



**BITS** Pilani  
Pilani Campus

# Embeddings, Vector Search & Hybrid Retrieval

Lecture 2 & 3 | Module 1  
Course: AIMLCZG521

- Bhagath S



# What We'll Learn Today

## Part 1: Foundations

- Encoder models & embedding generation
- Transformer architecture for embeddings
- Vector similarity mathematics
- Embedding model landscape

## Part 2: Vector Databases

- ANN search algorithms (HNSW, IVF, PQ)
- Index structure mathematics
- Database solutions & tradeoffs
- Cost engineering

## Part 3: Sparse Retrieval

- TF-IDF fundamentals
- BM25 algorithm & mathematics
- Hybrid search rationale

## Part 4: Hybrid Systems

- Reciprocal Rank Fusion (RRF)



# Prerequisites Review

Basic transformer architecture  
(self-attention, feedforward layers),  
matrix operations.

We'll build everything else from scratch.

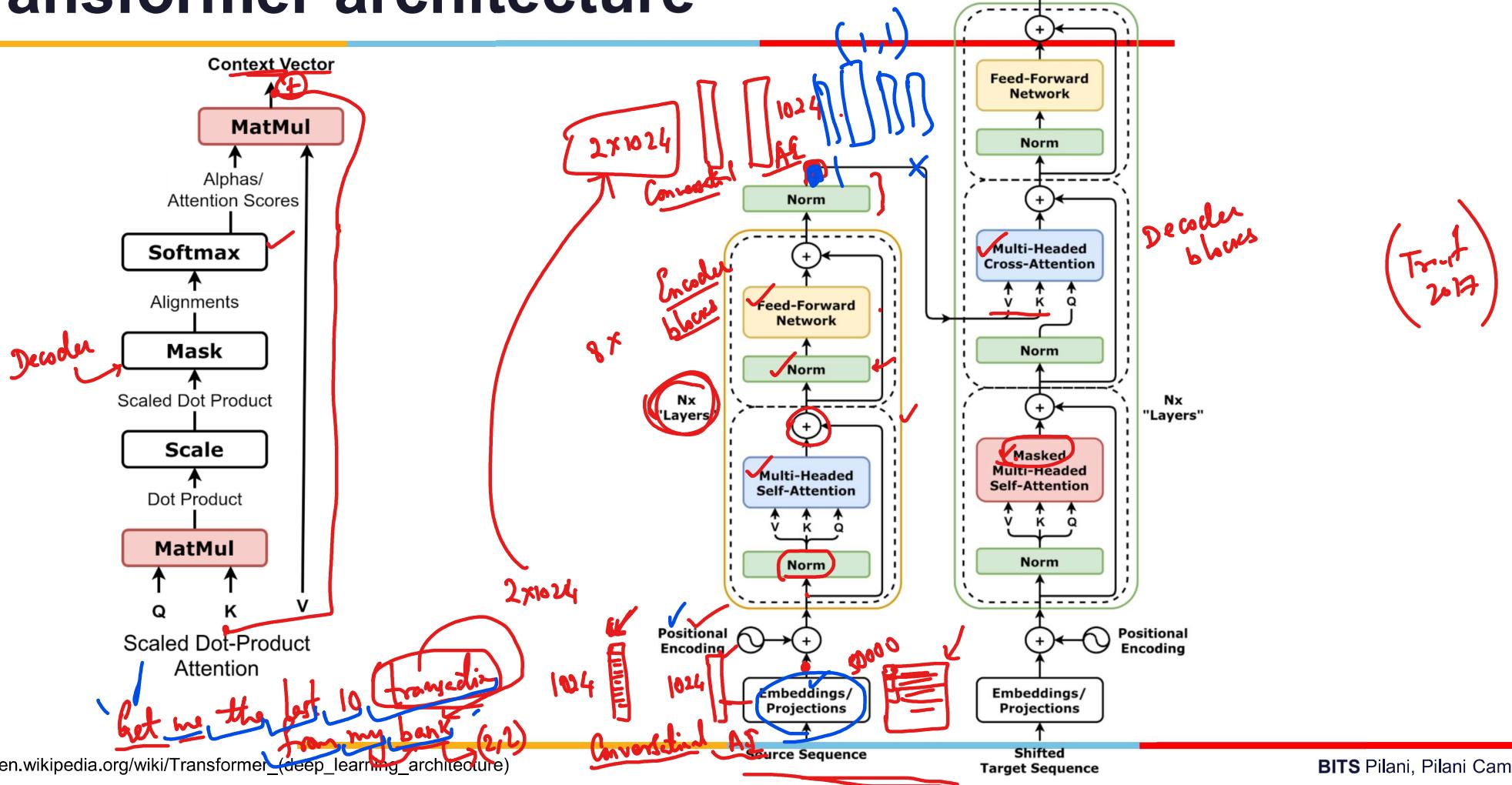


# Learning Objectives

By the end of this lecture, you will be able to:

- ✓ **Explain** how transformer encoder models create contextual embeddings through self-attention and pooling strategies
- ✓ **Calculate and compare** vector similarity using cosine similarity, L2 distance, and dot product metrics
- ✓ **Evaluate and select** appropriate ANN indexing strategies (HNSW, IVF, PQ) based on dataset size, memory constraints, and accuracy requirements
- ✓ **Implement** BM25 scoring from scratch, understanding term frequency saturation and document length normalization
- ✓ **Design** hybrid retrieval systems that combine dense and sparse methods using Reciprocal Rank Fusion (RRF)

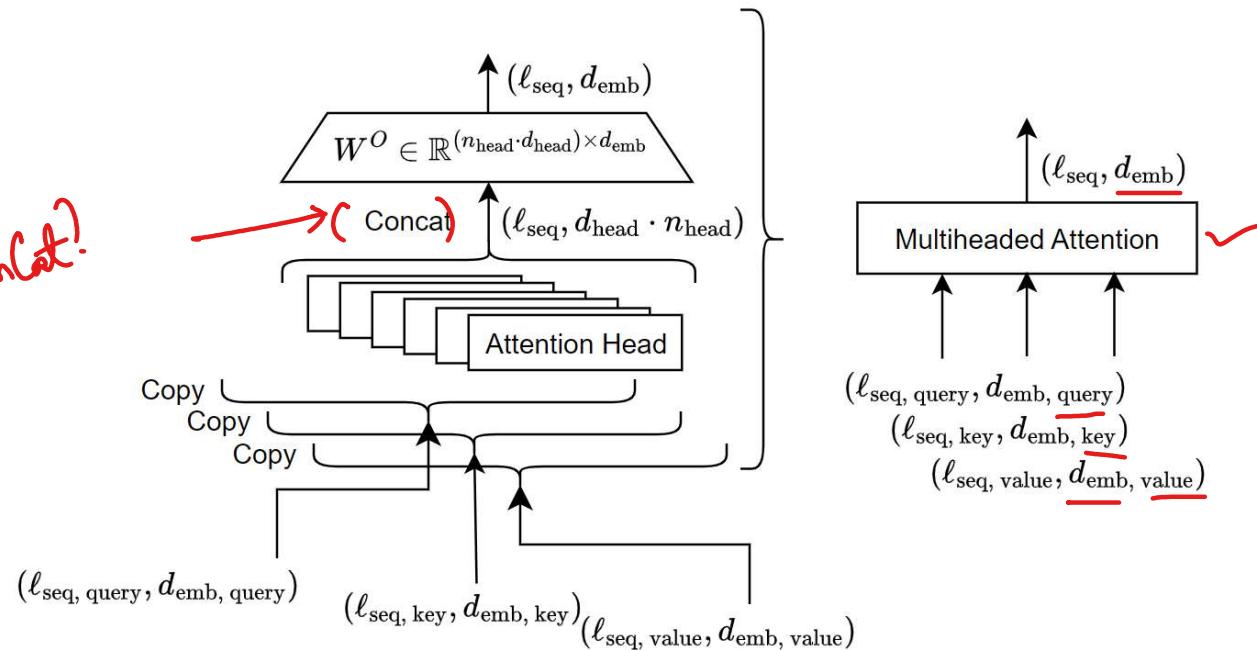
# Transformer architecture



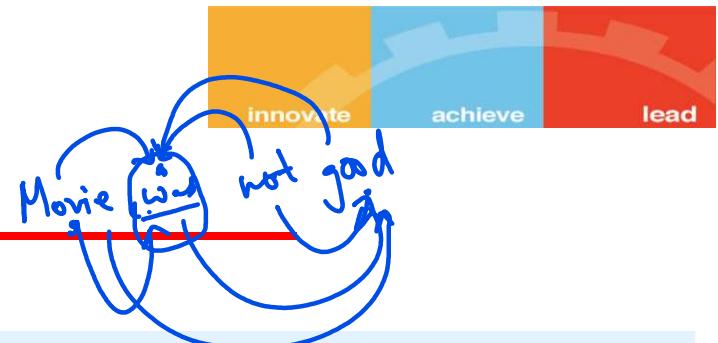


# Multi-headed Attention

why not add?  
why Concat?



# What are Encoder Models?



## Bidirectional Transformers for Understanding

**Core Capability:** Encoder models can see the entire input sequence at once (bidirectional context), making them ideal for:

- Understanding meaning (semantic representation)
- Classification tasks (sentiment, intent)
- Creating embeddings for similarity search



BERT (2018)

**Training:** Masked Language Modeling (MLM)

"The [MASK] brown fox [MASK] over the lazy dog" →  
Predict: "quick", "jumps"

**Key insight:** Random tokens replaced with [MASK], model predicts independently.  
Missing tokens can't be used for generation.



GPT (2018)

**Training:** Autoregressive (Causal LM)

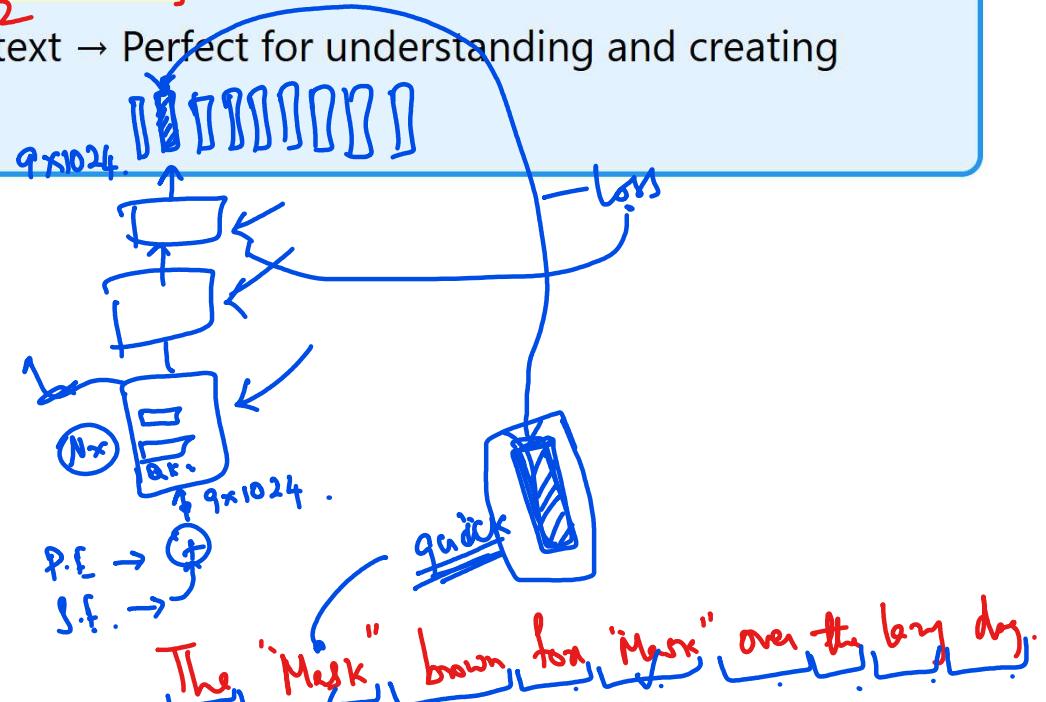
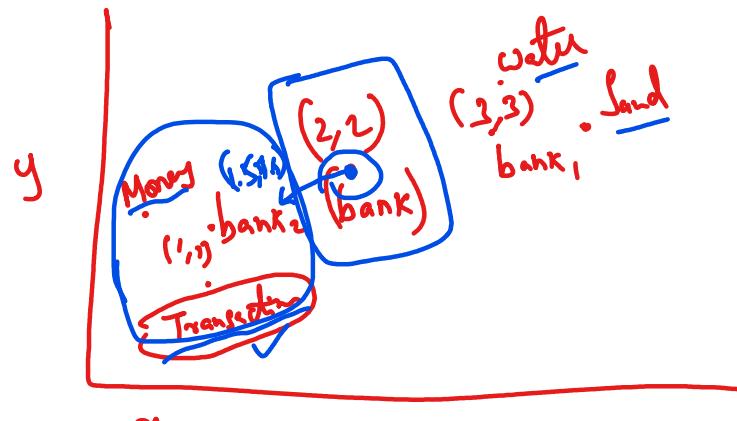
"<s> A B C D" → Predict each token auto-regressively

**Key insight:** Tokens predicted left-to-right. Can condition on leftward context, so it can be used for generation.



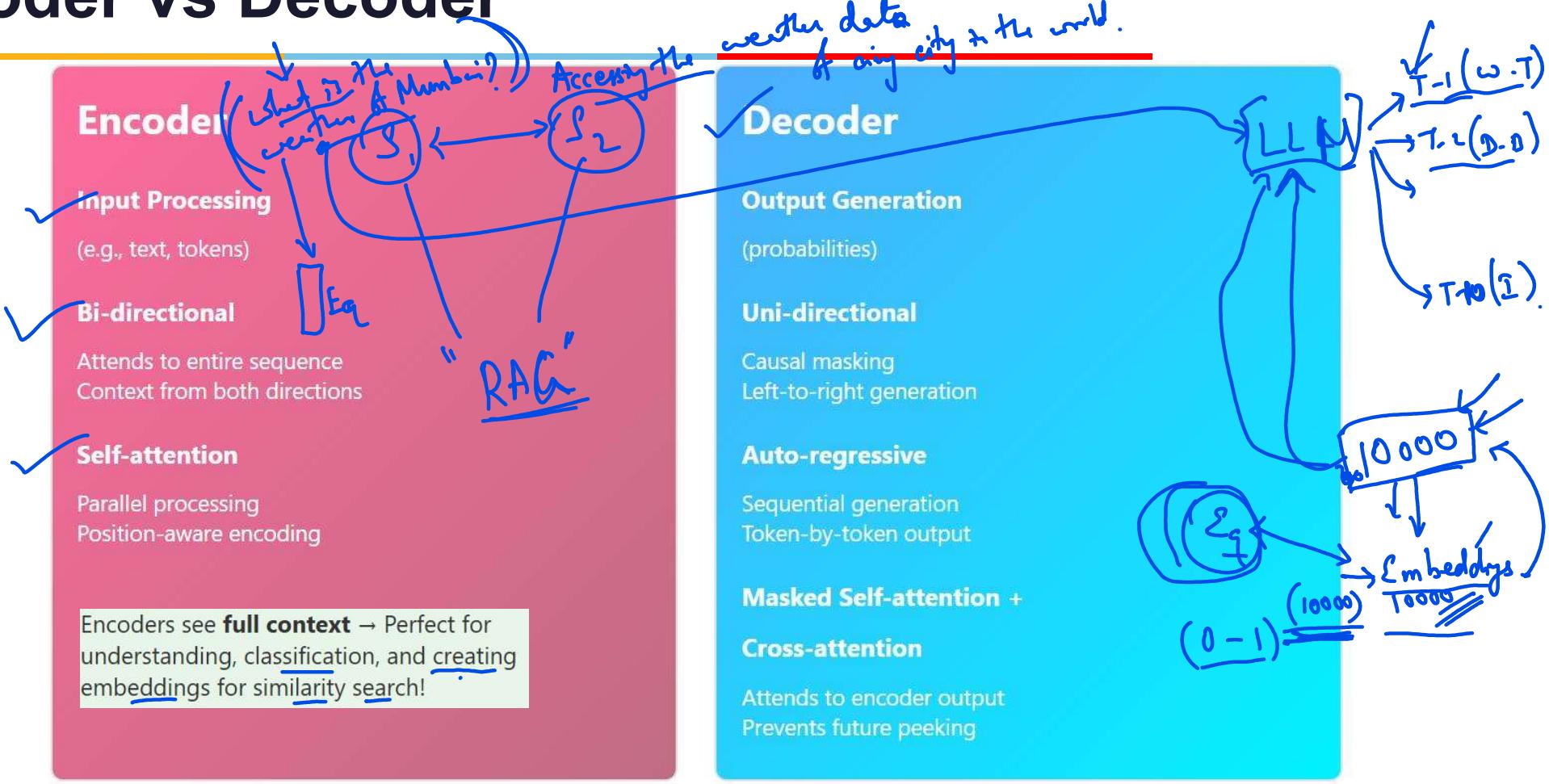
# Why Encoders for Embeddings?

The word "bank" in "river bank" vs ["bank account"] needs to see words on **BOTH sides** to understand meaning. Encoders see full context → Perfect for understanding and creating embeddings for similarity search!



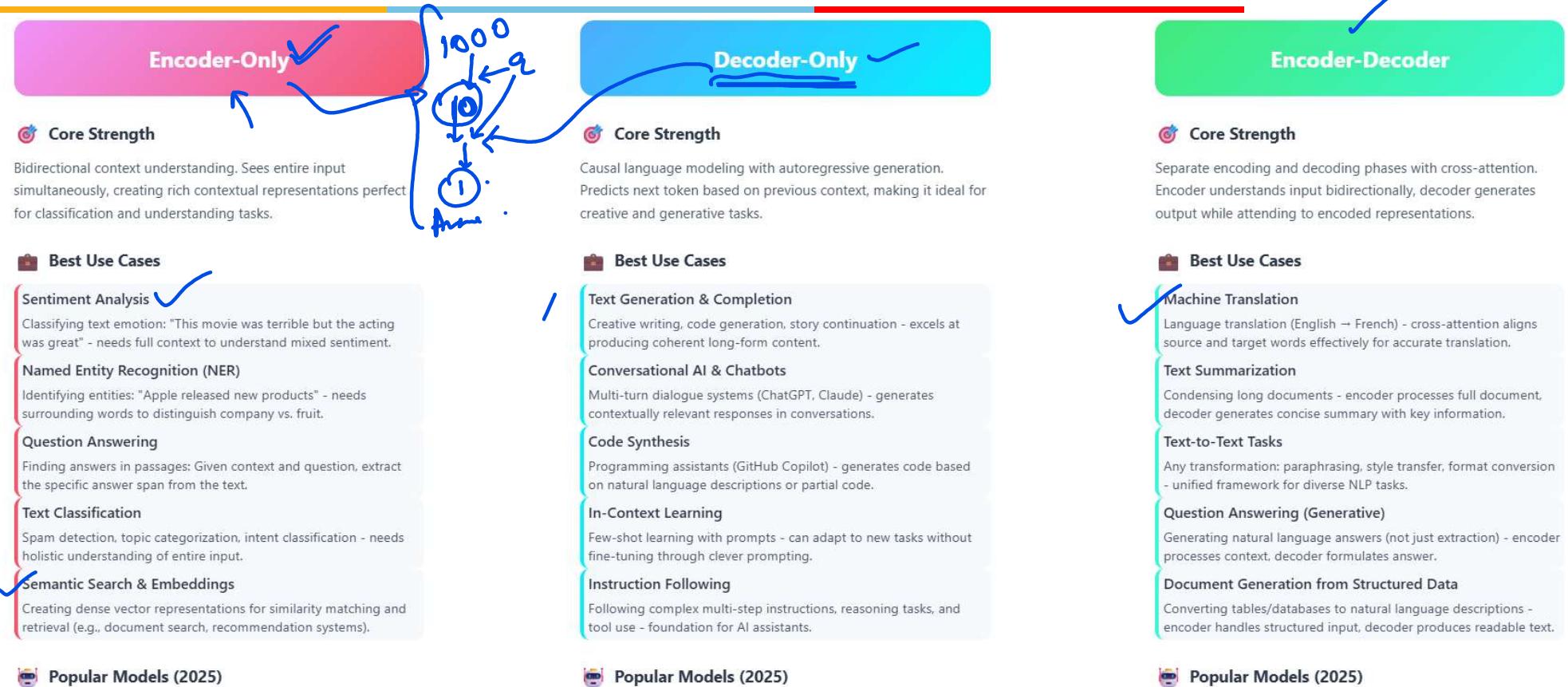


# Encoder vs Decoder





# Encoder vs Decoder vs Encoder-Decoder





# Comparison

Task Type	Encoder-Only	Decoder-Only	Encoder-Decoder	Reasoning
Sentiment Analysis	✓ Best	○ Possible	✗ Overkill	Needs bidirectional context, no generation required
Text Generation	✗ No	✓ Best	○ Possible	Requires autoregressive generation capability
Machine Translation	✗ No	○ Suboptimal	✓ Best	Cross-attention optimally aligns source-target pairs
Named Entity Recognition	✓ Best	✗ Inefficient	✗ Overkill	Token-level classification needs bidirectional context
Conversational AI	✗ No	✓ Best	○ Outdated	Decoder-only scales better for dialogue and instruction following
Summarization	✗ No	✓ Modern	✓ Traditional	Both work; decoder-only now preferred for unified architecture
Semantic Search	✓ Best	○ Possible	✗ Overkill	Encoder creates optimal dense representations for similarity
Code Generation	✗ No	✓ Best	○ Rare	Sequential generation with context, decoder-only excels



# How Encoder Transformers Create Embeddings

9 tokens → Single 768-dimensional vector

## Step-by-Step Process

**Input Text:** "Machine learning is fascinating"

### Step 1: Tokenization

→ [CLS] Machine learning is fascinating [SEP]  
→ Token IDs: [101, 8394, 4083, 2003, 17117, 102]

6

$\frac{3}{4} \times \# \text{tokens} = \# \text{words}$

### Step 2: Token Embeddings

Each ID → 768-dim vector  
8394 → [0.23, -0.15, 0.89, ..., 0.42]

6x 768

### Step 3: Position Embeddings

Position 0 → [0.01, 0.02, ...]  
Position 1 → [0.02, 0.03, ...]  
Sum with token embeddings

6x 768

Posing.  
D  
900 wpm

"1200 tokens"

### Step 4: Transformer Layers (12 layers for BERT-base)

Each layer: Self-Attention → LayerNorm → FFN → LayerNorm

### Step 5: Pooling Strategy

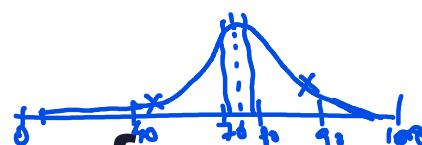
CLS Pooling (most common):  $v = h[\text{CLS}]$   
Mean Pooling:  $v = (1/n) \sum h_i$   
Max Pooling:  $v_j = \max(h_{1,j}, h_{2,j}, \dots, h_{n,j})$

6x 768

(1200 x 768)

1x 768

💡 **Default Strategy:** Most pre-trained models (BERT, RoBERTa, DistilBERT) use **CLS token pooling** by default. However, **mean pooling** often performs better for sentence embeddings (used by Sentence-BERT, BGE, GTE).



# How Encoder Transformers Create Embeddings

## Self-Attention Mathematics ✓

Query, Key, Value projections:

$$Q = X W_Q, K = X W_K, V = X W_V$$

Attention scores:

$$\text{Attention}(Q, K, V) = \text{softmax}(QK^T / \sqrt{d_k}) V$$

Where  $d_k$  is dimension of keys (typically 64)

## Pooling Strategies

Mean Pooling: ✓

$$v = (1/n) \sum h_i$$

CLS Pooling: ✓

$$v = h_{[CLS]}$$

Max Pooling:

$$v_j = \max(h_{1,j}, h_{2,j}, \dots, h_{n,j})$$

## Key Insight

The final embedding is **contextual** - "bank" gets different embeddings in different sentences because self-attention incorporates surrounding context. This is fundamentally different from word2vec or GloVe (static embeddings).



# Embedding Models: Key Players

2 Reasons for clustering?

- 1. One vector (mean, mean, ...)
- 2. Content of model.

## Open Source Models (Recommended for Production)

Model	Dimensions	Context (tokens)	Best For	Provider
bge-large-en-v1.5	1,024	512 <u>781</u>	General purpose, high quality	BAAI
gte-large-en-v1.5	1,024	8,192	Long documents	Alibaba
e5-mistral-7b-instruct	4,096	32,768	Instruction-following, very long context	Microsoft
jina-embeddings-v2	768	8,192	Multilingual, fast inference	Jina AI

Annotations:

- Handwritten notes: "V. DB" with arrows pointing to the "Dimensions" column, "C" with an arrow pointing to the "Context" row, and "H+C" with an arrow pointing to the "Dimensions" column.
- Handwritten notes: "10000 tokens" with an arrow pointing to the "Dimensions" column for the jina-embeddings-v2 model.
- Handwritten notes: "2000 token" with an arrow pointing to the "Dimensions" column for the gte-large-en-v1.5 model.
- Handwritten notes: "400 token" with an arrow pointing to the "Dimensions" column for the e5-mistral-7b-instruct model.
- Handwritten notes: "Semantic Structure (Perce)" with an arrow pointing to the "Dimensions" column for the jina-embeddings-v2 model.
- Diagram: A flowchart showing a process P-D branching into three parallel paths labeled P-1, P-2, and P-3. Each path has a box labeled "200 token". The outputs from P-1, P-2, and P-3 are combined into a final output C-1, C-2(400), and C-3(200).



# Embedding Models: Key Players



## Commercial API Models (Highest Quality)

Model	Dimensions	Context	Best For	Cost
<u>✓ text-embedding-3-large</u>	<u>3,072</u>	8,191	Highest quality, production	\$0.13 / 1M tokens
<u>✓ embed-english-v3.0</u>	1,024	<u>512</u>	Compression support, production	\$0.10 / 1M tokens

### 💡 Selection Guide:

Start with **bge-large-en-v1.5** for general use. Upgrade to commercial APIs if quality justifies cost. Use **gte-large** or **e5-mistral** for long documents (>1000 tokens).



# Embedding Models: Complete Reference

Model	Provider	Arch	Layers	Hidden	Output	Training	Context	Type
text-embedding-3-large	OpenAI	Transformer	N/A	N/A	3,072	Contrastive	8,191	API
embed-v3	Cohere	Transformer	N/A	N/A	1,024	Multi-task	512	API
bge-large-en-v1.5	BAAI	BERT	24	1,024	1,024	RetroMAE+Contrastive	512	Open
gte-large-en-v1.5	Alibaba	BERT	24	1,024	1,024	Contrastive	8,192	Open
e5-mistral-7b-instruct	Microsoft	Mistral 7B	32	4,096	4,096	Contrastive pre-training	32,768	Open
jina-embeddings-v2	Jina AI	BERT	12	768	768	Contrastive	8,192	Open

 **Reference:** Complete specifications for implementation decisions. See Part 1 for selection guidance.



# Key Concepts

- **Base Architecture:** The underlying neural network design (BERT, etc.)
- **Layers:** Number of transformer layers in the model
- **Hidden Size:** Dimensionality of hidden representations within the model
- **Output Dimensions:** Size of the final embedding vector
- **Training Objectives:**
  - *Contrastive Learning:* Learning by comparing similar vs dissimilar pairs
  - *RetroMAE:* Masked auto-encoding with enhanced reconstruction
- **Context Length:** Maximum number of tokens the model can process
- **Multilingual:** Models trained on multiple languages



# Training Objective - Contrastive Learning

## Core Objective

Train the model to place similar items close together in embedding space while pushing dissimilar items far apart.

## How It Works

The model learns by examining pairs or triplets of examples:

### Positive Pair (should be close):

- "The cat sat on the mat" ↔ "A feline rested on the rug"

### Negative Pair (should be far):

- "The cat sat on the mat" ↔ "How to bake chocolate cookies"

```
Loss = -log( exp(sim(a, p)) / Σ exp(sim(a, n)) )
```

where: sim = similarity, a = anchor, p = positive, n = negatives

### ✓ Advantages

- ✓ Creates semantically meaningful embeddings
- ✓ Works well with large datasets
- ✓ Naturally handles synonyms and paraphrases
- ✓ Self-supervised (doesn't need manual labels)

### ✗ Challenges

- ✗ Requires careful negative sampling
- ✗ Computationally expensive (many comparisons)
- ✗ Sensitive to batch size
- ✗ Hard negatives mining can be tricky



# Training Objective - Masked Language Modeling (MLM)

## Core Objective

Predict masked (hidden) words in a sentence using surrounding context. This teaches the model deep language understanding.

## How It Works

The model sees sentences with some words replaced by [MASK] tokens and must predict what was hidden.

**Original:** "The quick brown fox jumps over the lazy dog"

**Masked:** "The [MASK] brown fox [MASK] over the lazy dog"

**Model predicts:** [MASK<sub>1</sub>] = "quick", [MASK<sub>2</sub>] = "jumps"

$$\text{Loss} = -\sum \log P(\text{masked\_word} \mid \text{context})$$

Maximize probability of correct word given surrounding words

### ✓ Advantages

- ✓ Captures bidirectional context
- ✓ No labeled data required
- ✓ Learns rich linguistic patterns
- ✓ Foundation for BERT and similar models

### ✗ Challenges

- ✗ Slower training than causal LM
- ✗ Pretrain-finetune gap (no [MASK] at inference)
- ✗ Assumes independence of masked tokens
- ✗ Not optimal for generation tasks



# Training Objective – RetroMAE (Retrospective Masked Auto-Encoding)

## Core Objective

Enhanced masked auto-encoding that reconstructs the entire input from a heavily masked version, forcing the model to learn more compressed and meaningful representations.

## How It Works

Unlike standard MLM which masks ~15%, RetroMAE masks 50-70% and uses two encoders:

**Step 1 - Encoder:** Process heavily masked input (only 30-50% visible)

**Step 2 - Decoder:** Reconstruct the ENTIRE original sentence

### Example:

Original: "The quick brown fox jumps over the lazy dog"

Masked: "The [M] [M] fox [M] [M] [M] lazy [M]"

Task: Reconstruct all 9 words from just 3 visible ones

```
Loss = MSE(original_embeddings, reconstructed_embeddings)
```

With asymmetric encoder-decoder architecture

### ✓ Advantages

- ✓ Creates highly compressed embeddings
- ✓ Better semantic understanding than standard MLM
- ✓ Efficient for retrieval tasks
- ✓ Strong performance with less data

### ✗ Challenges

- ✗ More complex architecture (encoder + decoder)
- ✗ Requires careful tuning of masking ratio
- ✗ Longer training time
- ✗ Higher computational cost



# From Embeddings to Search

## What We've Learned

- ✓ Transformer encoders create rich, contextual embeddings
- ✓ Self-attention captures relationships between words
- ✓ Different pooling strategies extract sentence-level representations

## Next: Vector Databases

Learn how to measure similarity, build efficient indexes, and search at scale using approximate nearest neighbor algorithms.



# Vector Similarity: Mathematical Foundations

## 1. Cosine Similarity

$$\cos(\theta) = (\mathbf{A} \cdot \mathbf{B}) / (\|\mathbf{A}\| \|\mathbf{B}\|)$$

$$\cos(\theta) = \sum(A_i \times B_i) / (\sqrt{\sum A_i^2} \times \sqrt{\sum B_i^2})$$

**Range:** [-1, 1]

**Interpretation:**

- 1.0 = identical direction
- 0.0 = orthogonal (unrelated)
- -1.0 = opposite direction

## 2. Euclidean (L2) Distance

$$d(\mathbf{A}, \mathbf{B}) = \sqrt{\sum (A_i - B_i)^2}$$

**Range:** [0,  $\infty$ ]

**Interpretation:**

- 0 = identical vectors
- Larger = more different
- Sensitive to magnitude

*Must normalize vectors first!*

## 3. Dot Product (Inner Product)

$$\mathbf{A} \cdot \mathbf{B} = \sum(A_i \times B_i) = \|\mathbf{A}\| \|\mathbf{B}\| \cos(\theta)$$

**Range:**  $[-\infty, \infty]$  ( $[-1, 1]$  if normalized)

**Key insight:** If vectors are normalized ( $\|\mathbf{A}\| = \|\mathbf{B}\| = 1$ ), then dot product = cosine similarity!

# Vector Similarity: Visual Intuition

## 1. Cosine Similarity

$$\cos(\theta) = \frac{(A \cdot B)}{(\|A\| \|B\|)}$$



**Measures angle** between vectors

Range: [-1, 1]

- 1.0 = identical direction
- 0.0 = orthogonal (unrelated)
- -1.0 = opposite direction

## 2. Euclidean (L2) Distance

$$d(A, B) = \sqrt{\sum (A_i - B_i)^2}$$



**Measures distance** in space

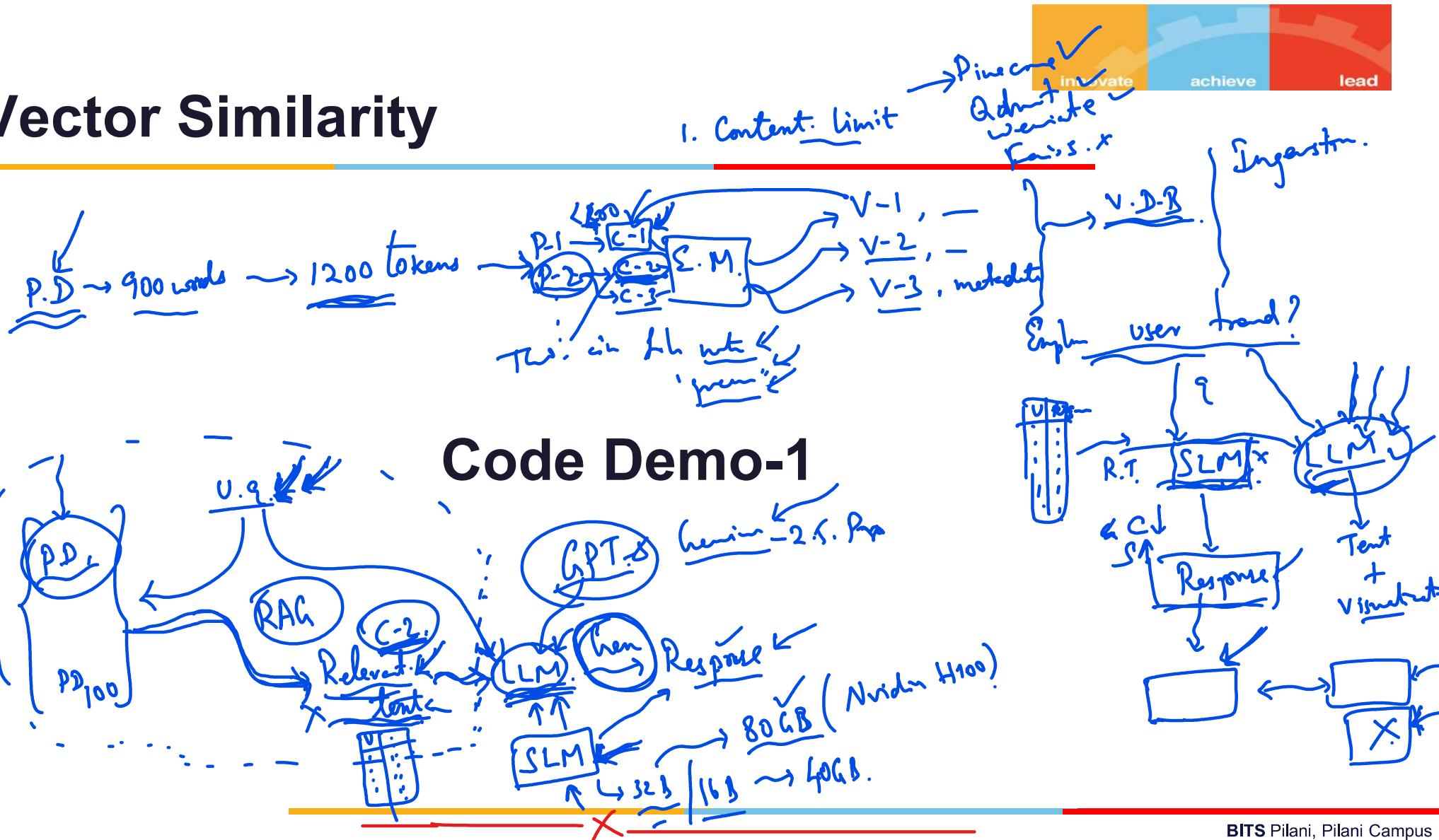
Range: [0, ∞]

- 0 = identical vectors
- Larger = more different
- **Sensitive to magnitude**



**Key Insight:** If vectors are normalized ( $\|A\| = \|B\| = 1$ ), then **dot product = cosine similarity**

# Vector Similarity





# The Computational Challenge

## Problem Statement

Given: 10 million document embeddings (768 dimensions each)

Task: Find top 10 most similar to query embedding

### Computational Complexity

#### For linear scan:

- Operations per query =  $n \times d$
- $n = 10,000,000$  documents
- $d = 768$  dimensions
- Total =  $7,680,000,000$  multiply-add operations ✓

#### At 1 billion ops/sec (modern CPU):

Time per query = 7.68 seconds

With 100 queries/sec: Need 768 CPU cores!



# Approximate Nearest Neighbor (ANN) ✓

## Scale Analysis

Documents	Linear Scan	Cost/month
10K	8ms	\$50
100K	77ms	\$50
1M	770ms	\$200
10M	7.7s ✗	\$5,000

## With ANN Index (HNSW)

Documents	HNSW	Cost/month
10K	2ms	\$50
100K	3ms	\$50
1M	5ms	\$120
10M	10ms ✓	\$250

## 🎯 The Solution: Approximate Nearest Neighbor (ANN)

**Key Insight:** We don't need the EXACT top 10 - getting 9/10 correct is often good enough!

ANN algorithms trade tiny accuracy loss (95-99% recall) for 100-1000x speedup by using smart data structures.



# ANN Indexing Strategies

## 1. Graph-Based ✓

### Example: HNSW

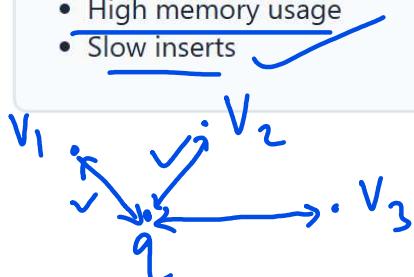
Build a navigable graph where nodes are vectors, edges connect similar vectors

#### Pros:

- Excellent recall (>95%)
- Fast queries ( $O(\log n)$ )
- No training needed

#### Cons:

- High memory usage
- Slow inserts



## 2. Partition-Based

### Example: IVF

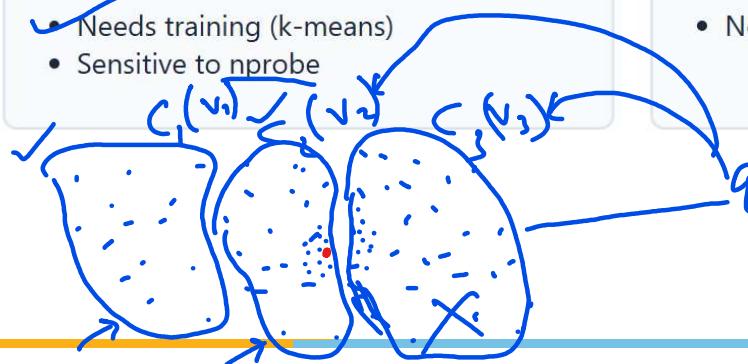
Cluster vectors into partitions, search only relevant partitions

#### Pros:

- Balanced performance
- Works with compression
- Scalable

#### Cons:

- Needs training (k-means)
- Sensitive to nprobe



## 3. Compression-Based ✓

### Example: PQ

Compress vectors to reduce memory, search in compressed space

#### Pros:

- 8-32x less memory
- Billions of vectors

#### Cons:

- Lower recall (~90%)
- Needs training

# HNSW: Hierarchical Navigable Small World

## Core Concept: Multi-Layer Skip List for Vectors

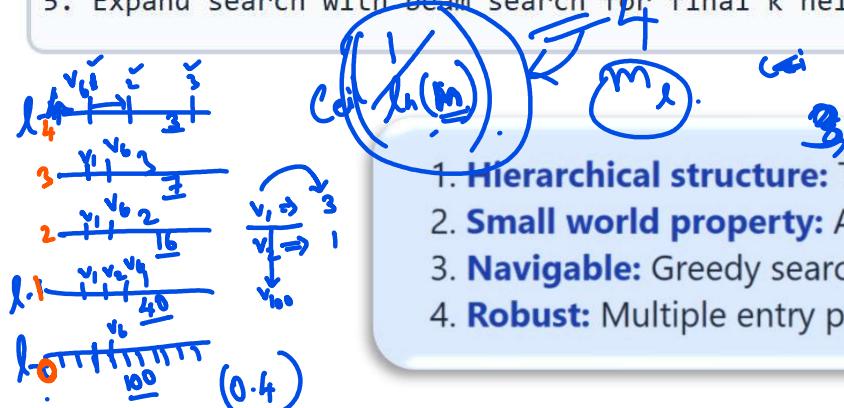
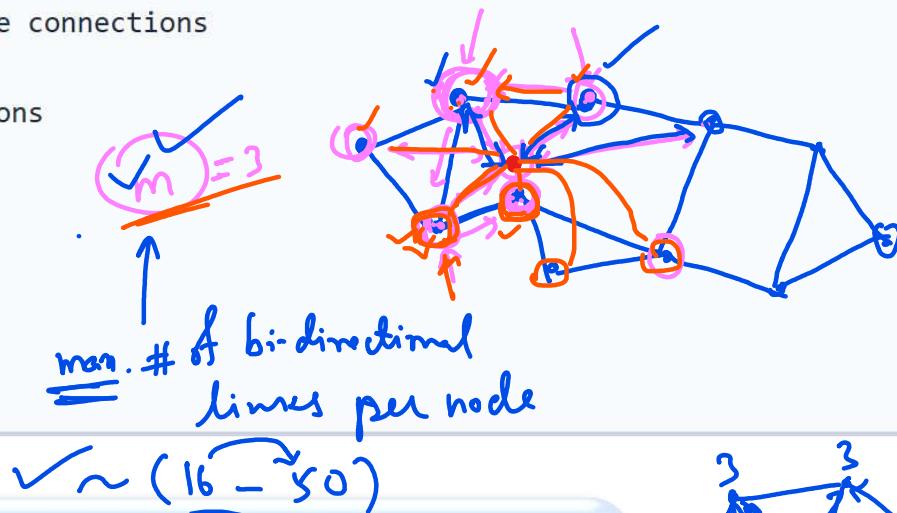
Layer 2: Entry → [ ] ← Sparse, long-range connections

Layer 1: [ ] ← Medium density

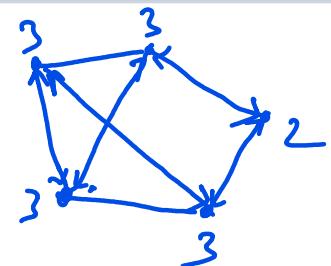
Layer 0: [ ] ← All vectors, dense connections

Search Process:

1. Enter at top layer (random entry point)"
2. Greedy search: move to nearest neighbor
3. When stuck (no closer neighbor), descend to next layer
4. Repeat until Layer 0
5. Expand search with beam search for final k neighbors

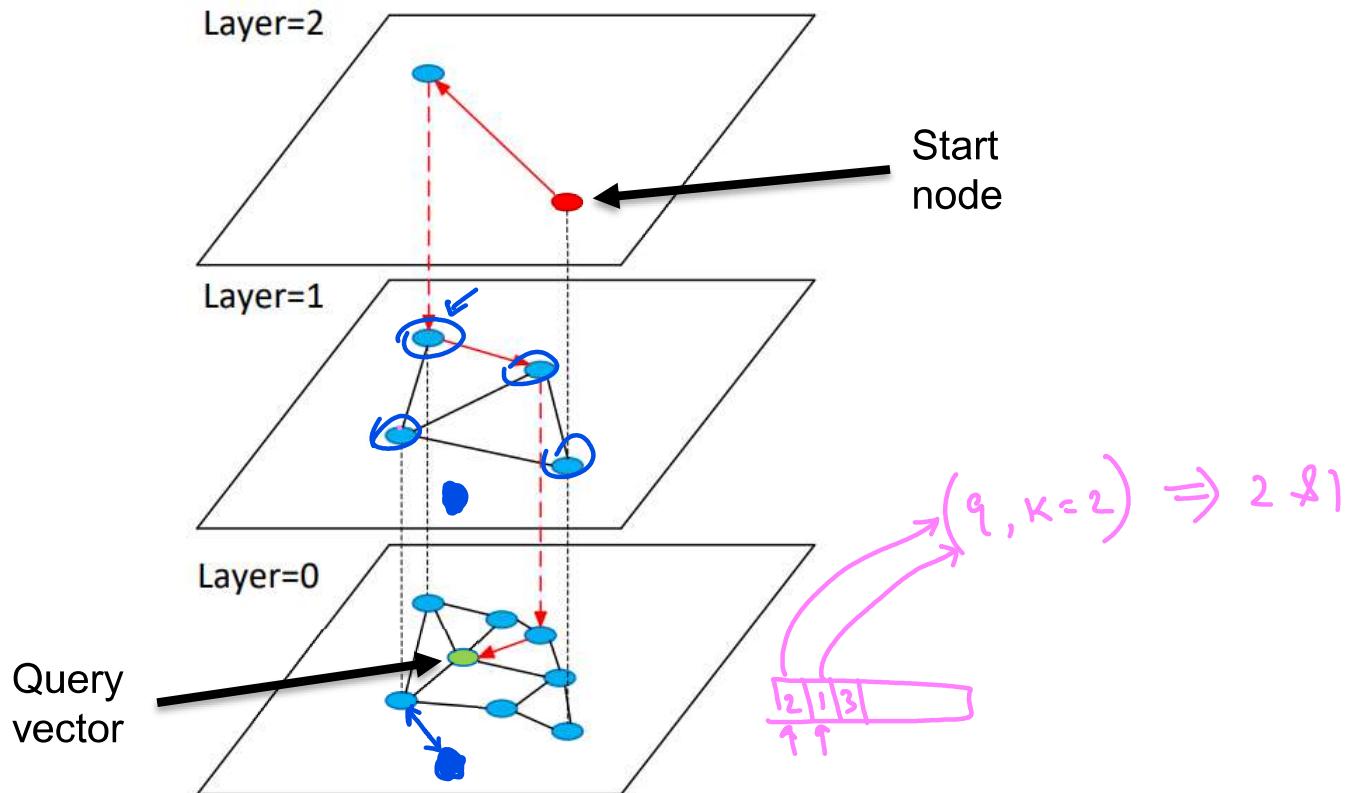


- 1. **Hierarchical structure:** Top layers skip across dataset for fast navigation
- 2. **Small world property:** Average path length is  $O(\log n)$
- 3. **Navigable:** Greedy search reliably finds near-optimal path
- 4. **Robust:** Multiple entry points prevent getting stuck in local minima





# HNSW Search: Step-by-Step



## ① Start at Top Layer (Entry Point)

Random entry point from sparse top layer

## ② Greedy Search

Move to nearest neighbor closer to query  
Repeat until no closer neighbors found

## ③ Descend to Next Layer

Use current position as entry for denser layer

## ④ Repeat Until Layer 0

Final greedy search on full graph

## ✓ Beam Search for Top-k

Expand search to find k nearest neighbors



# HNSW: Memory layout

## Per Vector Storage Components

### Storage Breakdown

- ✓ **Original vector:**  $d \times 4$  bytes
- ✓ **Layer info:** 1 byte
- ✓ **Connections:**  $M \times 4$  bytes  $\times$  layers

### Example (768-dim, M=16)

Vector:  $768 \times 4 = 3,072$  bytes

Layer: 1 byte

Connections: ~64 bytes avg

Total: ~3,136 bytes per vector



# HNSW: Memory Calculation by Scale

Scale	Vector Data	Graph Overhead	Total Memory
<u>1M vectors</u>	~3.1 GB (1.02× raw)	~0.5-1 GB (metadata)	<u>~3.6-4.1 GB</u> (1.2-1.3× raw)
<u>10M vectors</u>	~31 GB (1.02× raw)	~10-20 GB (adjacency)	<u>~41-51 GB</u> (1.4-1.7× raw)
<u>100M vectors</u>	~310 GB (1.02× raw)	~200-300 GB (multi-layer)	<u>~510-610 GB</u> (1.7-2.0× raw)

## ⚠ Production Planning: Memory Budget Guidelines

- **Per-vector data:** Minimal overhead (~1.02×)
- **Graph structure:** 0.4-1.0× additional (depends on M and scale)
- **Practical rule:** Budget **1.5-2.0× raw vector memory** for production HNSW



# Parameter Tuning: HNSW

## M (Connections per Layer) ✓

**16 - 64**

### Lower (16):

- Less memory
- Faster build
- Lower recall

### Higher (64):

- More memory
- Better recall
- Slower inserts

Default: M=16 for most cases

## ef\_construction ✓

**100 - 200**

### Lower (100):

- Faster build
- Lower quality index

### Higher (200+):

- Slower build
- Better index quality
- Higher recall

Default:  
ef\_construction=200

## ef\_search (Query Time) ✓

**50 - 500**

### Tune at runtime!

- ef=50: Fast, ~90% recall
- ef=100: Balanced
- ✓ ef=500: Slow, 99% recall

### Adjust per query needs

Typical: ef\_search=100-200



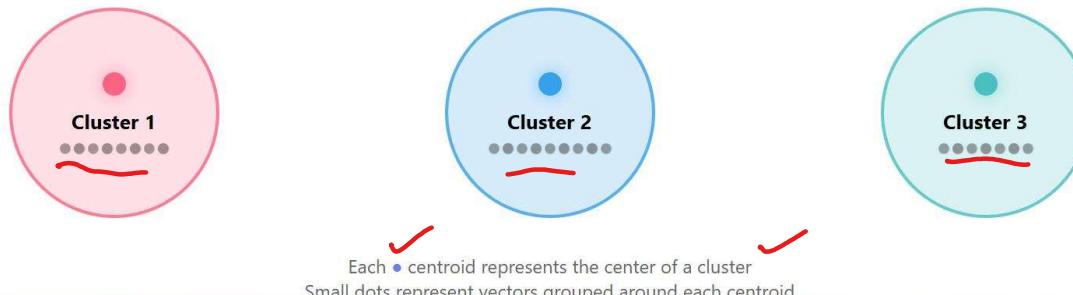
**Pro Tip:** Start with defaults (M=16, ef\_construction=200, ef\_search=100). Profile on your data, then adjust ef\_search for latency-recall tradeoff.



# IVF: Inverted File Index

## Step 1: Training Phase (k-means Clustering)

Partition the vector space into clusters using k-means algorithm



## Step 2: Vector Assignment

Assign each vector to its nearest centroid

For each vector  $v$ :

1. Calculate:  $\text{dist}(v, c_1)$ ,  $\text{dist}(v, c_2)$ ,  $\text{dist}(v, c_3)$ , ...
2. Find minimum distance
3. If  $\text{dist}(v, c_i)$  is minimum  $\rightarrow$  add  $v$  to  $\text{list}_i$



Result: An **inverted index** mapping each cluster to its vectors

- **Balanced:** Good speed/memory/accuracy tradeoff
- **Scalable:** Works well with billions of vectors
- **Composable:** Easily combined with PQ compression
- **Flexible:** nprobe tunable at query time



# IVF: Inverted File Index - Search

## Query Process:

1. Query vector  $q$  arrives

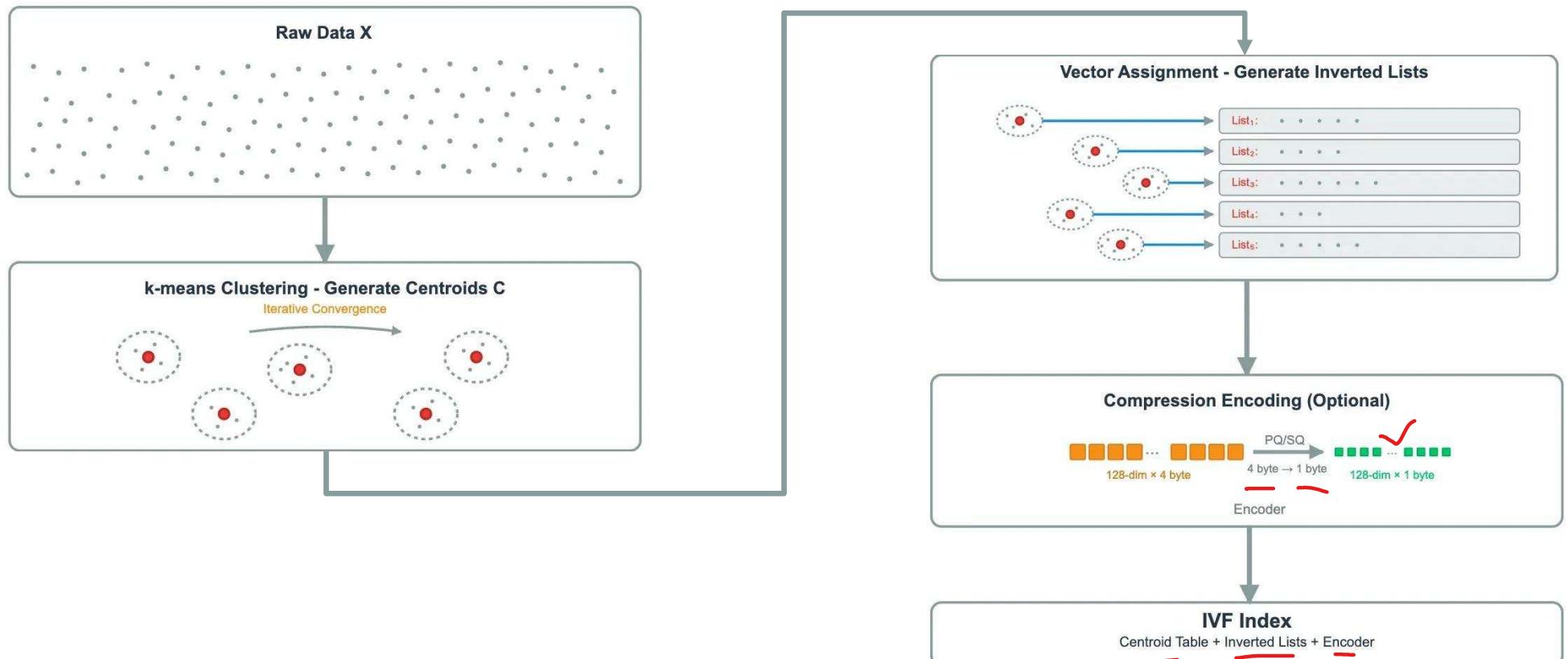
2. Find  $nprobe$  nearest centroids  
(e.g.,  $nprobe = 3 \rightarrow$  search top 3 closest clusters)

3. Search **only those clusters** *3 clusters*  
(Instead of searching ALL vectors in database!)

✓ 4. Merge and rank results across selected clusters



# IVF: Inverted File Index + PQ





# Parameter Tuning: IVF

## nlist (Number of Clusters)

$\sqrt{n}$  to  $n/1000$

### Too Few:

- Large partitions
- Slow search

### Too Many:

- Tiny partitions
- Many to search

### Sweet Spot:

- $4\sqrt{n}$  often works well

1M vectors  $\rightarrow$  nlist  $\approx 1,000 - 4,000$

## nprobe (Clusters to Search)

1 - 100

### Lower (1-5):

- Very fast
- Lower recall ( $\sim 70\text{-}85\%$ )

### Higher (50-100):

- Slower search
- Higher recall ( $\sim 95\%+$ )

### Tune at query time

Typical: nprobe=10-20

## PQ Subquantizers (m)

$d/8$  to  $d/4$

### More subquantizers:

- Better accuracy
- More memory

### Fewer subquantizers:

- More compression
- Lower accuracy

### Must divide dimension

768-dim  $\rightarrow$  m=96 (8 per subvector)

 **Training Time:** IVF requires training k-means on sample data (typically 10K-100K vectors). Budget:  $nlist \times 50$  iterations  $\times 10$  minutes for 1M vectors.



# Product Quantization (PQ) ✓

## Core Concept: Divide & Conquer

Split high-dimensional vectors into smaller chunks and quantize each independently



Each subvector is encoded independently using a trained codebook



# Product Quantization (PQ)

## Step 1: Train Codebooks (k-means per Subspace)

For each subvector position, train a codebook with k centroids (typically k=256)

### Codebook 1 (Sub 1)

- ID: 0 → [0.2, 0.5, 0.1, ...]
- ID: 1 → [0.8, 0.3, 0.6, ...]
- ID: 2 → [0.1, 0.9, 0.4, ...]
- ...
- ID: 255 → [0.4, 0.2, 0.7, ...]

### Codebook 2 (Sub 2)

- ID: 0 → [0.3, 0.7, 0.2, ...]
- ID: 1 → [0.6, 0.1, 0.8, ...]
- ID: 2 → [0.9, 0.4, 0.3, ...]
- ...
- ID: 255 → [0.2, 0.6, 0.5, ...]

### Codebook 3 & 4...

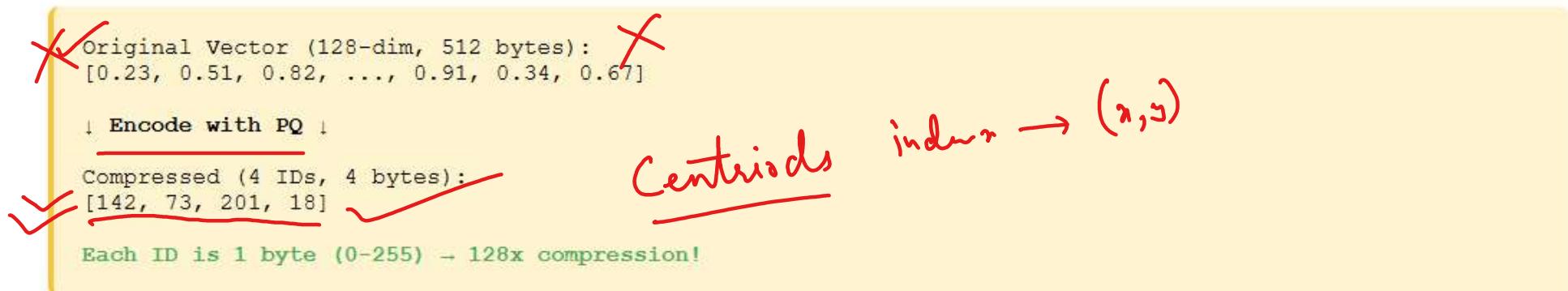
- Each subspace gets
- its own codebook
- trained independently
- using k-means
- clustering



# Product Quantization (PQ)

## 12 34 Step 2: Encode Vectors → Compact IDs

Replace each subvector with the ID of its nearest centroid in the codebook



Before PQ

**512 B**

per vector



After PQ

**4 B**

per vector



# Product Quantization: Step-3: Fast Distance Computation

Asymmetric  
Distance  
Computation (ADC)

## Step 1: Query Preparation ✓

Query vector  $\mathbf{q}$  arrives (full precision)

Split into subvectors:  $[\mathbf{q}_1, \mathbf{q}_2, \mathbf{q}_3, \mathbf{q}_4]$

$[0.23, 0.51, \dots] \rightarrow \text{splits} \rightarrow 4 \text{ chunks}$

## Step 2: Distance Tables

Pre-compute distances from each  $\mathbf{q}_i$  to all 256 centroids in codebook  $i$  ✓

$\text{table}_1[256], \text{table}_2[256], \text{table}_3[256], \text{table}_4[256]$

## Step 3: Database Vector Lookup

Each database vector stored as 4 IDs:  $[\text{ID}_1, \text{ID}_2, \text{ID}_3, \text{ID}_4]$

```
distance = table1[\text{ID}_1] + table2[\text{ID}_2] + table3[\text{ID}_3] + table4[\text{ID}_4]
```

**Just 4 lookups + 3 additions!** (vs 128 multiplications + additions)

⚡ **Speed Gain:** Distance computation becomes table lookup instead of expensive dot products. Enables searching billions of vectors in RAM.



# Product Quantization (PQ)

## Why Product Quantization is Powerful:

### Massive Compression

Reduce memory by 32-128x while maintaining good accuracy

### Fast Search

Distance = simple table lookups instead of expensive dot products

### Scalability

Store billions of vectors in RAM that would otherwise require terabytes

### Combinable

Often combined with IVF for even better performance (IVFPQ)

## When to Use PQ

- **Billion-scale datasets:** When RAM is the bottleneck
- **With IVF:** IVF+PQ is the standard for massive scale
- **Budget constraints:** Reduce infrastructure costs 10x
- **Acceptable recall loss:** 90-95% recall is sufficient



# Complexity Comparison

$n$  = vectors in database |  $d$  = vector dimension |  $k$  = clusters (IVF) |  $m$  = compressed dimension (PQ) |  $i$  = k-means iterations

Algorithm	Search Time	Memory	Build Time	Recall
Linear Scan	$O(n \cdot d)$	$O(n \cdot d)$	$O(1)$	100%
HNNSW	$O(\log n)$	$O(n \cdot d)$	$O(n \log n)$	95-99%
IVF	$O((n/k) \cdot d)$	$O(n \cdot d)$	$O(n \cdot d \cdot i)$	90-95%
PQ	$O(n \cdot m)$	$O(n \cdot m)$	$O(n \cdot d \cdot i)$	70-85%
IVF+PQ	$O((n/k) \cdot m)$	$O(n \cdot m)$	$O(n \cdot d \cdot i)$	85-92%
HNNSW+PQ	$O(\log n)$	$O(n \cdot m)$	$O(n \log n)$	90-96%

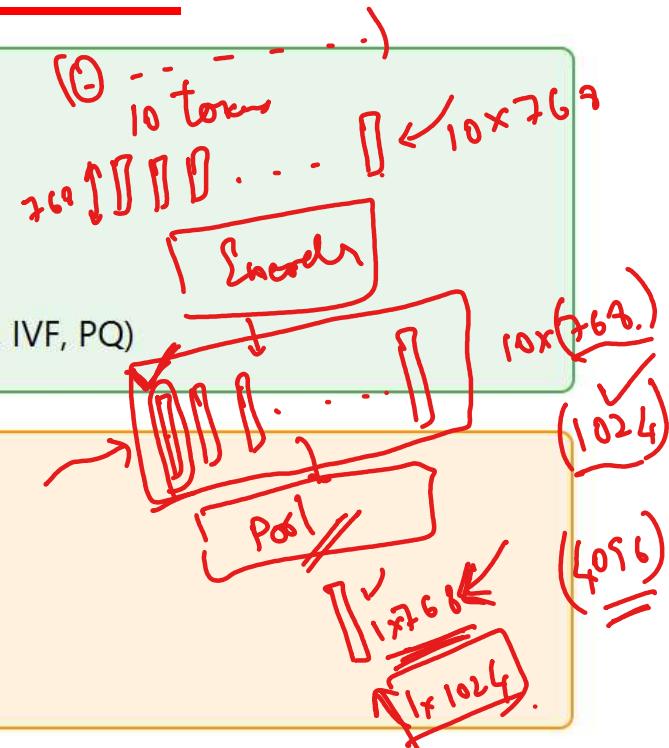
Key Insight: HNSW trades memory for speed. IVF+PQ trades accuracy for compression. HNSW+PQ is the sweet spot for production.



# From Dense to Sparse: Complementary Approaches

✓ **What We've Learned So Far**

- Transformer encoders create rich, contextual embeddings
- Self-attention captures relationships between words
- Different pooling strategies extract sentence-level representations
- We can search **billions of vectors efficiently** using ANN algorithms (HNSW, IVF, PQ)



✗ **But Vector Search Has Limitations...**

- Struggles with exact keyword matching (e.g., "ERROR code 500")
- Misses rare/unique terms not in training data (IDs, codes)
- Can't handle out-of-vocabulary words perfectly

**Next: Sparse retrieval methods (BM25)**  
that complement dense search for hybrid systems



# TF-IDF: Foundation of Sparse Retrieval

**Term Frequency - Inverse Document Frequency:** A statistical measure of word importance in documents

## ✓ Term Frequency (TF)

**Raw count:**

$$TF(t, d) = \text{count of term } t \text{ in document } d$$

**Normalized (common):**

$$TF(t, d) = f_{t,d} / |d|$$

$f_{t,d}$  = raw count,  $|d|$  = total terms

**Intuition:**

More occurrences = more important

But normalize by document length

## Inverse Document Frequency (IDF)

**Formula:**

$$IDF(t) = \log(N / df_t)$$

$N$  = total documents

$df_t$  = documents containing  $t$

**Intuition:**

Rare terms are more informative

"Qdrant" → high IDF

"the" → low IDF

## Combined TF-IDF Score

$$\begin{aligned} TF-IDF(t, d) &= TF(t, d) \times IDF(t) \\ &= (f_{t,d} / |d|) \times \log(N / df_t) \end{aligned}$$



# TF-IDF: Foundation of Sparse Retrieval

Corpus: 1000 documents

, 300 documents has the word "machine"

Document d: "machine learning is a subset of machine learning"

400,

Query q: "machine learning"

## Step 1: Calculate IDF for Each Query Term

### Term "machine"

```
f_machine,d = 2  
|d| = 9  
TF = 2/9 = 0.222  
df_machine = 300  
IDF = log(1000/300) = 1.20  
TF-IDF = 0.222 × 1.20 = 0.267
```

### Term "learning"

```
f_learning,d = 3  
|d| = 9  
TF = 3/9 = 0.333  
df_learning = 400  
IDF = log(1000/400) = 0.916  
TF-IDF = 0.333 × 0.916 = 0.305
```

Document score for query "machine learning":

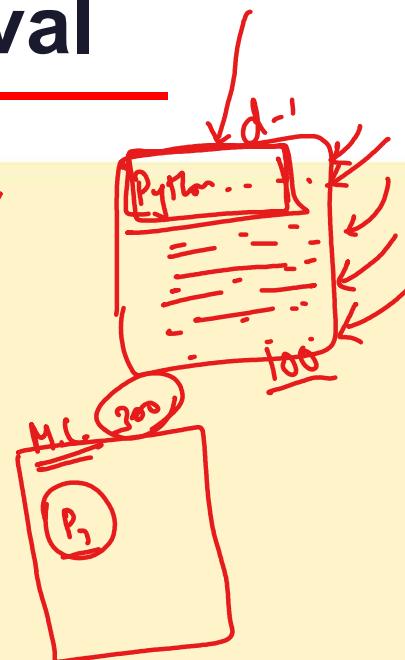
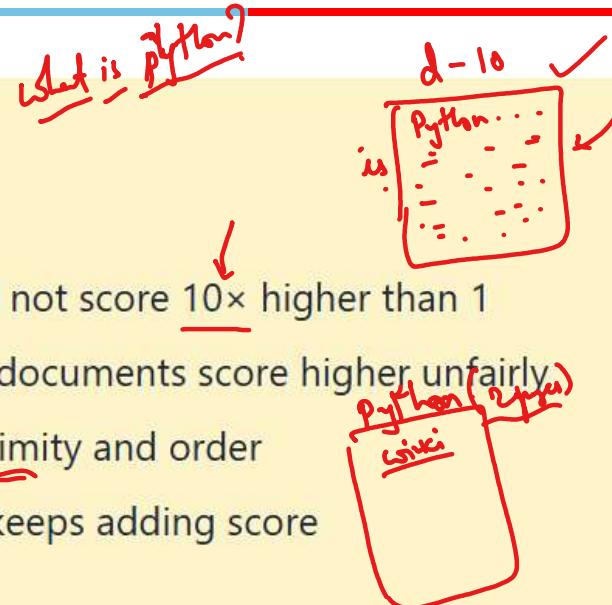
$$\text{score}(q, d) = 0.267 + 0.305 = 0.572$$

# TF-IDF: Foundation of Sparse Retrieval

## ✗ TF-IDF Limitations

- **Linear TF:** 10 occurrences should not score 10x higher than 1
- **No length normalization:** Long documents score higher unfairly
- **Bag of words:** Ignores term proximity and order
- **No term saturation:** Repetition keeps adding score

**Solution:** BM25 fixes these issues!





# BM25: Best Matching 25 (Improved TF-IDF)

## What BM25 Improves

BM25 is a **probabilistic ranking function** that addresses TF-IDF's limitations with:

1. **Term frequency saturation:** Diminishing returns for repeated terms
2. **Document length normalization:** Fair scoring regardless of length
3. **Probabilistic foundation:** Based on probability ranking principle

## BM25 Formula (Full Version)

$$\text{score}(Q, D) = \sum_{i=1}^n \text{IDF}(q_i) \cdot [f(q_i, D) \cdot (k_1 + 1)] / [f(q_i, D) + k_1 \cdot (1 - b + b \cdot |D| / \text{avgdl})]$$



# BM25: Best Matching 25 (Improved TF-IDF)

## IDF Component (Modified)

$$\text{IDF}(q_i) = \log[(N - df_i + 0.5) / (df_i + 0.5) + 1]$$

### Components:

- $N$  = total documents
- $df_i$  = docs containing term  $i$
- $+0.5$  = smoothing (avoid zero)
- $+1$  = ensure positive score

## TF Component (Saturating)

**Key innovation:** Saturation function

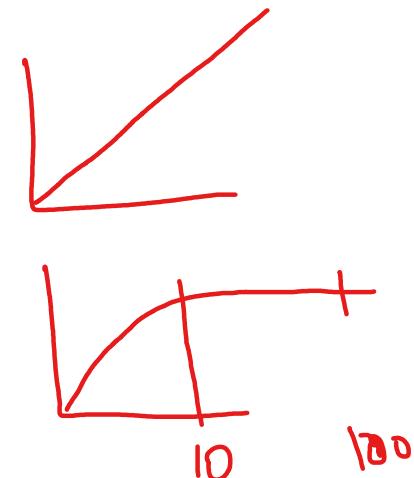
$$TF = f \cdot (k_1 + 1) / (f + k_1 \cdot \text{norm})$$

### Where:

$$\text{norm} = 1 - b + b \cdot (|D| / \text{avgdl})$$

$$\text{As } f \rightarrow \infty, TF \rightarrow (k_1 + 1) / (k_1 \cdot \text{norm})$$

Bounded, not linear!



100

Parameter	Default	Range	Effect
-----------	---------	-------	--------

✓  **$k_1$**  1.5 [1.2, 2.0] Controls TF saturation. Higher = more emphasis on term frequency.  
 **$k_1$  controls saturation:** With  $k_1=1.5$ , going from 1→2 occurrences helps more than 10→11

✓  **$b$**  0.75 [0, 1] Length normalization. 0 = no norm, 1 = full norm  
 **$b$  controls length penalty:**  $b=0.75$  means longer docs get penalized, but not fully

**Combined effect:** Short docs with focused terms rank high, long docs need more evidence



# BM25: Step-by-Step Example

## Scenario

Corpus: 1000 documents, average length: 50 terms

Query: "machine learning"

Document D: "Machine learning is a subset of machine learning and deep learning" (12 terms)

## Step 1: Calculate IDF for Each Query Term

**Term "machine":** appears in 300 documents

$$\text{IDF} = \log[(1000 - 300 + 0.5) / (300 + 0.5) + 1]$$

$$\text{IDF} = \log[700.5 / 300.5 + 1]$$

$$\text{IDF} = \log[2.331 + 1] = \log[3.331] = \mathbf{1.203}$$

**Term "learning":** appears in 400 documents

$$\text{IDF} = \log[(1000 - 400 + 0.5) / (400 + 0.5) + 1]$$

$$\text{IDF} = \log[600.5 / 400.5 + 1] = \log[2.499] = \mathbf{0.916}$$



# BM25: Step-by-Step Example

## Step 2: Calculate TF Component

For "machine" ( $f=2$ ,  $k_1=1.5$ ,  $b=0.75$ ):

$$\text{norm} = 1 - 0.75 + 0.75 \times (12/50)$$

$$\text{norm} = 0.25 + 0.18 = 0.43$$

$$\text{TF} = [2 \times (1.5 + 1)] / [2 + 1.5 \times 0.43]$$

$$\text{TF} = 5.0 / 2.645 = \mathbf{1.890}$$

For "learning"

( $f=3$ , same norm=0.43):

$$\text{TF} = [3 \times 2.5] / [3 + 0.645]$$

$$\text{TF} = 7.5 / 3.645 = \mathbf{2.058}$$

**Step 3: Combine Scores**

$$\begin{aligned}\text{BM25}(Q, D) &= \text{IDF}(\text{machine}) \times \text{TF}(\text{machine}) + \text{IDF}(\text{learning}) \times \text{TF}(\text{learning}) \\ &= 1.203 \times 1.890 + 0.916 \times 2.058 \\ &= 2.274 + 1.885 = \mathbf{4.159}\end{aligned}$$



# BM25: Step-by-Step Example

## Understanding the Difference

TF-IDF score (normalized by document length) :

$$\text{"machine": } (2/9) \times \log(1000/300) = 0.222 \times 1.20 = 0.267$$

$$\text{"learning": } (3/9) \times \log(1000/400) = 0.333 \times 0.916 = 0.305$$

✓ Total = 0.572

✓ BM25 score (with saturation & length normalization) :

$$\text{score}(q, d) = 4.159$$

### Why BM25 Scores Higher:

- **Document length normalization:** BM25 considers document is shorter than average (12 vs 50 tokens) → less penalty
- **Term frequency saturation:** 3 occurrences aren't 3x better than 1 → diminishing returns applied
- **Probabilistic foundation:** Based on probability ranking principle → better ranking properties



# Why Hybrid Search? Dense + Sparse

## X Dense-Only Failures

### Example 1: Exact Keywords

Query: "ERROR code 500"

Doc A: "Server error 500 occurred"  
Dense score: 0.72

Doc B: "Something went wrong"  
Dense score: 0.85 ← Wrong ranking!

Why? Embedding trained on general text  
doesn't know "500" is crucial

## ✓ Sparse (BM25) Catches These

### Example 1:

Query: "ERROR code 500"

Doc A: "Server error 500 occurred"  
BM25 score: 8.3 (high IDF for "500")

Doc B: "Something went wrong"  
BM25 score: 0.0 (no matching terms)

Correct ranking!



# Why Hybrid Search? Dense + Sparse

✗ Dense-Only Failures ✓

Example 2: Unique IDs

Query: "AIMLCZG521 schedule"

Doc A: "Course schedule for AI/ML"  
Dense: 0.88 ← Wrong!

Doc B: "AIMLCZG521 timing"  
Dense: 0.75

Unique terms not in training data

✓ Sparse (BM25) Catches These  
Example 2:

Query: "AIMLCZG521 schedule"

Doc A: "Course schedule for AI/ML"  
BM25: 3.2 ("schedule" matches)

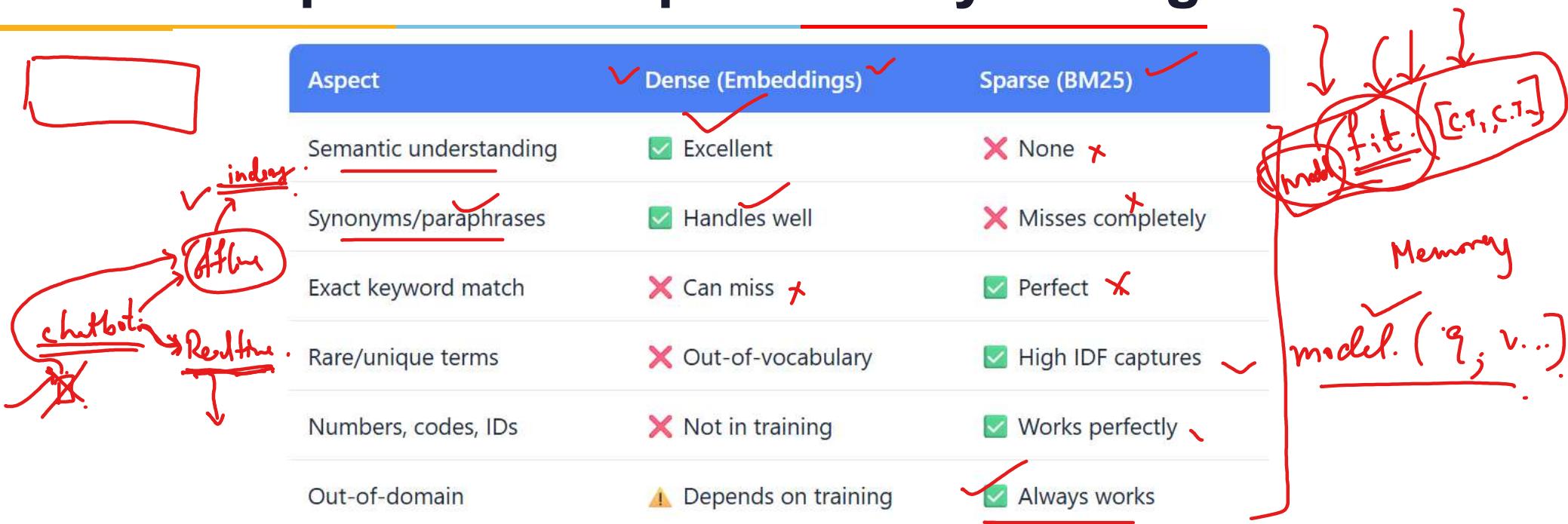
Doc B: "AIMLCZG521 timing"  
BM25: 12.5 (exact ID match!)

Correct ranking!



# Dense + Sparse - Complementary Strengths

Aspect	Dense (Embeddings)	Sparse (BM25)
Semantic understanding	✓ Excellent	✗ None
Synonyms/paraphrases	✓ Handles well	✗ Misses completely
Exact keyword match	✗ Can miss	✓ Perfect
Rare/unique terms	✗ Out-of-vocabulary	✓ High IDF captures
Numbers, codes, IDs	✗ Not in training	✓ Works perfectly
Out-of-domain	⚠ Depends on training	✓ Always works



## The Hybrid Solution

**Use BOTH retrieval methods and combine their results!**

Dense finds semantically similar content, Sparse ensures exact matches aren't missed.



# The Challenge: Combining Different Score Ranges

Dense retrieval gives cosine similarities  $[0, 1]$

BM25 gives unbounded scores  $[0, \infty]$

**Problem:** Can't just add them! Need score-agnostic fusion.

## Solution:

### Min-Max Normalization:

$$\text{score}' = (\text{score} - \min) / (\max - \min)$$

✗ Sensitive to outliers

### Z-score Normalization:

$$\text{score}' = (\text{score} - \mu) / \sigma$$

✗ Assumes normal distribution

### Reciprocal Rank Fusion (RRF):

✓ No assumptions about score distributions



# Reciprocal Rank Fusion (RRF): The Math

## RRF Formula

$$\text{RRF}(d) = \sum_{r \in \text{rankings}} \frac{1}{k + \underline{\text{rank}_r(d)}}$$

Where:

- $d$  = document being scored
- rankings = set of ranking lists (e.g., dense + sparse)
- $\text{rank}_r(d)$  = position of document  $d$  in ranking  $r$  (1, 2, 3, ...)

~~k~~ = constant (typically 60) to prevent division by small numbers

## Why k=60?

### Empirical finding from TREC competitions:

- $k=60$  balances contribution from high and low ranks
- Without  $k$ :  $1/1 = 1.0, 1/2 = 0.5$  (too steep drop)
- With  $k=60$ :  $1/61 = 0.0164, 1/62 = 0.0161$  (gentler)
- Prevents first rank from dominating too much

### Alternative k values:

- $k=0$ : Maximum emphasis on top ranks (aggressive)
- $k=60$ : Balanced (standard)
- $k=100$ : More weight to lower ranks (conservative)



# Properties of RRF

1. **Score-agnostic:** Only uses rank order, not raw scores
2. **Bounded:** Score  $\in [0, \text{number of rankings} / k]$
3. **Symmetric:** All rankings weighted equally
4. **Robust:** Works even if one ranking is poor
5. **No training:** No hyperparameters to tune (except k)

## ⌚ Why RRF Works So Well

Documents that rank high in MULTIPLE systems are likely truly relevant. RRF naturally gives bonus to "consensus" documents without explicit weighting.



# RRF: Example

Scenario: Query "machine learning tutorial"

## Dense Retrieval Results

Rank	Doc	Cosine
1	Doc A	0.92
2	Doc C	0.89
3	Doc B	0.85
4	Doc E	0.82
5	Doc D	0.79

## Sparse (BM25) Results

Rank	Doc	BM25
1	Doc B	87.3
2	Doc A	82.1
3	Doc D	79.5
4	Doc F	71.2
5	Doc C	68.9



# RRF: Example

NLU

## RRF Score Calculation (k=60)

### ✓ Doc A:

Dense rank: 1, Sparse rank: 2

$$\text{RRF} = 1/(60+1) + 1/(60+2) = 1/61 + 1/62 = 0.01639 + 0.01613 = \underline{\underline{0.03252}}$$

### Doc B:

Dense rank: 3, Sparse rank: 1

$$\text{RRF} = 1/(60+3) + 1/(60+1) = 1/63 + 1/61 = 0.01587 + 0.01639 = \underline{\underline{0.03226}}$$

### Doc C:

Dense rank: 2, Sparse rank: 5

$$\text{RRF} = 1/(60+2) + 1/(60+5) = 1/62 + 1/65 = 0.01613 + 0.01538 = \underline{\underline{0.03151}}$$

### Doc D:

Dense rank: 5, Sparse rank: 3

$$\text{RRF} = 1/(60+5) + 1/(60+3) = 1/65 + 1/63 = 0.01538 + 0.01587 = \underline{\underline{0.03125}}$$

### Doc E:

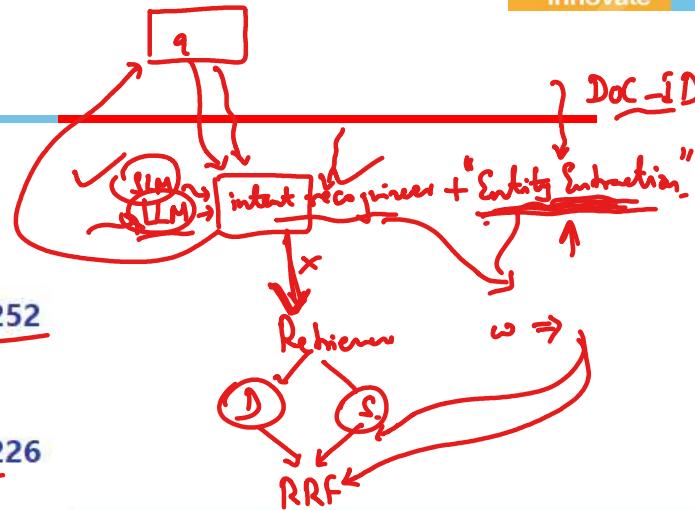
Dense rank: 4, Sparse rank: not in top 5 (treat as  $\infty$ )

$$\text{RRF} = 1/(60+4) + 0 = 1/64 = \underline{\underline{0.01562}}$$

### Doc F:

Dense rank: not in top 5, Sparse rank: 4

$$\text{RRF} = 0 + 1/(60+4) = 1/64 = \underline{\underline{0.01562}}$$



Final Ranking				
Rank	Doc	RRF Score	Dense Rank	Sparse Rank
1	Doc A	0.03252	1	2
2	Doc B	0.03226	3	1
3	Doc C	0.03151	2	5
4	Doc D	0.03125	5	3
5	Doc E, F	0.01562	4, -	-, 4

**Key Insight:** Doc A wins because it ranks high in BOTH systems (consensus), even though it wasn't #1 in sparse.



# Performance Characteristics (Typical)

Pipeline Stage	Latency	Recall	Precision
BM25 only	20ms	70%	60%
Vector only	50ms	75%	65%
Hybrid (BM25+Vector+RRF)	70ms	90%	75%



# Hybrid Search Implementation

**Code Demo**



# Key Takeaways

---

1. **Encoder Models:** Bidirectional transformers create contextual embeddings through self-attention and pooling
2. **Vector Similarity:** Cosine similarity (or normalized dot product) measures semantic closeness in embedding space
3. **ANN Algorithms:** HNSW (graph), IVF (partitions), PQ (compression) enable fast search at scale
4. **Sparse Retrieval:** BM25 improves TF-IDF with saturation and length normalization
5. **Hybrid Systems:** Dense + Sparse + RRF combines semantic and lexical strengths

## Formulae:

- Self-attention:  $\text{Attention}(Q, K, V) = \text{softmax}(QK^T / \sqrt{d_k})V$
- Cosine:  $\cos(A, B) = A \cdot B / (\|A\| \|B\|)$
- BM25:  $f(k_1+1) / (f + k_1(1-b+b|D|/\text{avgdl}))$
- RRF:  $\sum 1/(k+\text{rank})$



# Resources & Further Reading

- **Dense Passage Retrieval** (Karpukhin et al., 2020)  
Foundation for modern dense retrieval
- **HNSW** (Malkov & Yashunin, 2018)  
Efficient and robust approximate nearest neighbor search
- **Product Quantization** (Jégou et al., 2011)  
Vector compression for large-scale search
- **Sentence-BERT** (Reimers & Gurevych, 2019)  
Siamese networks for semantic similarity