DIFFERENT TYPES OF SIMILARITY SEARCHES TECHNIQUES

SWIPE

What is Similarity Search?

Similarity search is the process of finding data points that are most similar to a given query in a multi-dimensional vector space. When you work with vector databases, your data (text, images, audio, etc.) is converted into numerical vectors (embeddings) that capture semantic meaning. Similarity search then identifies which stored vectors are closest to your query vector.
Think of it like asking: "Show me the 5 most similar documents to this one" or "Find products that match this user's preferences."

How Similarity Search Works with Vector Databases

```
# What you write
results = vectorstore.similarity_search(query, k=5)

# What actually happens underneath
query_vector = embedding_model.embed(query)
similarities = calculate_similarity(query_vector, all_stored_vectors)
top_k_indices = get_top_k(similarities, k=5)
return retrieve_documents(top_k_indices)
```

The magic happens in the calculate_similarity() function, which uses mathematical formulas to measure how close vectors are to each other.

Common Use Cases

- Document Search: Finding relevant articles, papers, or web pages
- Recommendation Systems: Suggesting products, movies, or content
- Image Recognition: Finding similar images or detecting duplicates
- Chatbots & RAG: Retrieving relevant context for AI responses
- Anomaly Detection: Identifying unusual patterns in data
- Customer Segmentation: Grouping similar users or behaviors

Types of Similarity Measures

1. Cosine Similarity

What it is: Measures the angle between two vectors, focusing on direction rather than magnitude.

Mathematical Formula:

$$\text{cosine\_similarity}(A, B) = (A \cdot B) / (||A|| \times ||B||)$$

How it works step-by-step:

```
import numpy as np

def cosine_similarity(vec1, vec2):
    # Step 1: Calculate dot product (how much vectors align)
    dot_product = np.dot(vec1, vec2)

    # Step 2: Calculate magnitudes (length of each vector)
    magnitude_1 = np.linalg.norm(vec1)
    magnitude_2 = np.linalg.norm(vec2)

    # Step 3: Divide dot product by product of magnitudes
    similarity = dot_product / (magnitude_1 * magnitude_2)
    return similarity

# Example
vec1 = [1, 2, 3]
vec2 = [2, 4, 6]  # Same direction, different magnitude
similarity = cosine_similarity(vec1, vec2)
print(f"Cosine similarity: {similarity}")  # Output: 1.0 (identical direction)
```

Real-world example:

- Document 1: "The cat sat on the mat"   [0.2, 0.8, 0.1, 0.5, ...]
- Document 2: "A feline rested on the rug"   [0.3, 0.7, 0.2, 0.6, ...]
- Cosine similarity: 0.87 (high semantic similarity despite different words)

Best for:

- Text similarity and document search
- Recommendation systems
- High-dimensional data where vector length doesn't matter
- When you care about semantic meaning, not magnitude

Advantages:

- Normalized (always between -1 and 1)
- Ignores document length differences
- Great for sparse vectors

Disadvantages:

- Ignores magnitude completely
- Can be less intuitive than distance measures

2. Euclidean Distance (L2 Distance)

What it is: Calculates the straight-line distance between two points in multi-dimensional space.

Mathematical Formula:

$$\text{euclidean\_distance}(A, B) = \sqrt{\Sigma(A_i - B_i)^2}$$

How it works step-by-step:

```
def euclidean_distance(vec1, vec2):
    # Step 1: Calculate differences for each dimension
    differences = np.array(vec1) - np.array(vec2)

    # Step 2: Square all differences (removes negatives)
    squared_differences = differences ** 2

    # Step 3: Sum all squared differences and take square root
    distance = np.sqrt(np.sum(squared_differences))
    return distance
```

```
# Convert distance to similarity (smaller distance = higher similarity)
def euclidean_similarity(vec1, vec2):
    distance = euclidean_distance(vec1, vec2)
    return 1 / (1 + distance)

# Example
vec1 = [1, 2, 3]
vec2 = [2, 3, 4]  # Close but not identical
distance = euclidean_distance(vec1, vec2)
print(f"Euclidean distance: {distance}")  # Output: 1.73
```

Real-world example:

- Image 1 color histogram: [128, 64, 32, 16]
- Image 2 color histogram: [130, 66, 30, 18] (similar image)
- Euclidean distance: 4.47 (small distance = similar images)

Best for:

- Image similarity and computer vision
- Data where magnitude matters (like RGB values, coordinates)
- Low to medium-dimensional data
- When small differences in individual features are important

Advantages:

- Intuitive and easy to understand
- Considers all dimensions equally
- Good for continuous numerical data

Disadvantages:

- Sensitive to high-dimensional data (curse of dimensionality)
- Affected by scale differences between features
- Can be dominated by features with large ranges

Advantages:

- Less sensitive to outliers than Euclidean
- Works well with sparse vectors
- Computationally simpler (no squares or square roots)

Disadvantages:

- Less intuitive than Euclidean distance
- Doesn't consider the geometric structure of the space
- Can be too harsh on small differences

4. Dot Product Similarity

What it is: Simple multiplication and summation of corresponding vector elements.

Mathematical Formula:

$$\text{dot\_product}(A, B) = \Sigma(A_i \times B_i)$$

How it works step-by-step:

```
def dot_product_similarity(vec1, vec2):
  # Multiply corresponding elements and sum them up
  return np.dot(vec1, vec2)

# Example
vec1 = [1, 2, 3]
vec2 = [2, 1, 2]
similarity = dot_product_similarity(vec1, vec2)
print(f"Dot product: {similarity}")  # Output: 10 (1*2 + 2*1 + 3*2)
```

Real-world example:

- User preference: [0.8, 0.3, 0.9] (electronics, books, sports)
- Product features: [1.0, 0.1, 0.2] (electronics-heavy product)
- Dot product: 1.01 (strong match for electronics preference)

3. Manhattan Distance (L1 Distance)

What it is: Calculates distance by summing absolute differences along each dimension, like walking through city blocks.

Mathematical Formula:

```
manhattan_distance(A, B) = Σ|Ai - Bi|
```

How it works step-by-step:

```
def manhattan_distance(vec1, vec2):
    # Step 1: Calculate absolute differences for each dimension
    abs_differences = np.abs(np.array(vec1) - np.array(vec2))

    # Step 2: Sum all absolute differences
    distance = np.sum(abs_differences)
    return distance

# Example
vec1 = [1, 2, 3]
vec2 = [4, 1, 5]
distance = manhattan_distance(vec1, vec2)
print(f"Manhattan distance: {distance}")  # Output: 6 (|1-4| + |2-1| + |3-5|)
```

Real-world example:

- User behavior vector 1: [5, 0, 3, 0, 2] (ratings for 5 categories)
- User behavior vector 2: [4, 1, 2, 0, 3]
- Manhattan distance: 4 (sum of absolute differences)

Best for:

- Sparse data with many zero values
- High-dimensional spaces where you want to avoid curse of dimensionality
- When you want to penalize any differences equally
- Discrete or categorical data

Best for:

- When both magnitude and direction matter
- Normalized embeddings (where vectors have unit length)
- Simple similarity calculations
- When you want to reward strong positive correlations

Advantages:

- Computationally very efficient
- Simple to understand and implement
- Good for normalized vectors

Disadvantages:

- Sensitive to vector magnitudes
- Can produce unbounded similarity scores
- Not suitable for comparing vectors of different scales

Vector Database Implementations

Default Configurations by Database

Database	Default Similarity	Default Algorithm	Configurable Metrics
FAISS	L2 (Euclidean)	Flat/HNSW	L2, Inner Product, Custom
Pinecone	Cosine	HNSW	Cosine only
Weaviate	Cosine	HNSW	Cosine, Dot, L2, Manhattan, Hamming
ChromaDB	L2 (Euclidean)	HNSW	L2, Cosine, Inner Product
Qdrant	Cosine	HNSW	Cosine, Euclidean, Dot, Manhattan
Milvus	L2 (Euclidean)	IVF	L2, IP, Cosine, Hamming, Jaccard

Configuration Examples

FAISS:

```
import faiss
# L2 distance (Euclidean) - most common for embeddings
index = faiss.IndexFlatL2(dimension)

# Inner Product - for normalized vectors (equivalent to cosine)
index = faiss.IndexFlatIP(dimension)
```

```
Qdrant:

from qdrant_client.models import Distance, VectorParams

# Cosine similarity (default)
collection_config = VectorParams(
    size=768,
    distance=Distance.COSINE
)

ChromaDB:

# Configure distance metric
collection = client.create_collection(
    name="documents",
    metadata={"hnsw:space": "cosine"}  # or "l2", "ip"
)
```

Choosing the Right Similarity Measure

Decision Matrix

Use Case	Best Similarity	Reason
Text/Document Search	Cosine	Focuses on semantic meaning, not document length
Image Recognition	Euclidean	Pixel differences and visual features matter
Recommendation Systems	Cosine or Dot Product	User preference patterns and correlations
Anomaly Detection	Euclidean or Manhattan	Distance from normal patterns
Sparse Data	Manhattan	Less sensitive to outliers and zeros
High-Precision Needs	Exact Search	100% accuracy required
High-Speed Needs	Approximate (HNSW/LSH)	Speed over perfect accuracy

Practical Guidelines

1. **For text and language: Use Cosine Similarity**
   - Handles different document lengths well
   - Focuses on semantic relationships
   - Standard in NLP applications

2. **For images and computer vision: Use Euclidean Distance**
    - Pixel values and color differences matter
    - Geometric relationships are important
    - Good for feature vectors

3. **For recommendation systems: Use Cosine or Dot Product**
    - Cosine for user-item preferences
    - Dot product for content-based filtering
    - Handles sparse rating matrices well

4. **For high-dimensional sparse data: Use Manhattan Distance**
    - Less affected by curse of dimensionality
    - Robust to outliers
    - Works well with categorical features

Performance Optimization

Exact vs Approximate Search

Exact Search (Brute Force):

```
def brute_force_search(query_vector, all_vectors, k=5):
    similarities = []
    for i, vector in enumerate(all_vectors):
        sim = cosine_similarity(query_vector, vector)
        similarities.append((sim, i))

    similarities.sort(reverse=True)
    return similarities[:k]

# Time complexity: O(n*d) where n=vectors, d=dimensions
```

Approximate Search (HNSW):

- Builds multi-layer graph structure
- Searches from top layer down
- Uses "small world" navigation
- Time complexity: O(log n) average case

Best Practices

1. Vector Normalization: Normalize vectors for fair comparisons
2. Batch Processing: Use matrix operations for multiple queries
3. Memory Efficiency: Process data in chunks for large datasets
4. Index Selection: Choose appropriate index type based on data size
5. Parameter Tuning: Balance accuracy vs speed based on requirements

Conclusion

Understanding different similarity measures is crucial for building effective vector search systems. Each measure has specific strengths:

- **Cosine Similarity:** Best for text and semantic search
- **Euclidean Distance:** Ideal for images and continuous data
- **Manhattan Distance:** Great for sparse and high-dimensional data
- **Dot Product:** Simple and efficient for normalized vectors

STATFUSION Ai