# Distance Metrics and Similarity Measures in Vector Search Systems

## Introduction to Vector Similarity Search

Vector Similarity Search (VSS) represents one of the most transformative technological frameworks in modern data science and artificial intelligence. At its core, VSS enables machines to understand, organize, and retrieve information based on meaning and contextual relationships rather than exact matches or predefined categories. This paradigm shift has fundamentally altered how we interact with large-scale information systems and has become the backbone of numerous applications that we encounter daily.

In a world where data grows exponentially in both volume and complexity, traditional search and retrieval methods have reached their limitations. Vector Similarity Search addresses this need by representing data items as mathematical vectors in high-dimensional spaces where similarity becomes a measurable distance.

### Key Concepts in Vector Search

Before diving into specific distance metrics, let's establish some foundational concepts:

- **Vector**: A mathematical representation of data in an n-dimensional space, where each dimension corresponds to a feature of the data
- **Embedding**: The process of mapping raw data (text, images, audio, etc.) into a vector space in a way that preserves semantic relationships
- **Vector Space**: A mathematical space where vectors reside, with properties that allow for operations like addition, subtraction, and measuring distances
- **Distance/Similarity Metric**: A mathematical function that quantifies how "close" or "similar" two vectors are to each other
- **Vector Database**: A specialized database system designed to efficiently store, index, and query vector data

## Euclidean Distance (L2 Norm)

The Euclidean distance is perhaps the most intuitive distance metric, representing the "as-the-crow-flies" straight-line distance between two points in space.

### Mathematical Definition

For two vectors $A = (a_1, a_2, ..., a_n)$ and $B = (b_1, b_2, ..., b_n)$ in an n-dimensional space, the Euclidean distance is:

$$d_{Euclidean}(A, B) = \sqrt{\sum_{i=1}^{n}(a_i - b_i)^2}$$

In expanded form:

$$d_{Euclidean}(A, B) = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2 + \ldots + (a_n - b_n)^2}$$

### Implementation in Python

```python
import numpy as np

def euclidean_distance(vector_a, vector_b):
    """
    Calculate the Euclidean distance between two vectors.

    Args:
        vector_a: First vector (numpy array or list)
        vector_b: Second vector (numpy array or list)

    Returns:
        The Euclidean distance between the vectors
    """
    # Convert to numpy arrays if they aren't already
    a = np.array(vector_a)
    b = np.array(vector_b)

    # Calculate squared differences
    squared_diff = np.square(a - b)

    # Sum the squared differences and take the square root
    return np.sqrt(np.sum(squared_diff))

# Alternative implementation using numpy's built-in function
def euclidean_distance_numpy(vector_a, vector_b):
    return np.linalg.norm(np.array(vector_a) - np.array(vector_b))
```

### Properties and Characteristics

1. **Geometric Interpretation**: Represents the straight-line distance between points, preserving our intuitive understanding of physical space.
2. **Scale Sensitivity**: Euclidean distance is sensitive to the scale of the features. Features with larger scales will dominate the distance calculation.
3. **Curse of Dimensionality**: As dimensionality increases, Euclidean distance becomes less discriminative because distances between points tend to become more uniform.
4. **Computational Considerations**: Requires square root operations, which can be computationally expensive for very large datasets.

### Applications in Vector Search

- **Image Retrieval**: Finding visually similar images based on feature vectors
- **Sensor Data Analysis**: Identifying similar patterns in time-series data
- **Anomaly Detection**: Identifying outliers that are far from normal clusters
- **Geographical Applications**: When working with spatial coordinates

## Manhattan Distance (L1 Norm)

Manhattan distance, also known as L1 distance or taxicab distance, measures the sum of the absolute differences between the coordinates of two points.

### Mathematical Definition

For two vectors A and B in n-dimensional space:

$$d_{Manhattan}(A, B) = \sum_{i=1}^{n} |a_i - b_i|$$

In expanded form:

$$d_{Manhattan}(A, B) = |a_1 - b_1| + |a_2 - b_2| + \ldots + |a_n - b_n|$$

### Implementation in Python

```
import numpy as np

def manhattan_distance(vector_a, vector_b):
    """
    Calculate the Manhattan (L1) distance between two vectors.

    Args:
        vector_a: First vector (numpy array or list)
        vector_b: Second vector (numpy array or list)

    Returns:
        The Manhattan distance between the vectors
    """
    # Convert to numpy arrays if they aren't already
    a = np.array(vector_a)
    b = np.array(vector_b)

    # Calculate absolute differences and sum them
    return np.sum(np.abs(a - b))

# Alternative implementation using numpy's built-in function
def manhattan_distance_numpy(vector_a, vector_b):
    return np.linalg.norm(np.array(vector_a) - np.array(vector_b), ord=1)
```

### Properties and Characteristics

1. **Geometric Interpretation**: Represents the total distance traveled along the grid lines (like city blocks), hence the name "taxicab" distance.
2. **Robustness to Outliers**: Less sensitive to outliers compared to Euclidean distance, as it doesn't square differences.
3. **Feature Independence**: Each dimension contributes equally to the distance without interactions between dimensions.
4. **Computational Efficiency**: Only requires addition and subtraction operations, making it computationally less expensive than Euclidean distance.
5. **Sparse Data Handling**: Particularly effective for sparse high-dimensional data where many coordinates are zero.

#### Applications in Vector Search

- **Text Analysis**: Comparing document vectors where dimensions represent word frequencies
- **Recommendation Systems**: Finding similar user profiles in sparse preference spaces
- **Feature Selection**: In machine learning pipelines for feature importance assessment
- **Anomaly Detection**: When outliers should have less influence on the detection

## Cosine Similarity

Cosine similarity measures the cosine of the angle between two vectors, providing a measure of orientation similarity rather than magnitude similarity.

#### Mathematical Definition

For two vectors A and B:

$$\text{Cosine Similarity}(A, B) = \frac{A \cdot B}{||A|| \cdot ||B||} = \frac{\sum_{i=1}^{n} a_i b_i}{\sqrt{\sum_{i=1}^{n} a_i^2} \sqrt{\sum_{i=1}^{n} b_i^2}}$$

Where:

- A · B is the dot product of vectors A and B
- ||A|| and ||B|| are the norms (magnitudes) of vectors A and B

The resulting value ranges from -1 (completely opposite) to 1 (exactly the same), with 0 indicating orthogonality (decorrelation).

#### Implementation in Python

```
import numpy as np

def cosine_similarity(vector_a, vector_b):
    """
```

```
    Calculate the cosine similarity between two vectors.

    Args:
        vector_a: First vector (numpy array or list)
        vector_b: Second vector (numpy array or list)

    Returns:
        The cosine similarity between the vectors (range: -1 to 1)
    """
    # Convert to numpy arrays if they aren't already
    a = np.array(vector_a)
    b = np.array(vector_b)

    # Calculate dot product
    dot_product = np.dot(a, b)

    # Calculate magnitudes
    magnitude_a = np.linalg.norm(a)
    magnitude_b = np.linalg.norm(b)

    # Calculate cosine similarity
    if magnitude_a == 0 or magnitude_b == 0:
        return 0  # Handle zero vectors

    return dot_product / (magnitude_a * magnitude_b)

# Alternative implementation using scipy
def cosine_similarity_scipy(vector_a, vector_b):
    from scipy.spatial.distance import cosine
    return 1 - cosine(vector_a, vector_b)  # scipy.spatial.distance.cosine
returns distance, not similarity
```

## Properties and Characteristics

1. **Magnitude Independence**: Measures only the angle between vectors, ignoring their magnitudes. This makes it ideal for scenarios where magnitude is not relevant.
2. **Range Interpretation**:
   - Cosine similarity of 1: Vectors point in the same direction (perfectly similar)
   - Cosine similarity of 0: Vectors are perpendicular (completely dissimilar)
   - Cosine similarity of -1: Vectors point in opposite directions (perfectly dissimilar)
3. **Text Analysis Strength**: Particularly effective for text data where the frequency of terms (magnitude) is less important than their presence or absence.
4. **Normalization Effect**: By focusing on angles rather than distances, cosine similarity effectively normalizes for document length.
5. **Non-Metric Property**: Unlike Euclidean and Manhattan distances, cosine similarity is not a proper metric as it doesn't satisfy the triangle inequality.

### Applications in Vector Search

- **Document Similarity**: Comparing text documents regardless of their length
- **Topic Modeling**: Grouping documents with similar topics
- **Recommendation Systems**: Comparing user preference vectors
- **Natural Language Processing**: Analyzing semantic similarities between word embeddings
- **Clustering Algorithms**: Particularly effective in high-dimensional spaces

### Real-World Example: Document Similarity

Consider two documents represented as term frequency vectors:

- Document 1: [3, 2, 0, 5, 0, 1, 0]
- Document 2: [1, 0, 0, 3, 0, 0, 2]

The cosine similarity captures their topical similarity regardless of length differences:

```
doc1 = [3, 2, 0, 5, 0, 1, 0]
doc2 = [1, 0, 0, 3, 0, 0, 2]
similarity = cosine_similarity(doc1, doc2)
print(f"Document similarity: {similarity:.4f}")
# Output: Document similarity: 0.6693
```

## Advanced Distance Metrics

### Minkowski Distance (Lp Norm)

The Minkowski distance is a generalization of Euclidean and Manhattan distances, controlled by a parameter p:

$$d_{Minkowski}(A, B) = \left( \sum_{i=1}^{n} |a_i - b_i|^p \right)^{1/p}$$

- When p = 1: Manhattan distance
- When p = 2: Euclidean distance
- When p → ∞: Chebyshev distance (maximum coordinate difference)

Implementation:

```
def minkowski_distance(vector_a, vector_b, p):
    """
    Calculate the Minkowski distance between two vectors.

    Args:
        vector_a: First vector (numpy array or list)
        vector_b: Second vector (numpy array or list)
        p: The order of the norm (p=1 for Manhattan, p=2 for Euclidean)
```

```
    Returns:
        The Minkowski distance between the vectors
    """
    a = np.array(vector_a)
    b = np.array(vector_b)
    return np.power(np.sum(np.power(np.abs(a - b), p)), 1/p)
```

## Distance Metrics in Vector Databases

Vector databases are specialized systems designed to efficiently store, index, and query vector data. The choice of distance metric significantly impacts both the accuracy and performance of these systems.

### Impact of Distance Metrics on Vector Database Performance

1. **Indexing Strategies**: Different distance metrics require different indexing approaches:
   - Euclidean distance works well with spatial partitioning methods like k-d trees
   - Cosine similarity often requires specialized indexes that account for angular relationships
   - Manhattan distance can benefit from grid-based indexing schemes
2. **Computational Efficiency**: The computational complexity of distance calculations affects query performance:
   - Manhattan distance is typically faster to compute than Euclidean
   - Cosine similarity requires normalization steps
   - Advanced metrics like Mahalanobis are more computationally intensive
3. **Approximation Methods**: To scale to billions of vectors, databases often use approximation techniques:
   - Locality-Sensitive Hashing (LSH) works differently depending on the distance metric
   - Product Quantization compression efficiency varies with the distance function
   - Graph-based indices like HNSW need to be tailored to the specific distance metric

### Popular Vector Database Systems and Their Distance Metrics

1. **Faiss (Facebook AI Similarity Search)**:
   - Supports Euclidean distance (L2) and Inner Product (related to cosine similarity)
   - Optimized for high-performance GPU acceleration
   - Implements multiple indexing strategies including flat index, IVF, HNSW, and PQ
2. **Milvus**:

- Supports Euclidean, Manhattan, Hamming, Jaccard, and cosine similarity
- Distributed architecture for horizontal scaling
- Multiple index types including FLAT, IVF, HNSW, and ANNOY

3. **Pinecone**:
   - Supports Euclidean, cosine, and dot product similarity
   - Fully managed, cloud-native vector database
   - Optimized for both dense and sparse vectors
4. **Weaviate**:
   - Supports cosine similarity and cross-encoder reranking
   - Combines vector search with GraphQL for complex queries
   - Contextual indexing and filtering capabilities
5. **Qdrant**:
   - Supports Euclidean, cosine, dot product, and Manhattan distances
   - Designed for extended filtering alongside vector search
   - Optimized for fast updates and inserts alongside queries
6. **ChromaDB** :
   - Configure - Chroma Docs

> - distance metric - by default Chroma use L2 (Euclidean Distance Squared) distance metric for newly created collection. You can change it at creation time using `hnsw:space` metadata key. Possible values are `l2`, `cosine`, and 'ip' (inner product). (Note: `cosine` value returns `cosine distance` rather then `cosine similarity`. Ie. values close to 0 means the embeddings are more similar.)

- `hnsw:space` defines the distance function of the embedding space. The default is `l2` (squared L2 norm), and other possible values are `cosine` (cosine similarity), and `ip` (inner product).

| Distance | parameter | Equation |
|---|---|---|
| Squared L2 | `l2` | $d = \sum (A_i - B_i)^2$ |
| Inner product | `ip` | $d = 1.0 - \sum (A_i \times B_i)$ |
| Cosine similarity | `cosine` | $d = 1.0 - \frac{\sum (A_i \times B_i)}{\sqrt{\sum (A_i^2)} \cdot \sqrt{\sum (B_i^2)}}$ |

## Practical Considerations in Choosing Distance Metrics

### Data Characteristics

1. **Dimensionality**:
   - High-dimensional data (>100 dimensions) often suffers from the "curse of dimensionality"

- Manhattan distance tends to perform better than Euclidean in very high dimensions
- Cosine similarity can be more effective when dimensionality reduction is not feasible

2. **Sparsity**:
   - For sparse vectors (many zeros), cosine similarity and Manhattan distance are often more appropriate
   - Specialized sparse vector implementations can dramatically improve performance
3. **Scale and Distribution**:
   - If features have wildly different scales, Euclidean distance may give misleading results
   - Consider normalization or standardization before using distance metrics
   - Mahalanobis distance automatically accounts for different scales and correlations

### Application Requirements

1. **Semantic Search**:
   - Cosine similarity is often preferred for text embeddings
   - Domain-specific embeddings may perform better with specific distance metrics
2. **Image Similarity**:
   - Perceptual hashing often uses Hamming distance
   - CNN feature vectors may work better with Euclidean or cosine similarity
3. **Recommendation Systems**:
   - User preference vectors often use cosine similarity
   - Context-dependent recommendations might benefit from Mahalanobis distance
4. **Time Series Data**:
   - Dynamic Time Warping (DTW) for variable-length sequences
   - Euclidean distance for fixed-length, aligned sequences

### Performance and Scalability

1. **Query Speed vs. Accuracy Tradeoffs**:
   - Exact nearest neighbor search is often prohibitively slow for large datasets
   - Approximate methods trade some accuracy for dramatic speed improvements
   - The choice of distance metric affects which approximation methods work best
2. **Hardware Considerations**:
   - Some distance metrics can be efficiently computed on GPUs
   - SIMD instructions can accelerate certain distance calculations on CPUs
   - Memory bandwidth limitations may favor simpler metrics in practice
3. **Indexing Overhead**:

- Building and maintaining indices has costs in terms of memory and computation
- Different distance metrics require different index structures
- Update frequency affects which indexing approach is most efficient

## Evaluation of Distance Metrics

### Intrinsic Evaluation

1. **Recall@k**: Percentage of true nearest neighbors found among the top k results
2. **Precision@k**: Percentage of relevant items among the top k results
3. **Mean Average Precision (MAP)**: Average precision across multiple queries
4. **Discounted Cumulative Gain (DCG)**: Measures the quality of ranking

### Extrinsic Evaluation

1. **Task-Specific Performance**: How well does the chosen metric improve the final application?
2. **User Satisfaction**: Do users find the results relevant and useful?
3. **A/B Testing**: Direct comparison of different metrics in production environments

### Example: Comparing Distance Metrics for Text Embeddings

```
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity, manhattan_distances,
euclidean_distances
from transformers import AutoTokenizer, AutoModel
import torch

# Load pre-trained model and tokenizer
tokenizer = AutoTokenizer.from_pretrained("sentence-transformers/all-MiniLM-L6-
v2")
model = AutoModel.from_pretrained("sentence-transformers/all-MiniLM-L6-v2")

# Sample sentences
sentences = [
    "The cat sat on the mat",
    "A feline was resting on a rug",
    "The dog played in the yard",
    "My computer needs an upgrade"
]

# Generate embeddings
def generate_embeddings(texts):
    # Tokenize and prepare input
    inputs = tokenizer(texts, padding=True, truncation=True, return_tensors="pt")

    # Get model output
    with torch.no_grad():
```

```
        outputs = model(**inputs)

    # Mean pooling to get sentence embeddings
    embeddings = outputs.last_hidden_state.mean(dim=1)
    return embeddings.numpy()

embeddings = generate_embeddings(sentences)

# Compare similarities using different metrics
for i, sent1 in enumerate(sentences):
    print(f"\nSimilarities for: '{sent1}'")
    for j, sent2 in enumerate(sentences):
        if i != j:
            cos_sim = cosine_similarity([embeddings[i]], [embeddings[j]])[0][0]
            euc_dist = euclidean_distances([embeddings[i]], [embeddings[j]])[0]
[0]
            man_dist = manhattan_distances([embeddings[i]], [embeddings[j]])[0]
[0]

            print(f"  vs '{sent2}':")
            print(f"    Cosine similarity:  {cos_sim:.4f}")
            print(f"    Euclidean distance: {euc_dist:.4f}")
            print(f"    Manhattan distance: {man_dist:.4f}")
```

## Conclusion: Choosing the Right Distance Metric

Selecting the appropriate distance metric is a critical decision that impacts the effectiveness, efficiency, and accuracy of vector search systems. Key takeaways include:

1. **No Universal Best Metric**: The optimal choice depends on data characteristics, application requirements, and performance constraints.
2. **Start Simple**: Begin with standard metrics like Euclidean distance or cosine similarity before exploring more complex options.
3. **Empirical Testing**: Whenever possible, evaluate multiple metrics on your specific data and task rather than relying solely on theoretical properties.
4. **Consider the Full Pipeline**: The distance metric is just one component of a vector search system that includes embedding generation, indexing, filtering, and reranking.
5. **Hybrid Approaches**: Sometimes combining multiple distance metrics or using different metrics at different stages (coarse search vs. fine reranking) yields the best results.

Vector search systems continue to evolve rapidly, with new distance metrics, approximation techniques, and hardware optimizations constantly emerging. Understanding the fundamental properties and trade-offs of different distance

metrics provides the foundation needed to make informed decisions in this dynamic field.

### ChromaDB

```
collection.add(
    documents=["This is a document about cat", "This is a document about car"],
    metadatas=[{"category": "animal"}, {"category": "vehicle"}],
    ids=["id1", "id2"]
)

results = collection.query(
    query_texts=["vehicle"],
    n_results=2
)

results
```

The output is:

```
{'ids': [['id2', 'id1']],
 'distances': [[0.8069301247596741, 1.648103952407837]],
 'metadatas': [[{'category': 'vehicle'}, {'category': 'animal'}]],
 'embeddings': None,
 'documents': [['This is a document about car',
   'This is a document about cat']]}
```