The geometry of embeddings

How to build better RAG using linear algebra

Brendan Beh
AI.SEA

How do we match similarity in “concepts”?

“The cat sat on the mat”

“The dog slept on the carpet”

How does AI know these two are similar concepts when they share no words in common?

Words need to be converted to numbers for processing to happen

Everybody who has eaten at Taco Bell has ___

An input is first broken up into **tokens.**

Image from Georgia Tech Polo Club

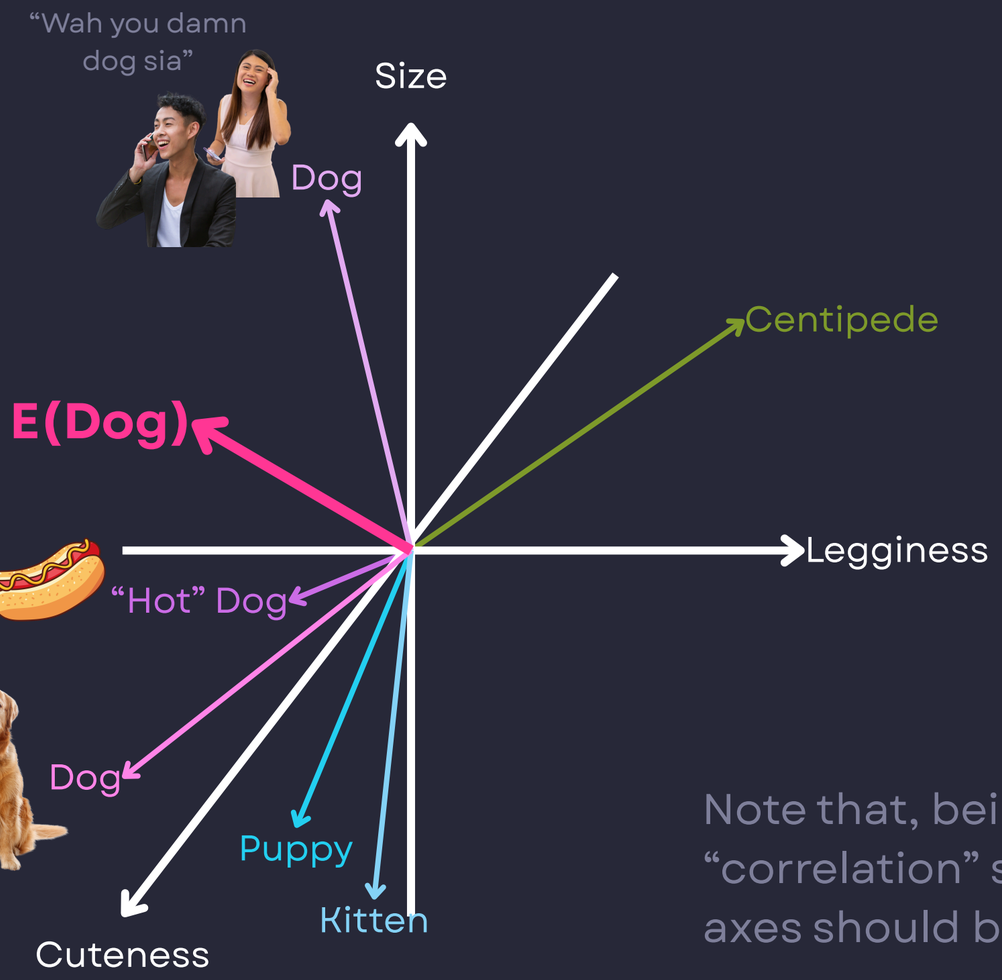
These tokens are encoded into a "**vector**" (i.e. a list of numbers) by the neural network.

This vector encodes the "meaning" of the token + its position in the input.

This is called the **embedding** process.

(i.e. converting a sentence into where it lives in "meaning" space.)

Vectors can be thought of as coordinates in “meaning” space



The process of converting a word into a vector is done through an **embedding** matrix.

$$W_E = \begin{pmatrix} 1.0 & -2.4 & 9.8 & 7.5 & .. & -6.2 & 3.6 \\ -2.5 & -4.4 & -8.7 & 9.6 & \dots & -3.1 & 8.8 \\ -7.3 & 3.5 & 9.3 & -4.4 & \dots & 2.5 & 8.3 \\ \dots & & & & & & \\ 3.2 & 2.4 & -0.6 & 2.5 & \dots & 7.3 & -0.7 \end{pmatrix}$$

The height of this matrix = the no. dimensions in the “meaning” space

The length of this matrix = the number of token permutations
(Think of as the no. words in the dictionary)

Note that, being accurate, the vectors are better described as positions in “correlation” space - we do not explicitly define what the meaning of each axes should be. We’ll explore this idea more in the next few sections.

The coordinates to embed into are found by the correlations of the word

The core idea behind embeddings is very simple: **if two things are similar in the real world, their coordinates in "meaning space" should be similar too.**

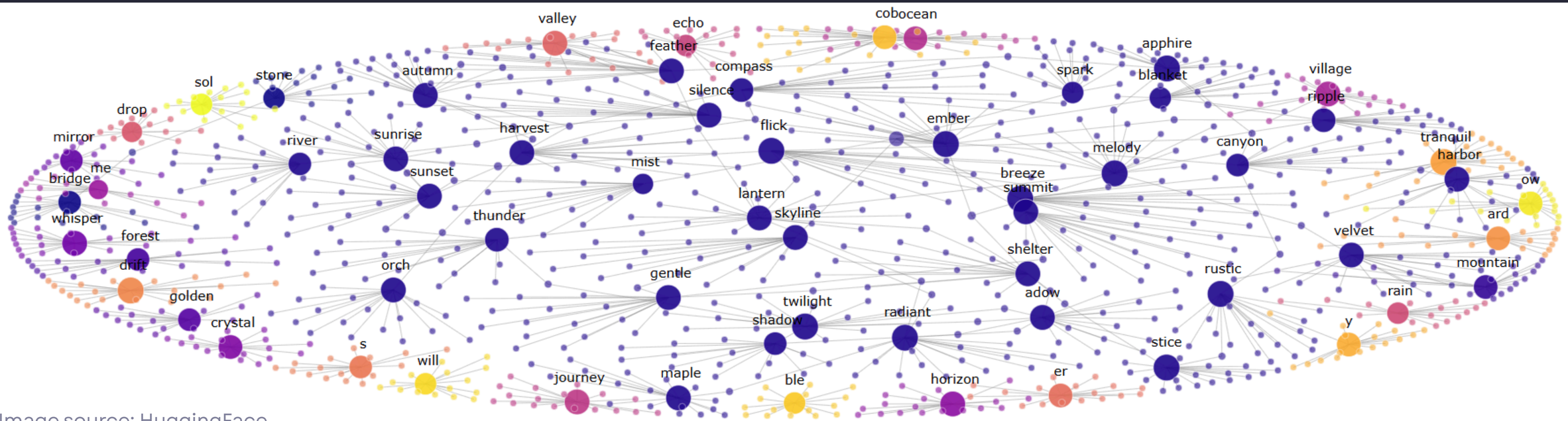


Image source: HuggingFace

The embedding atlas of 50 random words and their closest tokens in the embedding space of `deepseek-ai/DeepSeek-R1-Distill-Qwen-1.5B`.

How do we know what coordinates to assign to a given word?

We map it based on what it's usually used /associated with.

But how did we get here?

Image taken from HuggingFace “LLM Embeddings Explained: A Visual and Intuitive Guide“

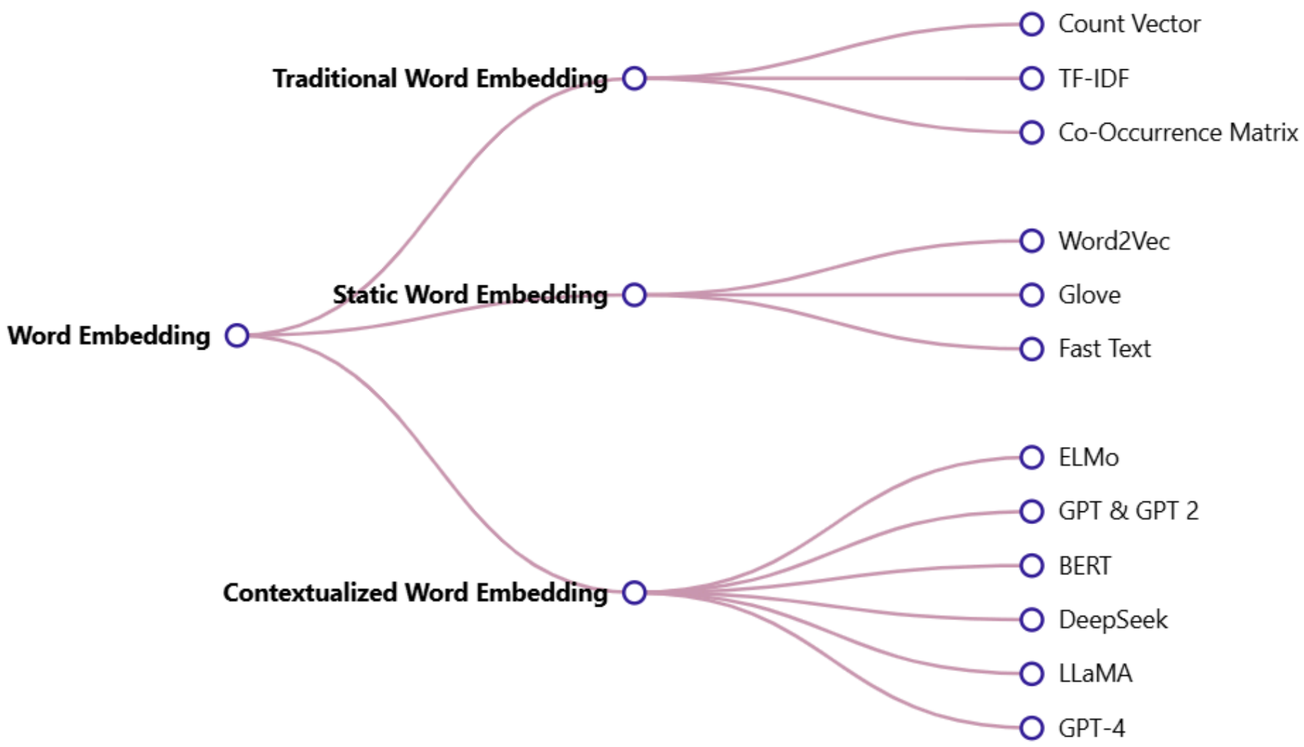


Figure 1: Overview of different word embedding techniques.

Let’s build an intuitive understanding of embeddings first.

OHE as a primitive example

$$A = \begin{pmatrix} 1 \\ 0 \\ 0 \\ \cdots \\ 0 \\ 0 \end{pmatrix} \quad B = \begin{pmatrix} 0 \\ 1 \\ 0 \\ \cdots \\ 0 \\ 0 \end{pmatrix} \quad \bullet\bullet\bullet$$

For an N-sized dictionary, you have a vector of length N

Imagine embedding Mandarin....

You can predict patterns... but

1) there's SO MUCH wasted space!
2) You don't capture any meaning!

The goal is to lower the dimension of the embedding space + capture meaning and context effectively

Counting to find importance - TF-IDF

TF-IDF (Term Frequency-Inverse Document Frequency) is a method used to determine how important a word is within a specific document relative to an entire collection of documents (corpus)

$$\text{TF-IDF}\ (t, d) = \underbrace{\frac{\text{Count of term t in document d}}{\text{Total no. terms in document d}}}_{\text{TF: How important is this term in a given document?}} \times \log\left(\underbrace{\frac{\text{Total number of documents}}{\text{Number of documents containing term } t}}_{\text{IDF: How common is this term across the whole corpus?}}\right)$$

The cat sat on the mat. The mat was soft.

The dog chased the cat across the yard.

Dogs and cats are common household pets.

If you break down a document enough, this is a primitive way of embedding text.

From keywords to coordinates

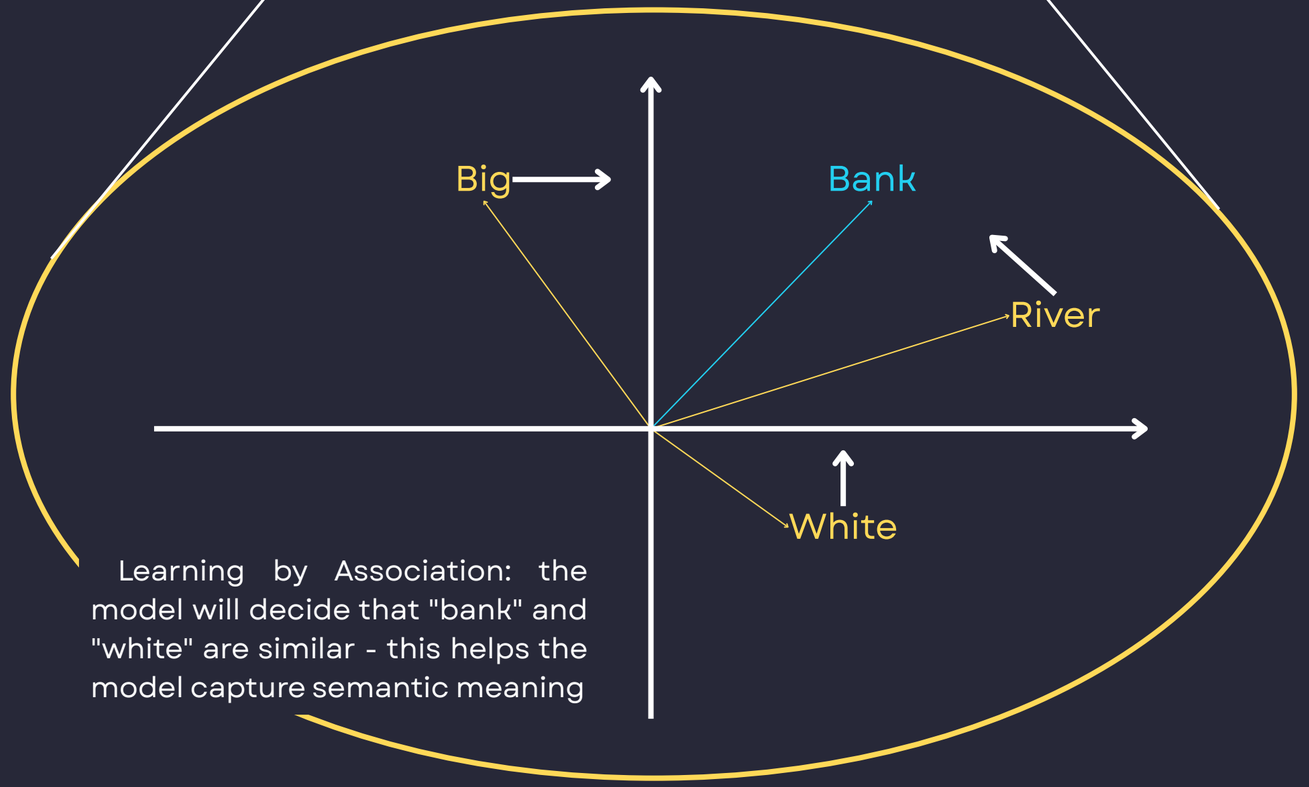
“doctor treats patient”

“physician cares for people”

Keyword counting fails here!

Word2Vec: going from single words to meanings

Near the east of the town, the big white bank by the river gave out a large loan to the poor farmer



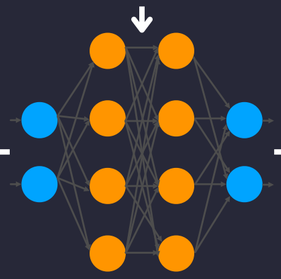
We want the model to push related words closer in embedding space

- Uses a shallow, two-layered neural network architecture to generate dense vector representations of words.
- It operates on the principle that words appearing in similar contexts are semantically related.

There are two ways to train a Word2Vec model

Near the east of the town, the big white bank by the river gave out a large loan to the poor farmer

OHE



Continuous-Bag-of-Words (CBOW)

....the big white ___ by the river...

bank

house

man

cow

fish

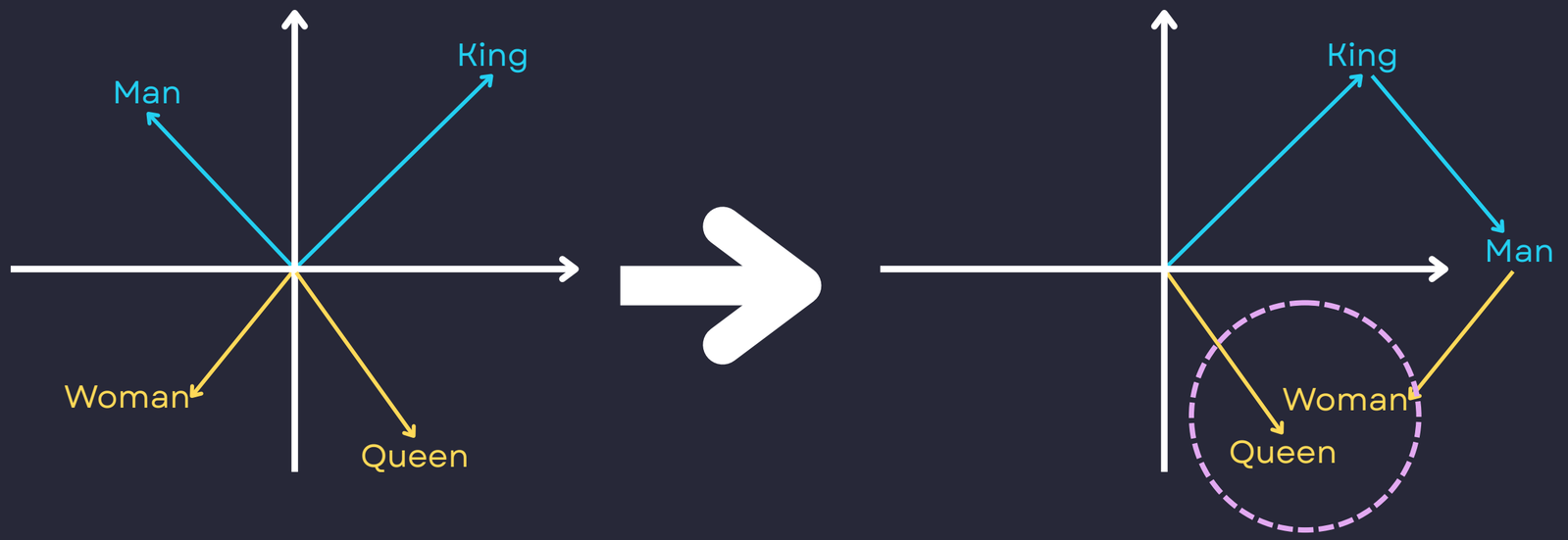
Skip-gram

___ ___ ___ bank ___ ___ ___

the	big
there	golden
inside	hidden
from	cold
under	lay
...	....

Word2Vec: a demo

“Directions” can encode meaning: this lets us do relational algebra

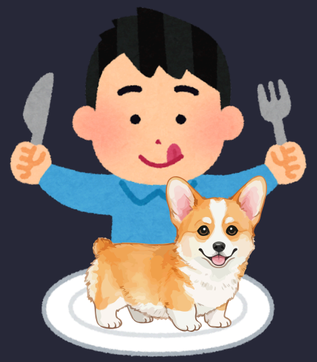


The vectors capture relationships between words as directions in the space. This allows you to perform simple math problems using the word vectors.

$$v_{king} - v_{man} + v_{woman} \approx v_{queen}$$

But words don’t have single meanings - context matters

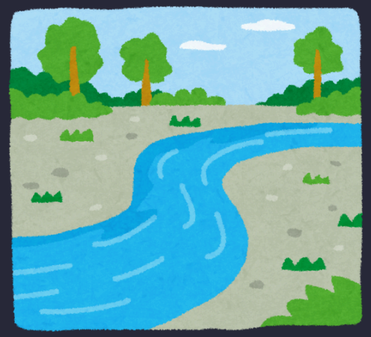
Dog bites man



Man bites dog

Static word embeddings alone may not capture order fully

Order matters!



River bank

Bank loan

Same words in different context = different meanings

Context matters!

BERT and contextual geometry

BERT (Bidirectional Encoder Representations from Transformers) is a transformer-based model that generates embeddings vectors that capture semantic and contextual meaning of text

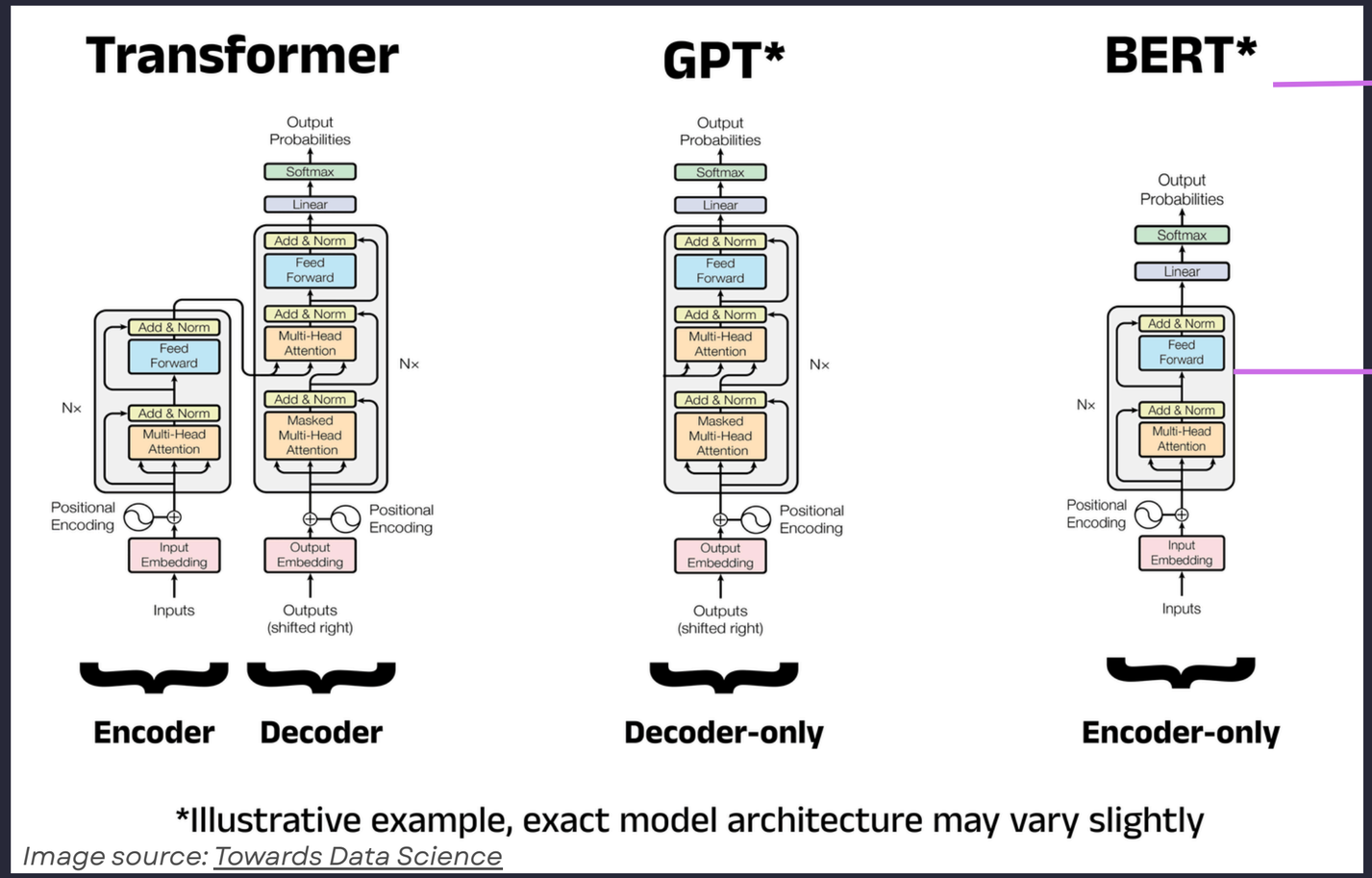
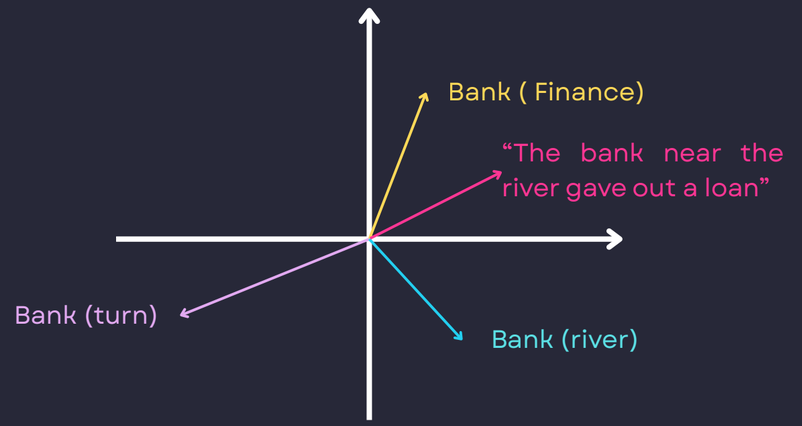


Image source: Towards Data Science

- BERT uses only the encoder part of the transformer architecture.
- The encoder's primary job is to produce high-quality text representations (vector embeddings)

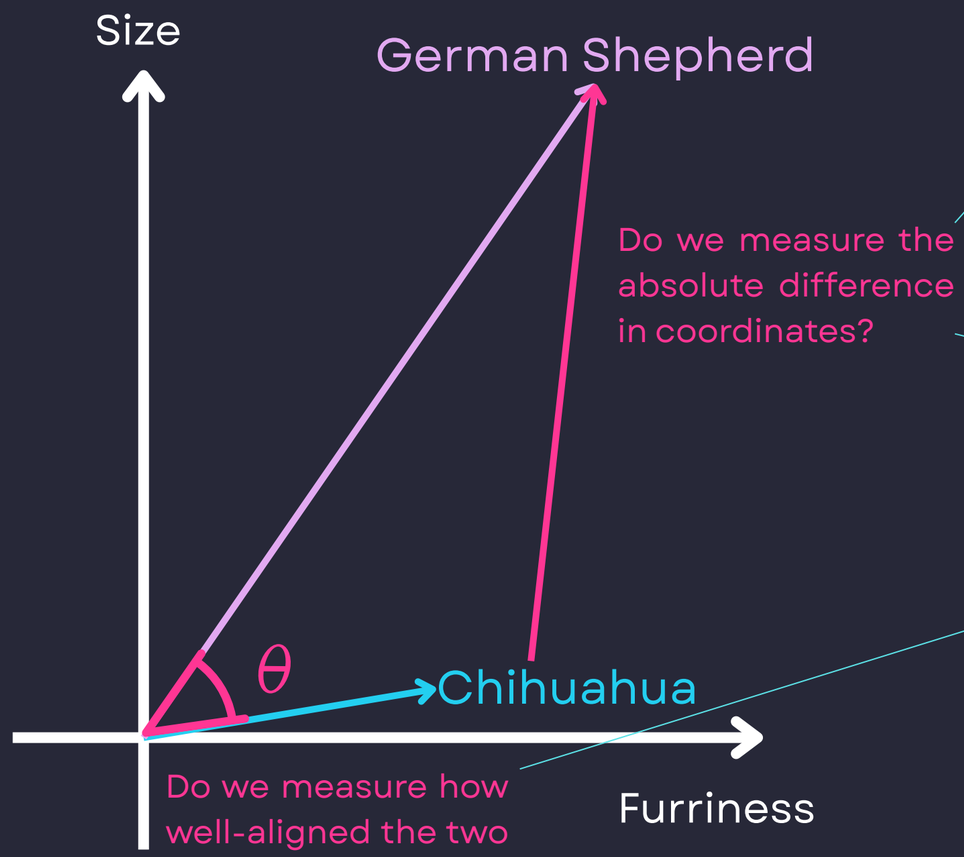
Importantly, BERT is **bidirectional**: it evaluates the text that precedes (left context) and follows (right context) of each word

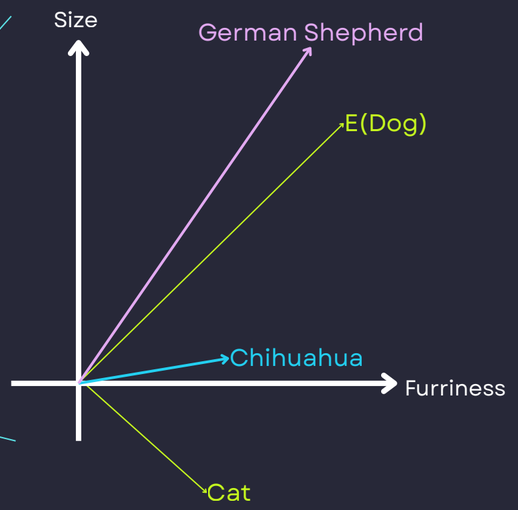
$$h_i = f_{transformer}(x_1, x_2, \ldots, x_n)[i]$$



BERT goes from "single coordinates" to "contextual coordinates"

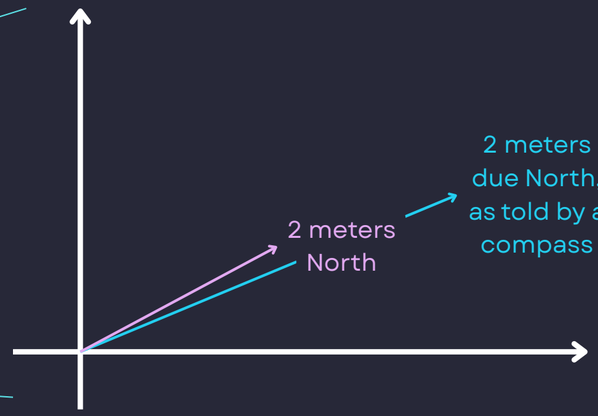
Not just one way to measure “distance”: we need to choose the best method





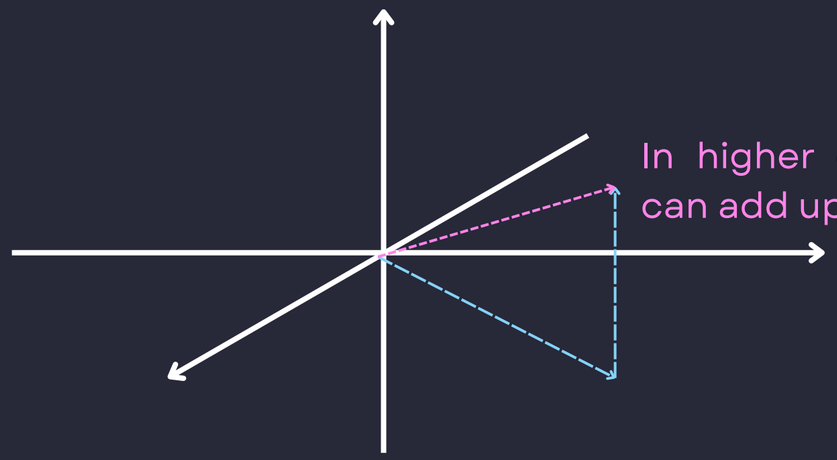
Not as simple as saying “just take the closest one”: if you like chihuahuas, you may get recommended a book on cats instead of dog.

Cosine Similarity is typically better for text embeddings or frequency-based data, where documents or words with similar semantic content may have different lengths or scales. It normalizes vector length, focusing on angular similarity, which makes it robust to differences in document length or feature scale.



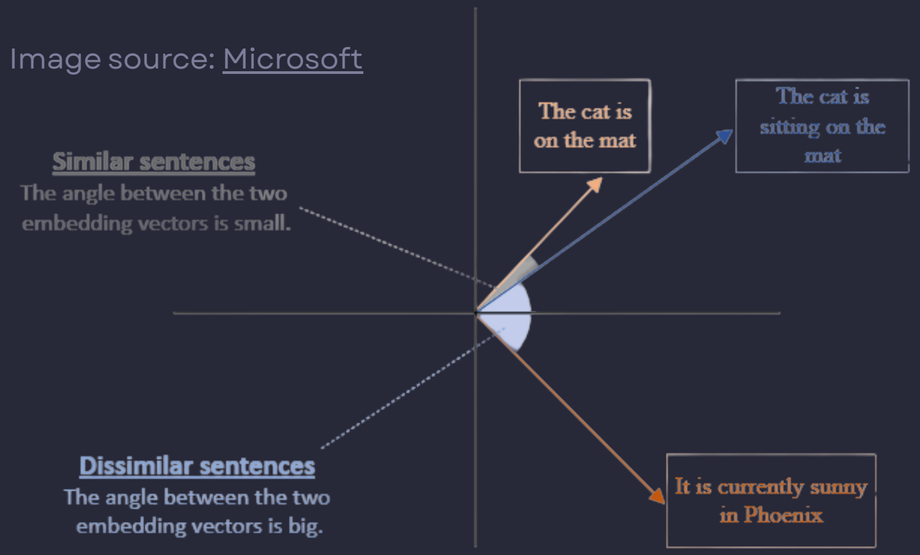
Euclidean is best when absolute differences in features are meaningful, e.g. physical measurements, user ratings. Euclidean distance captures absolute difference in feature values but can be sensitive to vector length and scale

Cosine similarity is the preferred method



In higher dimensions, small distances can add up - distances clump together

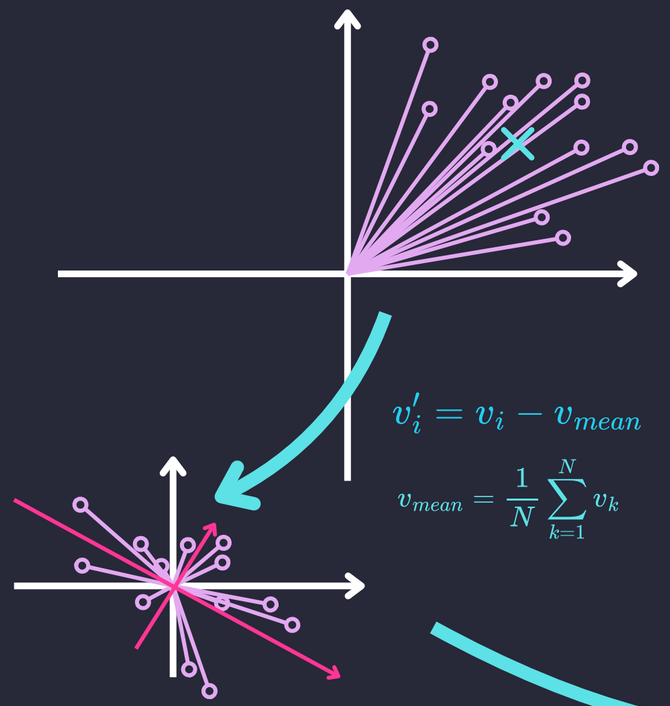
Image source: Microsoft



- Vector length is usually dictated by formatting / text length / etc.
- Cosine similarity is a better measure for semantic similarity
- Always L2-normalize, i.e. convert the vector lengths to 1

$$\hat{v} = \frac{v}{|v|}$$

What to do when “everything is a bit similar”



$$v_i' = v_i - v_{mean}$$

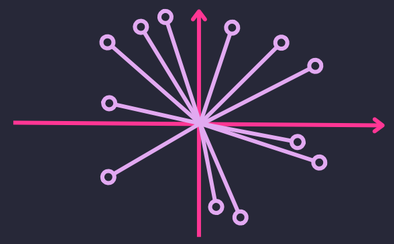
$$v_{mean} = \frac{1}{N}\sum_{k=1}^{N} v_k$$

Anisotropy: Some embedding models place most vectors in a narrow region

Fixes:

- *Compute the mean vector of your whole corpus and shift the origin to the mean, before you L2-normalize*
- *If still bad, try “whitening” (PCA transform): find the main directions (axes) of your cloud and divide each component by its standard deviation. Then re-normalize.*

```
# Suppose 'embeddings' is a 2D array: shape = (num_docs, dim)
# Step 1: Compute corpus mean vector
mean_vec = embeddings.mean(axis=0)  # average along each dimension

# Step 2: Subtract the mean from every embedding
centered = embeddings - mean_vec    # shift the cloud to be around 0

# Step 3: L2-normalize each vector
norms = np.linalg.norm(centered, axis=1, keepdims=True)
normalized = centered / norms

# 'normalized' is your mean-centered, unit-length embedding matrix
```



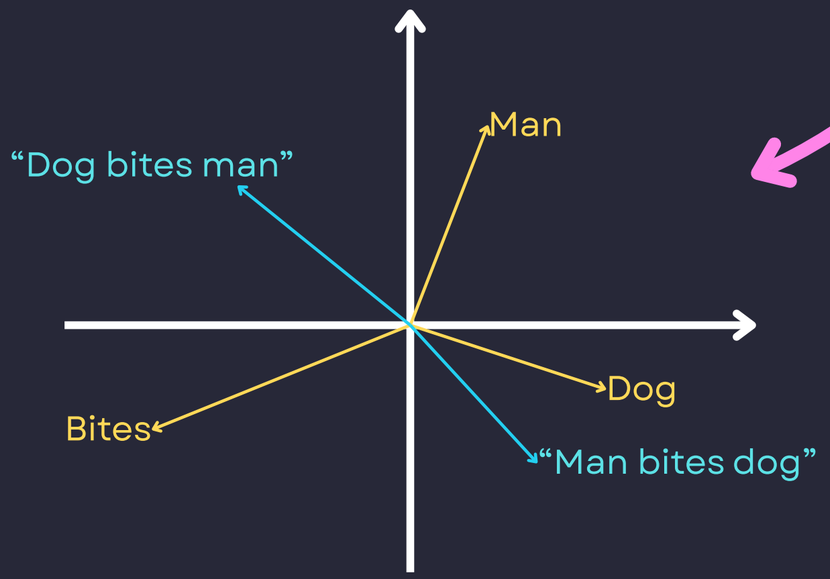
```
from sklearn.decomposition import PCA

# Assume 'centered' vectors from before
pca = PCA(whiten=True)
whitened = pca.fit_transform(centered)

# L2-normalize again
norms = np.linalg.norm(whitened, axis=1, keepdims=True)
final_vectors = whitened / norms
```

For documents, we need to scale up even further

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Comparing every word in a user's query to every word in every document would make searching extremely slow.

Need to compress into lower-dimensional reps (sentences, documents)



Comparing angles of sentences / documents lets us interpret how similar the sentences are

This scales upwards:
Words → Sentences → Paragraphs → Documents

We scale upwards through **pooling**.

Three ways to go from word → sentence embeddings

Mean pooling

Average all the word vectors

- Pros: simple, robust, works well with sentence-transformer models.
- Cons: treats every word equally (even “the,” “and”).

$$E_{sent} = \frac{1}{N} \sum_{i=1}^{N} v_i$$

Weighted mean pooling

Give important words more weight. Weights you can use:

- IDF
- Attention weights
- Part-of-speech weights

$$E_{sent} = \frac{\sum w_i v_i}{\sum v_i}$$

CLS token (for models trained that way)

Some models (e.g., BERT variants) provide a special [CLS] token at the beginning of the input sequence. This token's final vector output is intended to summarise information about the entire sequence.

- Pros: fast (just take one vector).
- Cons: not always best for semantic similarity unless the model was trained for it.
  - *Empirical studies often show that mean pooling performs better than the [CLS] token embedding for capturing comprehensive semantic meaning*

Rule of thumb:

If you’re using a sentence-transformer model (e.g., “all-MiniLM-L6-v2”), mean pooling is a great default.

Splitting long text without breaking meaning

Language and embedding models have limits in input length and semantic focus.

The big white bank by the river gave a loan to the poor farmer

Too short and noisy - not enough information to retrieve something meaningful

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Too long - long passages mix multiple ideas, averaging mixes and loses ideas

Chunking = the art of cutting text into pieces that are small enough to stay focused, but big enough to make sense.

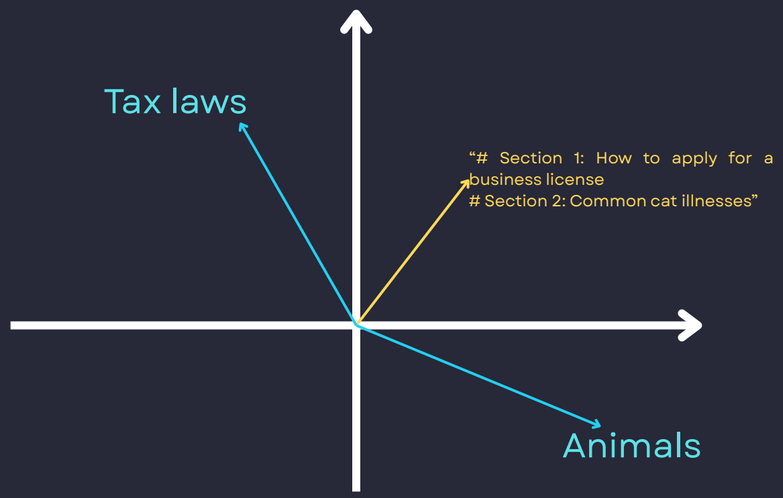
Good and bad chunking practices

Bad chunking: “Frankenstein” chunks

- *Combines unrelated ideas in the same text block.*
- *Resulting embedding points between meanings, not toward any single one.*

 # Section 1: How to apply for a business license
 # Section 2: Common cat illnesses

The chunk is far from “pet breeds” and “tax laws”, useless for both

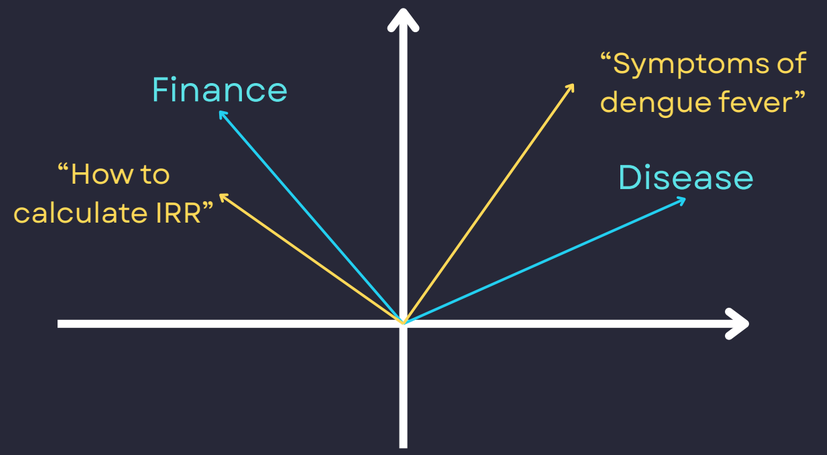


Good chunking: coherent chunks

- *Each chunk covers one main semantic theme.*
- *The embedding has a clean, stable direction.*
- *Retrieval queries about that topic land near the chunk in vector space*

“Symptoms of dengue fever”

“How to calculate IRR”



Visualize with PCA / UMAP

Blurred edges / bridges = bad chunking

```
from sklearn.decomposition import PCA
import matplotlib.pyplot as plt

pca = PCA(n_components=2)
proj = pca.fit_transform(chunk_embeddings)
plt.scatter(proj[:,0], proj[:,1])
```

How to chunk well

Respect semantic boundaries

###Section 1: Keeping your AWS server online
....

###Section 2: What to do in a zombie apocalypse
...

Use consistent size but flexible edges

Aim for 200–400 tokens (≈ 150–300 words), maintain 10–15 % overlap between consecutive chunks to preserve flow at the edges.

These are the first set of ideas. And here is the second set of ideas.

Tokens 1 - 350

Tokens 300 - 650

Detect topic drift automatically

You can use a smaller embedding model to measure how meaning changes sentence by sentence

```
current_chunk = []
for sent in sentences:
    if not current_chunk:
        current_chunk.append(sent)
        continue

    sim = cosine(embed(" ".join(current_chunk)), embed(sent))
    if sim < 0.75:   # topic changed
        save_chunk(current_chunk)
        current_chunk = [sent]
    else:
        current_chunk.append(sent)
save_chunk(current_chunk)
```

A simple RAG demo to see embeddings in action

Summary and recap

Step	Recommendation	Why
Split on	Headings / paragraphs / semantic drift	Respect natural topics
Chunk size	200–400 tokens	Enough info, not too diluted
Overlap	10–15 %	Preserve context at boundaries
Semantic check	cosine < 0.75 → new chunk	Detect topic change
Normalize	mean-center + L2	Keep geometry clean

Symptom	Likely Cause	Quick Fix
Everything seems similar	No mean-centering; anisotropy	Subtract mean, normalize
Unrelated chunks retrieved	Frankenstein chunks	Chunk by headings / semantic drift
Same doc appears twice	Overlap too large	Reduce overlap to 10 %
Cosine scores meaningless	No calibration	Build small gold set

Useful references

Harsh Vardhan: A Comprehensive Guide to Word Embeddings in NLP
Microsoft Ignite: **Design and develop a RAG solution**
https://www.singlestore.com/blog/beginner-guide-to-vector-embeddings/
https://towardsdatascience.com/a-complete-guide-to-bert-with-code-9f87602e4a11/
https://python.langchain.com/docs/tutorials/rag
https://www.kaggle.com/code/anshulgupta1502/rag-from-scratch/notebook