Vector-Database

Overview

- Database of high-dimensional vectors
- Vectors generated by transformation or embedding function
- Useful for natural language processing, computer vision, recommendation system and more

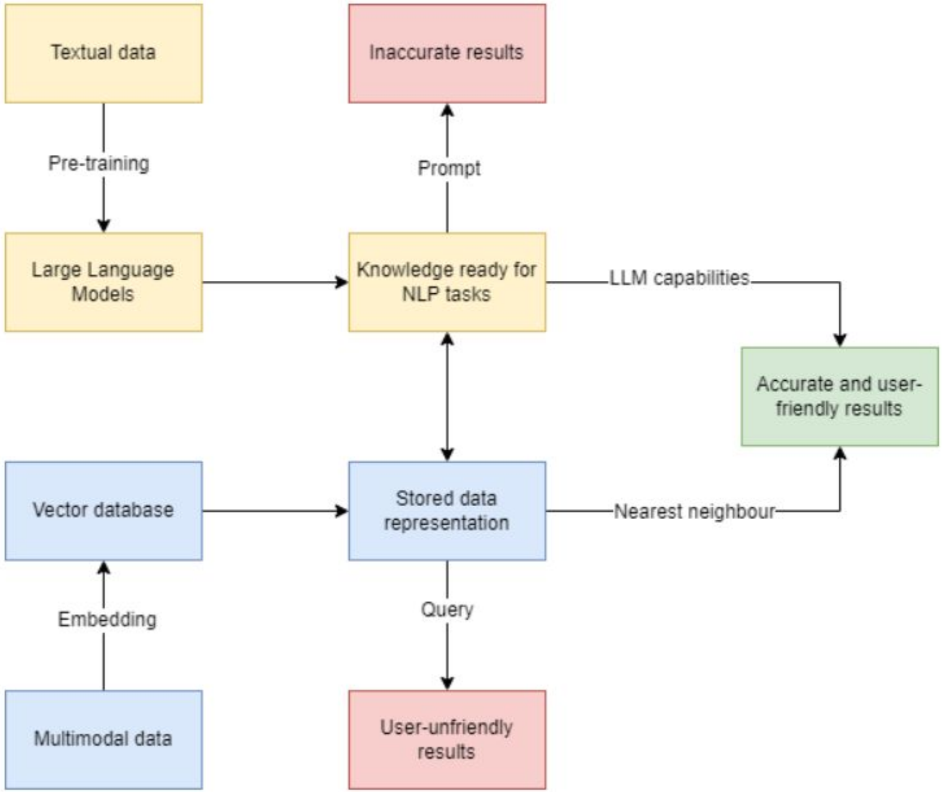
Advantages

- Fast and accurate similarity search and retrieval
- Support for complex and unstructured data
- Scalable and performant
  - Sharding, partitioning, caching, replication

Challenges

- Complex searching
- Different Vector Data Types
  - Sparse, different dimensions
- Easy API and connectors for frameworks to use
  - TensorFlow, PyTorch, Scikit-learn

LLMs



Textual data
Pre-training
Large Language Models
Inaccurate results
Prompt
Knowledge ready for NLP tasks
LLM capabilities
Accurate and user-friendly results
Vector database
Stored data representation
Nearest neighbour
Embedding
Query
Multimodal data
User-unfriendly results

LLMs ←→ Vector Databases

- Reduce hallucination
- Real-time knowledge
- Model compression
- Text generation
- Text augmentation
- Easy to update
- Can provide sources
- Reduce time and cost

Vector-Search: Brute-Force

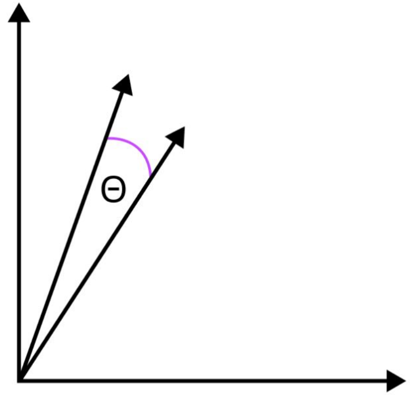
- Calculate distance from query vector to all vectors in the database
- Advantages
  - Guarantees the exact nearest neighbors.
  - Simple to implement.
- Disadvantages
  - Computationally expensive for large datasets
  - Does not scale well with the number of vectors or the dimensionality of the vectors
- Examples: Euclidean/cosine similarity

Vector-Search: Brute-Force

```python
def cosine_similarity(vec1, vec2):
    vec1 = np.array(vec1)
    vec2 = np.array(vec2)
    dot_product = np.dot(vec1, vec2)
    norm_vec1 = np.linalg.norm(vec1)
    norm_vec2 = np.linalg.norm(vec2)
    return dot_product / (norm_vec1 * norm_vec2)
```

Cosine Similarity

$$\cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \sqrt{\sum_{i=1}^{n} B_i^2}}$$

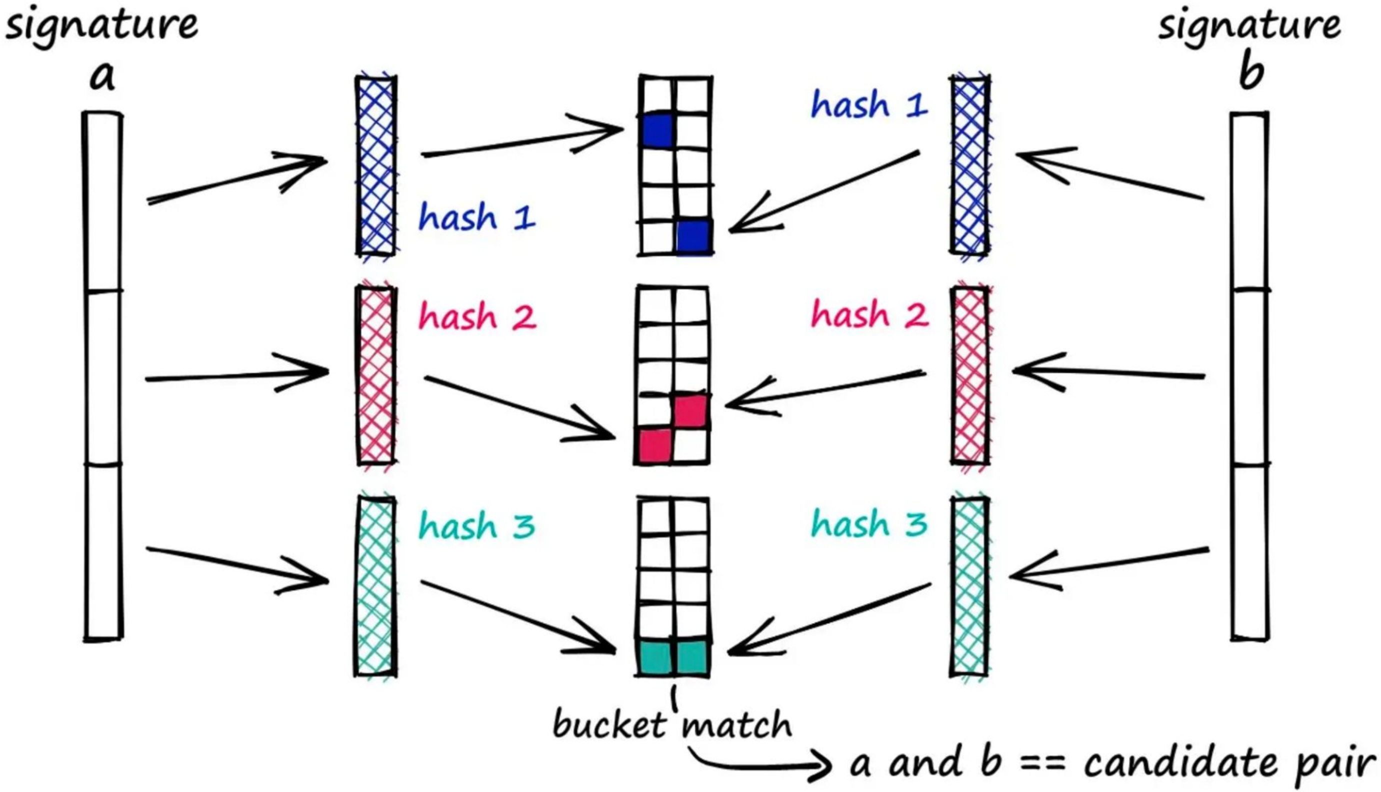


Vector-Search: ANN Search

- Approximate Nearest Neighbor (ANN) Search with Locality-Sensitive Hashing (LSH)
    - Hashing function that maximizes collision (for similar inputs)
- Advantages
    - Faster search for high-dimensional data
    - Tunable trade-offs between speed and accuracy
- Disadvantages
    - Provides approximate, not exact, results
    - Hash function selection and tuning can be complex

Vector-Search: ANN Search

-



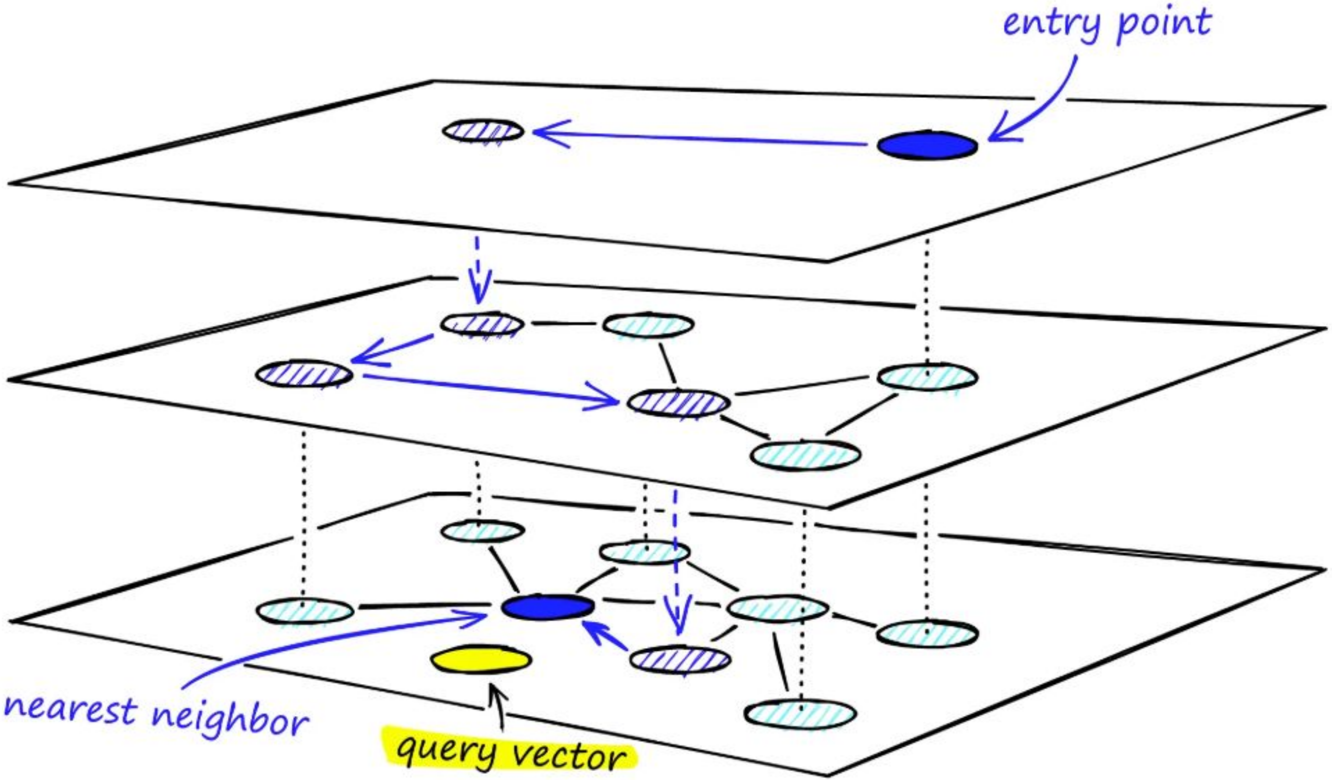
Vector-Search: ANN Search

```python
class LSH:
    def __init__(self, num_vectors, num_hash_tables, dimension):
        self.num_vectors = num_vectors
        self.num_hash_tables = num_hash_tables
        self.dimension = dimension
        self.hash_tables = [defaultdict(list) for _ in range(num_hash_tables)]
        self.random_vectors = [np.random.randn(dimension) for _ in range(num_hash_tables)]

    def _hash(self, vector, random_vector):
        return 1 if np.dot(vector, random_vector) > 0 else 0

    def _get_hash(self, vector):
        return tuple([self._hash(vector, rv) for rv in self.random_vectors])

    def add(self, vector, idx):
        for table_idx, hash_table in enumerate(self.hash_tables):
            hash_key = self._hash(vector, self.random_vectors[table_idx])
            hash_table[hash_key].append(idx)

    def query(self, vector, num_candidates=10):
        candidates = set()
        for table_idx, hash_table in enumerate(self.hash_tables):
            hash_key = self._hash(vector, self.random_vectors[table_idx])
            if hash_key in hash_table:
                candidates.update(hash_table[hash_key])

        return list(candidates)[:num_candidates]
```

Vector-Search: HNSW

- Hierarchical Navigable Small World
  - Graph structure where nodes represent vectors, and edges connect vectors that are close to each other
  - During search, it starts from a randomly selected node and navigates towards the query vector by exploring neighboring nodes iteratively
- Advantages
  - Very efficient for high-dimensional vector search
  - Balances speed and accuracy
- Disadvantages
  - High memory overhead due to graph structure
  - Initial graph construction is computationally expensive

Vector-Search: HNSW



Chroma Vector DB

- Create client:

```python
import chromadb
chroma_client = chromadb.Client()
```

- [Alternative] Create persistent client (creates sqlite3-file at location):

```python
client = chromadb.PersistentClient(path="/path/to/save/to")
```

- Possible to run in client-server mode

```python
import chromadb
chroma_client = chromadb.HttpClient(host='localhost', port=8000)
```

Chroma Vector DB

- Create collection (where the embeddings, docs, and metadata are stored)

```
collection = chroma_client.create_collection(name="my_collection")
```

- Simply adding documents to the collection

```
collection.add(
    documents=[
        "This is a document about pineapple",
        "This is a document about oranges"
    ],
    ids=["id1", "id2"]
)
```

Chroma Vector DB

- Querying

```python
results = collection.query(
    query_texts=["This is a query document about hawaii"],
    n_results=2 # how many results to return
)
print(results)
```

- Result:

```
{
  'documents': [[
      'This is a document about pineapple',
      'This is a document about oranges'
  ]],
  'ids': [['id1', 'id2']],
  'distances': [[1.0404009819030762, 1.243080496788025]],
  'uris': None,
  'data': None,
  'metadatas': [[None, None]],
  'embeddings': None,
}
```