy import OpenSearch
from util import getOpenAIClient
from initialize_db import generate_embeddings # Python file of Step-1

load_dotenv()
```

```python
host = os.environ.get('OPENSEARCH_HOST')
port = os.environ.get('OPENSEARCH_PORT')
username = os.environ.get('OPENSEARCH_USERNAME')
password = os.environ.get('OPENSEARCH_PASSWORD')

# OpenSearch configuration
OPENSEARCH_CONFIG = {
    "hosts": [{"host": host, "port": port}],
    "http_auth": (username, password),
    "http_compress": True,
    "use_ssl": True,
    "verify_certs": False,
    "ssl_assert_hostname": False,
    "ssl_show_warn": False
}

INDEX_NAME = "documents"

# Function to calculate cosine similarity between two vectors
# Cosine similarity is a measure of similarity between two non-zero vectors of an inner product space that measures the cosine of the angle between them.
# np.linalg: This function uses numpy's dot product and linear algebra norm functions to compute the cosine similarity.
def cosine_similarity(vec1, vec2):
    return np.dot(vec1, vec2) / (np.linalg.norm(vec1) * np.linalg.norm(vec2))

# Function to retrieve documents from OpenSearch based on cosine similarity
def retrieve_documents(opensearch_client, user_query, limit=3):
    # Generate the embedding for the query
    query_embedding = generate_embeddings(user_query)[0]

    # Perform the OpenSearch search to get all documents
    search_body = {
        "_source": ["content"], # Only retrieve necessary fields
        "query": {
            "knn": {
                "embedding":{
                    "vector": query_embedding,
```

```python
                "k": limit,
            }
        }
    }
  }
  response = opensearch_client.search(index=INDEX_NAME, body=search_body)

  # Extract documents and their embeddings
  documents_string = ''

  # # match_all query returns all documents, so we need to filter based on cosine similarity
  for hit in response["hits"]["hits"]:
    doc = hit["_source"]
    documents_string += doc['content']

  print('---------------------------------')
  for ele in documents_string.split('.'):
    print(ele, sep='\n')
  print('---------------------------------')

  return documents_string

# Function to interact with OpenAI and generate a response based on the retrieved documents
def generate_chat_response(user_query, retrieved_string):
  openai_client = getOpenAIClient()

  completion = openai_client.chat.completions.create(
    model="gpt-4o-2024-08-06",
    messages=[
      {"role": "system", "content": "You are a helpful assistant specialized about Animals answering questions using the context as the primary source of information and don't include content not in context"},
      #{"role": "system", "content": "You are a helpful assistant specialized about Animals answering questions using the context as the primary source of information and also include content not in context"},
      {"role": "user", "content": f"Question: {user_query} \n Context: {retrieved_string}"},
      #{"role": "assistant", "content": f"Relevant Document: {retrieved_string}"}
    ]
  )
  return completion.choices[0].message.content
```

```python
# Main function for testing
def main():
  # User query for information
  user_query = "I want to learn about animal sleep patterns"

  # Connect to OpenSearch
  opensearch_client = OpenSearch(**OPENSEARCH_CONFIG)

  # Retrieve documents based on the user query
  retrieved_string = retrieve_documents(opensearch_client, user_query, limit=3)

  if retrieved_string:
    # Generate a response based on the retrieved documents
    response = generate_chat_response(user_query, retrieved_string)
    print("Response from OpenAI Assistant:", response)
  else:
    print("No relevant documents found.")

if __name__ == "__main__":
  main()
```

More Vector Database Examples
https://github.com/openai/openai-cookbook/tree/main/examples/vector_databases