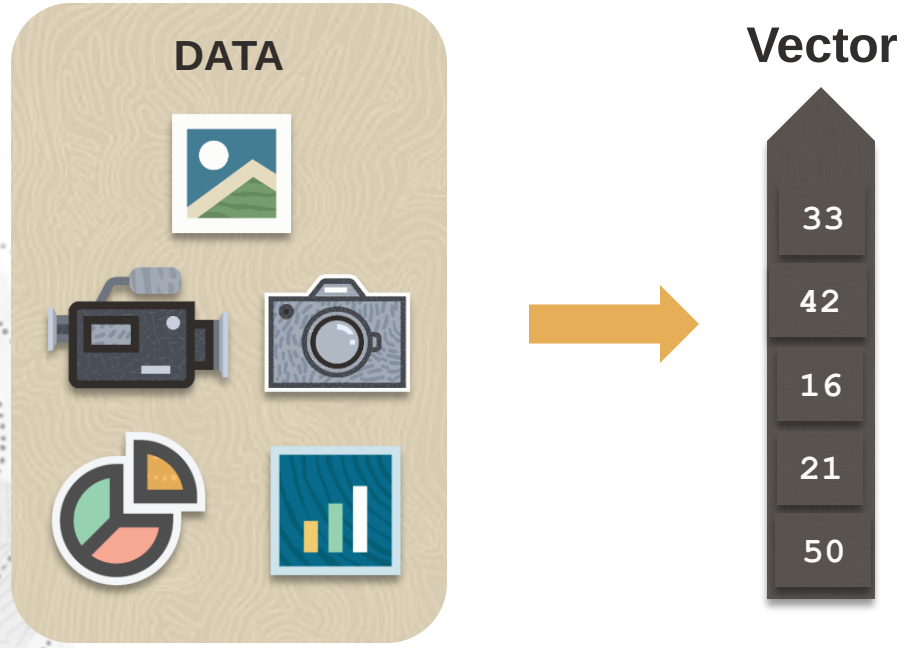
Vector Databases and Embeddings

- **Vector Databases**
- **Working with Embeddings**

Ram N Sangwan

Vector

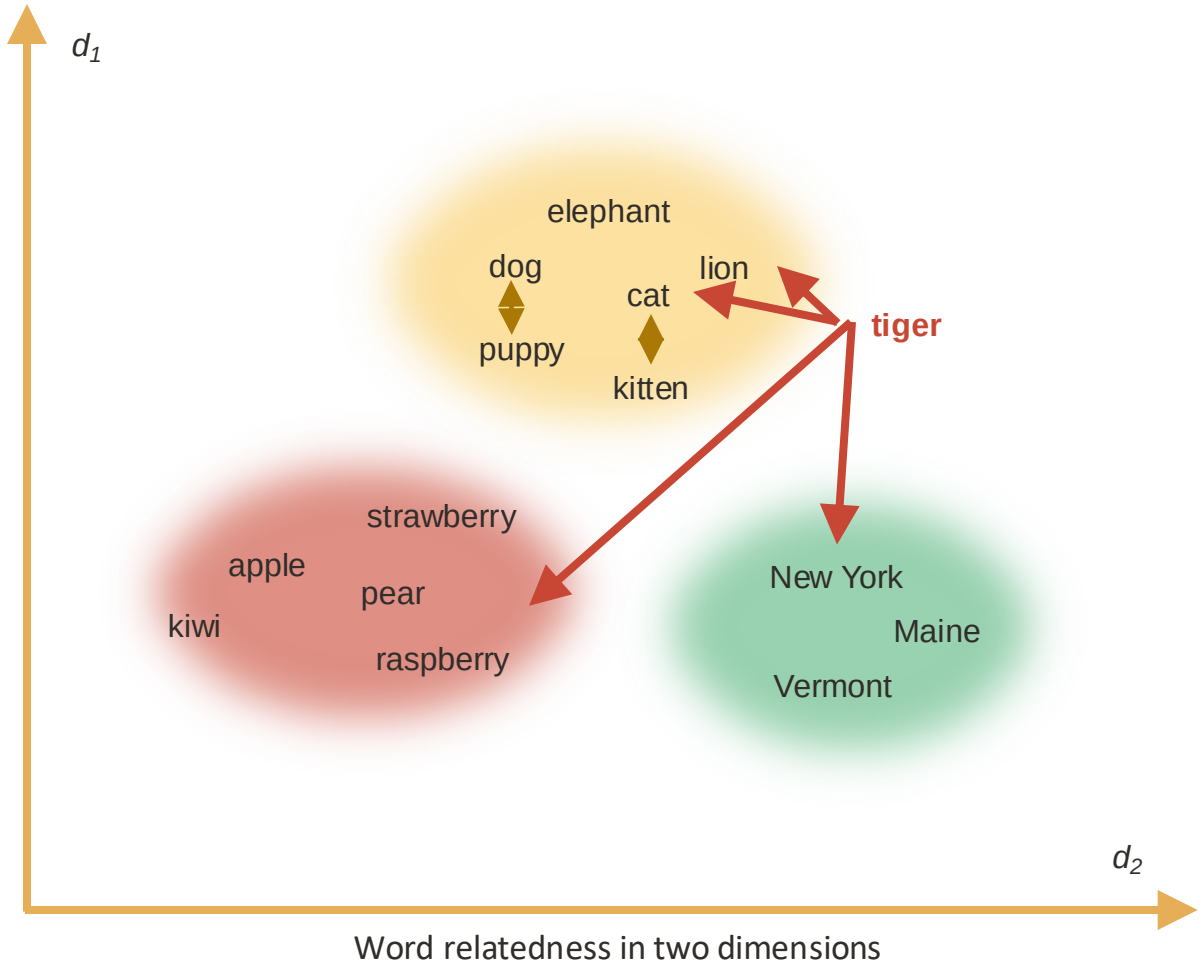


A vector is a sequence of numbers, called dimensions, used to capture the important "features" of the data.

Embeddings in LLMs are essentially high-dimensional vectors.

Vectors are generated using deep learning embedding models and represent the semantic content of data, not the underlying words or pixels.

Vector



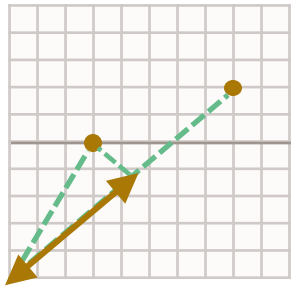
There are three groups of words here based on similarity: **animals, fruit, places.**

"Tiger" is closest to the Animals group, closer to cat family members.

Optimized for multidimensional spaces where the relationship is based on distances and similarities in a high-dimensional vector space

Embedding Distance

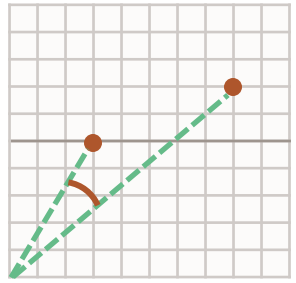
Dot Product



$$A.B = \sum_{i=1}^{n} A_i B_i$$

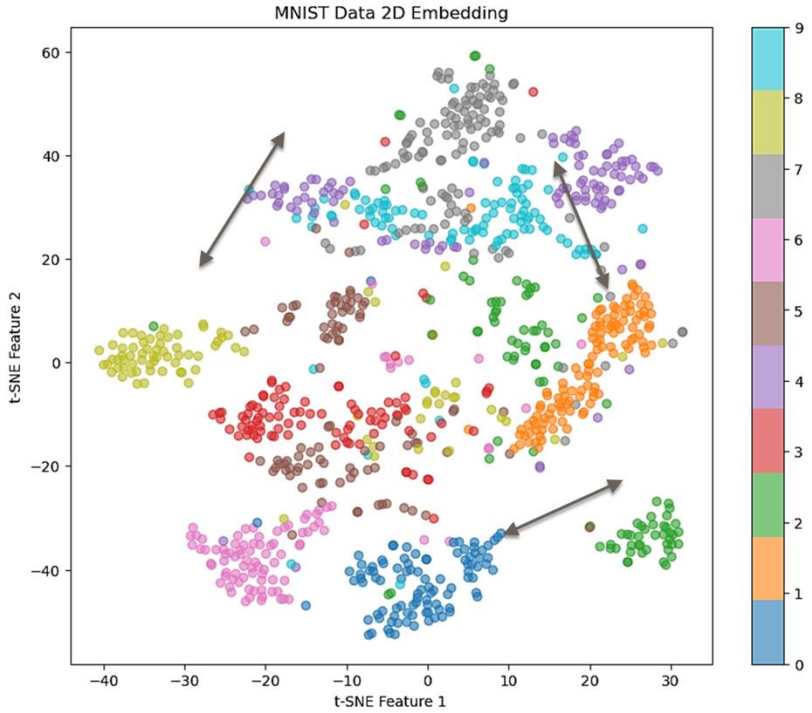
Dot Product is the Measure of the magnitude of the projection of one vector on to the other and gives you magnitude and direction.

Cosine Distance



$$1 - \frac{A \ . \ B}{||A|| \ \ ||B||}$$

Cosine Distance is the Measure of the difference in directionality between vectors so that only the angle between the vectors matters, not their magnitude.

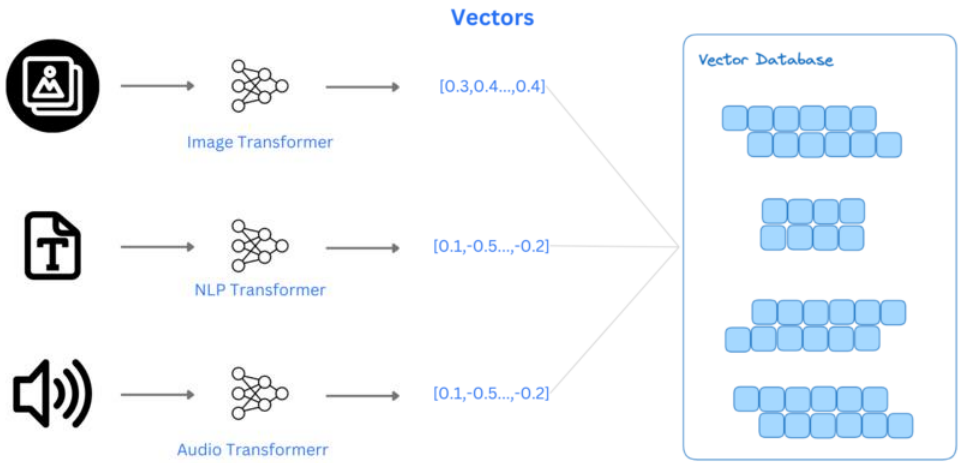


- This is a scatter plot of the MNIST handwritten dataset reduced to two dimensions using t-SNE, which is a technique for dimensionality reduction that is particularly well suited for the visualization of high-dimensional datasets.
- The plot shows clusters of points in different colors, each representing one of the digits from 0 to 9 from the dataset.
- You can see that how this was able to separate and group embeddings in case of images.

Vector Databases

What are Vector Databases?

- A **vector database** is a specialized type of **database** that **indexes** and **stores vector embeddings** for **fast retrieval** and **similarity search**.

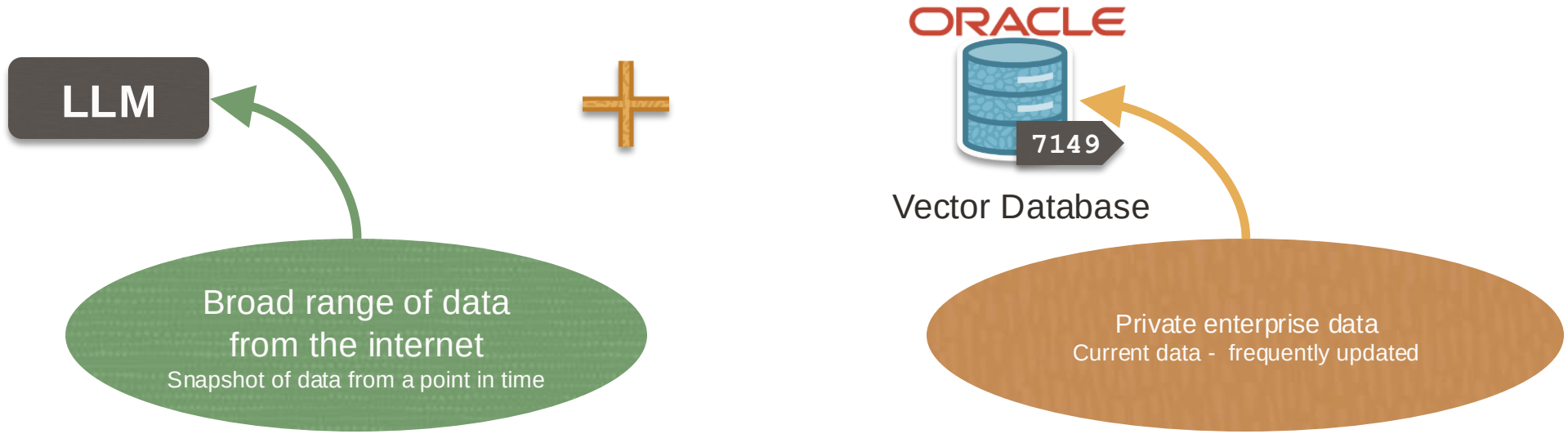


Vector Databases

- Oracle AI Vector Search, FAISS, Pinecone, Chroma and Weaviate are some prominent vector databases.
- Oracle already holds **most of the world's operational enterprise data.**
- **So, it will be an order of magnitude easier** to **add vector embeddings to already stored enterprise data in Oracle databases** than replicating enterprise data to a separate vector database.

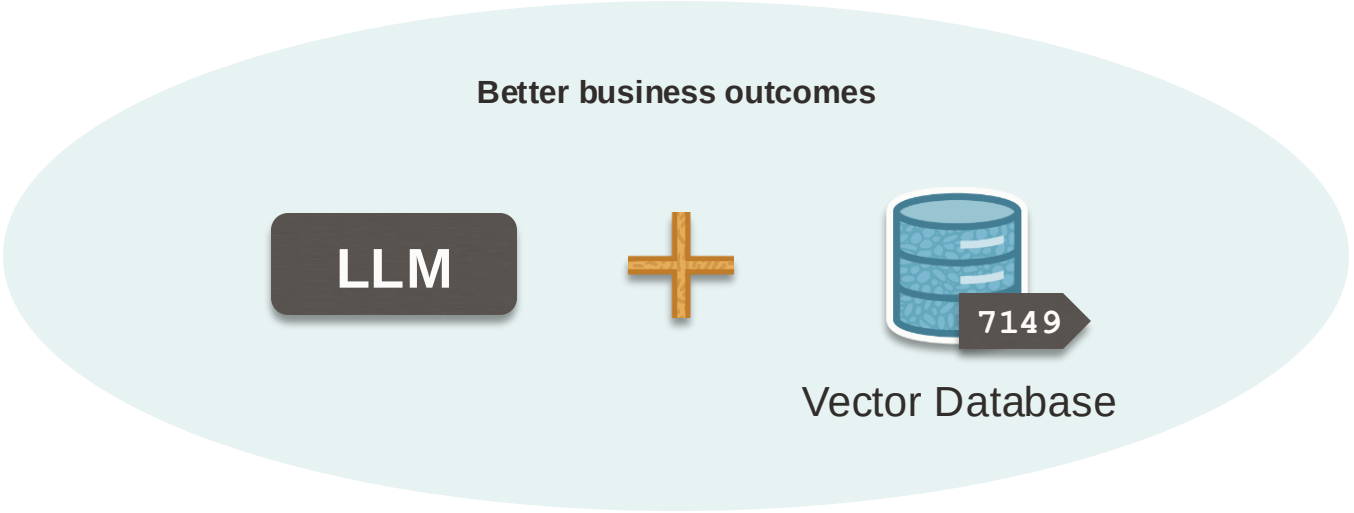
Role of Vector Databases with LLMs

- Address the hallucination (i.e., inaccuracy) problem inherent in LLM responses.
- Augment prompt with enterprise-specific content to produce better responses.
- Avoid exceeding LLM token limits by using most relevant content.

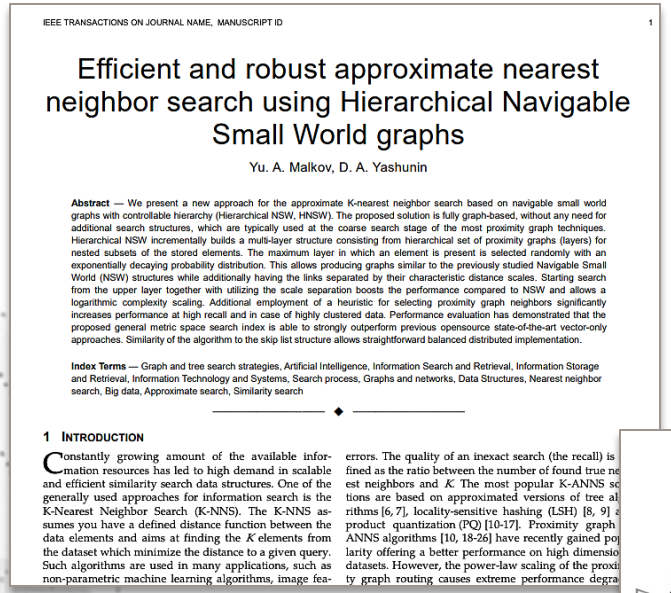


Role of Vector Databases with LLMs

- Cheaper than fine-tuning LLMs, which can be expensive to update
- Real-time updated knowledge base
- Cache previous LLM prompts/responses to improve performance and reduce costs



Similar Vectors



IEEE TRANSACTIONS ON JOURNAL NAME, MANUSCRIPT ID 1

Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs

Yu. A. Malkov, D. A. Yashunin

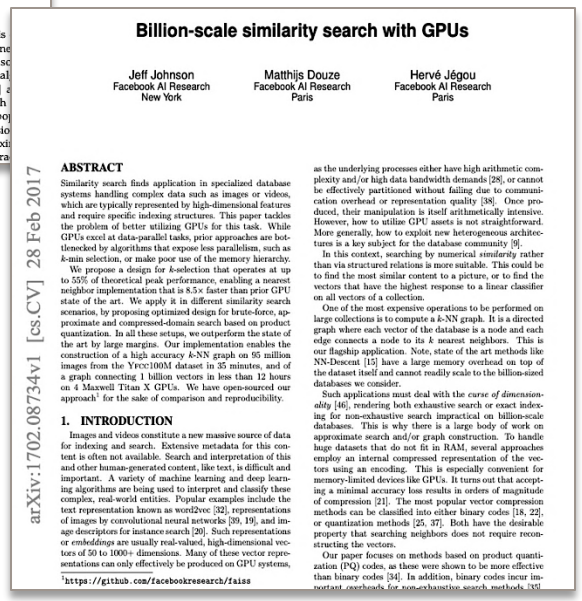
Abstract — We present a new approach for the approximate K-nearest neighbor search based on navigable small world graphs with controllable hierarchy (Hierarchical NSW, HNSW). The proposed solution is fully graph-based, without any need for additional search structures, which are typically used at the coarse search stage of the most proximity graph techniques. Hierarchical NSW incrementally builds a multi-layer structure consisting from hierarchical set of proximity graphs (layers) for nested subsets of the stored elements. The maximum layer in which an element is present is selected randomly with an exponentially decaying probability distribution. This allows producing graphs similar to the previously studied Navigable Small World (NSW) structures while additionally having the links separated by their characteristic distance scales. Starting search from the upper layer together with utilizing the scale separation boosts the performance compared to NSW and allows a logarithmic complexity scaling. Additional employment of a heuristic for selecting proximity graph neighbors significantly increases performance at high recall and in case of highly clustered data. Performance evaluation has demonstrated that the proposed general metric space search index is able to strongly outperform previous opensource state-of-the-art vector-only approaches. Similarity of the algorithm to the skip list structure allows straightforward balanced distributed implementation.

Index Terms — Graph and tree search strategies, Artificial Intelligence, Information Search and Retrieval, Information Storage and Retrieval, Information Technology and Systems, Search process, Graphs and networks, Data Structures, Nearest neighbor search, Big data, Approximate search, Similarity search

1 Introduction

Constantly growing amount of the available information resources has led to high demand in scalable and efficient similarity search data structures. One of the generally used approaches for information search is the K-Nearest Neighbor Search (K-NNS). The K-NNS assumes you have a defined distance function between the data elements and aims at finding the K elements from the dataset which minimize the distance to a given query. Such algorithms are used in many applications, such as non-parametric machine learning algorithms, image fea-

errors. The quality of an inexact search (the recall) is fined as the ratio between the number of found true ne est neighbors and K. The most popular K-ANNS sc tions are based on approximated versions of tree al rithms [6, 7], locality-sensitive hashing (LSH) [8, 9] product quantization (PQ) [10-17]. Proximity graph ANNS algorithms [10, 18-26] have recently gained po larity offering a better performance on high dimensio datasets. However, the power-law scaling of the proxi ty graph routing causes extreme performance degra



Billion-scale similarity search with GPUs

Jeff Johnson, Facebook AI Research, New York
Matthijs Douze, Facebook AI Research, Paris
Hervé Jégou, Facebook AI Research, Paris

arXiv:1702.08734v1 [cs.CV] 28 Feb 2017

ABSTRACT

Similarity search finds application in specialized database systems handling complex data such as images or videos, which are typically represented by high-dimensional features and require specific indexing structures. This paper tackles the problem of better utilizing GPUs for this task. While GPUs excel at data-parallel tasks, prior approaches are bottlenecked by algorithms that expose less parallelism, such as k-min selection, or make poor use of the memory hierarchy.

We propose a design for k-selection that operates at up to 55% of theoretical peak performance, enabling a nearest neighbor implementation that is 8.5× faster than prior GPU state of the art. We apply it in different similarity search scenarios, by proposing optimized design for brute-force, approximate and compressed-domain search based on product quantization. In all these setups, we outperform the state of the art by large margins. Our implementation enables the construction of a high accuracy k-NN graph on 95 million images from the YFCC100M dataset in 35 minutes, and of a graph connecting 1 billion vectors in less than 12 hours on 4 Maxwell Titan X GPUs. We have open-sourced our approach[1] for the sake of comparison and reproducibility.

1. INTRODUCTION

Images and videos constitute a new massive source of data for indexing and search. Extensive metadata for this content is often not available. Search and interpretation of this and other human-generated content, like text, is difficult and important. A variety of machine learning and deep learning algorithms are being used to interpret and classify these complex, real-world entities. Popular examples include the text representation known as word2vec [32], representations of images by convolutional neural networks [39, 19], and image descriptors for instance search [20]. Such representations or *embeddings* are usually real-valued, high-dimensional vectors of 50 to 1000+ dimensions. Many of these vector representations can only effectively be produced on GPU systems, as the underlying processes either have high arithmetic complexity and/or high data bandwidth demands [28], or cannot be effectively partitioned without failing due to communication overhead or representation quality [38]. Once produced, their manipulation is itself arithmetically intensive. However, how to utilize GPU assets is not straightforward. More generally, how to exploit new heterogeneous architectures is a key subject for the database community [9].

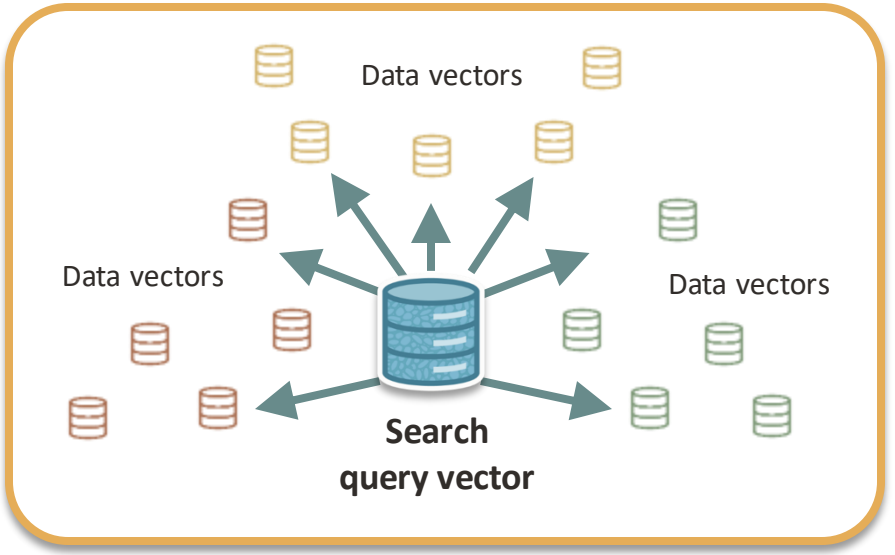
In this context, searching by numerical *similarity* rather than via structured relations is more suitable. This could be to find the most similar content to a picture, or to find the vectors that have the highest response to a linear classifier on all vectors of a collection.

One of the most expensive operations to be performed on large collections is to compute a k-NN graph. It is a directed graph where each vector of the database is a node and each edge connects a node to its k nearest neighbors. This is our flagship application. Note, state of the art methods like NN-Descent [15] have a large memory overhead on top of the dataset itself and cannot readily scale to the billion-sized databases we consider.

Such applications must deal with the *curse of dimensionality* [46], rendering both exhaustive search or exact indexing for non-exhaustive search impractical on billion-scale databases. This is why there is a large body of work on approximate search and/or graph construction. To handle huge datasets that do not fit in RAM, several approaches employ an internal compressed representation of the vectors using an encoding. This is especially convenient for memory-limited devices like GPUs. It turns out that accepting a minimal accuracy loss results in orders of magnitude of compression [21]. The most popular vector compression methods can be classified into either binary codes [18, 22], or quantization methods [25, 37]. Both have the desirable property that searching neighbors does not require reconstructing the vectors.

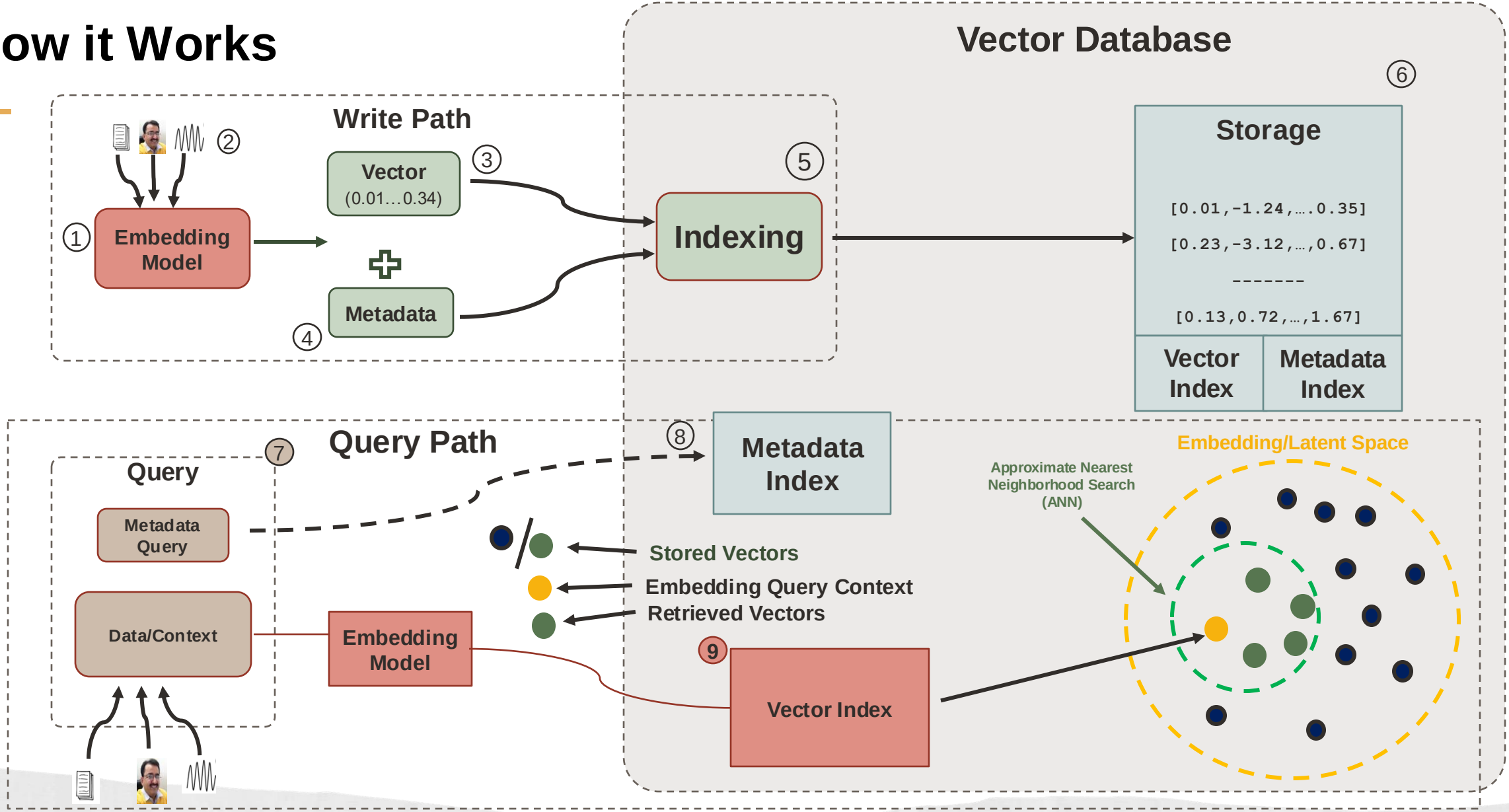
Our paper focuses on methods based on product quantization (PQ) codes, as these were shown to be more effective than binary codes [34]. In addition, binary codes incur important overheads for non-exhaustive search methods [35]

[1] https://github.com/facebookresearch/faiss



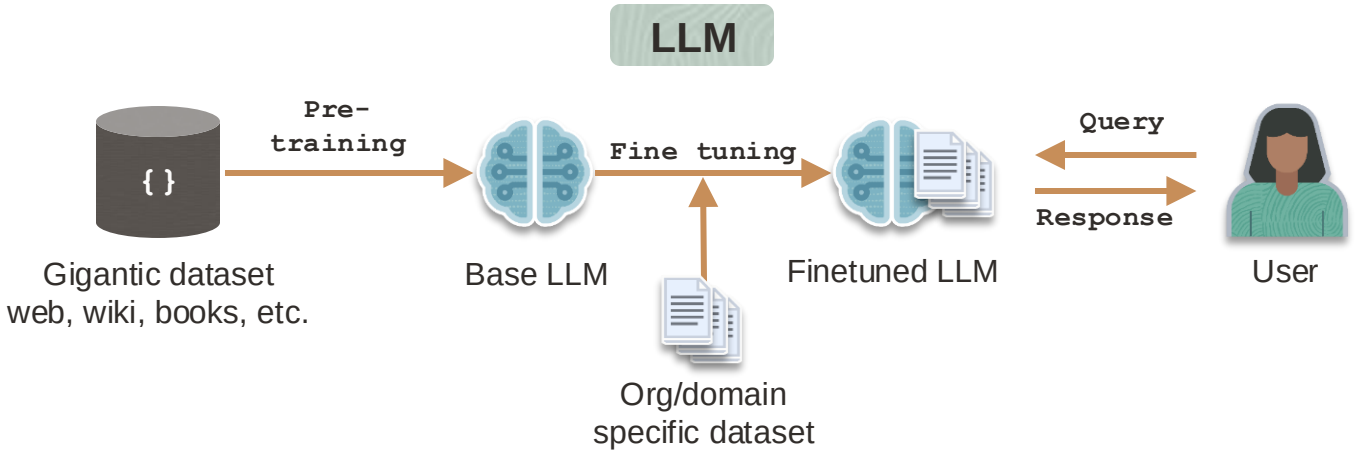
- K-Nearest Neighbors algorithm can be used to perform a vector or semantic search to obtain nearest vectors in embedding space to a query vector.
- ANN algorithms are designed to find near-optimal neighbors much faster than exact KNN searches.
- ANN methods such as HNSW, FAISS, Annoy are often preferred for large-scale similarity search tasks in embedding spaces due to their efficiency.

How it Works

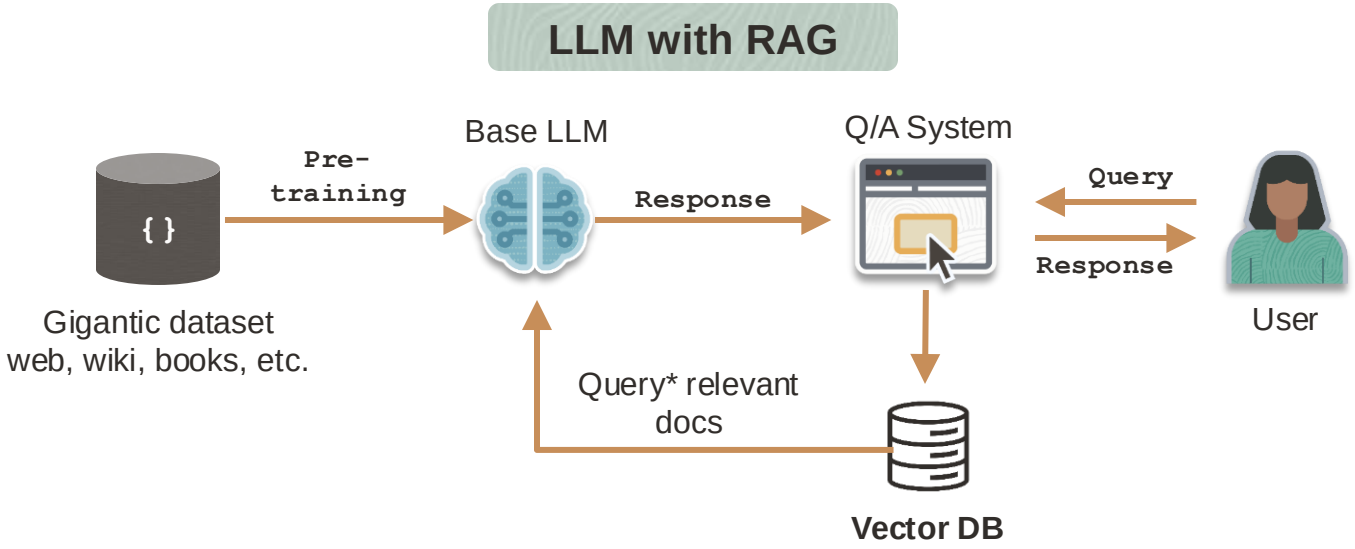

Vector Database
Write Path
Query Path
Embedding Model
Vector
(0.01…0.34)
Metadata
Indexing
Storage
[0.01,-1.24,….0.35]
[0.23,-3.12,…,0.67]
[0.13,0.72,…,1.67]
Vector Index
Metadata Index
Query
Metadata Query
Data/Context
Embedding Model
Metadata Index
Stored Vectors
Embedding Query Context
Retrieved Vectors
Vector Index
Embedding/Latent Space
Approximate Nearest Neighborhood Search (ANN)

LLM Versus LLM + RAG

LLMs without RAG rely on internal knowledge learned during pre-training on a large corpus of text. It may or may not use Fine-Tuning.



LLMs with RAG use an external database, which is a Vector Database.

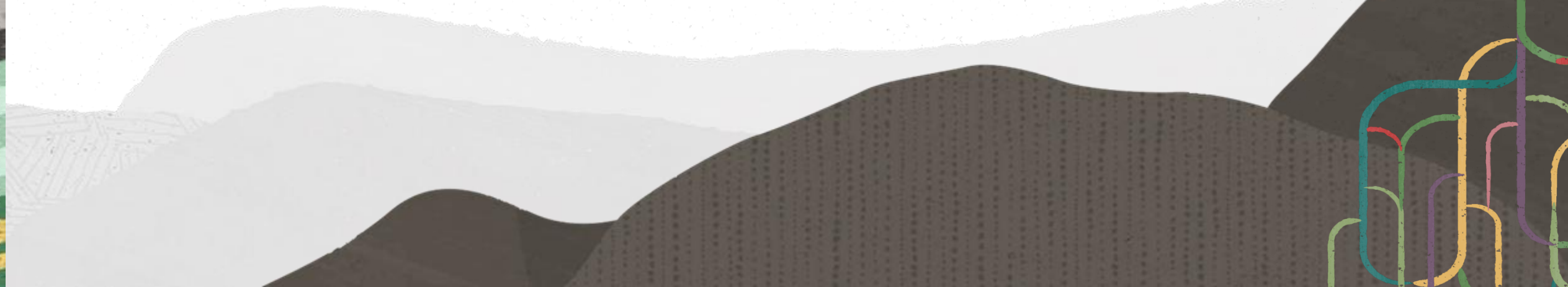


Applications in the Business World

Application	Key Features
Recommendation Systems	- Vectors for user -item similarity - Personalized recommendations
Semantic Search	- Text-to-vector conversion - Similarity-based search
Anomaly Detection	- Vectors for normal/anomalous behavior - Anomaly identification
Personalized Marketing	- Customer profiling - Customized offerings
Image Recognition	- Image-to-vector conversion - Similarity matching
Bioinformatics	- Genetic sequence storage - Protein structure querying

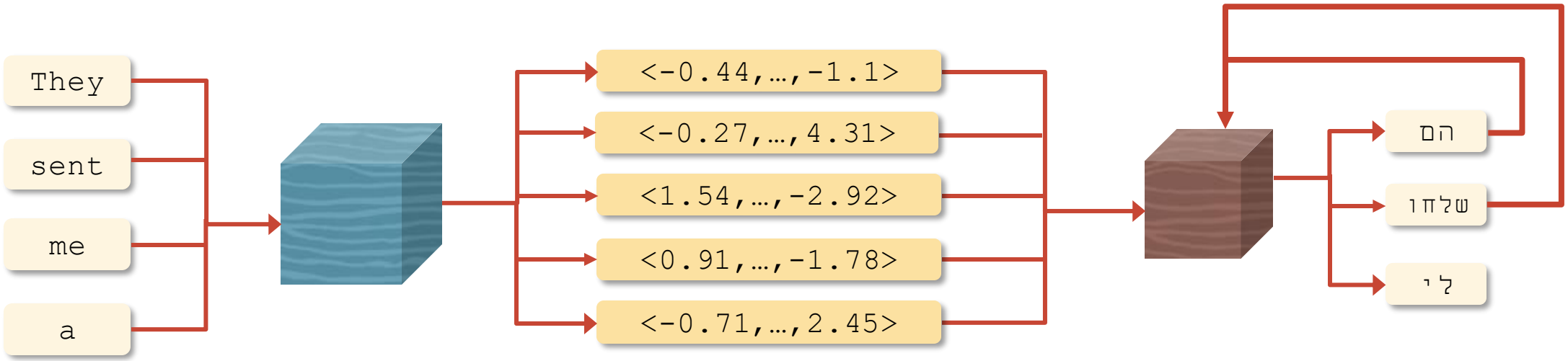
ORACLE
University

Embedding Models

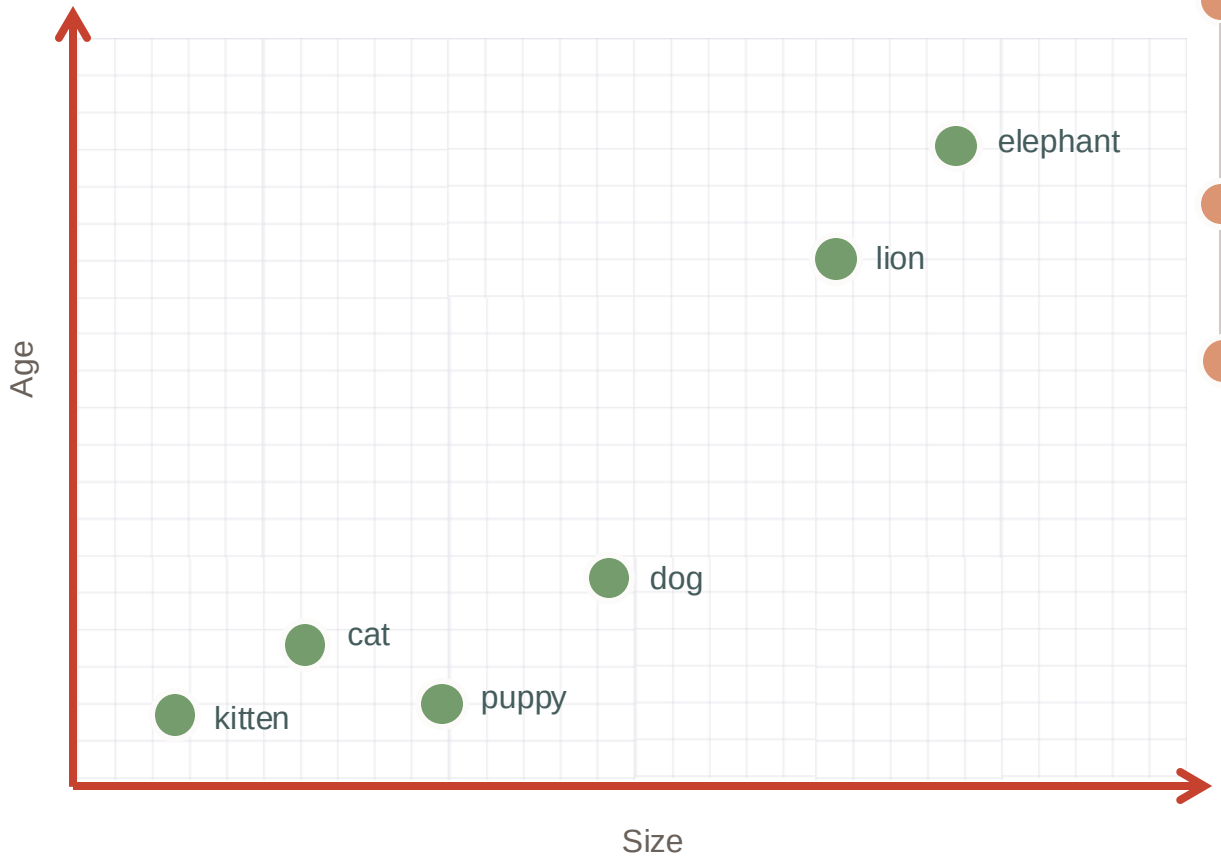


Embeddings

- Embeddings are numerical representations of a piece of text converted to number sequences.
- A piece of text could be a word, phrase, sentence, paragraph or one or more paragraphs.
- Embeddings make it easy for computers to understand the relationships between pieces of text.



Word Embeddings

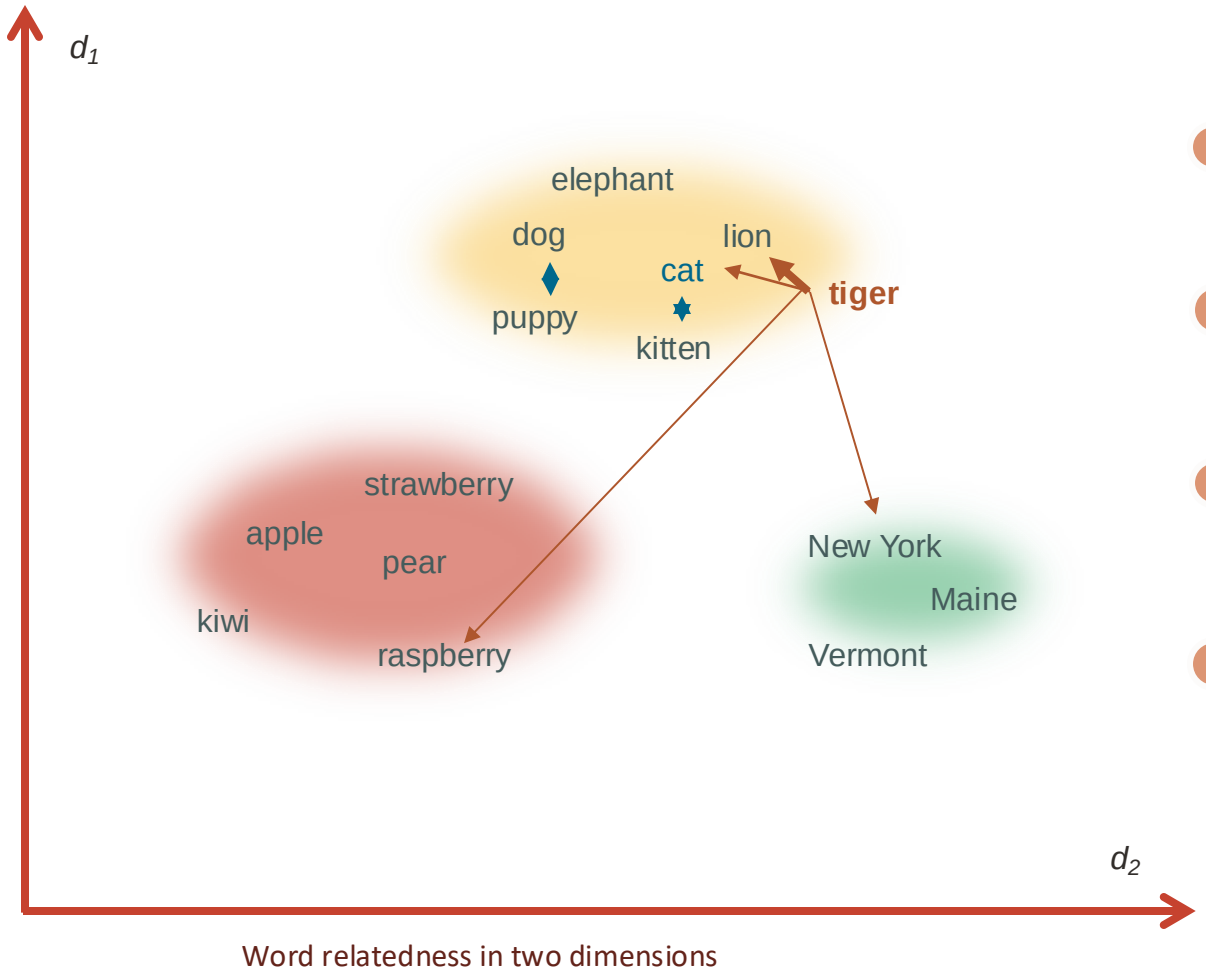


- Word Embeddings capture properties of the word.
- The example here shows two properties:
  - Age (vertical axis)
  - Size (horizontal axis)
- Actual Embeddings represent more properties (coordinates) than just two.
- These rows of coordinates are called vectors and represented as numbers

Word	Embeddings					
Puppy	0.02806	0.03906	0.0386	…	…	…
Kitten	0.0420	0.03006	0.5286	…	…	…
Cat	-0.024	0.0568	0.4280	0.91606	…	…
Dog	-0.0829	-0.4280	0.9280	0.8245	…	…

Age | Size | Other Properties

Semantic Similarity



- Cosine and Dot Product Similarity can be used to compute **numerical similarity**.
- Embeddings that are **numerically similar** are also **semantically similar**.
- E.g., embedding vector of “Puppy” will be more similar to “Dog” than that of “Lion.”
- There are three groups of words here based on similarity: Animals, Fruits, and Places.
- "Tiger" is closest to the Animals group, closer to cat family members.

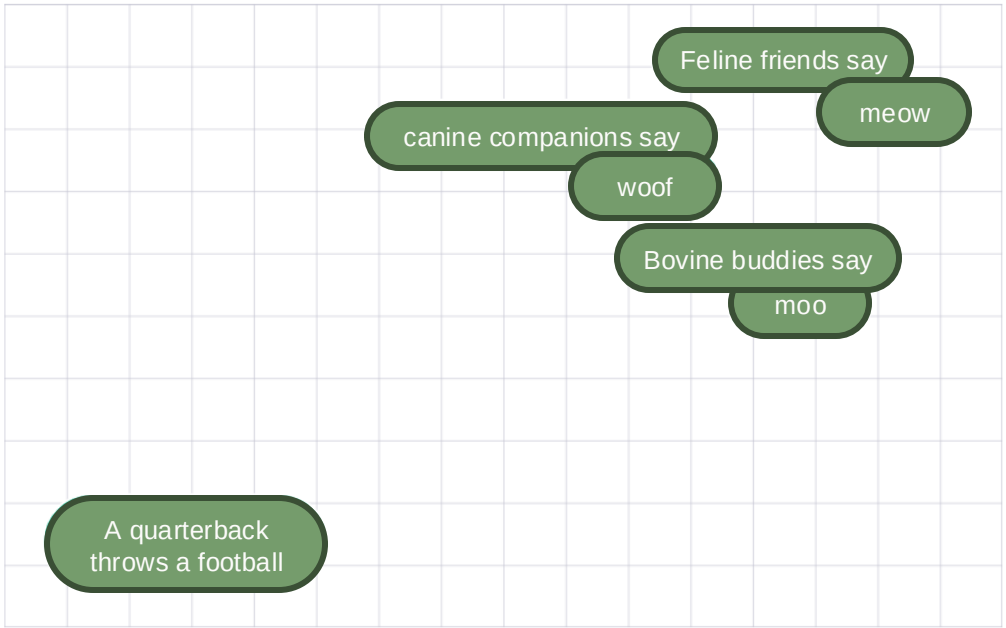
Sentence Embeddings

A sentence embedding associates every sentence with a vector of numbers.

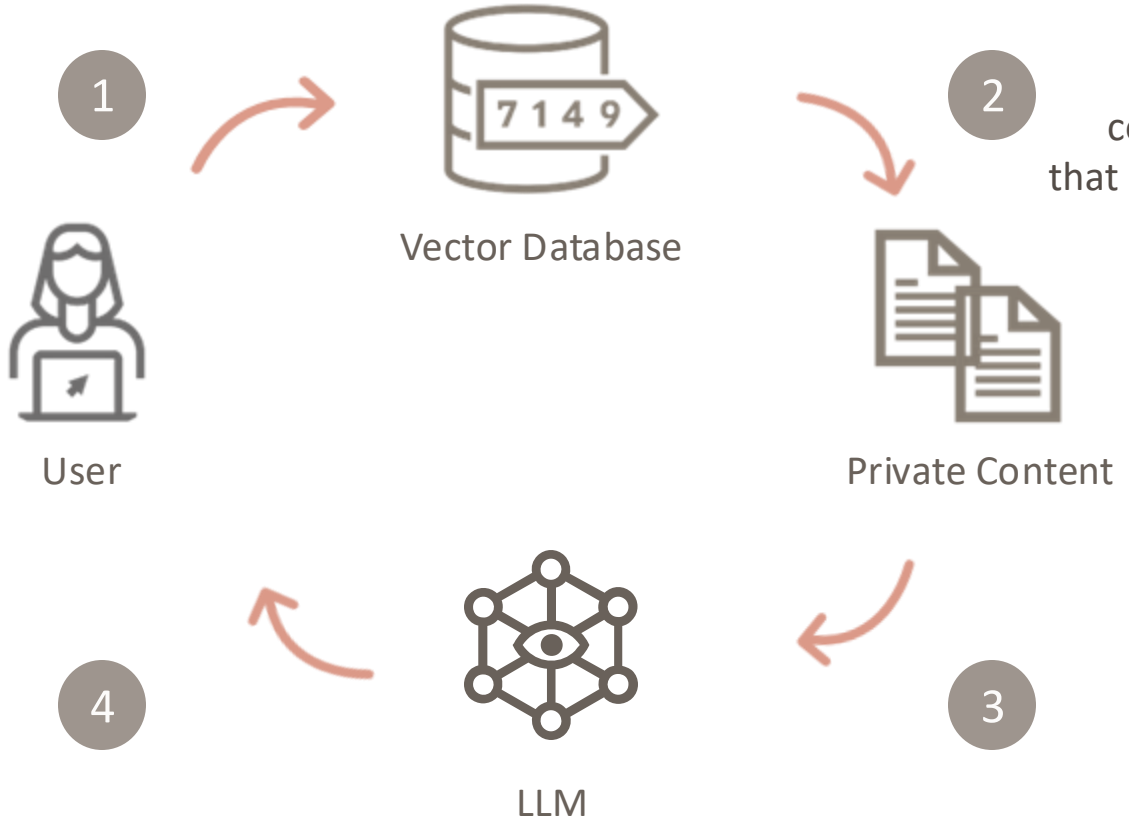
Similar sentences are assigned to similar vectors, different sentences are assigned to different vectors.

The embedding vector of “canine companions say” will be more similar to the embedding vector of “woof” than that of “meow.”

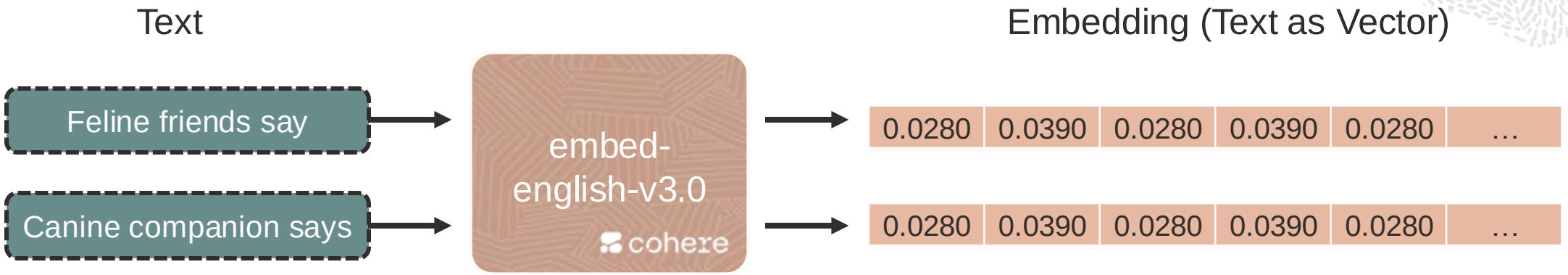
Sentences	Embeddings					
Feline friends say	0.02806	0.03906	…	…	…	…
Canine companion says	0.0420	0.03006	…	…	…	…
Bovine buddies say	-0.024	0.0568	…	…	…	…
A quarterback throws a football	-0.0829	-0.4280	…	….	…	…



Embeddings use case

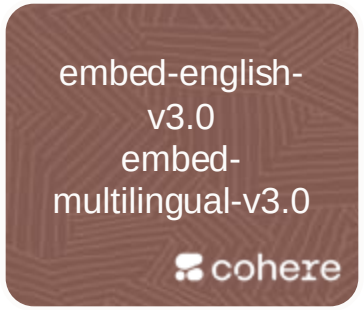


Embedding Models in Generative AI



- Cohere.embed-english converts English text into vector embeddings.
- Cohere.embed-english-light is the smaller and faster version of embed-english.
- Cohere.embed-multilingual is the state-of-the-art multilingual embedding model that can convert text in over 100 languages into vector embeddings.
- Use cases: Semantic search, Text classification, Text clustering

Embedding Models in Generative AI



- English and Multilingual
- Model creates a 1024-dimensional vector for each embedding
- Max 512 tokens per embedding

- Smaller, faster version; English and Multilingual
- Model creates a 384-dimensional vector for each embedding.
- Max 512 tokens per embedding

- Previous generation models, English
- Model creates a 1024 dimensional vector for each embedding
- Max 512 tokens per embedding

Creating Embeddings

Create Embeddings

- Create sentence embeddings using Cohere Embed Model.
- We will load the Text REtrieval Conference (TREC) question classification dataset which contains 5.5K labeled questions.
- We will take the first 1K samples, but this can be scaled to millions or even billions of samples.

```
from datasets import load_dataset

# load the first 1K rows of the TREC dataset
trec = load_dataset('trec', split='train[:1000]')
trec
```

```
trec[0]
```

b. We can then pass these questions to Cohere to create embeddings.

```
embeds = co.embed(
    texts=trec['text'],
    model='small',
    truncate='LEFT'
).embeddings
```

c. We can check the dimensionality of the returned vectors, for this we will convert it from a list of lists to a Numpy array. We will need to save the embedding dimensionality from this to be used when initializing our Pinecone index later.

```
import numpy as np

shape = np.array(embeds).shape
shape
```

Vector Databases - **Storing the Embeddings**

Storing the Embeddings

- Storing embeddings in the Pinecone vector database.
- Initialize the connection to Pinecone and then create a new index for storing the embeddings, making sure to specify that we would like to use the cosine similarity metric to align with Cohere's embedding Model.

```
from pinecone import Pinecone, ServerlessSpec

pc = Pinecone(api_key=PINECONE_KEY)
pc.create_index(
    name="cohere-pinecone-user21", # Add your User ID. For Example cohere-pinecone-user21
    dimension=1024,
    metric="cosine",
    spec=ServerlessSpec(
        cloud="aws",
        region="us-west-2"
    )
)
```

Vector Databases - **Storing the Embeddings**

Storing the Embeddings

- Pinecone expects us to provide a list of tuples in the format *(id, vector, metadata)*, where the *metadata* field is an optional extra field where we can store anything we want in a dictionary format.
- For this example, we will store the original text of the embeddings.
- While uploading our data, we will batch everything to avoid pushing too much data in one go.

Vector Databases - Example Flow

```python
batch_size = 128
index = pc.Index("cohere-pinecone-user21") # Add your User ID. For Example cohere-pinecone-user21
ids = [str(i) for i in range(shape[0])]
# create list of metadata dictionaries
meta = [{'text': text} for text in trec['text']]

# create list of (id, vector, metadata) tuples to be upserted
to_upsert = list(zip(ids, embeds, meta))

for i in range(0, shape[0], batch_size):
    i_end = min(i+batch_size, shape[0])
    index.upsert(vectors=to_upsert[i:i_end])

# let's view the index statistics
index_description = pc.describe_index("cohere-pinecone-user21") # Add your User ID. For Example cohere-pinecone-user21
```

Your vectorstore store your embeddings (☝) and make them easily searchable

Thank You