Embeddings and Vector Databases

Author: Luna Wang

What are embeddings

Embeddings are numerical representations of real-world objects, like text, images or audio. They are generated by machine learning models as vectors, an array of numbers, where the distance between vectors can be seens as the degree of similarity between objects. While an embedding model may have its own meaning for each of the dimensions, there's no guarantee between embedding models of the meaning for each of the dimensions used by the embedding models.

For example, the word "cat", "dog" and "apple" might be embedded into the following vectors:

cat -> (1, -1, 2)
dog -> (1.5, -1.5, 1.8)
Apple -> (-1, 2, 0)

These vectors are made-up for a simpler example. Real vectors are much larger, see the Dimension section for details.

Visualizing these vectors as points in a 3D space, we can see that "cat" and "dog" are closer, while "apple" is positioned further away.

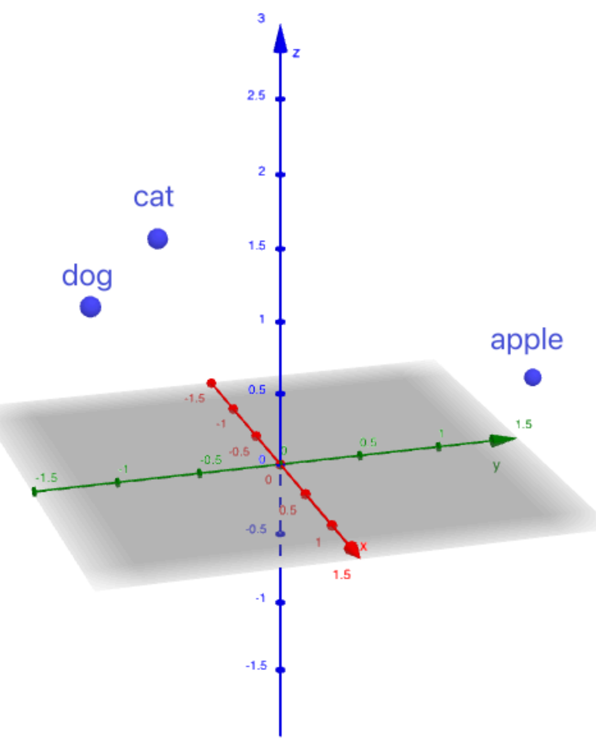

Figure 1. Vectors as points in a 3D space

By embedding words and contexts into vectors, we enable systems to assess how related two embedded items are to each other via vector comparison.

Dimension of embeddings

The dimension of embeddings refers to the length of the vector representing the object.

In the previous example, we embedded each word into a 3-dimensional vector. However, a 3-dimensional embedding inevitably leads to a massive loss of information. In reality, word embeddings typically require hundreds or thousands of dimensions to capture the nuances of language.

For example,

- OpenAI's text-embedding-ada-002 model outputs a 1536-dimensional vector
- Google Gemini's text-embedding-004 model outputs a 768-dimensional vector
- Amazon Titan's amazon.titan-embed-text-v2:0 model outputs a default 1024-dimensional vector

"I have a calico cat." → [-0.014509869, -0.009199489, 0.010892092, ... , 0.008430712, -0.04829732]

a 1536-dimensional vector

Figure 2. Using text-embedding-ada-002 to embed the sentence “I have a calico cat.”

In short, an embedding is a vector that represents a real-world object. The distance between these vectors indicates the similarity between the objects.

Limitation of embedding models

Embedding models are subject to a crucial limitation: the token limit, where a token can be a word, punctuation mark, or subword part. This constraint defines the maximum amount of text a model can process in a single input. For instance, the [Amazon Titan Text Embeddings models]() can handle up to 8,192 tokens.

When input text exceeds the limit, the model typically truncates it, discarding the remaining information. This can lead to a loss of context and diminished embedding quality, as crucial details might be omitted.

To address this, several strategies can help mitigate its impact:

- Text Summarization or Chunking: Long texts can be summarized or divided into smaller, manageable chunks before embedding.
- Model Selection: Different embedding models have varying token limits. Choosing a model with a higher limit can accommodate longer inputs.

What is a Vector Database

Vector databases are optimized for storing embeddings, enabling fast retrieval and similarity search. By calculating the similarity between the query vector and the other vectors in the database, the system returns the vectors with the highest similarity, indicating the most relevant content.

The following diagram illustrates a vector database search. A query vector 'favorite sport' is compared to a set of stored vectors, each representing a text phrase. The nearest neighbor, 'I like football', is returned as the top result.

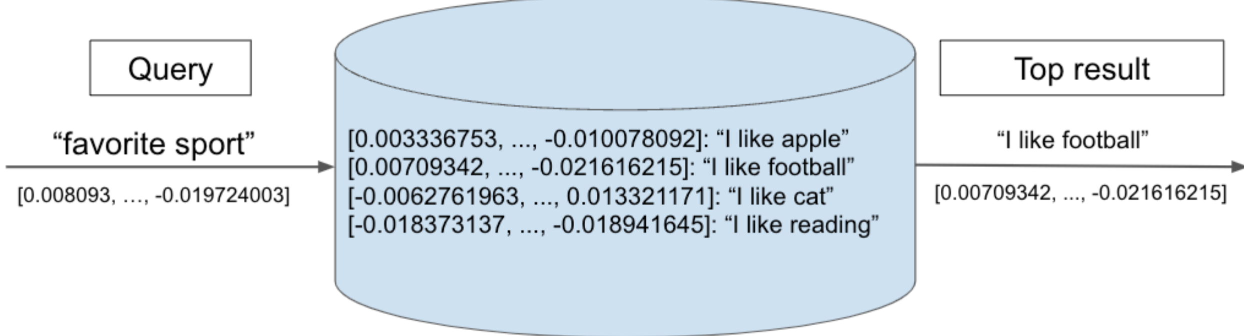

Figure 3. Vector Query Example

Prepare Content → **Embed** → **Store**

Prepare your content for vectorization. This often involves pre-processing, such as breaking down long documents into smaller chunks for more focused embeddings.

Each object to be stored requires a vector generated by an embedding model. Consistent dimensionality is essential for vector comparison and subsequent similarity calculation.

The generated vectors, linked to their original texts and potentially other associated data, will be stored in the database.

Figure 4. **Store** Vectors into Database

Embed the Query → **Retrieve Vectors**

The query will be transformed into a vector using the identical embedding model employed for the database objects.

The database will search for the vectors closest to the query vector. The original texts corresponding to these nearest vectors are considered the most relevant content.

Figure 5. **Retrieve** Vectors from Database

When working with vector databases, two key parameters come into play: **Top K** and **similarity measure** (or distance function).

Top K

When querying a vector database, the goal is often to retrieve the most similar items to a given query vector. This is where the Top K concept comes into play. Top K refers to retrieving the top K most similar items based on a similarity metric.

For instance, if you're building a product recommendation system, you might want to find the top 10 products similar to the one a user is currently viewing. In this case, K would be 10. The vector database would return the 10 product vectors closest to the query product's vector.

Similarity Measures

To determine the similarity between vectors, various distance metrics are employed, including:

- **Cosine Similarity:** This measures the cosine of the angle between two vectors. It is often used for text-based applications as it captures semantic similarity well. A value closer to 1 indicates higher similarity.
- **Euclidean Distance:** This calculates the straight-line distance between two points in Euclidean space. It is sensitive to magnitude differences between vectors.
- **Manhattan Distance:** Also known as L1 distance, it calculates the sum of the absolute differences between corresponding elements of two vectors. It is less sensitive to outliers compared to Euclidean distance.

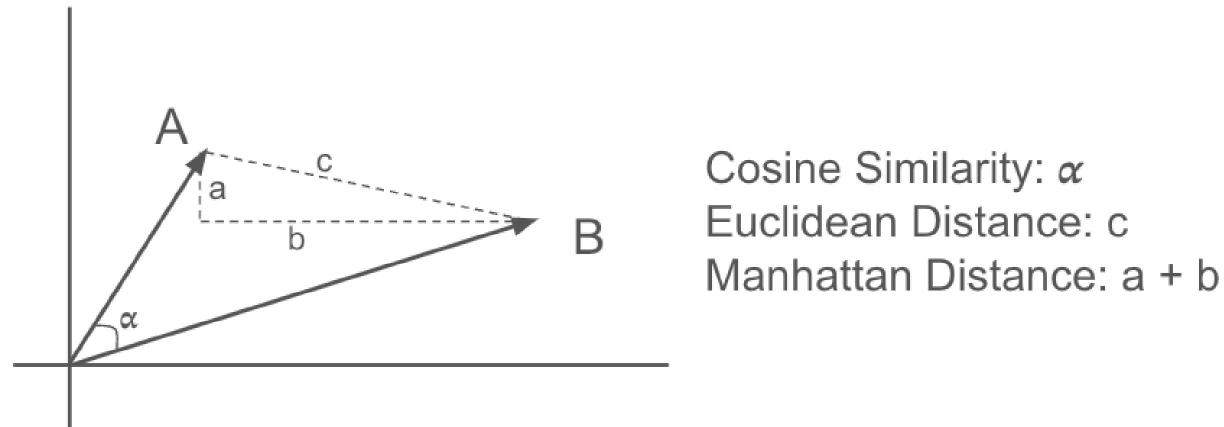

Figure 6. Similarity Measures

There are many other similarity measures not listed here. The choice of distance metric depends on the specific application and the nature of the data. It is recommended to experiment with various similarity metrics to see which one produces better results.

What embedders are supported in SnapLogic

As of August 2024, SnapLogic has supported embedders for major models and continues to expand its support. Supported embedders include:

- Amazon Titan Embedder
- OpenAI Embedder
- Azure OpenAi Embedder
- Google Gemini Embedder

What vector databases are supported in SnapLogic

- Pinecone
- OpenSearch
- MongoDB
- Snowflake
- Postgres
- AlloyDB

Pipeline examples

Embed a text file

1. Read the file using the **File Reader** snap.
2. Convert the binary input to a document format using the **Binary to Document** snap, as all embedders require document input.
3. Embed the document using your chosen **embedder** snap.

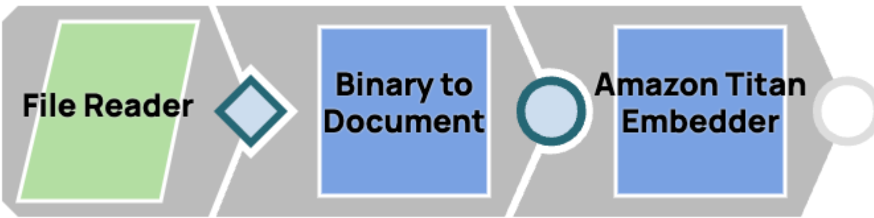

Figure 7. Embed a File


```
[
  {
    "embedding": [0.203125, -0.07861328, 0.28710938, 0.16308594, 0.4296875, 0.055664062, 0.023071289, 5.874634E-4, -0.... ],
    "inputTextTokenCount": 498,
    "original": {"content-type": "text/plain; charset=utf-8", "date": "Tue, 06 Aug 2024 19:47:53 GMT", "server": "env...}
  }
]
```

Figure 8. Output of the Embedder Snap

Store a Vector

1. Utilize the **JSON Generator** snap to simulate a document as input, containing the original text to be stored in the vector database.
2. Vectorize the original text using the **embedder** snap.
3. Employ a **mapper** snap to format the structure into the format required by Pinecone - the vector field is named "values", and the original text and other relevant data are placed in the "metadata" field.
4. Store the data in the vector database using the vector database's **upsert/insert** snap.

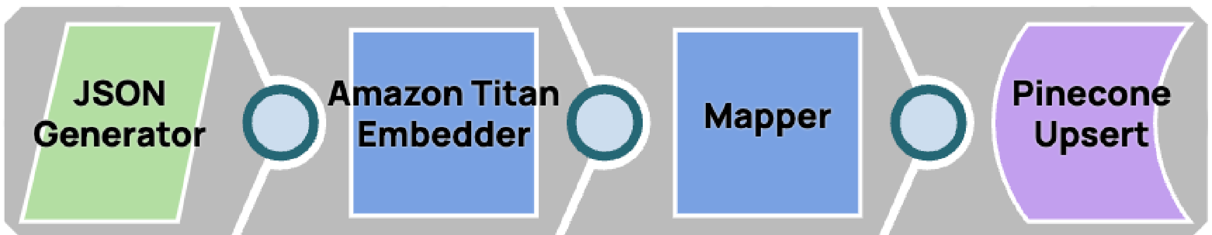

Figure 9. Store a Vector into Database

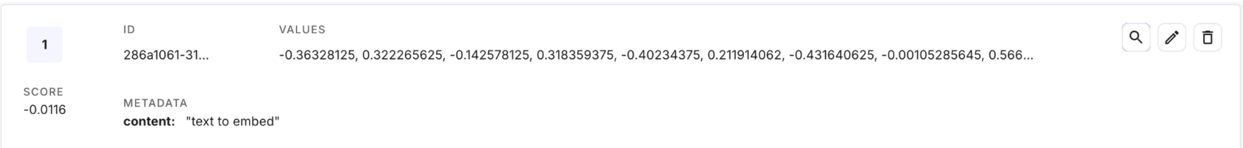

Figure 10. A Vector in the Pinecone Database

Retrieve Vectors

1. Utilize the JSON Generator snap to simulate the text to be queried.
2. Vectorize the original text using the embedder snap.
3. Employ a mapper snap to format the structure into the format required by Pinecone, naming the query vector as "vector".
4. Retrieve the top 1 vector, which is the nearest neighbor.

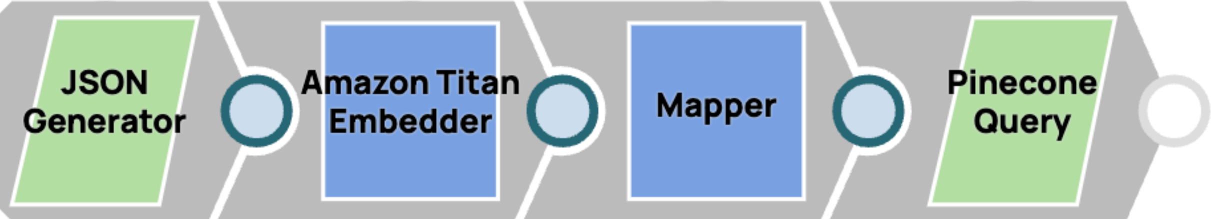

Figure 11. Retrieve Vectors from a Database

```javascript
[
    {
        "content" : "favorite sport"
    }
]
```

Figure 12. Query Text

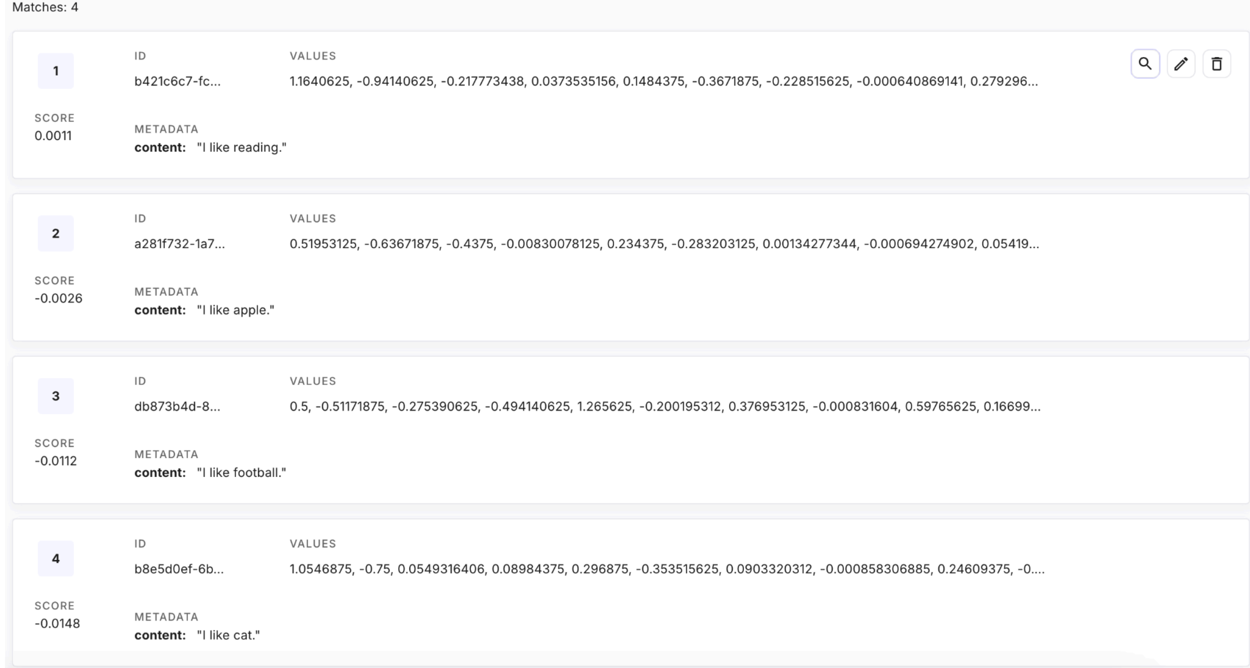

Figure 13. All Vectors in the Database

```javascript
"matches":[
        {
                "id": "db873b4d-81d9-421c-9718-5a2c2bd9e720"
                "score": 0.547461033
                "values":[]
                "metadata":{
```

```
            "content":"I like football."
        }
    }
]
```

Figure 14. Pipeline Output: the Closest Neighbor to the Query

Embedder and vector databases are widely used in applications such as Retrieval Augmented Generation (RAG) and building chat assistants.

Multimodal Embeddings

While the focus thus far has been on text embeddings, the concept extends beyond words and sentences. Multimodal embeddings represent a powerful advancement, enabling the representation of various data types, such as images, audio, and video, within a unified vector space. By projecting different modalities into a shared semantic space, complex relationships and interactions between these data types can be explored.

For instance, an image of a cat and the word "cat" might be positioned closely together in a multimodal embedding space, reflecting their semantic similarity. This capability opens up a vast array of possibilities, including image search with text queries, video content understanding, and advanced recommendation systems that consider multiple data modalities.