DeepLearning.AI

Vector Databases: from Embeddings to Applications

DeepLearning.AI

Outline

- How to obtain vector representations of data?
  - Embeddings
- Searching for Similar Vectors
  - Distance Metrics
- Approximate Nearest Neighbors
  - ANN - Trade recall for accuracy
  - HNSW
- Vector DB's
  - CRUD operations
  - Objects + Vectors
  - Inverted Index - filtered search
- Sparse vs Dense Search
  - ANN search over Dense embeddings
  - Sparse search
  - Hybrid Search
- Applications of Vector DBs in Industry

How can we measure the distance between these Image and Sentence Embeddings?

There are many ways to calculate the distances between two vectors.

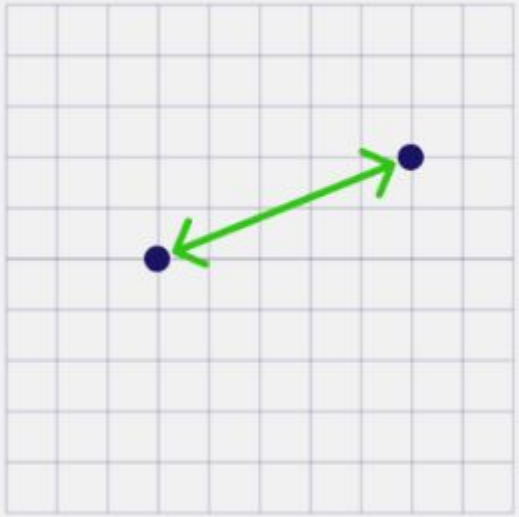
Here we will cover 4 distance metrics that you might find being used in the context of vector databases:

- Euclidean Distance(L2)
- Manhattan Distance(L1)
- Dot Product
- Cosine Distance

Euclidean Distance(L2)

Euclidean Distance(L2)

The length of the shortest path between two points or vectors.



Squared Euclidean (L2 Squared)

$$\sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

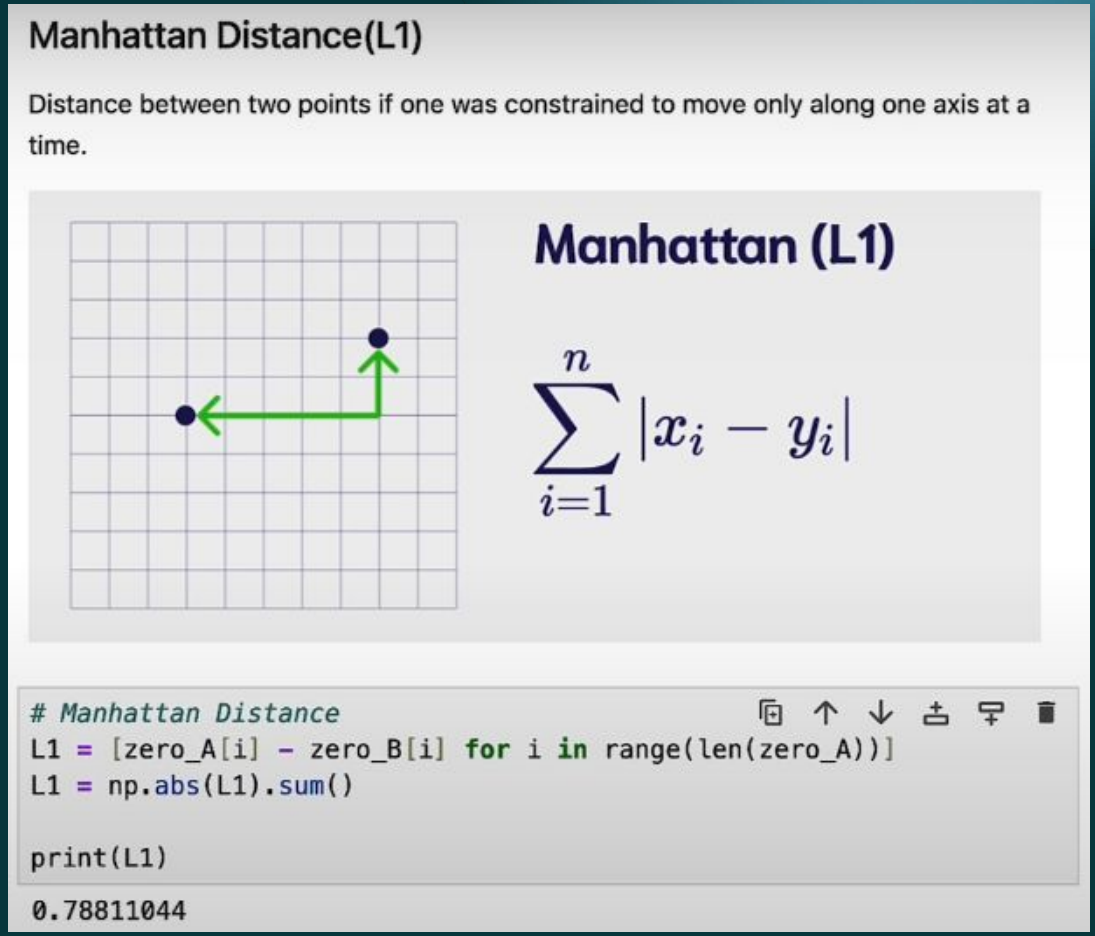
Manhattan Distance(L1)

Manhattan distance is one way to measure the distance between two points, like navigating on a chessboard or city blocks, where you can only move horizontally and vertically to reach the shortest path. Not being able to move diagonally.

Imagine we have two points:

- **Point A:** Located at coordinates (x1,x2)
- **Point B:** Located at coordinates (y1,y2)

Manhattan Distance
$=|x1-y1|+|x2-y2|$



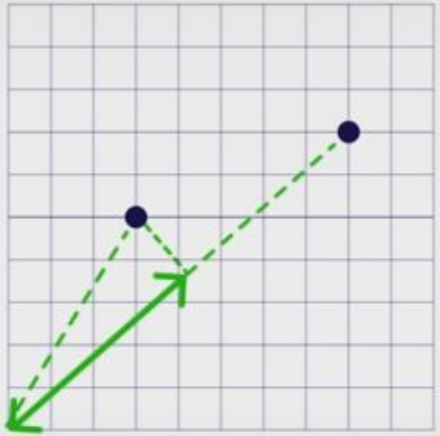
The Dot Product is a way to represent, with a single number, **how much two vectors (quantities with both direction and magnitude) are pointing in the same direction, and what their combined "magnitude" is.**

Example:
Let's say
Vector A = (2, 3) & Vector B = (4, 1)

Dot Product A · B = (2 × 4) + (3 × 1) = 8 + 3 = 11.

Dot Product

Measures the magnitude of the projection of one vector onto the other.



Dot Product

$$A \cdot B = \sum_{i=1}^{n} A_i B_i$$

```
# Dot Product
np.dot(zero_A,zero_B)
```

```
3.6484127
```

Cosine Distance is a method to measure how much two vectors are pointing in different directions. In essence, it quantifies the 'difference' in their orientation.

Key Meanings:
✳️Directional Similarity/Difference:
🔆If two vectors point in exactly the same direction, their cosine distance approaches 0.

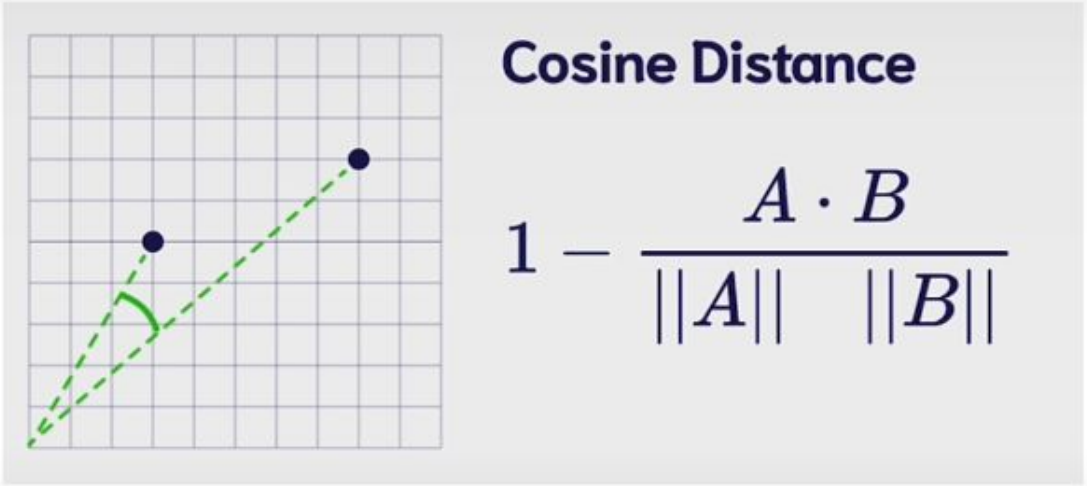
🔆If two vectors point in completely opposite directions, their cosine distance approaches 2.

🔆If two vectors are at a right angle (90 degrees) to each other, their cosine distance becomes 1.

✳️Magnitude Doesn't Matter: Unlike Manhattan or Euclidean distance, cosine distance does not care about the 'length' or 'magnitude' of the vectors; it only considers their 'direction'. No matter how long or short the vectors are, if they point in the same direction, they are considered similar in terms of cosine distance.

Cosine Distance

Measure the difference in directionality between vectors.

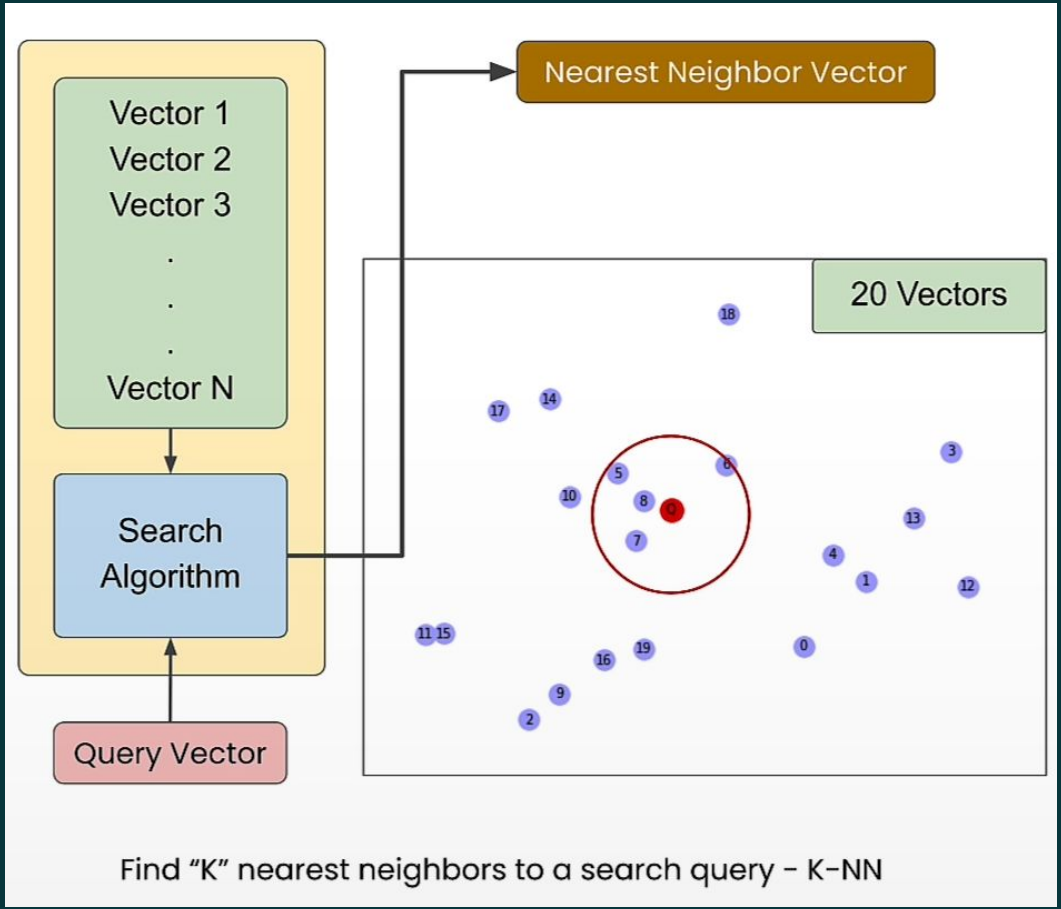



```
# Cosine Distance
cosine = 1 - np.dot(zero_A,zero_B)/(np.linalg.norm(zero_A)*np.linalg
print(f"{cosine:.6f}")
```

```
0.007732
```

Search for Similar Vectors

Searching for Similar Vectors

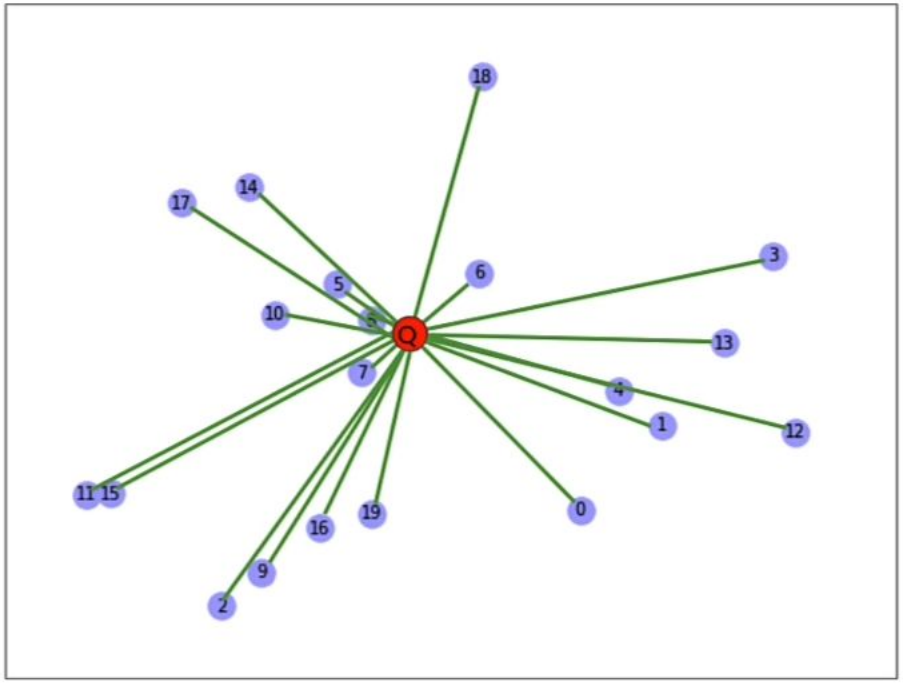


Find "K" nearest neighbors to a search query - K-NN

Weaviate

DeepLearning.AI

‘brute force’ search algorithm



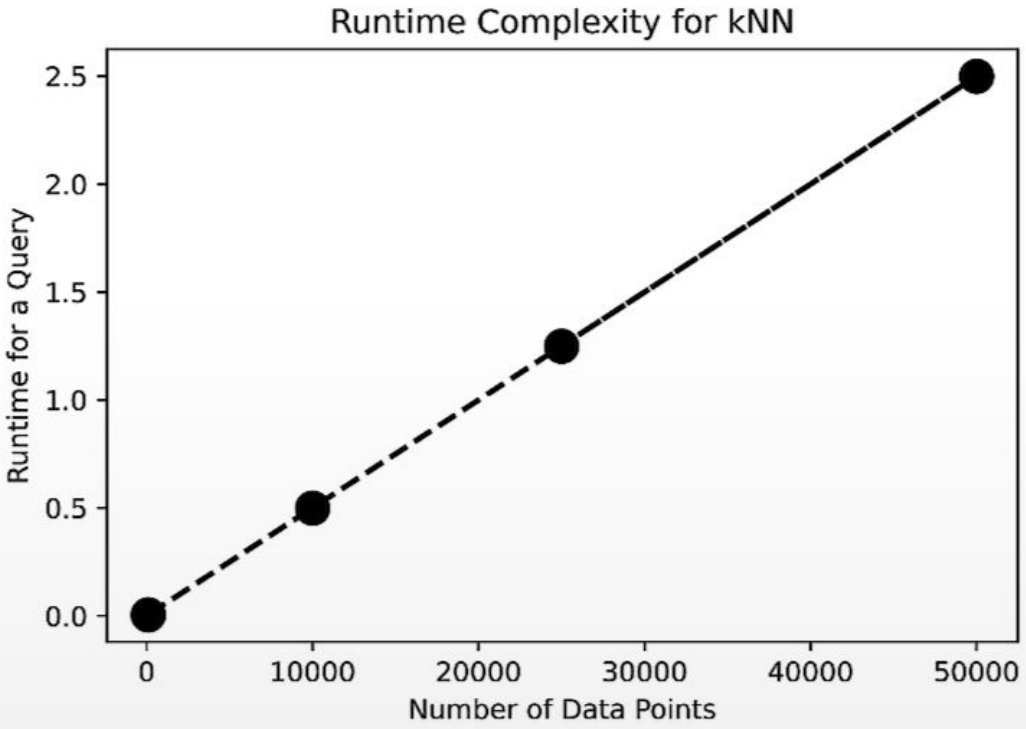
1. Measure the L2 distance between the Query and each vector.

2. Sort all those distances.

3. Return the top k matches. These are the most semantically similar points.

Runtime complexity of kNN

- ***O(dN)* runtime complexity for search**

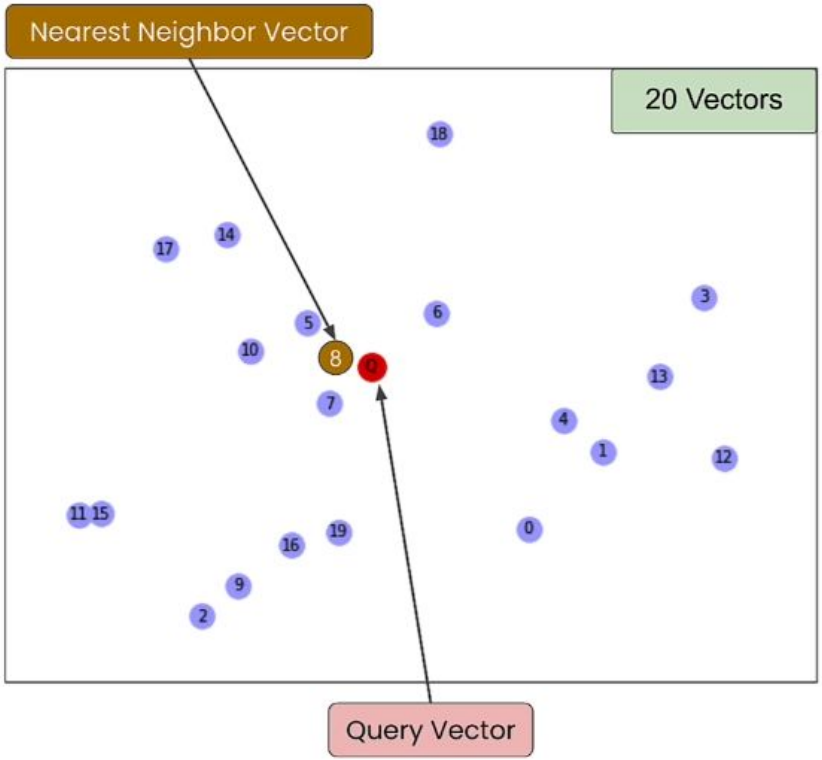


ANN

(Approximate nearest neighbours)

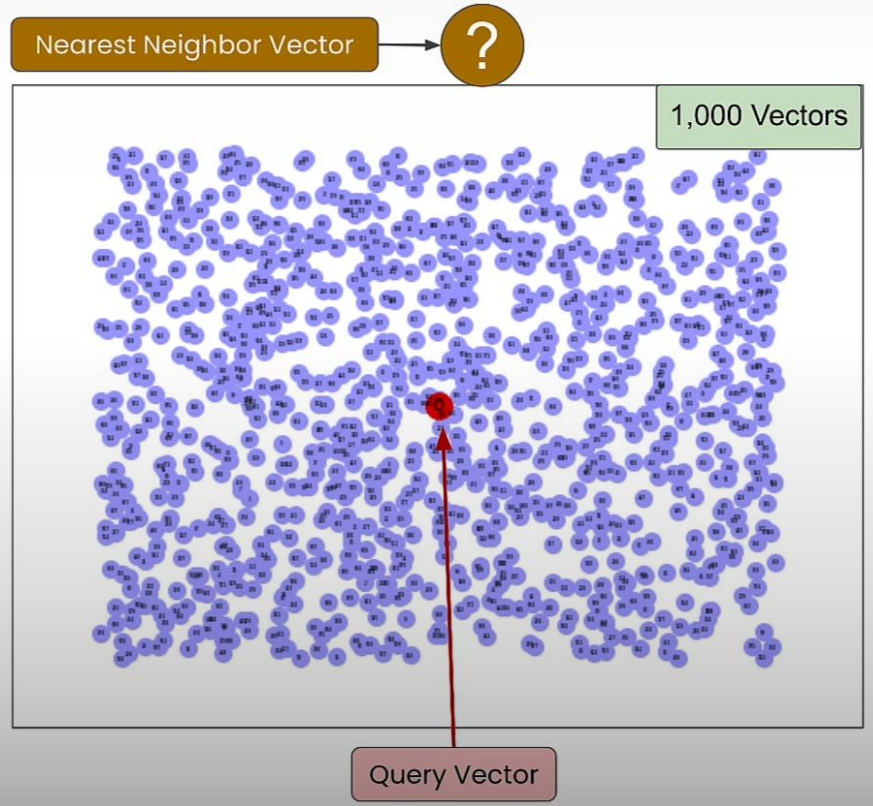
Weaviate DeepLearning.AI

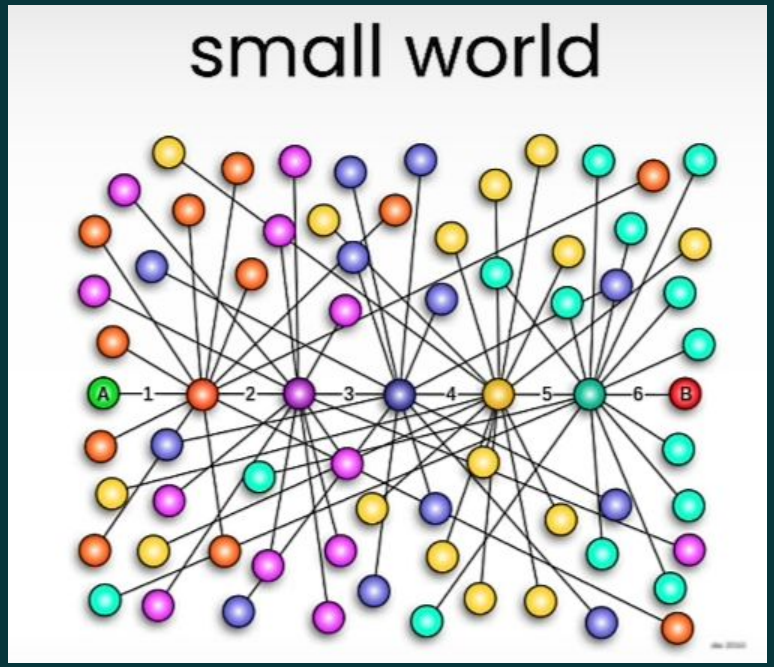
Not a Big Problem



Weaviate DeepLearning.AI

The Problem

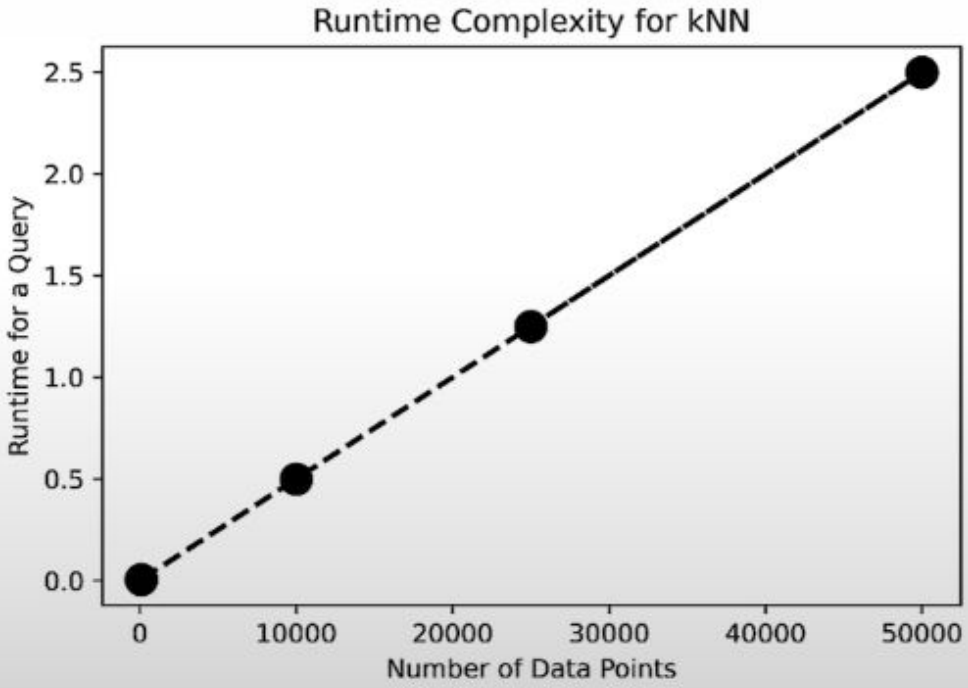




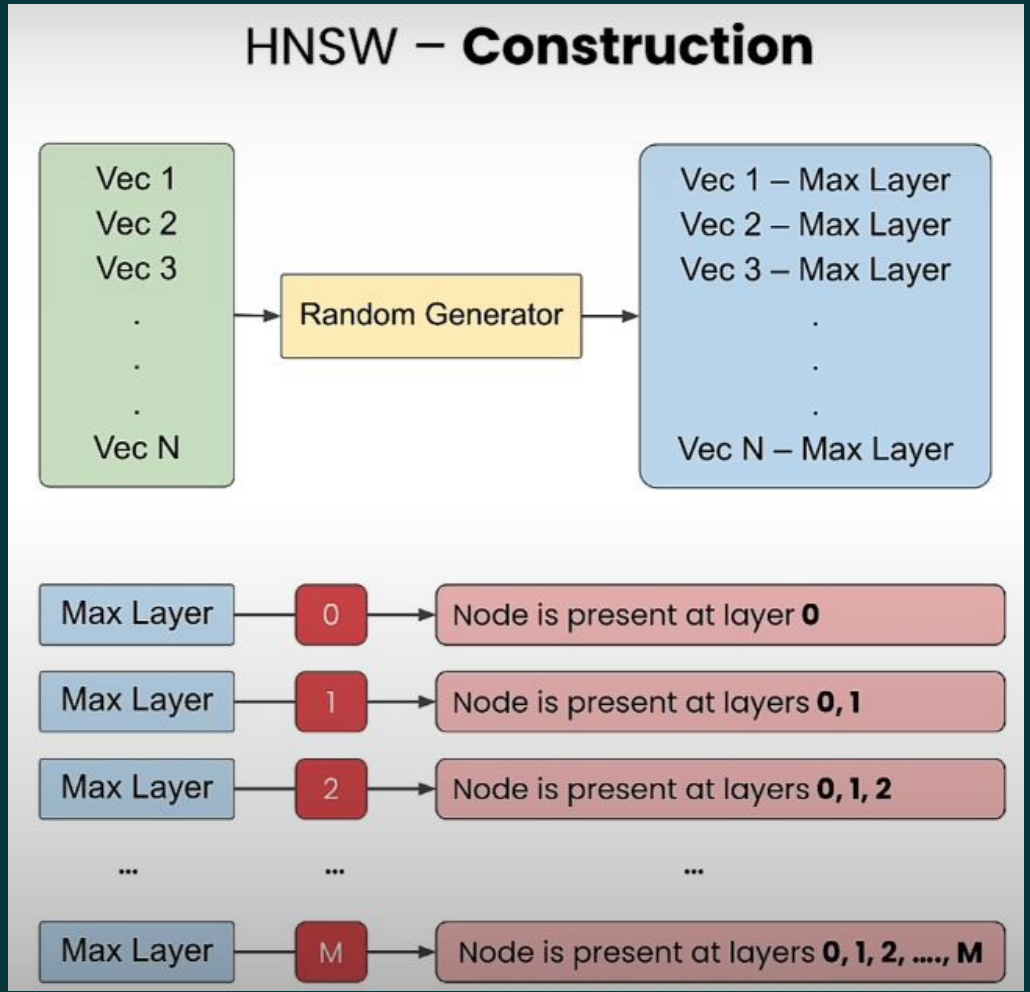
- Observed phenomenon in human social networks where everyone is closely connected.
- **Six Degrees of Separation:** Popularized concept that any two individuals are on average separated by six acquaintance links.
- **High Clustering:** Nodes form tightly-knit groups, with members of groups connecting to one another.
- **Short Path Lengths:** Despite high clustering, only a few steps are needed to connect any two nodes.

Runtime Complexity of kNN

- ***O(dN)* runtime complexity for search**



Large Computational Cost



1. **Vector Input**: You start with many vectors (like sentence or image embeddings).

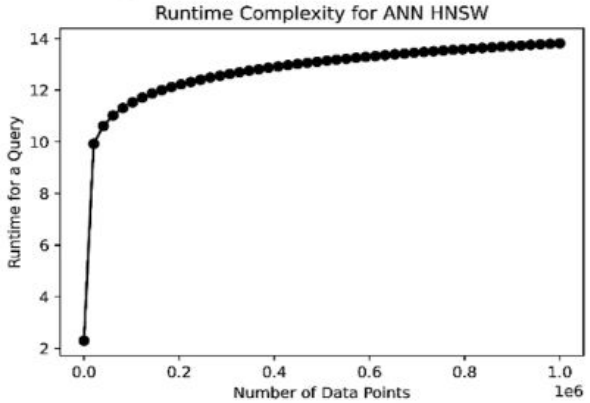
2. **Random Generator**: Each vector is randomly assigned a **maximum layer level**.
   - The **higher the layer**, the **less likely** a vector is placed there.

3. **Node Placement**:
   - Some nodes are only in **layer 0**
   - Others in **layers 0 and 1**
   - Some in **layers 0, 1, 2**, and so on...
     → The result looks like a **pyramid-shaped graph** where higher layers have fewer nodes.

📌 Why?
This structure allows you to **start the search from the top (broad scan)** and **refine it as you go down to lower layers (more accurate search)**.

HNSW Runtime

- **Low likelihood in Higher Levels**
  - The likelihood of a vector being found on the higher up levels of our HNSW graph goes down exponentially
- **Query time increases Logarithmically**
  - This means that as the number of datapoints increases the number of comparisons to perform vector search only goes up logarithmically.
- ***O(log(N))* runtime complexity**
  - As a results we have a *O(log(N))* runtime complexity for the HNSW algorithm



Key Ideas:

🔻 **Fewer Nodes in Higher Layers**

- The chance of a vector appearing in upper layers decreases exponentially.

📈 **Logarithmic Query Time**

- As the number of vectors increases, the **search time increases slowly** — not linearly.

⏱️ **Runtime Complexity = O(log N)**

- This means even with a **large dataset**, search remains **very fast**.
- For example, even with a million vectors, search time stays manageable.

📊 The graph shows that **as data grows**, **query time grows very slowly**, which is ideal for large-scale vector search.

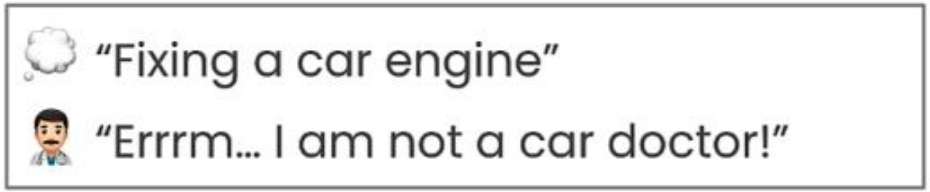
Sparse vs Dense Search

- **Dense Search (Semantic Search)**
  - Uses vector embedding representation of data to perform search.
    (as described in the previous lessons)
  - This types of search allows one to capture and return semantically similar objects.

Weaviate DeepLearning.AI

Dense Search Challenge

- **Out of domain data -** will provide poor accuracy
  - A Neural Network is as good as the data it was trained on.



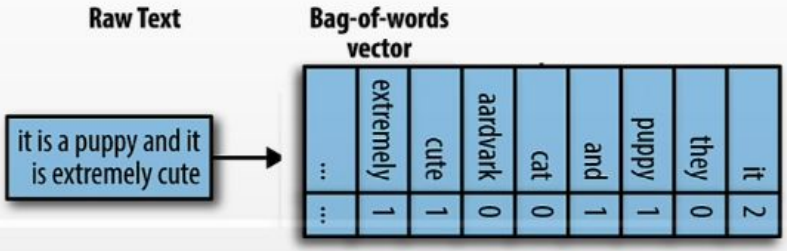
Dense Search Challenge

- **Product with a Serial Number**
  - Searching for seemingly random data (like serial numbers) will also yield poor results.

- **String or Word Matching**
  - Here we would be better off doing exact string or word matching to see where which product has the same serial number as the input serial number.
- This is known as **keyword search or sparse search!**

Sparse vs Dense Search

- **Bag of Words**
  - The easiest way to do keyword matching is using Bag of Words - to count how many times a word occurs in the query and the data vector and then return objects with the highest matching word frequencies.
- **This is known as Sparse Search**
  - because the text is embedded into vectors by counting how many times every unique word in your vocabulary occurs in the query and stored sentences.
- **Mostly Zeroes (Sparse Embedding)**
  - Since the likelihood of any given sentence containing every word in your vocabulary is quite low the embeddings is mostly zeroes and thus is known as a sparse embedding.



Weaviate DeepLearning.AI

BM25

Best Matching 25 (BM25): In practice when performing keyword search we use a modification of simple word frequencies called *best matching 25*

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

Learn more about BM25: https://en.wikipedia.org/wiki/Okapi_BM25

"If you can look into the seeds of time, And say which grain will grow and which will not."

[0.,0.,0.,0.33052881,0.,.,0.,0.20961694,0.,0.,.,0.,0.,0.,0.,.,0.,0.,0.20961694,0.20961694,.,0.20961694,0.20961694,0.,0.20961694,.20961694,0.20961694,0.,0.,0.,.,0.,0.,0.,0.20961694,0.,0.20961694,0.10938682,.,0.,0.20961694,0.41923388,0.41923388,0.,0.,0.20961694]

Term Explanation

Symbol	Meaning	In Simple Terms
q_i	A keyword from the query	e.g., "grain", "time"
$f(q_i, D)$	Frequency of keyword q_i in document D	How many times the word appears in the document
(	D	)
$avgdl$	Average document length	Average number of words across all documents
$IDF(q_i)$	Inverse Document Frequency	Gives more weight to rare words
k_1, b	Tuning parameters	Adjust the score sensitivity (e.g., k_1=1.2, b=0.75)

💡 Really Simple Summary

BM25 works like this:

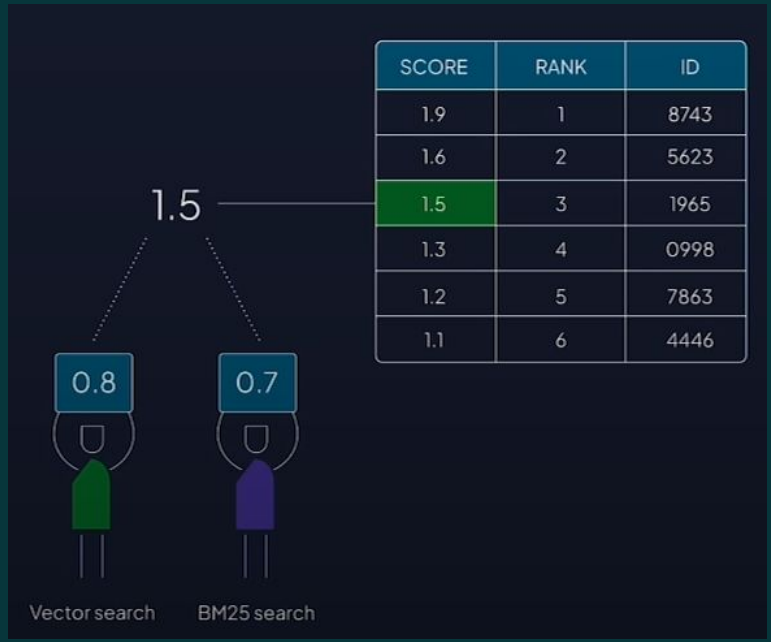
1. **Each keyword in your query**
2. Is checked for **how often it appears in the document**
3. Then **boosted if the word is rare across the whole dataset**
4. And **adjusted to not over-reward longer documents**
5. The final score tells how well the document matches your query.

Weaviate

DeepLearning.AI

Hybrid Search

- **What is Hybrid Search?**
  - Hybrid search is the process of performing both vector/dense search and keyword/sparse search and then combining the results
- **Combination based on a Scoring System**
  - This combination can be done based on a scoring system that measures how well each objects matches the query using both dense and sparse searches.



SCORE	RANK	ID
1.9	1	8743
1.6	2	5623
1.5	3	1965
1.3	4	0998
1.2	5	7863
1.1	6	4446

Multilingual Search

- Because embedding produces vectors that convey meaning, vectors for the same phrase in different languages produces similar results.

"Vacation spots in California"

[-0.2479, -0.1360, -0.1075, 0.0973,
...
, -0.0055, -0.0283, -0.313, 0.1390]

"加利福尼亚州的度假胜地"

[-0.2479, -0.1360, -0.1075, 0.0973,
...
, -0.0055, -0.0283, -0.313, 0.1390]

Weaviate DeepLearning.AI

Retrieval Augmented Generation (RAG)

- **Using a Vector Database as an External Knowledge Base**
  - Enable a large language model (LLM) to leverage a vector database as an external knowledge base of factual information
- **Retrieve Relevant Info and provide to LLM**
  - Improve a LLM by enabling it to retrieve relevant source material from the vector database and read it as part of the prompt prior to generating an answer to the prompt
- **Synergizes with a Vector Database**
  - vector databases can be queried for concepts using natural language
- **Better to do RAG**
  - Performing RAG is a lot more practical than having the LLM attend over its trained knowledge base
- **Example: Visiting a Library**
  - It's akin to a human visiting a library and checking out and reading source material and books prior to writing a well thought out response to a question.

Weaviate

DeepLearning.AI

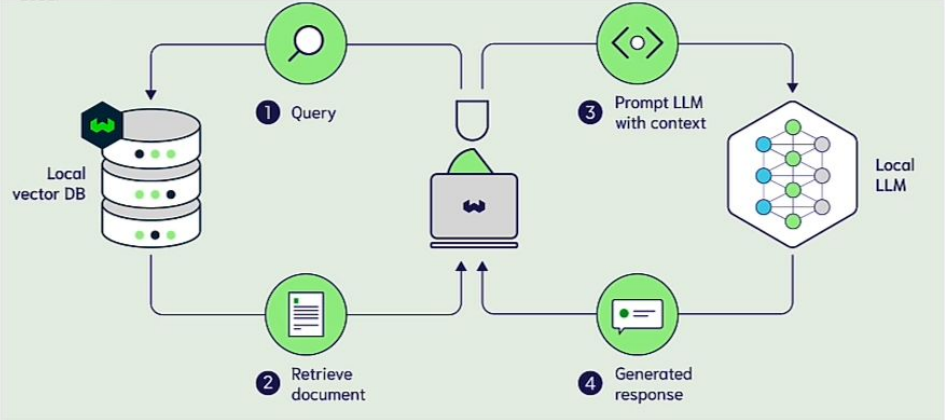
Advantages of RAG

Here we can list out the advantages of RAG if we have time

- **Reduce hallucinations** - furthermore allow the user to identify hallucinations by comparing generated text to source text
- **Enable a LLM to cite sources** that it used to generate answers
- **Solve knowledge intensive tasks** and prompts more accurately

The RAG Workflow

- At its simplest the **RAG workflow consists of 4 steps:**
  - Step 1: **Query a vector database**
  - Step 2: **Obtain relevant source objects**
  - Step 3: **Stuff the objects into the prompt**
  - Step 4: **Send the modified prompt to the LLM to generate an answer**



Performing RAG with Weaviate

The 4 step RAG workflow can be executed in Weaviate using one call!

🔄 Full Workflow: From Natural Language to Vector Search with Filters

1️⃣ User Input

- The user enters a natural language query.

📌 Example input:

```
"Show me laptops under $1000 with long battery life"
```

2️⃣ Convert the text to an embedding (vector) for similarity computation

- An embedding model (e.g., OpenAI, BERT, SentenceTransformer) converts the full sentence into a dense vector that captures its semantic meaning.

📌 Example output:

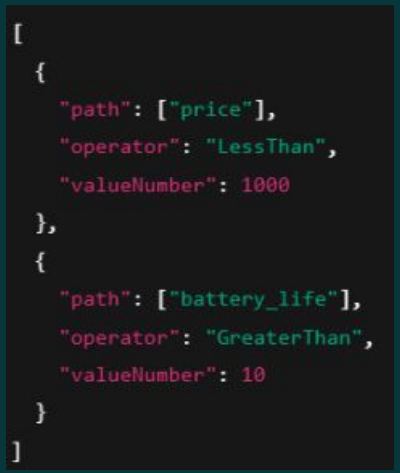
```
[0.23, -0.45, 0.87, ..., 0.13]
```

(This vector will be used in `.with_near_vector()` to find similar meanings in the database.)

The vector is used to find the top k most similar documents, and then filters are applied to narrow down the results

3️⃣ **Extract filter conditions from the sentence**

- A **parser or LLM** identifies structured conditions within the sentence.



```
[
  {
    "path": ["price"],
    "operator": "LessThan",
    "valueNumber": 1000
  },
  {
    "path": ["battery_life"],
    "operator": "GreaterThan",
    "valueNumber": 10
  }
]
```

4️⃣ Generate JSON filters using a parser logic or LCEL

- Tools like **LangChain Expression Language (LCEL)** or an LLM prompt will convert the extracted conditions into a JSON format compatible with `.with_where()`.

📌 **Generated filter (for Weaviate, Pinecone, etc.):**

```
{
  "operator": "And",
  "operands": [
    {
      "path": ["price"],
      "operator": "LessThan",
      "valueNumber": 1000
    },
    {
      "path": ["battery_life"],
      "operator": "GreaterThan",
      "valueNumber": 10
    }
  ]
}
```

5️⃣ Run vector search + apply filters

- This is where the system combines both the semantic vector and structured filters in a search query.

📌 **Example API call using Weaviate-style client**:

```
response = (
    client.query
    .get("LaptopCollection", ["title", "price", "battery_life"])
    .with_near_vector({ "vector": user_query_vector })
    .with_where(parsed_filter_json)
    .with_limit(5)
    .do()
)
```

6️⃣ Return the results

- The system retrieves the most **semantically relevant** and **filter-matching** items and returns them to the user.

📌 **Example output**: